APPENDIX J

US006182114B1

(12) **United States Patent**
**Yap et al.**

(10) **Patent No.: US 6,182,114 B1**
(45) **Date of Patent: Jan. 30, 2001**

(54) **APPARATUS AND METHOD FOR REALTIME VISUALIZATION USING USER-DEFINED DYNAMIC, MULTI-FOVEATED IMAGES**

(75) Inventors: **Chee K. Yap**; **Ee-Chien Chang**, both of New York, NY (US); **Ting-Jen Yen**, Jersey City, NJ (US)

(73) Assignee: **New York University**, New York, NY (US)

( * ) Notice: Under 35 U.S.C. 154(b), the term of this patent shall be extended for 0 days.

(21) Appl. No.: **09/005,174**

(22) Filed: **Jan. 9, 1998**

(51) **Int. Cl.**$^{7}$ ........................................ **G06F 15/16**

(52) **U.S. Cl.** ........................................ **709/203**; 709/246

(58) **Field of Search** ........................................ 709/217, 219, 709/246, 247, 203; 707/10; 382/103, 233, 235, 232, 240, 302

(56) **References Cited**

U.S. PATENT DOCUMENTS

| | | | | |
|---|---|---|---|---|
| 4,622,632 | | 11/1986 | Tanimoto . | |
| 5,341,466 | | 8/1994 | Perlin . | |
| 5,481,622 | * | 1/1996 | Gerhardt et al. | 382/103 |
| 5,568,598 | * | 10/1996 | Mack et al. | 382/302 X |
| 5,710,835 | * | 1/1998 | Bradley | 382/233 |
| 5,724,070 | * | 3/1998 | Denninghoff et al. | 382/235 X |
| 5,861,920 | * | 1/1999 | Mead et al. | 382/232 X |
| 5,880,856 | * | 3/1999 | Ferriere | 382/240 X |
| 5,920,865 | * | 7/1999 | Ariga | 707/10 |

OTHER PUBLICATIONS

Tams Frajka et al., Progressive Image Coding with Spatially Variable Resolution, IEEE, Proceedings International Conference on Image Processing 1997, Oct. 1997, vol. 1, pp. 53–56.*

E. C. Chang et al., "Realtime Visualization of Large . . . " Mar. 31, 11997,pp. 1–9, Courant Institute of Mathematical Sciences, New York University, N.Y. U.S.A.

E. C. Chang et al., "A Wavelet Approach to Foveating Images", Jan. 10, 1997,pp. 1–11, Courant Institute of Mathematical Sciences, New York University, N.Y. U.S.A.

S.G. Mallat, "A Theory for Multiresolutional Signal Decomposition . . . ", IEEE Transactions on Pattern Analysis and Machine Intelligence,pp. 3–23, Jul. 1989, vol. 11, No. 7, IEEE Computer Society.

News Release, "Wavelet Image Features",Summus'Wavelet Image Compression,Summus 14 pages.

R.L. White et al., "Compression and Progressive Transmission of Astronomical Images", SPIE Technical Conference 2199, 1994.

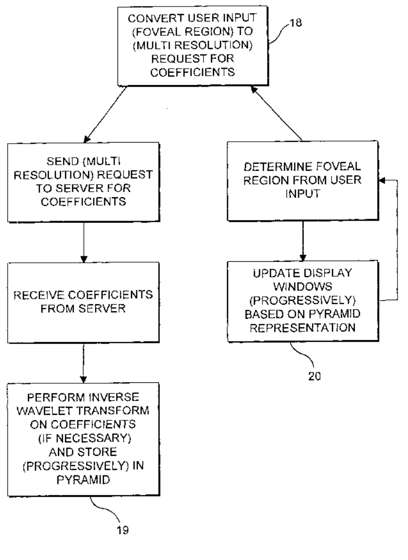
(List continued on next page.)

*Primary Examiner*—Zarni Maung
*Assistant Examiner*—Patrice Winder
(74) *Attorney, Agent, or Firm*—Baker Botts, L.L.P.

(57) **ABSTRACT**

A client apparatus which enables a realtime visualization of at least one image. The client apparatus includes a storage device which stores first data corresponding to a multifoveated representation of an original image, and a user input device which providing second data corresponding to at least one visualization command of at least one user. In addition, the client apparatus includes a processing arrangement which generates third data corresponding to a multifoveated image using the first data, the second data and a foveation operator.

**8 Claims, 6 Drawing Sheets**


CONVERT USER INPUT (FOVEAL REGION) TO (MULTI RESOLUTION) REQUEST FOR COEFFICIENTS
18
SEND (MULTI RESOLUTION) REQUEST TO SERVER FOR COEFFICIENTS
DETERMINE FOVEAL REGION FROM USER INPUT
RECEIVE COEFFICIENTS FROM SERVER
UPDATE DISPLAY WINDOWS (PROGRESSIVELY) BASED ON PYRAMID REPRESENTATION
20
PERFORM INVERSE WAVELET TRANSFORM ON COEFFICIENTS (IF NECESSARY) AND STORE (PROGRESSIVELY) IN PYRAMID
19

APPENDIX J

OTHER PUBLICATIONS

E.L. Schwartz, "The Development of Specific Visual . . . " Journal of Theoretical Biology, 69:655–685, 1977.

F.S. Hill Jr. et al.,"Interactive Image Query . . . " Computer Graphics, 17(3), 1983.

T.H. Reeves et al., "Adaptive Foveation of MPEG Video", Proceedings of the 4th ACM International Multimedia Conference, 1996.

R.S. Wallace et al., "Space–variant image processing". Int'l. J. of Computer Vision, 13:1(1994) 71–90.

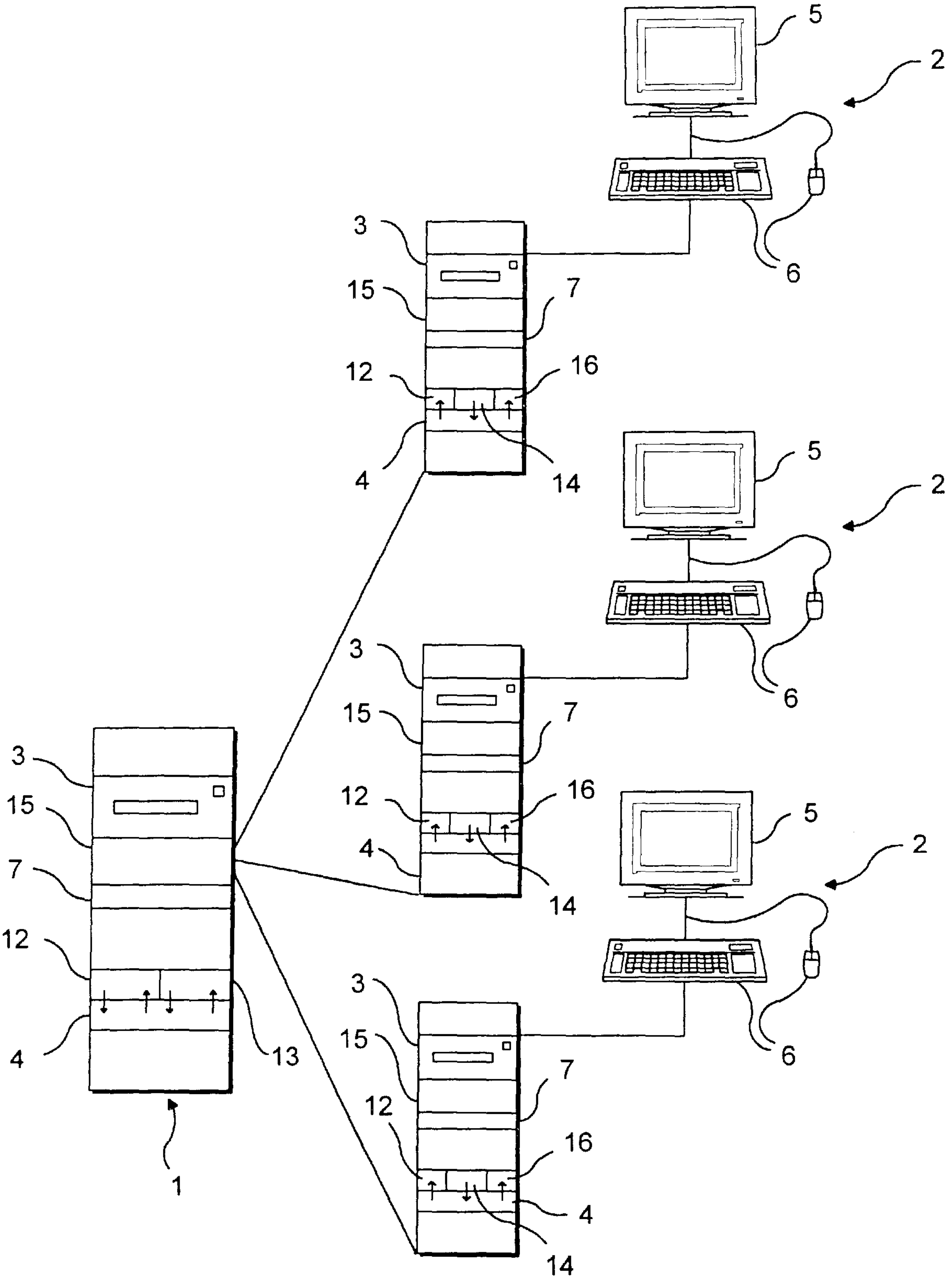
E.L. Schwartz A quantitative model of the functional architecture: Biological cybernetics, 37(1980) 63–76.

P. Kortum et al., "Implementation of a Foveated Image . . . " Human Vision and Electronic Imagining, SPIE Proceedings vol. 2657, 350–360, 1996.

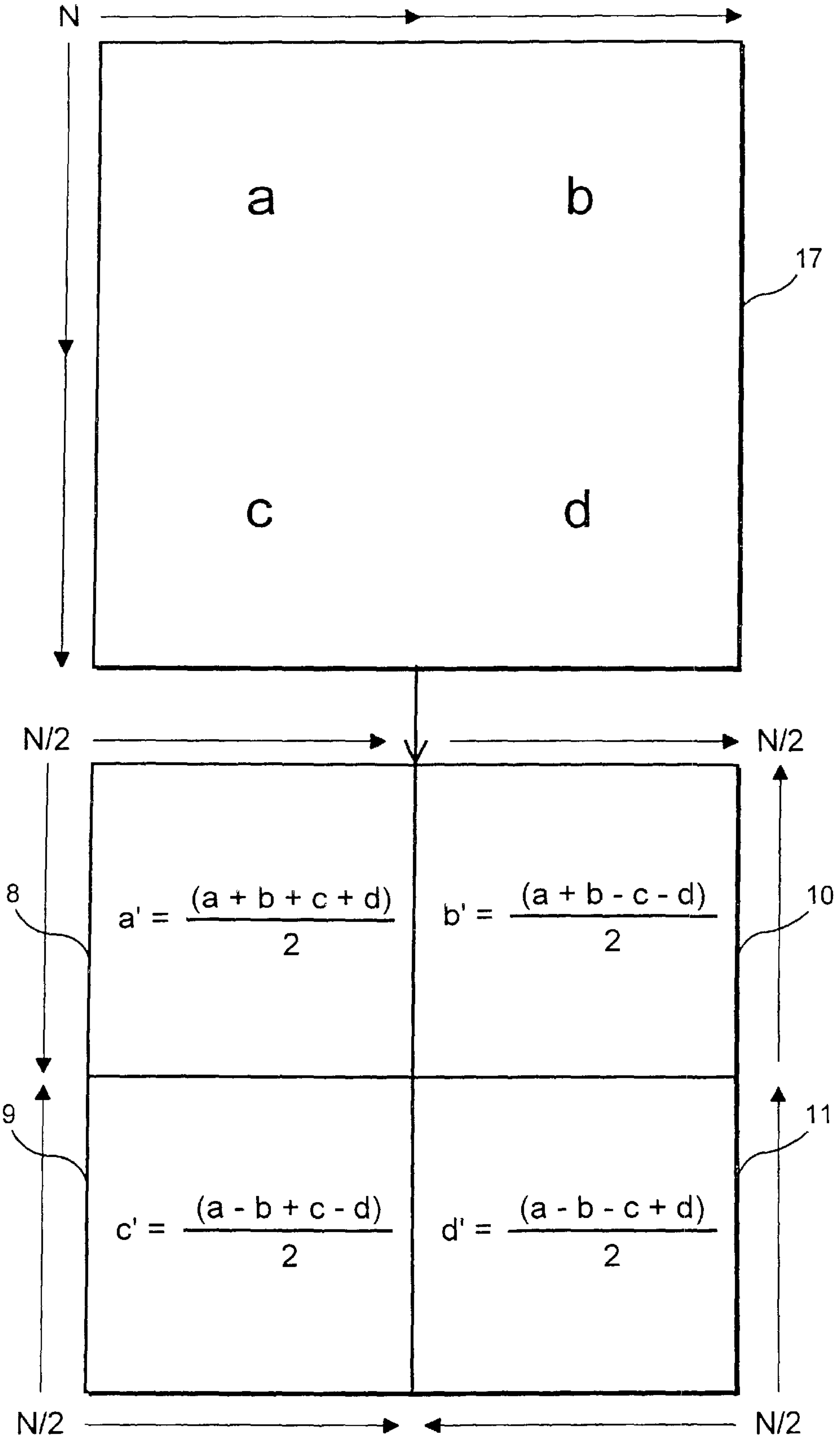
M.H. Gross et al., "Efficient triangular surface . . . ", IEEE Trans on Visualization and Computer Graphics, 2(2) 1996.

* cited by examiner

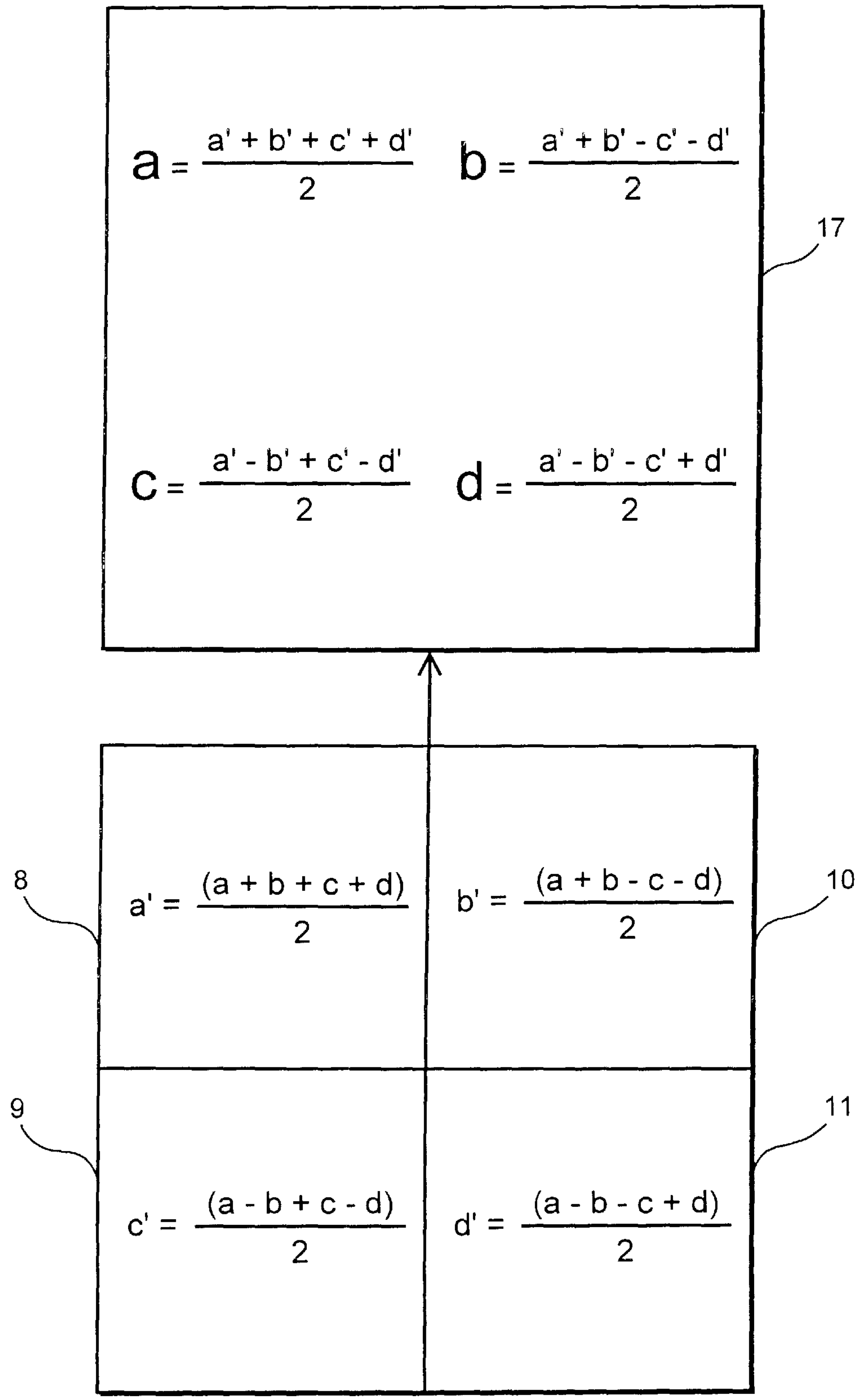
APPENDIX J



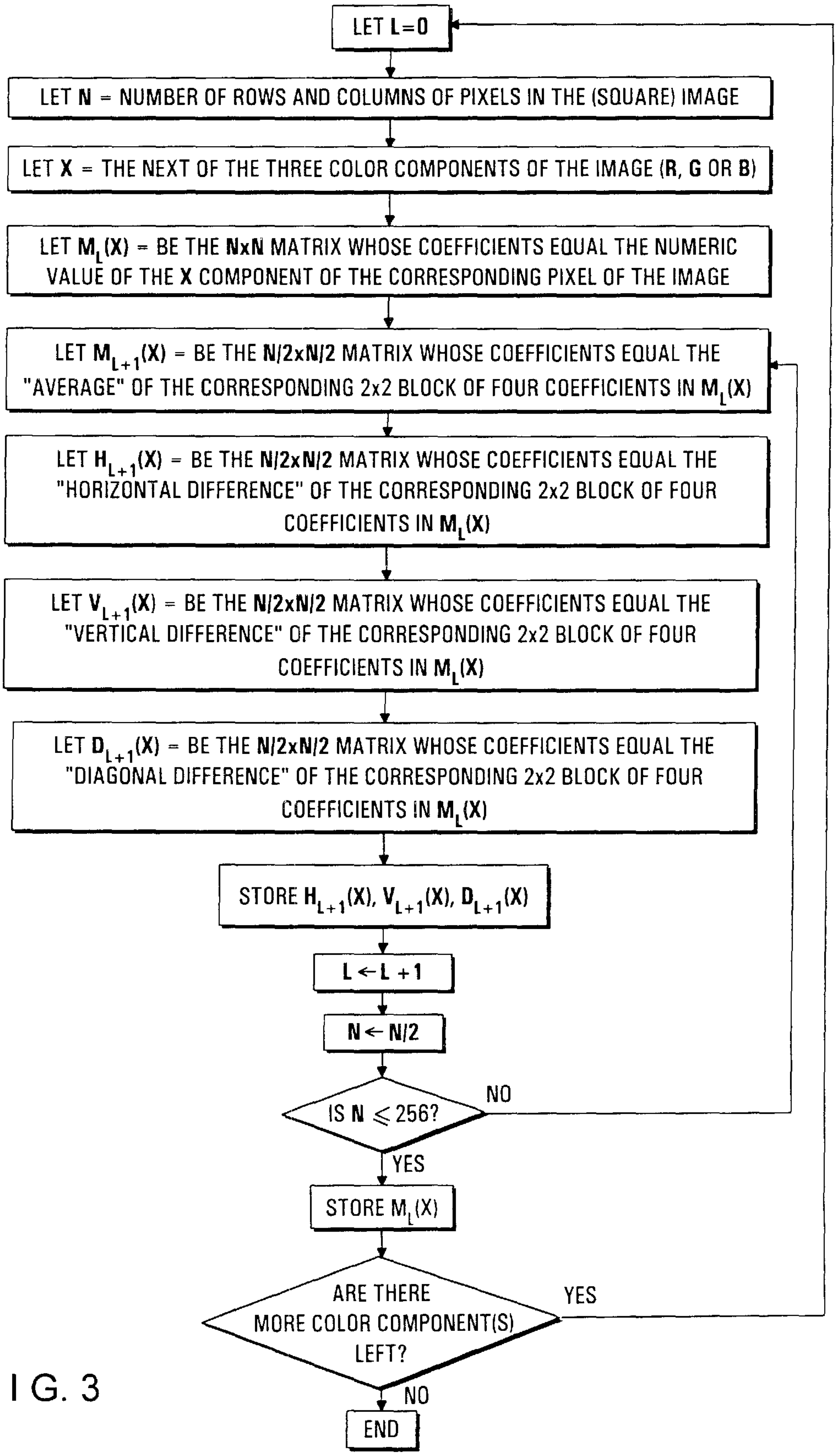
F I G. 1



F I G. 2A


$a = \frac{a' + b' + c' + d'}{2}$
$b = \frac{a' + b' - c' - d'}{2}$
$c = \frac{a' - b' + c' - d'}{2}$
$d = \frac{a' - b' - c' + d'}{2}$
17
8
$a' = \frac{(a + b + c + d)}{2}$
$b' = \frac{(a + b - c - d)}{2}$
10
9
$c' = \frac{(a - b + c - d)}{2}$
$d' = \frac{(a - b - c + d)}{2}$
11

F I G. 2B


LET L=0
LET N = NUMBER OF ROWS AND COLUMNS OF PIXELS IN THE (SQUARE) IMAGE
LET X = THE NEXT OF THE THREE COLOR COMPONENTS OF THE IMAGE (R, G OR B)
LET $M_L(X)$ = BE THE NxN MATRIX WHOSE COEFFICIENTS EQUAL THE NUMERIC VALUE OF THE X COMPONENT OF THE CORRESPONDING PIXEL OF THE IMAGE
LET $M_{L+1}(X)$ = BE THE N/2xN/2 MATRIX WHOSE COEFFICIENTS EQUAL THE "AVERAGE" OF THE CORRESPONDING 2x2 BLOCK OF FOUR COEFFICIENTS IN $M_L(X)$
LET $H_{L+1}(X)$ = BE THE N/2xN/2 MATRIX WHOSE COEFFICIENTS EQUAL THE "HORIZONTAL DIFFERENCE" OF THE CORRESPONDING 2x2 BLOCK OF FOUR COEFFICIENTS IN $M_L(X)$
LET $V_{L+1}(X)$ = BE THE N/2xN/2 MATRIX WHOSE COEFFICIENTS EQUAL THE "VERTICAL DIFFERENCE" OF THE CORRESPONDING 2x2 BLOCK OF FOUR COEFFICIENTS IN $M_L(X)$
LET $D_{L+1}(X)$ = BE THE N/2xN/2 MATRIX WHOSE COEFFICIENTS EQUAL THE "DIAGONAL DIFFERENCE" OF THE CORRESPONDING 2x2 BLOCK OF FOUR COEFFICIENTS IN $M_L(X)$
STORE $H_{L+1}(X)$, $V_{L+1}(X)$, $D_{L+1}(X)$
$L \leftarrow L+1$
$N \leftarrow N/2$
IS N $\leqslant$ 256?
NO
YES
STORE $M_L(X)$
ARE THERE MORE COLOR COMPONENT(S) LEFT?
YES
NO
END

F I G. 3

APPENDIX J

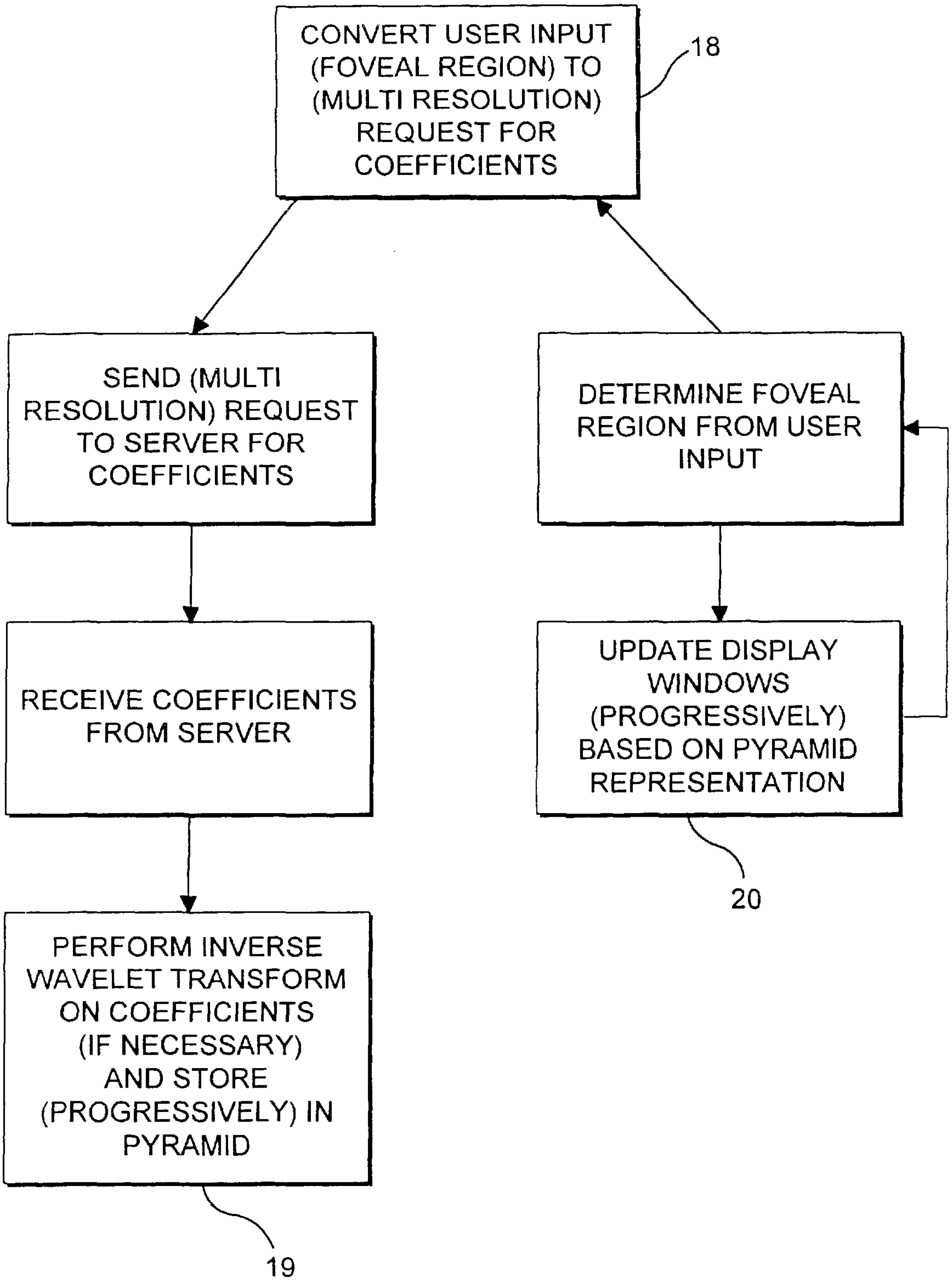
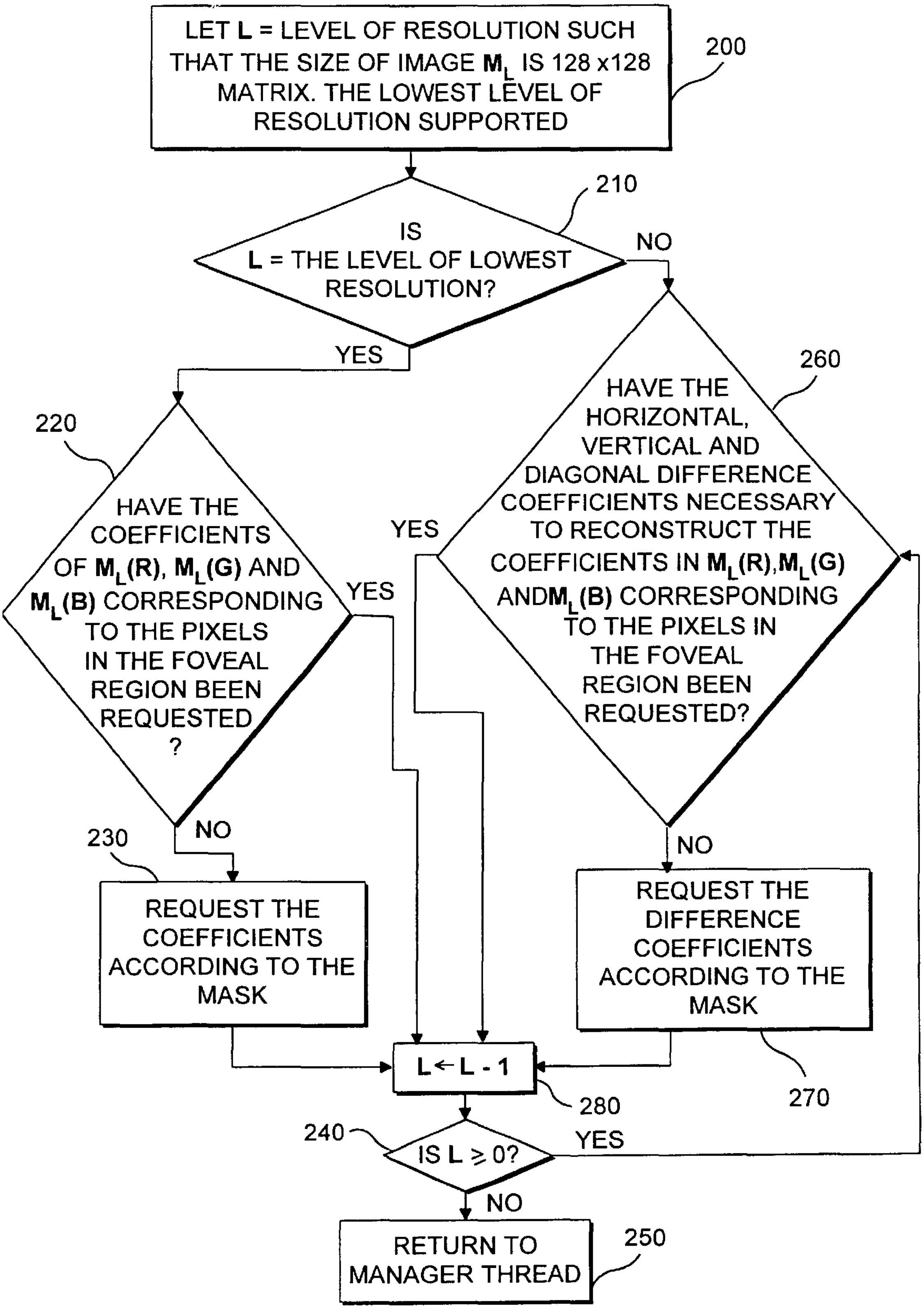

CONVERT USER INPUT (FOVEAL REGION) TO (MULTI RESOLUTION) REQUEST FOR COEFFICIENTS
18
SEND (MULTI RESOLUTION) REQUEST TO SERVER FOR COEFFICIENTS
DETERMINE FOVEAL REGION FROM USER INPUT
RECEIVE COEFFICIENTS FROM SERVER
UPDATE DISPLAY WINDOWS (PROGRESSIVELY) BASED ON PYRAMID REPRESENTATION
20
PERFORM INVERSE WAVELET TRANSFORM ON COEFFICIENTS (IF NECESSARY) AND STORE (PROGRESSIVELY) IN PYRAMID
19

FIG. 4

APPENDIX J


LET L = LEVEL OF RESOLUTION SUCH THAT THE SIZE OF IMAGE $M_L$ IS 128 x128 MATRIX. THE LOWEST LEVEL OF RESOLUTION SUPPORTED
200
IS L = THE LEVEL OF LOWEST RESOLUTION?
210
NO
YES
220
HAVE THE COEFFICIENTS OF $M_L(R)$, $M_L(G)$ AND $M_L(B)$ CORRESPONDING TO THE PIXELS IN THE FOVEAL REGION BEEN REQUESTED ?
YES
NO
260
HAVE THE HORIZONTAL, VERTICAL AND DIAGONAL DIFFERENCE COEFFICIENTS NECESSARY TO RECONSTRUCT THE COEFFICIENTS IN $M_L(R)$,$M_L(G)$ AND$M_L(B)$ CORRESPONDING TO THE PIXELS IN THE FOVEAL REGION BEEN REQUESTED?
YES
NO
230
REQUEST THE COEFFICIENTS ACCORDING TO THE MASK
REQUEST THE DIFFERENCE COEFFICIENTS ACCORDING TO THE MASK
270
L←L - 1
280
240
IS L ≥ 0?
YES
NO
RETURN TO MANAGER THREAD
250

F I G. 5

APPENDIX J

# APPARATUS AND METHOD FOR REALTIME VISUALIZATION USING USER-DEFINED DYNAMIC, MULTI-FOVEATED IMAGES

## FIELD OF THE INVENTION

The present invention relates to a method and apparatus for serving images, even very large images, over a "thinwire" (e.g., over the Internet or any other network or application having bandwidth limitations).

## BACKGROUND INFORMATION

The Internet, including the World Wide Web, has gained in popularity in recent years. The Internet enables clients/users to access information in ways never before possible over existing communications lines.

Often, a client/viewer desires to view and have access to relatively large images. For example, a client/viewer may wish to explore a map of a particular geographic location. The whole map, at highest (full) level of resolution will likely require a pixel representation beyond the size of the viewer screen in highest resolution mode.

One response to this restriction is for an Internet server to pre-compute many smaller images of the original image. The smaller images may be lower resolution (zoomed-out) views and/or portions of the original image. Most image archives use this approach. Clearly this is a sub-optimal approach since no preselected set of views can anticipate the needs of all users.

Some map servers (see, e.g., URLs http://www.mapquest.com and http://www.MapOnUs.com) use an improved approach in which the user may zoom and pan over a large image. However, transmission over the Internet involves significant bandwidth limitations (i.e transmission is relatively slow). Accordingly, such map servers suffer from at least three problems:

Since a brand new image is served up for each zoom or pan request, visual discontinuities in the zooming and panning result. Another reason for this is the discrete nature of the zoom/pan interface controls.

Significantly less than realtime response.

The necessarily small fixed size of the viewing window (typically about 3"×4.5"). This does not allow much of a perspective.

To generalize, what is needed is an apparatus and method which allows realtime visualization of large scale images over a "thinwire" model of computation. To put it another way, it is desirable to optimize the model which comprises an image server and a client viewer connected by a low bandwidth line.

One approach to the problem is by means of progressive transmission. Progressive transmission involves sending a relatively low resolution version of an image and then successively transmitting better resolution versions. Because the first, low resolution version of the image requires far less data than the full resolution version, it can be viewed quickly upon transmission. In this way, the viewer is allowed to see lower resolution versions of the image while waiting for the desired resolution version. This gives the transmission the appearance of continuity. In addition, in some instances, the lower resolution version may be sufficient or may in any event exhaust the display capabilities of the viewer display device (e.g., monitor).

Thus, R. L. White and J. W. Percival, "Compression and Progressive Transmission of Astronomical Images," *SPIE Technical Conference* 2199, 1994, describes a progressive transmission technique based on bit planes that is effective for astronomical data.

However, utilizing progressive transmission barely begins to solve the "thinwire" problem. A viewer zooming or panning over a large image (e.g., map) desires realtime response. This of course is not achieved if the viewer must wait for display of the desired resolution of a new quadrant or view of the map each time a zoom and pan is initiated. Progressive transmission does not achieve this realtime response when it is the higher resolution versions of the image which are desired or needed, as these are transmitted later.

The problem could be effectively solved, if, in addition to variable resolution over time (i.e, progressive transmission), resolution is also varied over the physical extent of the image.

Specifically, using foveation techniques, high resolution data is transmitted at the user's gaze point but with lower resolution as one moves away from that point. The very simple rationale underlying these foveation techniques is that the human field of vision (centered at the gaze point) is limited. Most of the pixels rendered at uniform resolution are wasted for visualization purposes. In fact, it has been shown that the spatial resolution of the human eye decreases exponentially away from the center gaze point. E. L. Schwartz, "The Development of Specific Visual Projections in the Monkey and the Goldfish: Outline of a Geometric Theory of Receptotopic Structure," *Journal of Theoretical Biology,* 69:655–685, 1977

The key then is to mimic the movements and spatial resolution of the eye. If the user's gaze point can be tracked in realtime and a truly multi-foveated image transmitted (i.e., a variable resolution image mimicking the spatial resolution of the user's eye from the gaze point), all data necessary or useful to the user would be sent, and nothing more. In this way, the "thinwire" model is optimized, whatever the associated transmission capabilities and bandwidth limitations.

In practice, in part because eye tracking is imperfect, using multi-foveated images is superior to atempting display of an image portion of uniform resolution at the gaze point.

There have in fact been attempts to achieve multifoveated images in a "thinwire" environment.

F. S. Hill Jr., Sheldon Walker Jr. and Fuwen Gao, "Interactive Image Query System Using Progressive Transmission," *Computer Graphics,* 17(3), 1983, describes progressive transmission and a form of foveation for a browser of images in an archive. The realtime requirement does not appear to be a concern.

T. H. Reeves and J. A. Robinson, "Adaptive Foveation of MPEG Video," *Proceedings of the* $4^{th}$ *ACM International Multimedia Conference,* 1996, gives a method to foveate MPEG-standard video in a thin-wire environment. MPEG-standard could provide a few levels of resolution but they consider only a 2-level foveation. The client/viewer can interactively specify the region of interest to the server/sender.

R. S. Wallace and P. W. Ong and B. B. Bederson and E. L. Schwartz, "Space-variant image processing". Intl. J. Of Computer Vision, 13:1 (1994) 71–90 discusses space-variant images in computer vision. "Space-Variant" may be regarded as synonymous with the term "multifoveated" used above. A biological motivation for such images is the complex logmap model of the transformation from the retina to the visual cortex (E. L. Schwartz, "A quantitative model of the functional architecture of human striate cortex with application to visual illusion and cortical texture analysis", Biological Cybernetics, 37(1980) 63–76).

Philip Kortum and Wilson S. Geisler, "Implementation of a Foveated Image Coding System For Image Bandwidth Reduction," *Human Vision and Electronic Imaging, SPIE Proceedings Vol.* 2657, 350–360, 1996, implement a real time system for foveation-based visualization. They also noted the possibility of using foveated images to reduce bandwidth of transmission.

M. H. Gross, O. G. Staadt and R. Gatti, "Efficient triangular surface approximations using wavelets and quadtree data structures", IEEE Trans, On Visualization and Computer Graphics, 2(2), 1996, uses wavelets to produce multifoveated images.

Unfortunately, each of the above attempts are essentially based upon fixed super-pixel geometries, which amount to partitioning the visual field into regions of varying (predetermined) sizes called super-pixels, and assigning the average value of the color in the region to the super-pixel. The smaller pixels (higher resolution) are of course intended to be at the gaze point, with progressively larger super-pixels (lower resolution) about the gaze point.

However, effective real-time visulization over a "thin wire" requires precision and flexibility. This cannot be achieved with a geometry of predetermined pixel size. What is needed is a flexible foveation technique which allows one to modify the position and shape of the basic foveal regions, the maximum resolution at the foveal region and the rate at which the resolution falls away. This will allow the "thin-wire" model to be optimized.

In addition, none of the above noted references addresses the issue of providing multifoveated images that can be dynamically (incrementally) updated as a function of user input. This property is crucial to the solution of the thinwire problem, since it is essential that information be "streamed" at a rate that optimally matches the bandwidth of the network with the human capacity to absorb the visual information.

## SUMMARY OF THE INVENTION

The present invention overcomes the disadvantages of the prior art by utilizing means for tracking or approximating the user's gaze point in realtime and, based on the approximation, transmitting dynamic multifoveated image (s) (i.e., a variable resolution image over its physical extent mimicking the spatial resolution of the user's eye about the approximated gaze point) updated in realtime.

"Dynamic" means that the image resolution is also varying over time. The user interface component of the present invention may provide a variety of means for the user to direct this multifoveation process in real time.

Thus, the invention addresses the model which comprises an image server and a client viewer connected by a low bandwidth line. In effect, the invention reduces the bandwidth from server to client, in exchange for a very modest increase of bandwidth from the client to the server

Another object of the invention is that it allows realtime visualization of large scale images over a "thinwire" model of computation.

An additional advantage is the new degree of user control provided for realtime, active, visualization of images (mainly by way of foveation techniques). The invention allows the user to determine and change in realtime, via input means (for example, without limitation, a mouse pointer or eye tracking technology), the variable resolution over the space of the served up image(s).

An additional advantage is that the invention demonstrates a new standard of performance that can be achieved by large-scale image servers on the World Wide Web at current bandwidth or even in the near future.

Note also, the invention has advantages over the traditional notion of progressive transmission, which has no interactivity. Instead, the progressive transmission of an image has been traditionally predetermined when the image file is prepared. The invention's use of dynamic (constantly changing in realtime based on the user's input) multifoveated images allows the user to determine how the data are progressively transmitted.

Other advantages of the invention include that it allows the creation of the first dynamic and a more general class of multifoveated images. The present invention can use wavelet technology. The flexibility of the foveation approach based on wavelets allows one to easily modify the following parameters of a multifoveated image: the position and shape of the basic foveal region(s), the maximum resolution at the foveal region(s), and the rate at which the resolution falls away. Wavelets can be replaced by any multi resolution pyramid schemes. But it seems that wavelet-based approaches are preferred as they are more flexible and have the best compression properties.

Another advantage is the present invention's use of dynamic data structures and associated algorithms. This helps optimize the "effective real time behavior" of the system. The dynamic data structures allow the use of "partial information" effectively. Here information is partial in the sense that the resolution at each pixel is only partially known. But as additional information is streamed in, the partial information can be augmented. Of course, this principle is a corollary to progressive transmission.

Another advantage is that the dynamic data structures may be well exploited by the special architecture of the client program. For example, the client program may be multi-threaded with one thread (the "manager thread") designed to manage resources (especially bandwidth resources). This manager is able to assess network congestion, and other relevant parameters, and translate any literal user request into the appropriate level of demand for the network. For example, when the user's gaze point is focused on a region of an image, this may be translated into requesting a certain amount, say, X bytes of data. But the manager can reduce this to a request over the network of (say) X/2 bytes of data if the traffic is congested, or if the user is panning very quickly.

Another advantage of the present invention is that the server need send only that information which has not yet been served. This has the advantage of reducing communication traffic.

Further objects and advantages of the invention will become apparent from a consideration of the drawings and ensuing description.

## BRIEF DESRIPTION OF DRAWINGS

FIG. **1** shows an embodiment of the present invention including a server, and client(s) as well as their respective components.

FIG. **2***a* illustrates one level of a particular wavelet transform, the Haar wavelet transform, which the server may execute in one embodiment of the present invention.

FIG. **2***b* illustrates one level of the Haar inverse wavelet transform.

FIG. **3** is a flowchart showing an algorithm the server may execute to perform a Haar wavelet transform in one embodiment of the present invention.

FIG. **4** shows Manager, Display and Network threads, which the client(s) may execute in one embodiment of the present invention.

FIG. **5** is a more detailed illustration of a portion of the Manager thread depicted in FIG. **4**.

## DETAILED DESCRIPTION OF THE INVENTION

FIG. **1** depicts an overview of the components in an exemplary embodiment of the present invention. A server **1** is comprised of a storage device **3**, a memory device **7** and a computer processing device **4**. The storage device **3** can be implemented as, for example, an internal hard disk, Tape Cartridge, or CD-ROM. The faster access and greater storage capacity the storage device **3** provides, the more preferable the embodiment of the present invention. The memory device **7** can be implemented as, for example, a collection of RAM chips.

The processing device **4** on the server **1** has network protocol processing element **12** and wavelet transform element **13** running off it. The processing device **4** can be implemented with a single microprocessor chip (such as an Intel Pentium chip), printed circuit board, several boards or other device. Again, the faster the speed of the processing device **4**, the more preferable the embodiment. The network protocol processing element **12** can be implemented as a separate "software" (i.e., a program, sub-process) whose instructions are executed by the processing device **4**. Typical examples of such protocols include TCP/IP (the Internet Protocol) or UDP (User Datagram Protocol). The wavelet transform element **13** can also be implemented as separate "software" (i.e., a program, sub-process) whose instructions are executed by the processing device **4**.

In a preferred embodiment of the present invention, the server **1** is a standard workstation or Pentium class system. Also, TCP/IP processing may be used to implement the network protocol processing element **12** because it reduces complexity of implementation. Although a TCP/IP implementation is simplest, it is possible to use the UDP protocol subject to some basic design changes. The relative advantage of using TCP/IP as against UDP is to be determined empirically. An additional advantage of using modern, standard network protocols is that the server **1** can be constructed without knowing anything about the construction of its client(s) **2**.

According to the common design of modern computer systems, the most common embodiments of the present invention will also include an operating system running off the processing means device **4** of the server **1**. Examples of operating systems include, without limitation, Windows 95, Unix and Windows NT. However, there is no reason a processing device **4** could not provide the functions of an "operating system" itself.

The server **1** is connected to a client(s) **2** in a network. Typical examples of such servers **1** include image archive servers and map servers on the World Wide Web.

The client(s) **2** is comprised of a storage device **3**, memory device **7**, display **5**, user input device **6** and processing device **4**. The storage device **3** can be implemented as, for example, an internal hard disks, Tape Cartridge, or CD-ROM. The faster access and greater storage capacity the storage device **3** provides, the more preferable the embodiment of the present invention. The memory device **7** can be implemented as, for example, a collection of RAM chips. The display **5** can be implemented as, for example, any monitor, whether analog or digital. The user input device **6** can be implemented as, for example, a keyboard, mouse, scanner or eye-tracking device.

The client **2** also includes a processing device **4** with network protocol processing element **12** and inverse wavelet transform element means **14** running off it. The processing device **4** can be implemented as, for example, a single microprocessor chip (such as an Intel Pentium chip), printed circuit board, several boards or other device. Again, the faster the run time of the processing device **4**, the more preferable the embodiment. The network protocol processing element **12** again can be implemented as a separate "software" (i.e., a program, sub-process) whose instructions are executed by the processing device **4**. Again, TCP/IP processing may be used to implement the network protocol processing element **12**. The inverse wavelet transform element **14** also may be implemented as separate "software." Also running off the processing device **4** is a user input conversion mechanism **16**, which also can be implemented as "software."

As with the server **1**, according to the common design of modern computer systems, the most common embodiments of the present invention will also include an operating system running off the processing device **4** of the client(s) **2**.

In addition, if the server **1** is connected to the client(s) **2** via a telephone system line or other systems/lines not carrying digital pulses, the server **1** and client(s) **2** both also include a communications converter device **15**. A communications converter device **15** can be implemented as, for example, a modem. The communications converter device **15** converts digital pulses into the frequency/signals carried by the line and also converts the frequency/signals back into digital pulses, allowing digital communication.

In the operation of the present invention, the extent of computational resources (e.g., storage capacity, speed) is a more important consideration for the server **1**, which is generally shared by more than one client **2**, than for the client(s) **2**.

In typical practice of the present invention, the storage device **3** of the server **1** holds an image file, even a very large image file. A number of client **2** users will want to view the image.

Prior to any communication in this regard between the server **1** and client(s) **2**, the wavelet transform element **13** on the server **1** obtains a wavelet transform on the image and stores it in the storage device **3**.

There has been extensive research in the area of wavelet theory. However, briefly, to illustrate, "wavelets" are defined by a group of basis functions which, together with coefficients dependant on an input function, can be used to approximate that function over varying scales, as well as represent the function exactly in the limit. Accordingly, wavelet coefficients can be categorized as "average" or "approximating coefficients" (which approximate the function) and "difference coefficients" (which can be used to reconstruct the original function exactly). The particular approximation used as well as the scale of approximation depend upon the wavelet bases chosen. Once a group of basis functions is chosen, the process of obtaining the relevant wavelet coefficients is called a wavelet transform.

In the preferred embodiment, the Haar wavelet basis functions are used. Accordingly, in the preferred embodiment, the wavelet transform element **13** on the server **1** performs a Haar wavelet transform on a file representation of the image stored in the storage device **3**, and then stores the transform on the storage device **3**. However, it is readily apparent to anyone skilled in the art that any of the wavelet family of transforms may be chosen to implement the present invention.

Note that once the wavelet transform is stored, the original image file need not be kept, as it can be reconstructed exactly from the transform.

FIG. **2** illustrates one step of the Haar wavelet transform. Start with an n by n matrix of coefficients **17** whose entries correspond to the numeric value of a color component (say, Red, Green or Blue) of a square screen image of n by n pixels. Divide the original matrix **17** into 2 by 2 blocks of four coefficients, and for each 2×2 block, label the coefficient in the first column, first row "a,"; second column, first row "b"; second row, first column "c"; and second row, second column "d."

Then one step of the Haar wavelet transform creates four n/2 by n/2 matrices. The first is an n/2 by n/2 approximation matrix **8** whose entries equal the "average" of the corresponding 2 by 2 block of four coefficients in the original matrix **17**. As is illustrated in FIG. **2**, the coefficient entries in the approximation matrix **8** are not necessarily equal to the average of the corresponding four coefficients a, b, c and d (i.e., a'=(a+b+c+d)/4) in the original matrix **17**. Instead, here, the "average" is defined as (a+b+c+d)/2.

The second is an n/2 by n/2 horizontal difference matrix **10** whose entries equal b'=(a+b−c−d)/2, where a, b, c and d are, respectively, the corresponding 2×2 block of four coefficients in the original matrix **17**. The third is an n/2 by n/2 vertical difference matrix **9** whose entries equal c'=(a−b+c−d)/2, where a, b, c and d are, respectively, the corresponding 2×2 block of four coefficients in the original matrix **17**. The fourth is an n/2 by n/2 diagonal difference matrix **11** whose entries equal d'=(a−b−c+d)/2, where a, b, c and d are, respectively, the corresponding 2×2 block of four coefficients in the original matrix **17**.

A few notes are worthy of consideration. First, the entries a', b', c', d' are the wavelet coefficients. The approximation matrix **8** is an approximation of the original matrix **17** (using the "average" of each 2×2 group of 4 pixels) and is one fourth the size of the original matrix **17**.

Second, each of the 2×2 blocks of four entries in the original matrix **17** has one corresponding entry in each of the four n/2 by n/2 matrices. Accordingly, it can readily be seen from FIG. **2** that each of the 2×2 blocks of four entries in the original matrix **17** can be reconstructed exactly, and the transformation is invertible. Therefore, the original matrix **17** representation of an image can be discarded during processing once the transform is obtained.

Third, the transform can be repeated, each time starting with the last approximation matrix **8** obtained, and then discarding that approximation matrix **8** (which can be reconstructed) once the next wavelet step is obtained. Each step of the transform results in approximation and difference matrices ½ the size of the approximation matrix **8** of the prior step.

Retracing each step to synthesize the original matrix **17** is called the inverse wavelet transform, one step of which is depicted in FIG. **2***b*.

Finally, it can readily be seen that the approximation matrix **8** at varying levels of the wavelet transform can be used as a representation of the relevant color component of the image at varying levels of resolution.

Conceptually then, the wavelet transform is a series of approximation and difference matrices at various levels (or resolutions). The number of coefficients stored in a wavelet transform is equal to the number of pixels in the original matrix **17** image representation. (However, the number of bits in all the coefficients may differ from the number of bits in the pixels. Applying data compression to coefficients turns out to be generally more effective on coefficients.) If we assume the image is very large, the transform matrices must be further decomposed into blocks when stored on the storage means **3**.

FIG. **3** is a flowchart showing one possible implementation of the wavelet transform element **13** which performs a wavelet transform on each color component of the original image. As can be seen from the flowchart, the transform is halted when the size of the approximation matrix is 256×256, as this may be considered the lowest useful level of resolution.

Once the wavelet transform element **13** stores a transform of the image(s) in the storage means **3** of the server **1**, the server **1** is ready to communicate with client(s) **2**.

In typical practice of the invention the client **2** user initiates a session with an image server **1** and indicates an image the user wishes to view via user input means **6**. The client **2** initiates a request for the 256 by 256 approximation matrix **8** for each color component of the image and sends the request to the server **1** via network protocol processing element **12**. The server **1** receives and processes the request via network protocol processing element **12**. The server **1** sends the 256 by 256 approximation matrices **8** for each color component of the image, which the client **2** receives in similar fashion. The processing device **4** of the client **2** stores the matrices in the storage device **3** and causes a display of the 256 by 256 version of the image on the display **5**. It should be appreciated that the this low level of resolution requires little data and can be displayed quickly. In a map server application, the 256 by 256, coarse resolution version of the image may be useful in a navigation window of the display **5**, as it can provide the user with a position indicator with respect to the overall image.

A more detailed understanding of the operation of the client **2** will become apparent from the discussion of the further, continuous operation of the client **2** below.

Continuous operation of the client(s) **2** is depicted in FIG. **4**. In the preferred embodiment, the client(s) **2** processing device may be constructed using three "threads," the Manager thread **18**, the Network Thread **19** and the Display Thread **20**. Thread programming technology is a common feature of modern computers and is supported by a variety of platforms. Briefly, "threads" are processes that may share a common data space. In this way, the processing means can perform more than one task at a time. Thus, once a session is initiated, the Manager Thread **18**, Network Thread **19** and Display Thread **20** run simultaneously, independently and continually until the session is terminated. However, while "thread technology" is preferred, it is unnecessary to implement the client(s) **2** of the present invention.

The Display Thread **20** can be based on any modern windowing system running off the processing device **4**. One function of the Display Thread **20** is to continuously monitor user input device **6**. In the preferred embodiment, the user input device **6** consists of a mouse or an eye-tracking device, though there are other possible implementations. In a typical embodiment, as the user moves the mouse position, the current position of the mouse pointer on the display **5** determines the foveal region. In other words, it is presumed the user gaze point follows the mouse pointer, since it is the user that is directing the mouse pointer. Accordingly, the display thread **20** continuously monitors the position of the mouse pointer.

In one possible implementation, the Display Thread **20** places user input requests (i.e., foveal regions determined from user input device **6**) as they are obtained in a request queue. Queue's are data structures with first-in-first-out characteristics that are generally known in the art.

The Manager Thread **18** can be thought of as the brain of the client **2**. The Manager Thread **18** converts the user input request in the request queue into requests in the manager request queue, to be processed by the Network Thread **19**. The user input conversion mechanism **16** converts the user determined request into a request for coefficients.

A possible implementation of user input conversion mechanism **16** is depicted in the flow chart in FIG. **5**. Essentially, the user input conversion mechanism **16** requests all the coefficient entries corresponding to the foveal region in the horizontal difference **10** matrices, vertical difference **9** matrices, diagonal difference matrices **11** and approximation matrix **8** of the wavelet transform of the image at each level of resolution. (Recall that only the last level approximation matrix **8** needs to be stored by the server **1**.) That is, wavelet coefficients are requested such that it is possible to reconstruct the coefficients in the original matrix **17** corresponding to the foveal region.

As the coefficients are included in the request, they are masked out. The use of a mask is commonly understood in the art. The mask is maintained to determine which coefficients have been requested so they are not requested again. Each mask can be represented by an array of linked lists (one linked list for each row of the image at each level of resolution).

As shown in FIG. **5**, the input conversion mechanism **16** determines the current level of resolution ("L") of an image ("$M_L$") such that the image $M_L$ is, e.g., 128×128 pixel matrix (for example, the lowest supported resolution), as shown in Step **200**. Then, the input conversion mechanism **16** determines if the current level L is the lowest resolution level (Step **210**). If so, it is determined if the three color coefficients (i.e., $M_L(R)$, $M_L(G)$, and $M_L(B)$) correspond to the foveal region that has been requested (Step **220**). If that is the case, then the input conversion mechanism **16** confirms that the current region L is indeed the lowest resolution region (Step **240**), and returns the control to the Manager Thread **18** (Step **250**). If, in Step **220**, it is determined that the three color coefficients have not been requested, these coefficients are requested using the mask described above, and the process continues to Step **240**, and the control is returned to the Manager Thread **18** (Step **250**).

If, in Step **210**, it is determined that the current level L is not the lowest resolution level, then the input conversion mechanism **16** determines whether the horizontal, vertical and diagonal difference coefficients (which are necessary to reconstruct the three color coefficients) have been requested (Step **260**). If so, then the input conversion mechanism **16** skips to Step **280** to decrease the current level L by 1. Otherwise a set of difference coefficients may be requested. This set depends on the mask and the foveal parameters (e.g., a shape of the foveal region, a maximum resolution, a rate of decay of the resolution, etc.). The user may select "formal" values for these foveal parameters, but the Manager Thread **18** may, at this point, select the "effective" values for these parameters to ensure a trade-off between (1) achieving a reasonable response time over the estimated current network bandwidth, and (2) achieving a maximum throughput in the transmission of data. The process then continues to Step **280**. Thereafter, the input conversion mechanism **16** determines whether the current level L is greater or equal to zero (Step **240**). If that is the case, the process loops back to step **260**. Otherwise, the control is returned to the Manager Thread **18** (Step **250**).

The Network Thread **19** includes the network protocol processing element **12**. The Network Thread obtains the (next) multi-resolution request for coefficients corresponding to the foveal region from request queue and processes and sends the request to the server **1** via network protocol processing element **12**.

Notice that the data requested is "local" because it represents visual information in the neighborhood of the indicated part of the image. The data is incremental because it represents only the additional information necessary to increase the resolution of the local visual information. (Information already available locally is masked out).

The server **1** receives and processes the request via network protocol processing element **12**, and sends the coefficients requested. When the coefficients are sent, they are masked out. The mask is maintained to determine which coefficients have been sent and for deciding which blocks of data can be released from main memory. Thus, an identical version of the mask is maintained on both the client **2** side and server **1** side.

The Network Thread **19** of the client **2** receives and processes the coefficients. The Network Thread **19** also includes inverse wavelet transform element **14**. The inverse wavelet transform element **14** performs an inverse wavelet transform on the received coefficients and stores the resulting portion of an approximation matrix **8** each time one is obtained (i.e., at each level of resolution) in the storage device **3** of the client **2**. The sub-image is stored at each (progressively higher, larger and less course) level of its resolution.

Note that as the client **2** knows nothing about the image until it is gradually filled in as coefficients are requested. Thus, sparse matrices (sparse, dynamic data structures) and associated algorithms can be used to store parts of the image received from the server **1**. Sparse matrices are known in the art and behave like normal matrices except that the memory space of the matrix are not allocated all at once. Instead the memory is allocated in blocks of sub-matrices. This is reasonable as the whole image may require a considerable amount of space.

Simultaneously, the Display thread **20** (which can be implemented using any modern operating system or windowing system) updates the display **5** based on the pyramid representation stored in the storage device **3**.

Of course, the Display thread **20** continues its monitoring of the user input device **6** and the whole of client **2** processing continues until the session is terminated.

A few points are worthy of mention. Notice that since lower, coarser resolution images will be stored on the client **2** first, they are displayed first Also, the use of foveated images ensures that the incremental data to update the view is small, and the requested data can arrive within the round trip time of a few messages using, for example, the TCP/IP protocol.

Also notice, that a wavelet coefficient at a relatively coarser level of resolution corresponding to the foveal region affects a proportionately larger part of the viewer's screen than a coefficient at a relatively finer level of resolution corresponding to the foveal region (in fact, the resolution on the display **5** exponentially away from the mouse pointer). Also notice the invention takes advantage of progressive transmission, which gives the image perceptual continuity. But unlike the traditional notion of progressive transmission, it is the client **2** user that is determining transmission ordering, which is not pre-computed because the server **1** doesn't know what the client(s) **2** next request will be. Thus, as noted in the objects and advantages section, the "thinwire" model is optimized.

Note that in the event the thread technology is utilized to implement the present invention, semaphores data structures are useful if the threads share the same data structures (e.g., the request queue). Semaphores are well known in the art and ensure that only one simultaneous process (or "thread") can access and modify a shared data structure at one time. Semaphores are supported by modern operating systems.

## CONCLUSION

It is apparent that various useful modifications can be made to the above description while remaining within the scope of the invention.

For example, without limitation, the user can be provided with two modes for display: to always fill the pixels to the highest resolution that is currently available locally or to fill them up to some user specified level. The client **2** display **5** may include a re-sizable viewing window with minimal penalty on the realtime performance of the system. This is not true of previous approaches. There also may be an auxiliary navigation window (which can be re-sized but is best kept fairly small because it displays the entire image at a low resolution). The main purpose of such a navigation window would be to let the viewer know the size and position of the viewing window in relation to the whole image.

It is readily seen that further modifications within the scope of the invention provide further advantages to the user. For example, without limitation, the invention may have the following capabilities: continuous realtime panning, continuous realtime zooming, foveating, varying the foveal resolution and modification of the shape and size of the foveal region. A variable resolution feature may also allow the server **1** to dynamically adjust the amount of transmitted data to match the effective bandwidth of the network.

While the above description contains many specificities, these should not be construed as limitations on the scope of the invention, but rather as an exemplification of one preferred embodiment thereof. Many other variations are possible. Accordingly, the scope of the invention should be determined not by the embodiment(s) illustrated, but by the appended claims and their legal equivalents.

What is claimed is:

**1**. A client apparatus for enabling a realtime visualization of at least one image, the client apparatus comprising:

a storage device storing first data corresponding to a multifoveated representation of an original image,

a user input device providing second data corresponding to at least one visualization command of at least one user; and

a processing arrangement generating third data corresponding to a multifoveated image using the first data, the second data and a foveation operator.

**2**. The client apparatus of claim **1**, further comprising a network protocol processing element which provides the third data using a TCP/IP protocol.

**3**. The client apparatus of claim **1**, wherein the processing element transmits the third data to the at least one client via the Internet.

**4**. The client apparatus of claim **1**, wherein the user input device includes a mouse device.

**5**. The client apparatus of claim **1**, wherein the user input device includes at least one of an eye-tracking device and a keyboard.

**6**. The client apparatus of claim **1**, wherein the foveation operator is specified using parameters that include at least one of:

a set of foveation points,

a shape of a foveated region,

a maximum resolution of the foveated region, and

a rate at which a maximum resolution of the foveal region decays.

**7**. The client apparatus of claim **1**,

wherein the processing arrangement receives the original image from a server, and

wherein the memory arrangement stores a data structure representing the multifoveated image, the data structure that is optimized for the client apparatus being independent of an image representation provided by a server.

**8**. The client apparatus of claim **1**, wherein the third data corresponding to the multifoveated image is generated for at least one of

a first arbitrary-shaped foveal region,

a second arbitrarily-fine foveal region, and

an arbitrary union of the first and second foveal regions.

* * * * *

APPENDIX K

US005179638A

# United States Patent [19]

**Dawson et al.**

[11] **Patent Number: 5,179,638**

[45] **Date of Patent: Jan. 12, 1993**

[54] **METHOD AND APPARATUS FOR GENERATING A TEXTURE MAPPED PERSPECTIVE VIEW**

[75] Inventors: **John F. Dawson; Thomas D. Snodgrass; James A. Cousens,** all of Albuquerque, N. Mex.

[73] Assignee: **Honeywell Inc.,** Minneapolis, Minn.

[21] Appl. No.: **514,598**

[22] Filed: **Apr. 26, 1990**

[51] **Int. Cl.**$^5$ .............................................. **G06F 15/62**

[52] **U.S. Cl.** .................................... **395/125;** 395/127; 395/130

[58] **Field of Search** ............... 395/125, 126, 127, 130; 364/443, 723; 340/729

[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 4,677,576 | 6/1987 | Berlin, Jr. et al. | 395/127 X |
| 4,876,651 | 10/1989 | Dawson et al. | 395/126 X |
| 4,884,220 | 11/1989 | Dawson et al. | 395/125 |
| 4,899,293 | 2/1990 | Dawson et al. | 395/125 X |
| 4,940,972 | 7/1990 | Mouchot et al. | 395/125 X |
| 4,985,854 | 1/1991 | Wittenburg | 395/126 X |
| 5,020,014 | 5/1991 | Miller et al. | 364/723 |

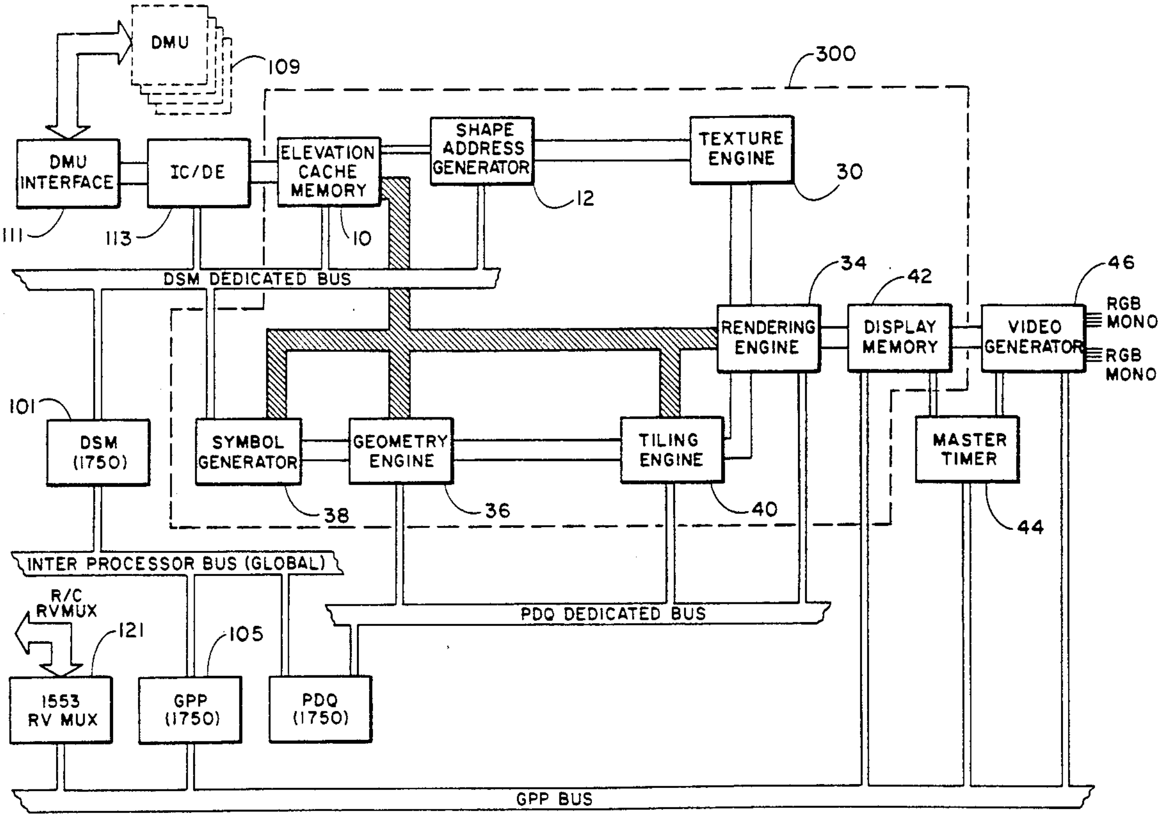
*Primary Examiner*—Gary V. Harkcom
*Assistant Examiner*—Mark K. Zimmerman

*Attorney, Agent, or Firm*—Ronald E. Champion; George A. Leone, Sr.

[57] **ABSTRACT**

A method and apparatus for providing a texture mapped perspective view for digital map systems. The system includes apparatus for storing elevation data, apparatus for storing texture data, apparatus for scanning a projected view volume from the elevation data storing apparatus, apparatus for processing, apparatus for generating a plurality of planar polygons and apparatus for rendering images. The processing apparatus further includes apparatus for receiving the scanned projected view volume from the scanning apparatus, transforming the scanned projected view volume from object space to screen space, and computing surface normals at each vertex of each polygon so as to modulate texture space pixel intensity. The generating apparatus generates the plurality of planar polygons from the transformed vertices and supplies them to the rendering apparatus which then shades each of the planar polygons. In one alternate embodiment of the invention, the polygons are shaded by apparatus of the rendering apparatus assigning one color across the surface of each polygon. In yet another alternate embodiment of the invention, the rendering apparatus interpolates the intensities between the vertices of each polygon in a linear fashion as in Gouraud shading.

**8 Claims, 7 Drawing Sheets**



APPENDIX K

Fig.-1

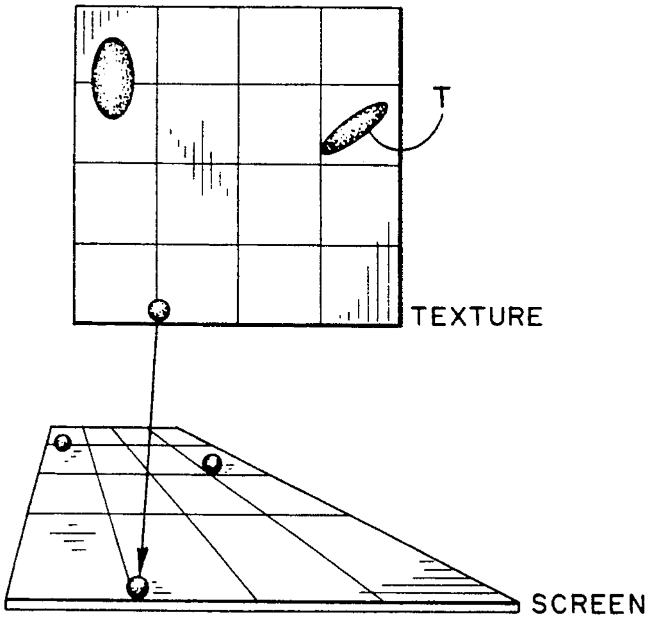


Fig.-2

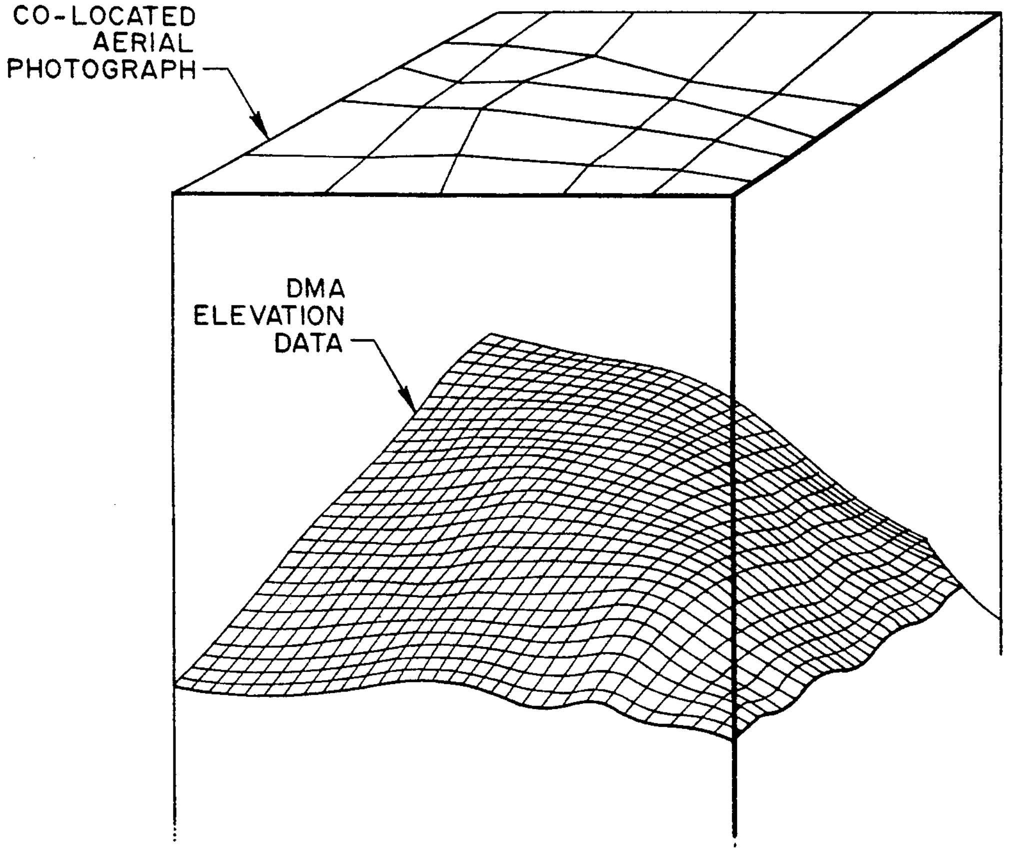


*Fig.-3*

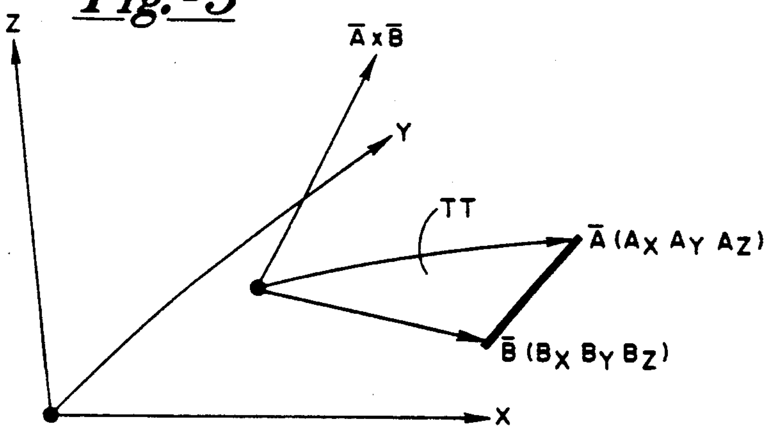
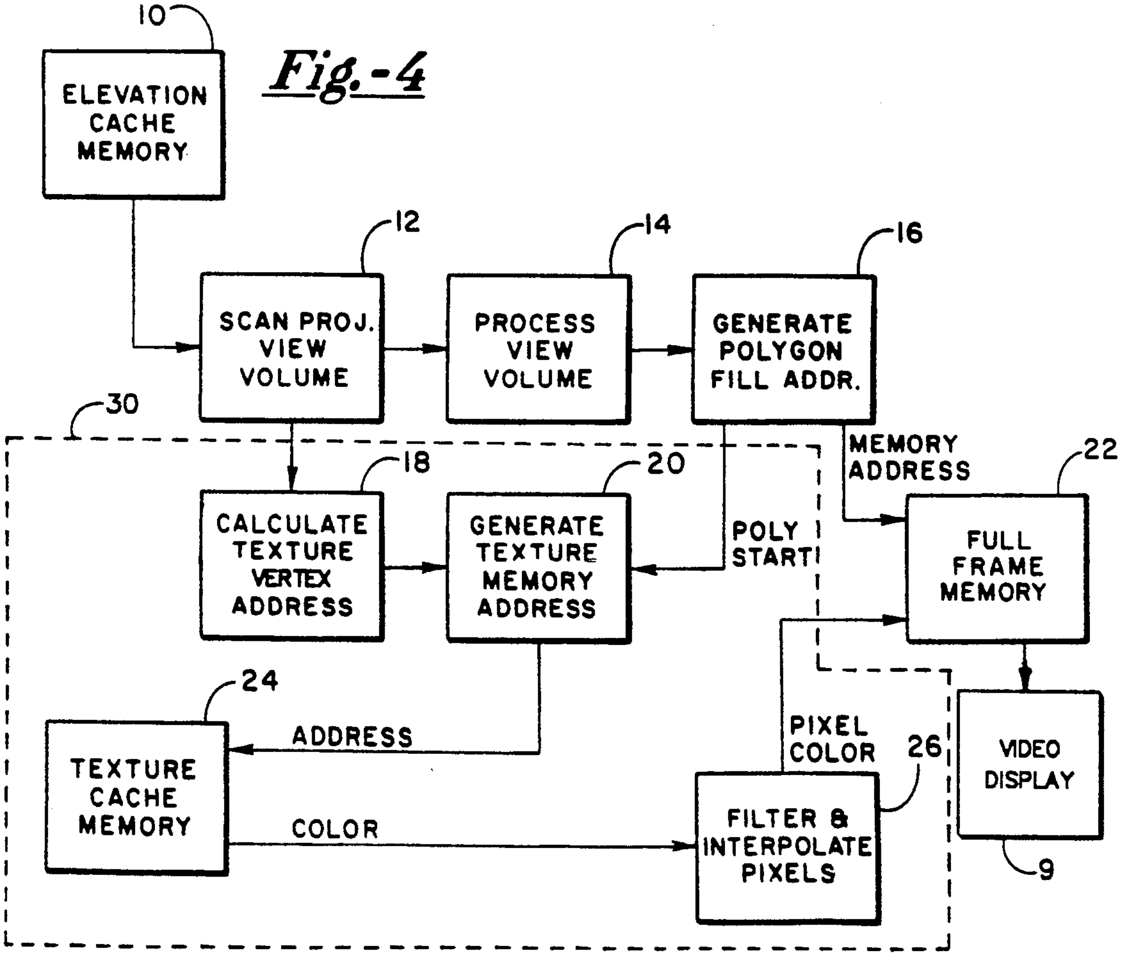

Z
$\bar{A} \times \bar{B}$
Y
TT
$\bar{A}$ ($A_X$ $A_Y$ $A_Z$)
$\bar{B}$ ($B_X$ $B_Y$ $B_Z$)
X

*Fig.-4*


10
ELEVATION CACHE MEMORY
12
SCAN PROJ. VIEW VOLUME
14
PROCESS VIEW VOLUME
16
GENERATE POLYGON FILL ADDR.
30
MEMORY ADDRESS
22
18
CALCULATE TEXTURE VERTEX ADDRESS
20
GENERATE TEXTURE MEMORY ADDRESS
POLY START
FULL FRAME MEMORY
24
TEXTURE CACHE MEMORY
ADDRESS
PIXEL COLOR
26
VIDEO DISPLAY
COLOR
FILTER & INTERPOLATE PIXELS
9

APPENDIX K

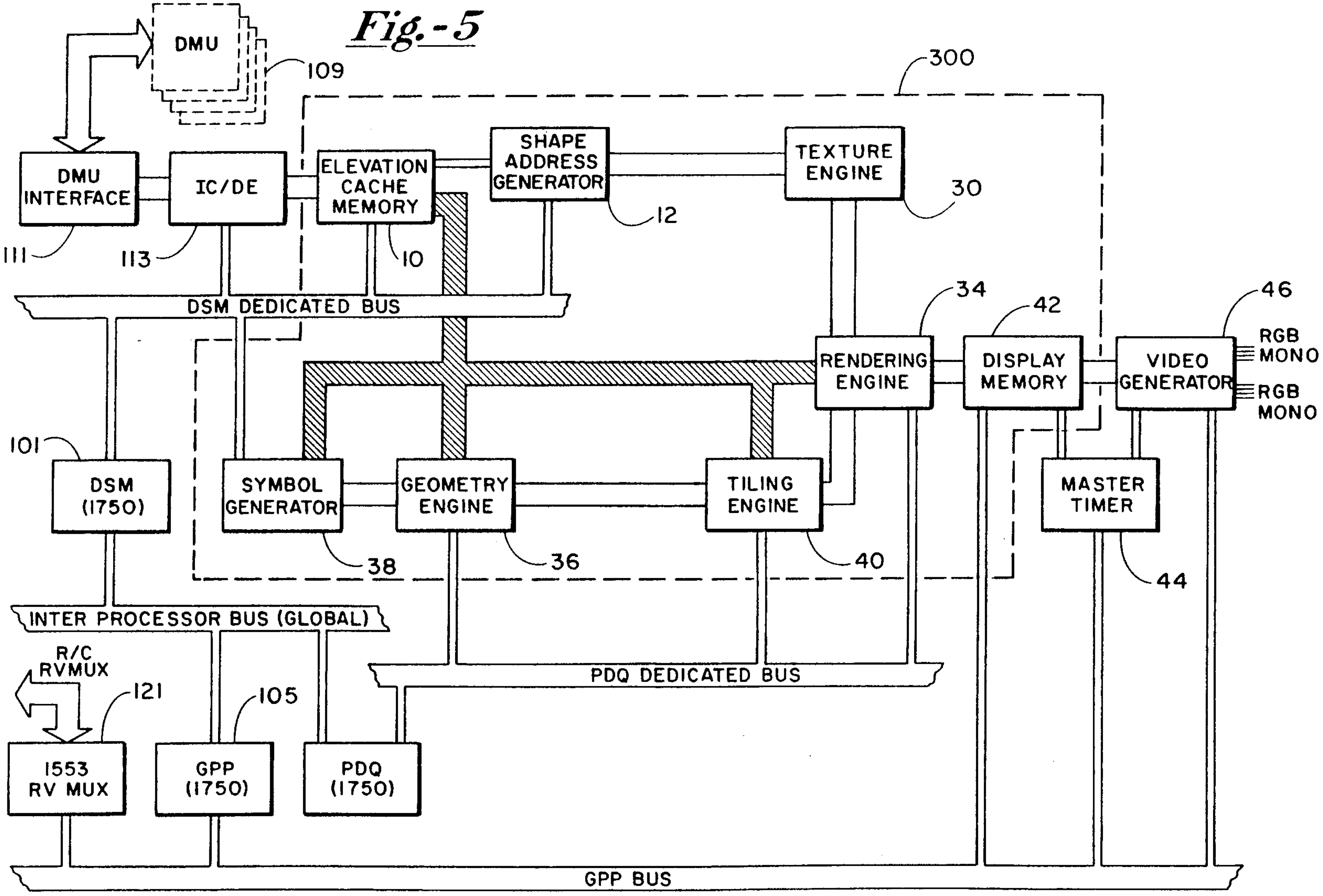


*Fig.-5*

*Fig.-6*

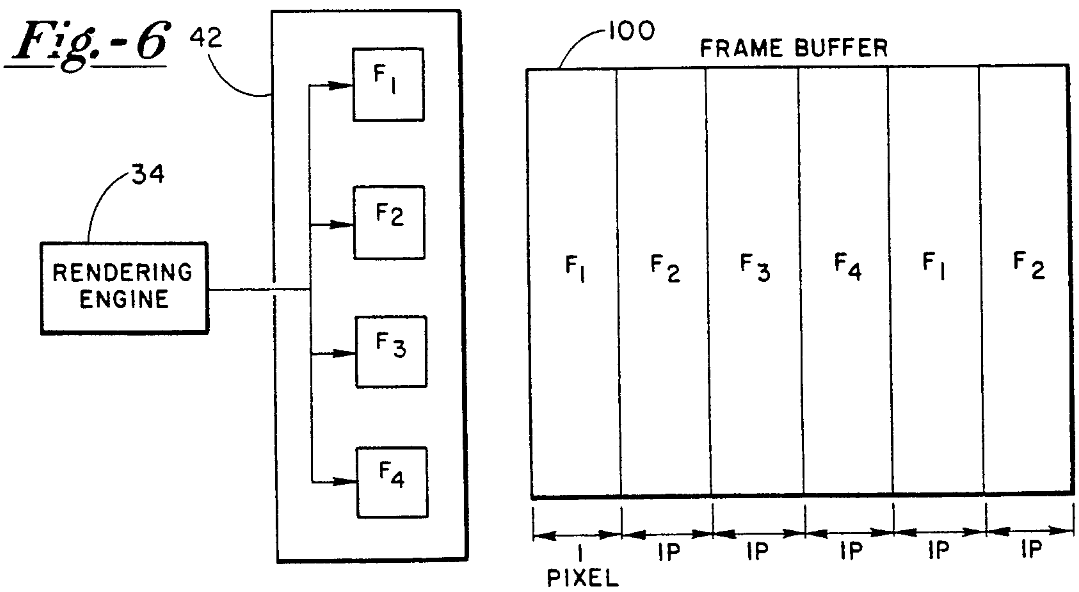


*Fig.-7A*

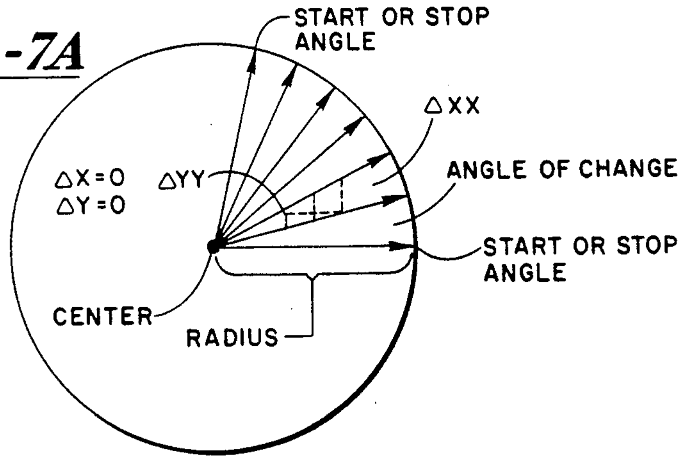


*Fig.-7B*

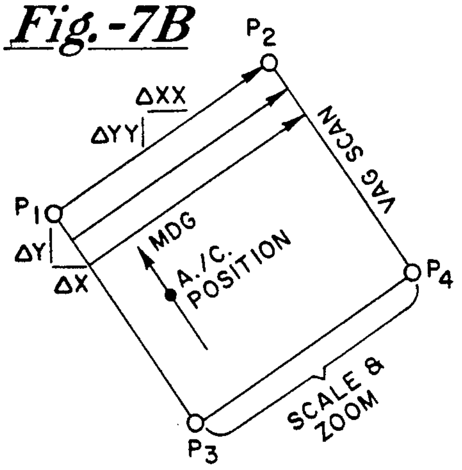


*Fig.-7C*

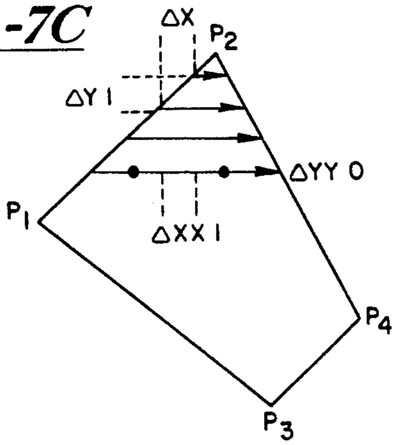
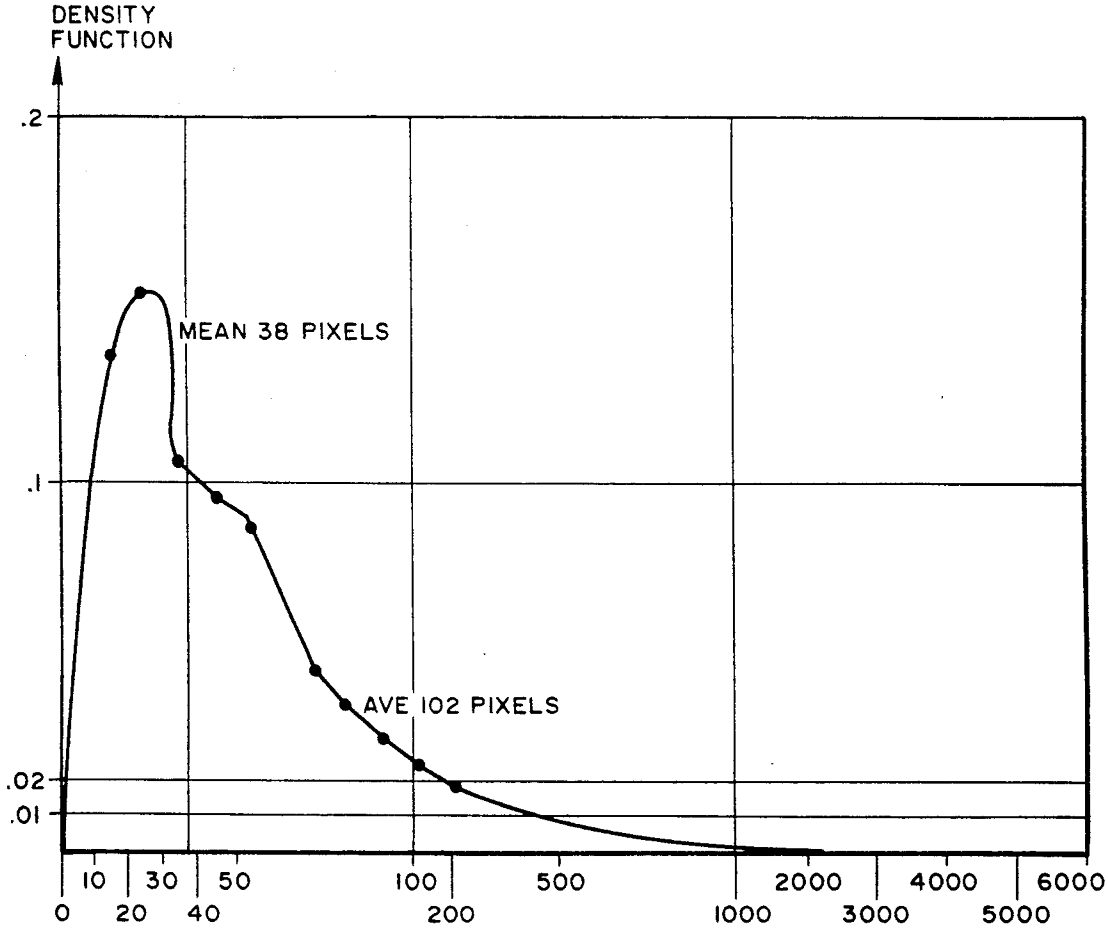


Fig.-8


DENSITY
FUNCTION
.2
.1
.02
.01
MEAN 38 PIXELS
AVE 102 PIXELS
0
10
20
30
40
50
100
200
500
1000
2000
3000
4000
5000
6000

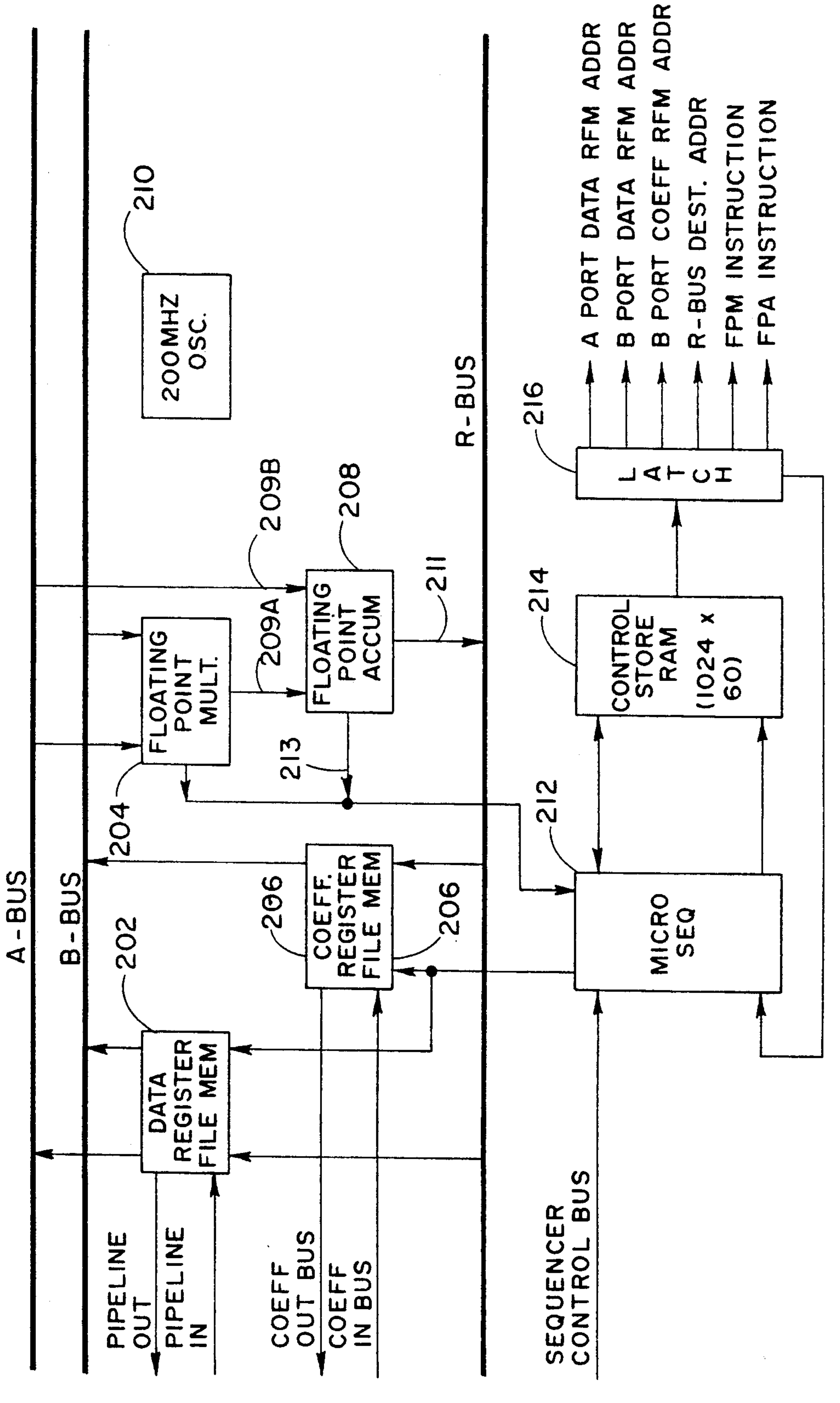

A-BUS
B-BUS
R-BUS
200MHZ OSC.
210
FLOATING POINT MULT.
204
FLOATING POINT ACCUM
208
209A
209B
211
213
DATA REGISTER FILE MEM
202
COEFF. REGISTER FILE MEM
206
206
MICRO SEQ
212
CONTROL STORE RAM (1024 x 60)
214
LATCH
216
A PORT DATA RFM ADDR
B PORT DATA RFM ADDR
B PORT COEFF RFM ADDR
R-BUS DEST. ADDR
FPM INSTRUCTION
FPA INSTRUCTION
PIPELINE OUT
PIPELINE IN
COEFF OUT BUS
COEFF IN BUS
SEQUENCER CONTROL BUS

Fig.-9

APPENDIX K

*Fig.-10A*

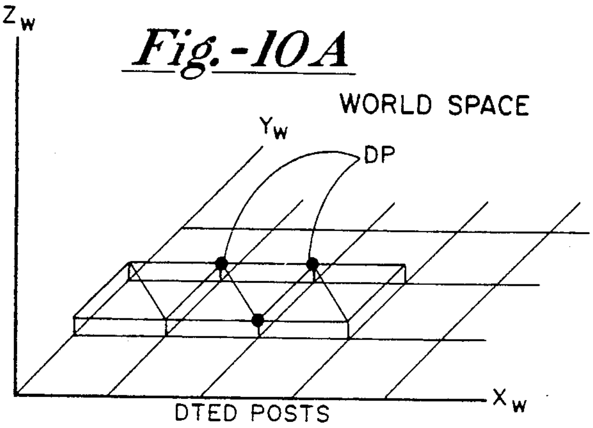


*Fig.-10B*

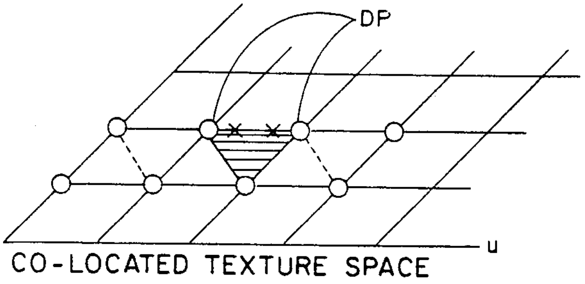


*Fig.-10C*

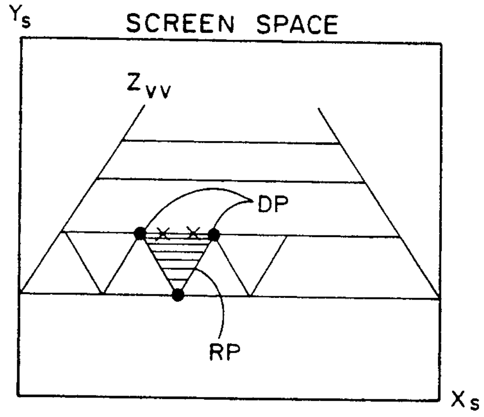
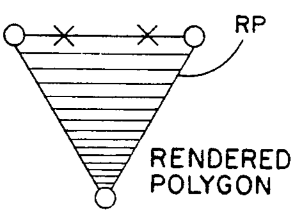


*Fig.-10D*



# METHOD AND APPARATUS FOR GENERATING A TEXTURE MAPPED PERSPECTIVE VIEW

The present invention is directed generally to graphic display systems and, more particularly, to a method and apparatus for generating texture mapped perspective views for a digital map system.

## RELATED APPLICATIONS

The following applications are included herein by reference:

(1) U.S. Pat. No. 4,876,651 filed May 11, 1988, issued Oct. 24, 1989 entitled "Digital Map System" which was assigned to the assignee of the present invention;

(2) Assignee copending application Ser. No. 09/514,685 filed Apr. 26, 1990, entitled "High Speed Processor for Digital Signal Processing";

(3) U.S. Pat. No. 4,884,220 entitled "Generator with Variable Scan Patterns" filed Jun. 7. 1988, issued Nov. 28, 1989, which is assigned to the assignee of the present invention;

(4) U.S. Pat. No. 4,899,293 entitled "A method of Storage and Retrieval of Digital Map Data Based Upon a Tessellated Geoid System", filed Dec. 14, 1988, issued Feb. 6, 1990;

(5) U.S. Pat. No. 5,020,014 entitled "Generic Interpolation Pipeline Processor", filed Feb. 7, 1989, issued May 28, 1991, which is assigned to the assignee of the present invention;

(6) Assignee's copending patent application Ser. No. 07/732,725 filed Jul. 18, 1991 entitled "Parallel Polygon/Pixel Rendering Engine Architecture for Computer Graphics" which is a continuation of patent application 07/419,722 filed Oct. 11, 1989 now abandoned;

(7) Assignee's copending patent application Ser. No. 07/514,724 filed Apr. 26, 1990 entitled "Polygon Tiling Engine";

(8) Assignee's copending patent application Ser. No. 07/514,723 filed Apr. 26, 1990 entitled "Polygon Sort Engine"; and

(9) Assignee's copending patent application Ser. No. 07/514,742 filed Apr. 26, 1990 entitled "Three Dimensional Computer Graphic Symbol Generator".

## BACKGROUND OF THE INVENTION

Texture mapping is a computer graphics technique which comprises a process of overlaying aerial reconnaissance photographs onto computer generated three dimensional terrain images. It enhances the visual reality of raster scan images substantially while incurring a relatively small increase in computational expense. A frequent criticism of known computer-generated synthesized imagery has been directed to the extreme smoothness of the image. Prior art methods of generating images provide no texture, bumps, outcroppings, or natural abnormalities in the display of digital terrain elevation data (DTED).

In general, texture mapping maps a multidimensional image to a multidimensional space. A texture may be thought of in the usual sense such as sandpaper, a plowed field, a roadbed, a lake, woodgrain and so forth or as the pattern of pixels (picture elements) on a sheet of paper or photographic film. The pixels may be arranged in a regular pattern such as a checkerboard or may exhibit high frequencies as in a detailed photograph of high resolution LandSat imagery. Texture may also be three dimensional in nature as in marble or

woodgrain surfaces. For the purposes of the invention, texture mapping is defined to be the mapping of a texture onto a surface in three dimensional object space. As is illustrated schematically in FIG. 1, a texture space object T is mapped to a display screen by means of a perspective transformation.

The implementation of the method of the invention comprises two processes. The first process is geometric warping and the second process is filtering. FIG. 2 illustrates graphically the geometric warping process of the invention for applying texture onto a surface. This process applies the texture onto an object to be mapped analogously to a rubber sheet being stretched over a surface. In a digital map system application, the texture typically comprises an aerial reconnaissance photograph and the object mapped is the surface of the digital terrain data base as shown in FIG. 2. After the geometric warping has been completed, the second process of filtering is performed. In the second process, the image is resampled on the screen grid.

The invention provides a texture mapped perspective view architecture which addresses the need for increased aircraft crew effectiveness, consequently reducing workload, in low altitude flight regimes characterized by the simultaneous requirement to avoid certain terrain and threats. The particular emphasis of the invention is to increase crew situational awareness. Crew situational awareness has been increased to some degree through the addition of a perspective view map display to a plan view capability which already exists in digital map systems. See, for example, assignee's copending application Ser. No. 07/192,798, for a DIGITAL MAP SYSTEM, filed May 11, 1988, issued Oct. 24, 1989 as U.S. Pat. No. 4,876,651 which is incorporated herein by reference in its entirety. The present invention improves the digital map system capability by providing a means for overlaying aerial reconnaissance photographs over the computer generated three dimensional terrain image resulting in a one-to-one correspondence from the digital map image to the real world. In this way the invention provides visually realistic cues which augment the informational display of such a computer generated terrain image. Using these cues an aircraft crew can rapidly make a correlation between the display and the real world.

The architectural challenge presented by texture mapping is that of distributing the processing load to achieve high data throughput using parallel pipelines and then recombining the parallel pixel flow into a single memory module known as a frame buffer. The resulting contention for access to the frame buffer reduces the effective throughput of the pipelines in addition to requiring increased hardware and board space to implement the additional pipelines. The method and apparatus of the invention addresses this challenge by effectively combining the low contention attributes of a single high speed pipeline with the increased processing throughput of parallel pipelines.

## SUMMARY OF THE INVENTION

A method and apparatus for providing a texture mapped perspective view for digital map systems is provided. The invention comprises means for storing elevation data, means for storing texture data, means for scanning a projected view volume from the elevation data storing means, means for processing the projected view volume, means for generating a plurality of planar polygons and means for rendering images. The process-

APPENDIX K

ing means further includes means for receiving the scanned projected view volume from the scanning means, transforming the scanned projected view volume from object space to screen space, and computing surface normals at each vertex of each polygon so as to modulate texture space pixel intensity. The generating means generates the plurality of planar polygons from the transformed vertices and supplies them to the rendering means which then shades each of the planar polygons.

A primary object of the invention is to provide a technology capable of accomplishing a fully integrated digital map display system in an aircraft cockpit.

In one alternate embodiment of the invention, the polygons are shaded by means of the rendering means assigning one color across the surface of each polygon.

In yet another alternate embodiment of the invention, the rendering means interpolates the intensities between the vertices of each polygon in a linear fashion as in Gouraud shading.

It is yet another object of the invention to provide a digital map system including capabilities for perspective view, transparency, texture mapping, hidden line removal, and secondary visual effects such as depth cues and artifact (i.e., anti-aliasing) control.

It is yet another object of the invention to provide the capability for displaying forward looking infrared (FLIR) data and radar return images overlaid onto a plan and perspective view digital map image by fusing images through combining or subtracting other sensor video signals with the digital map terrain display.

It is yet another object of the invention to provide a digital map system with an arbitrary warping capability of one data base onto another data base which is accommodated by the perspective view texture mapping capability of the invention.

Other objects, features and advantages of the invention will become apparent to those skilled in the art through the drawings, description of the preferred embodiment and claims herein. In the drawings, like numerals refer to like elements.

## BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** shows the mapping of a textured object to a display screen by a perspective transformation.

FIG. **2** illustrates graphically the geometric warping process of the invention for applying texture onto a surface.

FIG. **3** illustrates the surface normal calculation as employed by the invention.

FIG. **4** presents a functional block diagram of one embodiment of the invention.

FIG. **5** illustrates a top level block diagram of one embodiment of the texture mapped perspective view architecture of the invention.

FIG. **6** schematically illustrates the frame buffer configuration as employed by one embodiment of the invention.

FIGS. *7a, 7b* and *7c* illustrate three examples of display format shapes.

FIG. **8** graphs the density function for maximum pixel counts.

FIG. **9** is a block diagram of one embodiment of the geometry array processor as employed by the invention.

FIGS. **10**A, **10**B, **10**C and **10**D illustrated the tagged architectural texture mapping as provided by the invention.

## DESCRIPTION OF THE PREFERRED EMBODIMENT

Generally, perspective transformation from texture space having coordinates U, V to screen space having coordinates X, Y requires an intermediate transformation from texture space to object space having coordinates $X_0$, $Y_0$, $Z_0$. Perspective transformation is accomplished through the general perspective transform equation as follows:

$$[X\ Y\ Z\ H] = [X\ Y\ Z\ 1]\ X \begin{bmatrix} A & B & C & | & P \\ D & E & F & | & Q \\ G & H & I & | & R \\ L & M & N & | & S \end{bmatrix}$$

where a point (X,Y,Z) in 3-space is represented by a four dimensional position vector [X Y Z H] in homogeneous coordinates.

The 3×3 sub-matrix

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}$$

accomplishes scaling, shearing, and rotation.

The 1×3 row matrix [L M N] produces translation.

The 3×1 column matrix

$$\begin{bmatrix} P \\ Q \\ R \end{bmatrix}$$

produces perspective transformation.

The 1×1 scalar [S] produces overall scaling.

The Cartesian cross-product needed for surface normal requires a square root. As shown in FIG. **3**, the surface normal shown is a vector A×B perpendicular to the plane formed by edges of a polygon as represented by vectors A and B, where A×B is the Cartesian cross-product of the two vectors. Normalizing the vector allows calculation for sun angle shading in a perfectly diffusing Lambertian surface. This is accomplished by taking the vector dot product of the surface normal vector with the sun position vector. The resulting angle is inversely proportional to the intensity of the pixel of the surface regardless of the viewing angle. This intensity is used to modulate the texture hue and intensity value.

$$\frac{A \times B}{||A||\ ||B||} \text{ where } \begin{aligned} A &= Ax^2 + Ay^2 + Az^2 \\ B &= Bx^2 + By^2 + Bz^2 \end{aligned}$$

A terrain triangle TT is formed by connecting the endpoints of vectors A and B, from point $B_X$, $B_Y$, $B_Z$ to point $A_X$, $A_Y$, $A_Z$.

Having described some of the fundamental basis for the invention, a description of the method of the invention will now be set out in more detail below.

Referring now to FIG. **4**, a functional block diagram of one embodiment of the invention is shown. The invention functionally comprises a means for storing elevation data **10**, a means for storing texture data **24**, a means for scanning a projected view volume from the elevation data storing means **12**, means for processing view volume **14** including means for receiving the scanned projected view volume from the scanning means **12**, means for generating polygon fill addresses **16**, means for calculating texture vertices addresses **18**, means for generating texture memory addresses **20**, means for filtering and interpolating pixels **26**, a full-frame memory **22**, and video display **9**. The processing means **14** further includes means for transforming the scanned projected view volume from object space to screen space and means for computing surface normals at each vertex of each polygon so as to calculate pixel intensity.

The means for storing elevation data **10** may preferably be a cache memory having at least a 50 nsec access time to achieve 20 Hz bi-linear interpolation of a 512×512 pixel resolution screen. The cache memory further may advantageously include a 256×256 bit buffer segment with 2K bytes of shadow RAM used for the display list. The cache memory may arbitrarily be reconfigured from 8 bits deep (data frame) to 64 bits (i.e., comprising the sum of texture map data (24 bits)+DTED (16 bits)+aeronautical chart data (24 bits)). A buffer segment may start at any cache address and may be written horizontally or vertically. Means for storing texture data **24** may advantageously be a texture cache memory which is identical to the elevation cache memory except that it stores pixel information for warping onto the elevation data cache. Referring now to FIG. **5**, a top level block diagram of the texture mapped perspective view architecture is shown. The architecture implements the functions as shown in FIG. **4** and the discussion which follows shall refer to functional blocks in FIG. **4** and corresponding elements in FIG. **5**. In some cases, such as element **14**, there is a one-to-one correspondence between the functional blocks in FIG. **4** and the architectural elements of FIG. **5**. In other cases, as explained hereinbelow, the functions depicted in FIG. **4** are carried out by a plurality of elements shown in FIG. **5**. The elements shown in FIG. **5** comprising the texture mapped perspective view system **300** of the invention include elevation cache memory **10**, shape address generator (SHAG) **12**, texture engine **30**, rendering engine **34**, geometry engine **36**, symbol generator **38**, tiling engine **40**, and display memory **42**. These elements are typically part of a larger digital map system including a digital map unit (DMU) **109**, DMU interface **111**, IC/DE **113**, a display stream manager (DSM) **101**, a general purpose processor (GPP) **105**, RV MUX **121**, PDQ **123**, master time **44**, video generator **46** and a plurality of data bases. The latter elements are described in assignee's Digital Map System U.S. Pat. No. 4,876,651.

## GEOMETRY ENGINE

The geometry engine **36** is comprised of one or more geometry array processors (GAPs) which process the 4×4 Euler matrix transformation from object space (sometimes referred to as "world" space) to screen space. The GAPs generate X and Y values in screen coordinates and Zvv values in range depth. The GAPs also compute surface normals at each vertex of a polygon representing an image in object space via Cartesian cross-products for Gouraud shading, or they may assign one surface normal to the entire polygon for flat shading and wire mesh. Intensity calculations are performed using a vector dot product between the surface normal or normals and the illumination source to implement a Lambertian diffusely reflecting surface. Hue and intensity values are then assigned to the polygon. The method and apparatus of the invention also provides a dot rendering scheme wherein the GAPs only transform one vertex of each polygon and the tiling engine **40**, explained in more detail below, is inhibited. In this dot rendering format, hue and intensity are assigned based on the planar polygon containing the vertex and the rendering engine is inhibited. Dot polygons may appear in the same image as multiple vertex polygons or may comprise the entire image itself. The "dots" are passed through the polygon rendering engine **34**. A range to the vertices or polygon (Zvv) is used if a fog or "DaVinci" effect are invoked as explained below. The GAPs also transform three dimensional overlay symbols from world space to screen space.

Referring now to FIG. **9**, a block diagram of one example embodiment of a geometry array processor (GAP) is shown. The GAP comprises a data register file memory **202**, a floating point multiplier **204**, a coefficient register file memory **206**, a floating point accumulator **208**, a 200 MHz oscillator **210**, a microsequencer **212**, a control store RAM **214**, and latch **216**.

The register file memory may advantageously have a capacity of 512 by 32 bits. The floating point accumulator **208** includes two input ports **209**A and **209**B with independent enables, one output port **211**, and a condition code interface **212** responsive to error codes. The floating point accumulator operates on four instructions, namely, multiply, no-op, pass A, and pass B. The microsequencer **212** operates on seven instructions including loop on count, loop on condition, jump, continue, call, return and load counter. The microsequencer includes a debug interface having a read/-write (R/W) internal register, R/W control store memory, halt on address, and single step, and further includes a processor interface including a signal interrupt, status register and control register. The GAP is fully explained in the assignee's co-pending application No. 07/514,685 filed Apr. 26, 1990 entitled High Speed Processor for Digital Signal Processing which is incorporated herein by reference in its entirety.

In one alternative embodiment of the invention, it is possible to give the viewer of the display the visual effect of an environment enshrouded in fog. The fog option is implemented by interpolating the color of the triangle vertices toward the fog color. As the triangles get smaller with distance, the fog particles become denser. By using the known relationship between distance and fog density, the fog thickness can be "dialed" or adjusted as needed. The vertex assignment interpolates the vertex color toward the fog color as a function of range toward the horizon. The fog technique may be implemented in the hardware version of the GAP such as may be embodied in a GaAs semiconductor chip. If a linear color space (typically referred to as "RGB" to reflect the primary colors, red, green and blue) is assumed, the amount of fog is added as a function of range to the polygon vertices' color computation by well known techniques. Thus, as the hue is assigned by elevation banding or monochrome default value, the fog color is tacked on. The rendering engine **34**, explained in more detail below, then straight forwardly interpolates the interior points.

In another alternative embodiment of the invention, a DaVinci effect is implemented. The DaVinci effect causes the terrain to fade into the distance and blend with the horizon. It is implemented as a function of range of the polygon vertices by the GAP. The horizon color is added to the vertices similarly to the fog effect.

## SHAPE ADDRESS GENERATOR (SHAG)

The SHAG **12** receives the orthographically projected view volume outline onto cache from the DSM. It calculates the individual line lengths of the scans and the delta x and delta y components. It also scans the elevation posts out of the elevation cache memory and passes them to the GAPs for transformation. In one embodiment of the invention, the SHAG preferably includes two arithmetic logic units (ALUs) to support the **50** nsec cache **10**. In the SHAG, data is generated for the GAPs and control signals are passed to the tiling engine **40**. DFAD data is downloaded into overlay RAM (not shown) and three dimensional symbols are passed to the GAPs from symbol generator **38**. Elevation color banding hue assignment is performed in this function. The SHAG generates shapes for plan view, perspective view, intervisibility, and radar simulation. These are illustrated in FIG. **7**. The SHAG is more fully explained in assignee's copending application, Ser. No. 203,660, Generator With Variable Scan Patterns, filed Jun. 7, 1988 issued as U.S. Pat. No. 4,884,220 on Nov. 28, 1989 which is incorporated herein by reference in its entirety.

A simple Lambertian lighting diffusion model has proved adequate for generating depth cueing in one embodiment of the invention. The sun angle position is completely programmable in azimuth and zenith. It may also be self-positioning based on time of day, time of year, latitude and longitude. A programmable intensity with gray scale instead of color implements the moon angle position algorithm. The display stream manager (DSM) programs the sun angle registers. The illumination intensities of the moon angle position may be varied with the lunar waxing and waning cycles.

## TILING ENGINE AND TEXTURE ENGINE

Still referring to FIGS. **4** and **5**, the means for calculating texture vertex address **18** may include the tiling engine **40**. Elevation posts are vertices of planar triangles modeling the surface of the terrain. These posts are "tagged" with the corresponding U,V coordinate address calculated in texture space. This tagging eliminates the need for interpolation by substituting an address lookup. Referring to FIGS. **10A**, **10B**, **10C** and **10D**, with continuing reference to FIGS. **4** and **5**, the tagged architectural texture mapping as employed by the invention is illustrated. FIG. **10A** shows an example of DTED data posts, DP, in world space. FIG. **10B** shows the co-located texture space for the data posts. FIG. **10C** shows the data posts and rendered polygon in screen space. FIG. **10D** illustrates conceptually the interpolation of tagged addresses into a rendered polygon RP. The texture engine **30** performs the tagged data structure management and filtering processes. When the triangles are passed to the rendering engine by the tiling engine for filling with texture, the tagged texture address from the elevation post is used to generate the texture memory address. The texture value is filtered by filtering and interpolation means **26** before being written to full-frame memory **22** prior to display.

The tiling engine generates the planar polygons from the transformed vertices in screen coordinates and passes them to the rendering engine. For terrain polygons, a connectivity offset from one line scan to the next is used to configure the polygons. For overlay symbols, a connectivity list is resident in a buffer memory (not shown) and is utilized for polygon generation. The tiling engine also informs the GAP if it is busy. In one embodiment 512 vertices are resident in a 1K buffer.

All polygons having surface normals more than 90 degrees from LOS are eliminated from rendering. This is known in the art as backface removal. Such polygons do not have to be transformed since they will not be visible on the display screen. Additional connectivity information must be generated if the polygons are nonplanar as the transformation process generates implied edges. This requires that the connectivity information be dynamically generated. Thus, only planar polygons with less than 513 vertices are implemented. Non-planar polygons and dynamic connectivity algorithms are not implemented by the tiling engine. The tiling engine is further detailed in assignee's copending applications of even filing date herewith entitled Polygon Tiling Engine, as referenced hereinabove and Polygon Sort Engine, as referenced hereinabove, both of which are incorporated herein by reference.

## RENDERING ENGINE

Referring again to FIG. **5**, the rendering engine **34** of the invention provides a means of drawing polygons in a plurality of modes. The rendering engine features may include interpolation algorithms for processing coordinates and color, hidden surface removal, contour lines, aircraft relative color bands, flat shading, Gourand shading, phong shading, mesh format or screen door effects, ridgeline display, transverse slice, backface removal and RECE (aerial reconnaissance) photo modes. With most known methods of image synthesis, the image is generated by breaking the surfaces of the object into polygons, calculating the color and intensity at each vertex of the polygon, and drawing the results into a frame buffer while interpolating the colors across the polygon. The color information at the vertices is calculated from light source data, surface normal, elevation and/or cultural features.

The interpolation of coordinate and color (or intensity) across each polygon must be performed quickly and accurately. This is accomplished by interpolating the coordinate and color at each quantized point or pixel on the edges of the polygon and subsequently interpolating from edge to edge to generate the fill lines. For hidden surface removal, such as is provided by a Z-buffer in a well-known manner, the depth or Z-value for each pixel is also calculated. Furthermore, since color components can vary independently across a surface or set of surfaces, red, green and blue intensities are interpolated independently. Thus, a minimum of six different parameters (X,Y,Z,R,G,B) are independently calculated when rendering polygons with Gouraud shading and interpolated Z-values.

Additional features of the rendering engine include a means of providing contour lines and aircraft relative color bands. For these features the elevation also is interpolated at each pixel. Transparency features dictate that an alpha channel be maintained and similarly interpolated. These requirements imply two additional axes of interpolation bringing the total to eight. The rendering engine is capable of processing polygons of one vertex in its dot mode, two vertices in its line mode, and three to 512 coplanar vertices in its polygon mode.

In the flat shading mode the rendering engine assigns the polygon a single color across its entire surface. An arbitrary vertex is selected to assign both hue and intensity for the entire polygon. This is accomplished by assigning identical RGB values to all vertices. Interpolation is performed normally but results in a constant value. This approach will not speed up the rendering process but will perform the algorithm with no hardware impact.

The Gouraud shading algorithm included in the rendering engine interpolates the intensities between the vertices of each polygon rendered in a linear fashion. This is the default mode. The Phong shading algorithm interpolates the surface normals between the vertices of the polygon between applying the intensity calculations. The rendering engine would thus have to perform an illumination calculation at each pixel after interpolation. This approach would significantly impact the hardware design. This algorithm may be simulated, however, using a weighing function (typically a function of cosine (Θ)) around a narrow band of the intensities. This results in a non-linear interpolation scheme and provides for a simulated specular reflectance. In an alternative embodiment, the GAP may be used to assign the vertices of the polygon this non-linear weighing via the look-up table and the rendering engine would interpolate as in Gouraud shading.

Transparency is implemented in the classical sense using an alpha channel or may be simulated with a screen door effect. The screen door effect simply renders the transparent polygon as normal but then only outputs every other or every third pixel. The mesh format appears as a wire frame overlay with the option of rendering either hidden lines removed or not. In the case of a threat dome symbol, all polygon edges must be displayed as well as the background terrain. In such a case, the fill algorithm of the rendering engine is inhibited and only the polygon edges are rendered. The intensity interpolation is performed on the edges which may have to be two pixels wide to eliminate strobing. In one embodiment, an option for terrain mesh includes the capability for tagging edges for rendering so that the mesh appears as a regular orthogonal grid.

Typical of the heads up display (HUD) format used in aircraft is the ridgeline display and the transverse slice. In the ridgeline format, a line drawing is produced from polygon edges whose slopes change sign relative to the viewpoint. All polygons are transformed, tiled, and then the surface normals are computed and compared to the viewpoint. The tiling engine strips away the vertices of non-ridge contributing edges and passes only the ridge polygons to the rendering engine. In transverse slice mode, fixed range bins relative to the aircraft are defined. A plane orthogonal to the view LOS is then passed through for rendering. The ridges then appear to roll over the terrain as the aircraft flies along. These algorithms are similar to backface removal. They rely upon the polygon surface normal being passed to the tiling engine.

One current implementation of the invention guarantees non-intersecting polygon sides by restricting the polygons rendered to be planar. They may have up to 512 vertices. Polygons may also consist of one or two vertices. The polygon "end" bit is set at the last vertex and processed by the rendering engine. The polygon is tagged with a two bit rendering code to select mesh, transparent, or Gouraud shading. The rendering engine

also accomplishes a fine clip to the screen for the polygon and implements a smoothing function for lines.

An optional aerial reconnaissance (RECE) photo mode causes the GAP to texture map an aerial reconnaissance photograph onto the DTED data base. In this mode the hue interpolation of the rendering engine is inhibited as each pixel of the warping is assigned a color from the RECE photo. The intensity component of the color is dithered in a well known manner as a function of the surface normal as well as the Z-depth. These pixels are then processed by the rendering engine for Z-buffer rectification so that other overlays such as threats may be accommodated. The RECE photos used in this mode have been previously warped onto a tessellated geoid data base and thus correspond pixel-for-pixel to the DTED data. See assignee's aforereferenced copending application for A Method of Storage and Retrieval of Digital Map Data Based Upon A Tessellated Geoid System, which is hereby incorporated by reference in its entirety. The photos may be denser than the terrain data. This implies a deeper cache memory to hold the RECE photos. Aeronautical chart warping mode is identical to RECE photos except that aeronautical charts are used in the second cache. DTED warping mode utilizes DTED data to elevation color band aeronautical charts.

The polygon rendering engine may preferably be implemented in a generic interpolation pipeline processor (GIPP) of the type as disclosed in assignee's aforereferenced patent entitled Generic Interpolation Pipeline Processor, which is incorporated herein by reference in its entirety. In one embodiment of the invention, the GIPPs fill in the transformed polygons using a bilinear interpolation scheme with six axes (X,Y,Z,R,G,B). The primitive will interpolate a 16 bit pair and 8 bit pair of values simultaneously, thus requiring 3 chips for a polygon edge. One embodiment of the system of the invention has been sized to process one million pixels each frame time. This is sufficient to produce a 1K×1K high resolution chart, or a 512×512 DTED frame with an average of four overwrites per pixel during hidden surface removal with GIPPs outputting data at a 60 nsec rate, each FIFO, F1–F4, as shown in FIG. **6**, will receive data on the average of every 240 nsec. An even distribution can be assumed by decoding on the lower 2X address bits. Thus, the memory is divided into one pixel wide columns FIG. **6** is discussed in more detail below.

Referring again to FIGS. **4** and **5**, the "dots" are passed through the GIPPs without further processing. Thus, the end of each polygon's bit is set. A ZB buffer is needed to change the color of a dot at a given pixel for hidden dot removal. Perspective depth cuing is obtained as the dots get closer together as the range from the viewpoint increases.

Bi-linear interpolation mode operates in plan view on either DLMS or aeronautical charts. It achieves 20 Hz interpolation on a 512×512 display. The GIPPs perform the interpolation function.

## DATA BASES

A Level I DTED data base is included in one embodiment of the invention and is advantageously sampled on three arc second intervals. Buffer segments are preferably stored at the highest scales (104.24 nm) and the densest data (13.03 nm). With such a scheme, all other scales can be created. A Level II DTED data base is also included and is sampled at one arc second intervals. Buffer segments are preferably stored only at the densest data (5.21 nm).

A DFAD cultural feature data base is stored in a display list of 2K words for each buffer segment. The data structure consists of an icon font call, a location in cache, and transformation coefficients from model space to world space consisting of scaling, rotation, and position (translation). A second data structure comprises a list of polygon vertices in world coordinates and a color or texture. The DFAD data may also be rasterized and overlaid on a terrain similar to aerial reconnaissance photos.

Aeronautical charts at the various scales are warped into the tessellated geoid. This data is 24 bits deep. Pixel data such as LandSat, FLIR, data frames and other scanned in source data may range from one bit up to 24 bits in powers of two (1,2,4,8,16,24).

## FRAME BUFFER CONFIGURATION

Referring again to FIG. **6**, the frame buffer configuration of one embodiment of the invention is shown schematically. The frame buffer configuration is implemented by one embodiment of the invention comprises a polygon rendering chip **34** which supplies data to full-frame memory **42**. The full-frame memory **42** advantageously includes first-in, first-out buffers (FIFO) $F_1$, $F_2$, $F_3$ and $F_4$. As indicated above with respect to the discussion of the rendering engine, the memory is divided up into one pixel wide columns as shown in FIG. **6**. By doing so, however, chip select must changed on every pixel when the master timer **44** shown in FIG. **5** reads the memory. However, by orienting the SHAG scan lines at 90 degrees to the master timer scan lines, the chip select will change on every line. The SHAG starts scanning at the bottom left corner of the display and proceeds to the upper left corner of the display.

With the image broken up in this way, the probability that the GIPP will write to the same FIFO two times in a row, three times, four, and so on can be calculated to determine how deep the FIFO must be. Decoding on the lower order address bits means that the only time the rendering engine will write to the same FIFO twice in a row is when a new scan line is started. At four deep as shown in the frame buffer graph **100**, the chances of the FIFO filling up are approximately one in 6.4K. With an image of 1 million pixels, this will occur an acceptably small number of times for most applications. The perspective view transformations for 10,000 polygons with the power and board area constraints that are imposed by an avionics environment is significant. The data throughput for a given scene complexity can be achieved by adding more pipeline in parallel to the architecture. It is desirable to have as few pipelines as possible, preferably one, so that the image reconstruction at the end of the pipeline does not suffer from an arbitration bottleneck for a Z-buffered display memory.

In one embodiment of the invention, the processing throughput required has been achieved through the use of GaAs VSLI technology for parallel pipelines and a parallel frame buffer design has eliminated contention bottlenecks. A modular architecture allows for additional functions to be added to further the integration of the digital map into the avionics suite. The system architecture of the invention has high flexibility while maintaining speed and data throughput. The polygonal data base structure approach accommodate arbitrary scene complexity and a diversity of data base types.

The data structure of the invention is tagged so that any polygon may be rendered via any of the implemented schemes in a single frame. Thus, a particular image may have Gouraud shaded terrain, transparent threat domes, flat shaded cultural features, lines, and dots. In addition, since each polygon is tagged, a single icon can be comprised of differently shaded polygons. The invention embodies a 24 bit color system, although a production map would be scaled to 12 bits. A 12 bit system provides 4K colors and would require a 32K by 8 RGB RAM look-up table (LUT).

## MISCELLANEOUS FEATURES

The display formats in one example of the invention are switchable at less than 600 milliseconds between paper chart, DLMS plan and perspective view. A large cache (1 megabit D-RAMs) is required for texture mapping. Other format displays warp chart data over DTED, or use DTED to pseudo-color the map. For example, change the color palate LUT for transparency. The GAP is used for creating a true orthographic projection of the chart data.

An edit mode for three dimensions is supported by the apparatus of the invention. A three dimensional object such as a "pathway in the sky" may be tagged for editing. This is accomplished by first, moving in two dimensions at a given AGL, secondly, updating the AGL in the three dimensional view, and finally, updating the data base.

The overlay memory from the DMC may be video mixed with the perspective view display memory.

Freeze frame capability is supported by the invention. In this mode, the aircraft position is updated using the cursor. If the aircraft flies off the screen, the display will snap back in at the appropriate place. This capability is implemented in plan view only. There is data frame software included to enable roaming through cache memory. This feature requires a two axis roam joystick or similar control. Resolution of the Z-buffer is 16 bits. This allows 64K meters down range.

The computer generated imagery has an update rate of 20 Hz. The major cycle is programmable and variable with no frame extend invoked. The system will run as fast as it can but will not switch ping-pong display memories until each functional unit issues a "pipeline empty" message to the display memory. The major cycle may also be locked to a fixed frame in multiples of 16.6 milliseconds. In the variable frame mode, the processor clock is used for a smooth frame interpolation for roam or zoom. The frame extend of the DMC is eliminated in perspective view mode. Plan view is implemented in the same pipeline as the perspective view. The GPP **105** loads the countdown register on the master timer to control the update rate.

The slowest update rate is 8.57 Hz. The image must be generated in this time or the memories will switch. This implies a pipeline speed of 40 million pixels per second. In a 512×512 image, it is estimated that there would be 4 million pixels rendered worst case with heavy hidden surface removal. In most cases, only million pixels need be rendered. FIG. **8** illustrates the analysis of pixel over-writes. The minimum requirement for surface normal resolution so that the best image is achieved is 16 bits. Tied to this is the way in which the normal is calculated. Averaging from surrounding tiles gives a smoother image on scale change or zoom. Using one tile is less complex, but results in poorer image quality. Surface normal is calculated on the fly in accordance with known techniques.

## DISPLAY MEMORY

This memory is a combination of scene and overlay with a Z-buffer. It is distributed or partitioned for optimal loading during write, and configured as a frame buffer during read-out. The master time speed required is approximately 50 MHz. The display memory resolution can be configured as 512×512×12 or as 1024×1024×12. The Z-buffer is 16 bits deep and 1K×1K resolution. At the start of each major cycle, the Z-values are set to plus infinity (FF Hex). Infinity (Zmax) is programmable. The back clipping plane is set by the DSM over the control bus.

At the start of each major cycle, the display memory is set to a background color. In certain modes such as mesh or dot, this color will change. A background color register is loaded by the DSM over the configuration bus and used to fill in the memory.

## VIDEO GENERATOR/MASTER TIMER

The video generator **46** performs the digital to analog conversion of the image data in the display memory to send to the display head. It combines the data stream from the overlay memory of the DMC with the display memory from the perspective view. The configuration bus loads the color map.

A 30 Hz interlaced refresh rate may be implemented in a system employing the present invention. Color pallets are loadable by the GPP. The invention assumes a linear color space in RGB. All colors at zero intensity go to black.

## THREE DIMENSIONAL SYMBOL GENERATOR

The three-dimensional symbol generator **38** performs the following tasks:

1. It places the model to world transformation coefficients in the GAP.

2. It operates in cooperation with the geometry engine to multiply the world to screen transformation matrix by the model to world transformation matrix to form a model to screen transformation matrix. This matrix is stored over the model to world transformation matrix.

3. It operates in cooperation with the model to screen transformation matrix to each point of the symbol from the vertex list to transform the generic icon to the particular symbol.

4. It processes the connectivity list in the tiling engine and forms the screen polygons and passes them to the rendering engine.

One example of a three-dimensional symbol generator is described in detail in the assignee's aforereferenced patent application entitled "Three Dimensional Computer Graphic Symbol Generator".

The symbol generator data base consists of vertex list library and 64K bytes of overlay RAM and a connectivity list. Up to 18K bytes of DFAD (i.e., 2K bytes display list from cache shadow RAM×9 buffer segments) are loaded into the overlay RAM for cultural feature processing. The rest of the memory holds the threat/intelligence file and the mission planning file for the entire gaming area. The overlay RAM is loaded over the control bus from the DSM processor with the threat and mission planning files. The SHAG loads the DFAD files. The symbol libraries are updated via the configuration bus.

The vertex list contains the relative vertex positions of the generic library icons. In addition, it contains a 16 bit surface normal, a one bit end of polygon flag, and a one bit end of symbol flag. The table is 32K×16 bits. A maximum of 512 vertices may be associated with any given icon. The connectivity list contains the connectivity information of the vertices of the symbol. A 64K by 12 bit table holds this information.

A pathway in the sky format may be implemented in this system. It consists of either a wire frame tunnel or an elevated roadbed for flight path purposes. The wire frame tunnel is a series of connected transparent rectangles generated by the tiling engine of which only the edges are visible (wire mesh). Alternatively, the polygons may be precomputed in world coordinates and stored in a mission planning file. The roadbed is similarly comprised of polygons generated by the tiler along a designated pathway. In either case, the geometry engine must transform these polygons from object space (world coordinate system) to screen space. The transformed vertices are then passed to the rendering engine. The parameters (height, width, frequency) of the tunnel and roadbed polygons are programmable.

Another symbol used in the system is a waypoint flag. Waypoint flags are markers consisting of a transparent or opaque triangle on a vertical staff rendered in perspective. The waypoint flag icon is generated by the symbol generator as a macro from a mission planning file. Alternatively, they may be precomputed as polygons and stored. The geometry engine receives the vertices from the symbol generator and performs the perspective transformation on them. The geometry engine passes the rendering engine the polygons of the flag staff and the scaled font call of the alphanumeric symbol. Plan view format consists of a circle with a number inside and is not passed through the geometry engine.

DFAD data processing consists of a generalized polygon renderer which maps 32K points possible down to 256 polygons or less for a given buffer segment. These polygons are then passed to the rendering engine. This approach may redundantly render terrain and DFAD for the same pixels but easily accommodates declutter of individual features. Another approach is to rasterize the DFAD and use a texture warp function to color the terrain. This would not permit declutter of individual features but only classes (by color). Terrain color show-through in sparse overlay areas would be handled by a transparent color code (screen door effect). No verticality is achieved.

There are 298 categories of aerial, linear, and point features. Linear features must be expanded to a double line to prevent interlace strobing. A point feature contains a length, width, and height which can be used by the symbol generator for expansion. A typical lake contains 900 vertices and produces 10 to 20 active edges for rendering at any given scan line. The number of vertices is limited to 512. The display list is 64K bytes for a 1:250K buffer segment. Any given feature could have 32K vertices.

Up to 2K bytes of display list per buffer segment DTED is accommodated for DFAD. The DSM can tag the classes or individual features for clutter/declutter by toggling bits in the overlay RAM of the SHAG.

The symbol generator processes macros and graphic primitives which are passed to the rendering engine. These primitives include lines, arcs, alphanumerics, and two dimensional symbology. The rendering engine draws these primitives and outputs pixels which are anti-aliased. The GAP transforms these polygons and passes them to the rendering engine. A complete 4×4 Euler transformation is performed. Typical macros include compass rose and range scale symbols. Given a macro command, the symbol generator produces the primitive graphics calls to the rendering engine. This mode operates in plan view only and implements two dimensional symbols. Those skilled in the art will appreciate that the invention is not limited to specific fonts.

Three dimensional symbology presents the problem of clipping to the view volume. A gross clip is handled by the DSM in the cache memory at scan out time. The base of a threat dome, for example, may lie outside the orthographic projection of the view volume onto cache, yet a part of its dome may end up visible on the screen. The classical implementation performs the functions of tiling, transforming, clipping to the view volume (which generates new polygons), and then rendering. A gross clip boundary is implemented in cache around the view volume projection to guarantee inclusion of the entire symbol. The anomaly under animation to be avoided is that of having symbology sporadically appear and disappear in and out of the frame at the frame boundaries. A fine clip to the screen is performed downstream by the rendering engine. There is a 4K boundary around the screen which is rendered. Outside of this boundary, the symbol will not be rendered. This causes extra rendering which is clipped away.

Threat domes are represented graphically in one embodiment by an inverted conic volume. A threat/intelligence file contains the location and scaling factors for the generic model to be transformed to the specific threats. The tiling engine contains the connectivity information between the vertices and generates the planar polygons. The threat polygons are passed to the rendering engine with various viewing parameters such as mesh, opaque, dot, transparent, and so forth.

Graticles represent latitude and longitude lines, UTM klicks, and so forth which are warped onto the map in perspective. The symbol generator produces these lines.

Freeze frame is implemented in plan view only. The cursor is flown around the screen, and is generated by the symbol generator.

Programmable blink capability is accommodated in the invention. The DSM updates the overlay RAM toggle for display. The processor clock is used during variable frame update rate to control the blink rate.

A generic threat symbol is modeled and stored in the three dimensional symbol generation library. Parameters such as position, threat range, and angular threat view are passed to the symbol generator as a macro call (similar to a compass rose). The symbol generator creates a polygon list for each threat instance by using the parameters to modify the generic model and place it in the world coordinate system of the terrain data base. The polygons are transformed and rendered into screen space by the perspective view pipeline. These polygons form only the outside envelope of the threat cone.

This invention has been described herein in considerable detail in order to comply with the Patent Statues and to provide those skilled in the art with the information needed to apply the novel principles and to construct and use such specialized components as are required. However, it is to be understood that the invention can be carried out by specifically different equipment and devices, and that various modifications, both as to the equipment details and operating procedures,

can be accomplished without departing from the scope of the invention itself.

What is claimed is:

**1**. A system for providing a texture mapped perspective view for a digital map system wherein objects are transformed from texture space having U, V coordinates to screen space having X, Y coordinates comprising:

(a) a cache memory means for storing terrain data including elevation posts, wherein the cache memory means includes an output and an address bus;

(b) a shape address generator means for scanning cache memory having an ADDRESS SIGNAL coupled to the cache memory means address bus wherein the shape address generator means scans the elevation posts out of the cache memory means;

(c) a geometry engine coupled to the cache memory means output to receive the elevation posts scanned from the cache memory by the shape address generator means, the geometry engine including means for

i. transformation of the scanned elevation posts from object space to screen space so as to generate transformed vertices in screen coordinates for each elevation post, and

ii. generating three dimensional coordinates;

(d) a tilling engine coupled to the geometry engine for generating planar polygons from the generated three dimensional coordinates;

(e) a symbol generator to the geometry engine for transmitting a vertex list to the geometry engine wherein the geometry engine operates on the vertex list to transform the vertex list into screen space X, Y coordinates and passes the screen space X, Y coordinates to the tilling engine for generating planar polygons which form icons for display and processing information from the tilling engine into symbols,

(f) a texture engine means coupled to receive the ADDRESS SIGNAL from the shape address generator means including a texture memory and including a means for generating a texture vertex address to texture space correlated to an elevation post address and further including a means for generating a texture memory address for scanning the texture memory wherein the texture memory provides texture data on a texture memory data bus in response to being scanned by the texture memory address;

(g) a rendering engine having an input coupled to the tilling engine and the texture memory data bus for generating image data from the planar polygons; and

(h) a display memory for receiving image data from the rendering engine output wherein the display memory includes at least four first-in, first-out memory buffers.

**2**. The apparatus of claim **1** wherein each polygon has a surface and the rendering means assigns one color across the surface of each polygon.

**3**. The apparatus of claim **1** wherein the vertices of each polygon have an intensity and the rendering means interpolates the intensities between the vertices of each polygon in a linear fashion.

**4**. The apparatus of claim **1** wherein the rendering means further includes means for generating transparent polygons and passing the transparent polygon to the display memory.

5. A method for providing a texture mapped perspective view for a digital map system having a cache memory, a geometry engine coupled to the cache memory, a shape address generator coupled to the cache memory, a tiling engine coupled to the geometry engine, a symbol generator coupled to the geometry engine and the tiling engine, a texture engine coupled to the cache memory, a rendering engine coupled to the tiling engine and the texture engine, and a display memory coupled to the rendering engine, wherein objects are transformed from texture space having U, V coordinates to screen space having X, Y coordinates, the method comprising the steps of:

(a) storing terrain data, including elevation posts, in the cache memory;

(b) scanning the cache memory to retrieve the elevation posts;

(c) transforming the terrain data from elevation posts in object space to transformed vertices in screen space, and

(d) generating planar polygons from the generated three dimensional coordinates;

(e) transmitting a vertex list to the geometry engine, operating the geometry engine to transform the vertex list into screen space X, Y coordinates and passing the screen space X, Y coordinates to the tiling engine for generating planar polygons which form icons for display;

(f) tagging elevation posts with corresponding addresses in texture space;

(g) generating image data in the rendering engine from the planar polygons and the tagged elevation posts; and

(h) storing the generated image data in the display memory wherein the display memory comprises at least four first-in, first-out memory buffers and the step of storing the generated images includes storing the generated image data in the at least four First-in, First-out memory buffers.

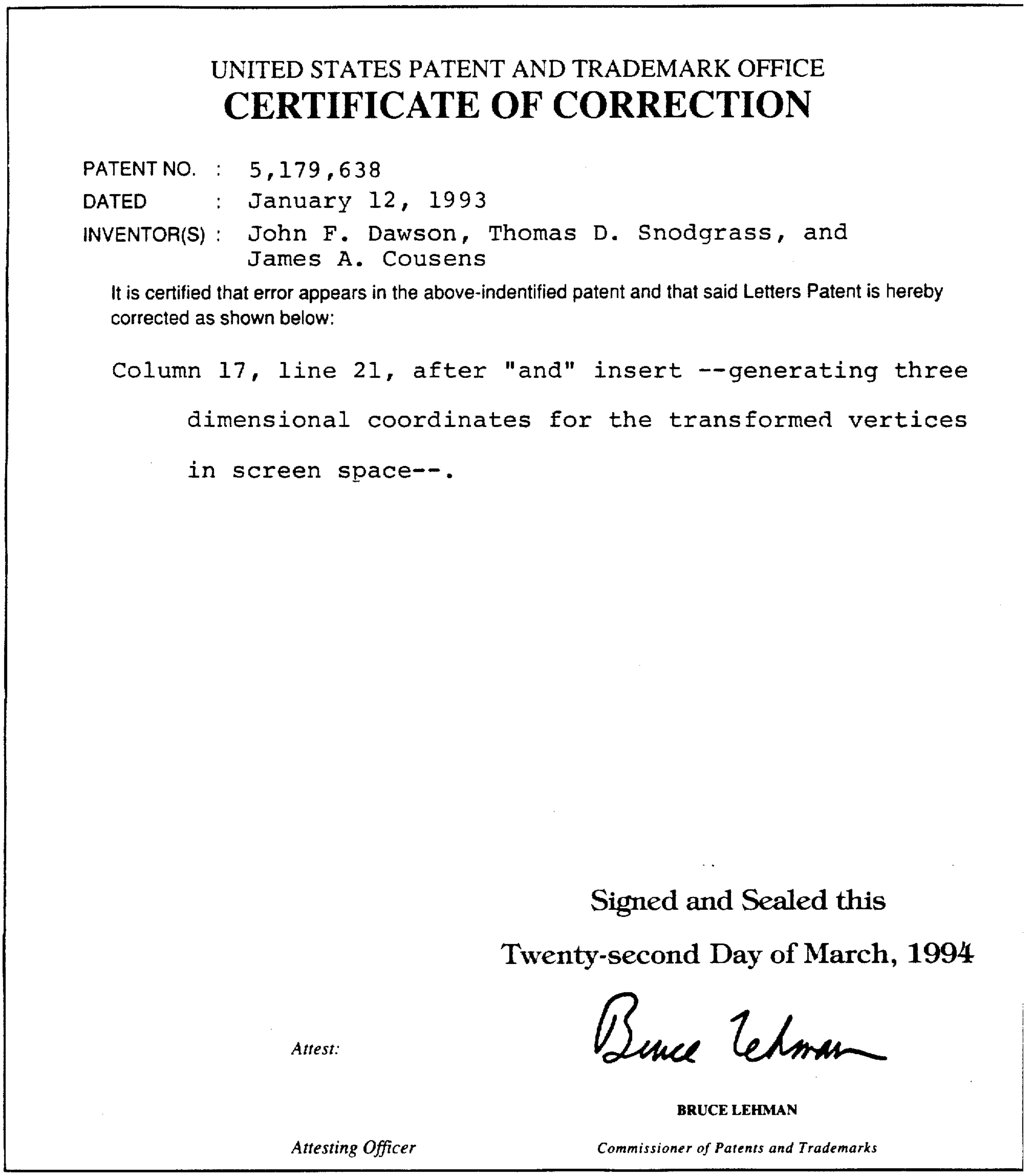
6. The method of claim 5 wherein each polygon has a surface and wherein the step of generating image data further includes the steps of assigning one color across the surface of each polygon.

7. The method of claim 5 wherein the vertices of each polygon have an intensity and the step of generating image data further includes the step of interpolating the intensities between the vertices of each polygon in a linear fashion.

8. The method of claim 5 wherein the step of generating image data further includes the step of generating transparent polygons and passing the transparent polygons to the display memory.

* * * * *

**APPENDIX K**



UNITED STATES PATENT AND TRADEMARK OFFICE

# CERTIFICATE OF CORRECTION

PATENT NO. : 5,179,638
DATED : January 12, 1993
INVENTOR(S) : John F. Dawson, Thomas D. Snodgrass, and
James A. Cousens

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Column 17, line 21, after "and" insert --generating three dimensional coordinates for the transformed vertices in screen space--.

Signed and Sealed this
Twenty-second Day of March, 1994

Attest:

Bruce Lehman

BRUCE LEHMAN

*Attesting Officer* *Commissioner of Patents and Trademarks*

APPENDIX L

# Pyramidal Parametrics

Lance Williams

Computer Graphics Laboratory
New York Institute of Technology
Old Westbury, New York

## Abstract

The mapping of images onto surfaces may substantially increase the realism and information content of computer-generated imagery. The projection of a flat source image onto a curved surface may involve sampling difficulties, however, which are compounded as the view of the surface changes. As the projected scale of the surface increases, interpolation between the original samples of the source image is necessary; as the scale is reduced, approximation of multiple samples in the source is required. Thus a constantly changing sampling window of view-dependent shape must traverse the source image.

To reduce the computation implied by these requirements, a set of prefiltered source images may be created. This approach can be applied to particular advantage in animation, where a large number of frames using the same source image must be generated. This paper advances a "pyramidal parametric" prefiltering and sampling geometry which minimizes aliasing effects and assures continuity within and between target images.

Although the mapping of texture onto surfaces is an excellent example of the process and provided the original motivation for its development, pyramidal parametric data structures admit of wider application. The aliasing of not only surface texture, but also highlights and even the surface representations themselves, may be minimized by pyramidal parametric means.

General Terms: Algorithms.

Keywords and Phrases: Antialiasing, Illumination Models, Modeling, Pyramidal Data Structures, Reflectance Mapping, Texture Mapping, Visible Surface Algorithms.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation--display algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling--curve, surface, solid and object representations, geometric algorithms, languages and systems; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism--color, shading, shadowing, and texture.

## 1. Pyramidal Data Structures

Pyramidal data structures may be based on various subdivisions: binary trees, quad trees, oct trees, or n-dimensional hierarchies [17]. The common feature of these structures is a succession of levels which vary the resolution at which the data is represented.

The decomposition of an image by two-dimensional binary subdivision was a pioneering strategy in computer graphics for visible surface determination [15]. The approach was essentially a synthesis-by-analysis: the image plane was subdivided into quadrants recursively until analysis of a subsection showed that surface ordering was sufficiently simple to permit rendering. Such subdivision and analysis has been subsequently adopted to generate spatial data structures [5], which have been used to represent images [9] both for pattern recognition [13] and for transmission [10], [14]. In the field of computer graphics, such data structures have been adopted for texture mapping [4], [16], and generalized to represent objects in space [11].

The application of pyramidal data to image storage and transmission may permit significant compression of the data to be stored or transmitted. This is so because highly detailed features may be localized within an otherwise low-frequency image, permitting the sampling rate to be reduced for large sections of the image. Besides permitting bandwidth compression, the representation orders data in such a way that the general character of images may be recalled or transmitted before the specific details.

Pattern recognition and classification often require the comparison of a candidate image against a set of canonical patterns. This is an operation the expense of which increases as the square of the resolution at which it is performed. The use of pyramidal data structures in pattern recognition and classification permits the comparison of the gross features of two-dimensional functions preliminary to the minute particulars; a good general reference on this application is [12].

© ACM 0-89791-109-1/83/007/0001 $00.75

APPENDIX L

In computer graphics, pyramidal texture maps may be used to perform arbitrary mappings of a function with minimal aliasing artifacts and reduced computation. Once again, images may be represented at different spatial bandwidths. The concern is that inappropriate resolution misrepresents the data; that is, sampling high-resolution data at larger sample intervals invites aliasing.

## 2. Parametric Interpolation

By a pyramidal parametric data structure, we will mean simply a pyramidal structure with both intra- and inter-level interpolation. Consider the case of an image represented as a two-dimensional array of samples. Interpolation is necessary to produce a continuous function of two parameters, U and V. If, in addition, a third parameter (call it D) moves us up and down a hierarchy of corresponding two-dimensional functions, with interpolation between (or among) the levels of the pyramid providing continuity, the structure is pyramidal parametric.

The practical distinction between such a structure and an ordinary interpolant over an n-dimensional array of samples is that the number of samples representing each level of the pyramid may be different.

## 3. Mip Mapping

"Mip" mapping is a particular format for two-dimensional parametric functions, which, along with its associated addressing scheme, has been used successfully to bandlimit texture mapping at New York Institute of Technology since 1979. The acronym "mip" is from the Latin phrase "multum in parvo," meaning "many things in a small place." Mip mapping supplements bilinear interpolation of pixel values in the texture map (which may be used to smoothly translate and magnify the texture) with interpolation between prefiltered versions of the map (which may be used to compress many pixels into a small place). In this latter capacity, mip offers much greater speed than texturing algorithms which perform explicit convolution over an area in the texture map for each pixel rendered [1], [6].

Mip owes its speed in compressing texture to two factors. First, a fair amount of filtering of the original texture takes place when the mip map is first created. Second, subsequent filtering is approximated by blending different levels of the mip map. This means that all filters are approximated by linearly interpolating a set of square box filters, the sides of which are powers-of-two pixels in length. Thus, mapping entails a fixed overhead, which is independent of the area filtered to compute a sample.

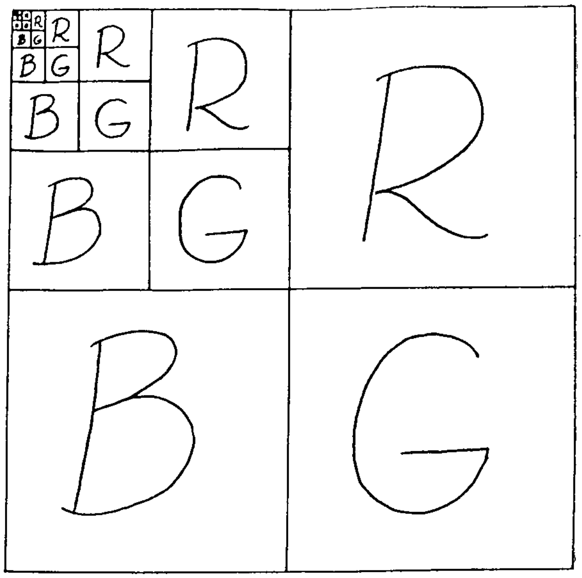


Figure (1)
Structure of a Color Mip Map
Smaller and smaller images diminish into the upper left corner of the map. Each of the images is averaged down from its larger predecessor.

(Below:)
Mip maps are indexed by three coordinates: U, V, and D. U and V are spatial coordinates of the map; D is the variable used to index, and interpolate between, the different levels of the pyramid.

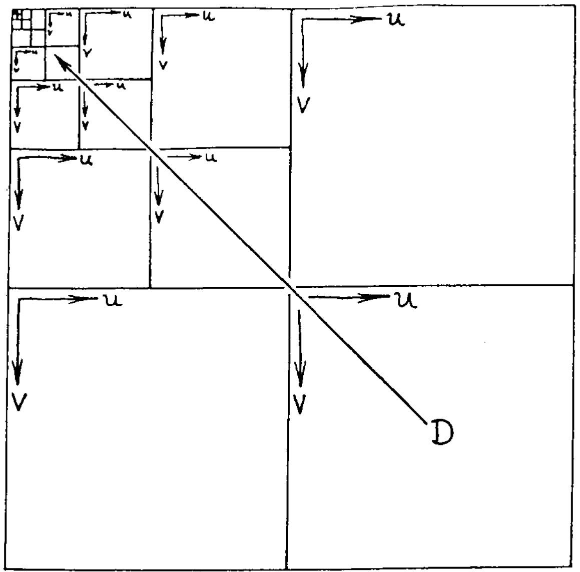


Figure (1) illustrates the memory organization of a color mip map. The image is separated into its red, green, and blue components (R, G, and B in the diagram). Successively filtered and down-sampled versions of each component are instanced above and to the left of the originals, in a series of smaller and smaller images, each half the linear dimension (and a quarter the number of

APPENDIX L

samples) of its parent. Successive divisions by four partition the frame buffer equally among the three components, with a single unused pixel remaining in the upper left-hand corner.

The concept behind this memory organization is that corresponding points in different prefiltered maps can be addressed simply by a binary shift of an input U, V coordinate pair. Since the filtering and sampling are performed at scales which are powers of two, indexing the maps is possible with inexpensive binary scaling. In a hardware implementation, the addresses in all the corresponding maps (now separate memories) would be instantly and simultaneously available from the U, V input.

The routines for creating and accessing mip maps at NYIT are based on simple box (Fourier) window prefiltering, bilinear interpolation of pixels within each map instance, and linear interpolation between two maps for each value of D (the pyramid's vertical coordinate). For each of the three components of a color mip map, this requires 8 pixel reads and 7 multiplications. This choice of filters is strictly for the sake of speed. Note that the bilinear interpolation of pixel values at the extreme edges of each map instance must be performed with pixels from the opposite edge(s) of that map, for texture which is periodic. For non-periodic texture, scaling or clipping of the U, V coordinates prevents the intrusion of an inappropriate map or color component into the interpolation.

The box (Fourier) window used to create the mip maps illustrated here, and the tent (Bartlett) window used to interpolate them, are far from ideal; yet probably the most severe compromise made by mip filtering is that it is symmetrical. Each of the prefiltered levels of the map is filtered equally in X and Y. Choosing a value of D trades off aliasing against blurring, which becomes a tricky proposition as a pixel's projection in the texture map deviates from symmetry. Heckbert [8] suggests:

$$d = \max\left(\sqrt{\left(\frac{\partial u}{\partial x}\right)^2+\left(\frac{\partial v}{\partial x}\right)^2}\ ,\sqrt{\left(\frac{\partial u}{\partial y}\right)^2+\left(\frac{\partial v}{\partial y}\right)^2}\right)$$

where D is proportional to the "diameter" of the area in the texture to be filtered, and the partials of U and V (the texture-map coordinates) with respect to X and Y (the screen coordinates) can be calculated from the surface projection.

Illustrations of mapping performed by the mip technique are the subject of Figures (2) through (10). The NYIT Test Frog in Figure (2) is magnified by simple point sampling in (3), and by interpolation in (4). The hapless amphibian is similarly compressed by point sampling in (5) and by mipping in (6).

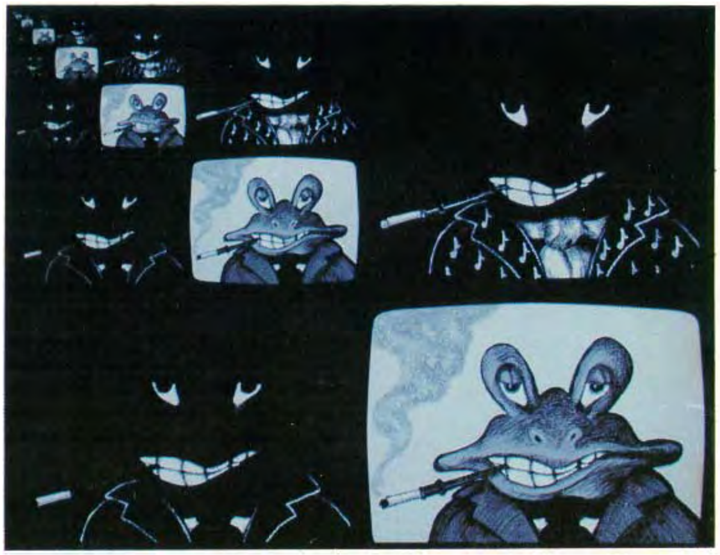


Figure (2)
Mip map of the flexible NYIT Test Frog.

The more general and interesting case -- continuously variable upsampling and downsampling of the original texture -- is illustrated in (7) on a variety of surfaces. Since the symmetry of mip filtering would be expected to show up badly when texture is compressed in only one dimension, figures (8) through (10) are of especial interest. These pictures, created by Ed Emshwiller at NYIT for his videotape, "Sunstone," were mapped using Alvy Ray Smith's TEXAS animation program, which in turn used MIP to antialias texture. As the panels rotate edge-on, the texture collapses to a line smoothly and without apparent artifacts.

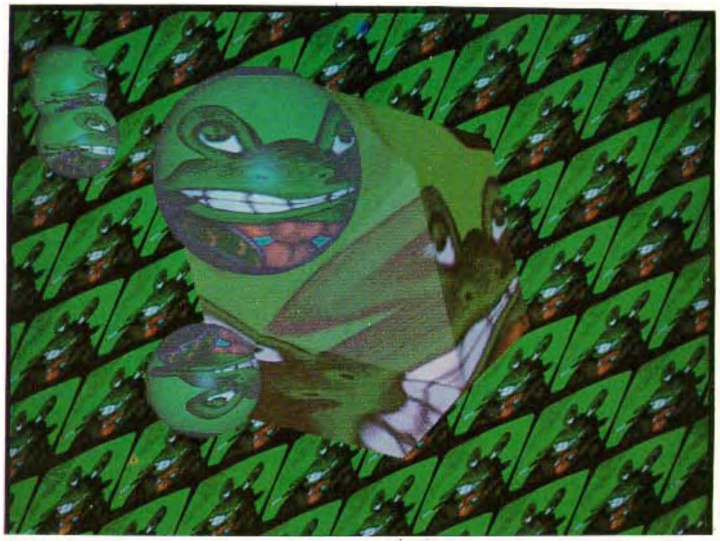


Figure (7)
General mapping: interpolation and pyramidal compression.

APPENDIX L

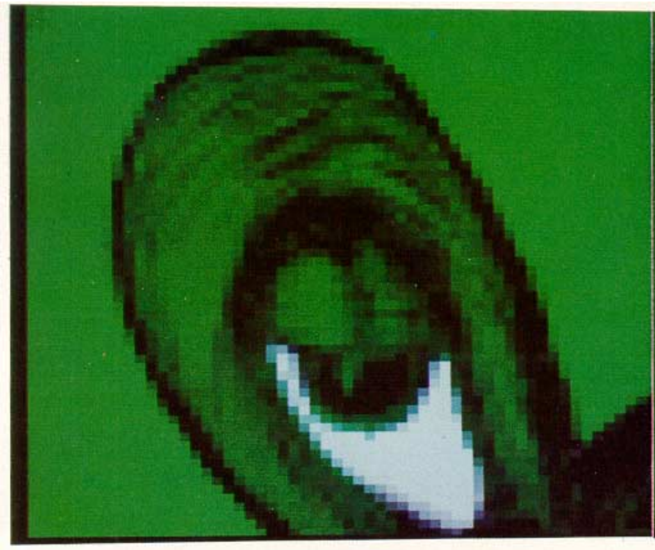


Figure (3)
Upsampling the frog: magnification by point sampling.

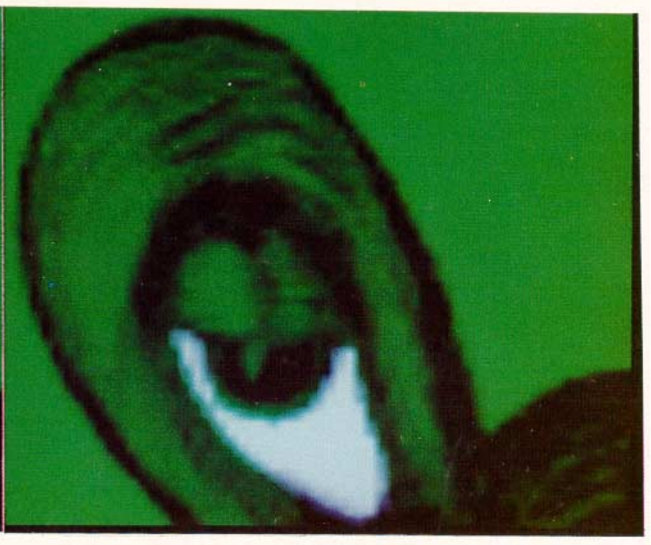


Figure (4)
Upsampling the frog: magnification by bilinear interpolation.

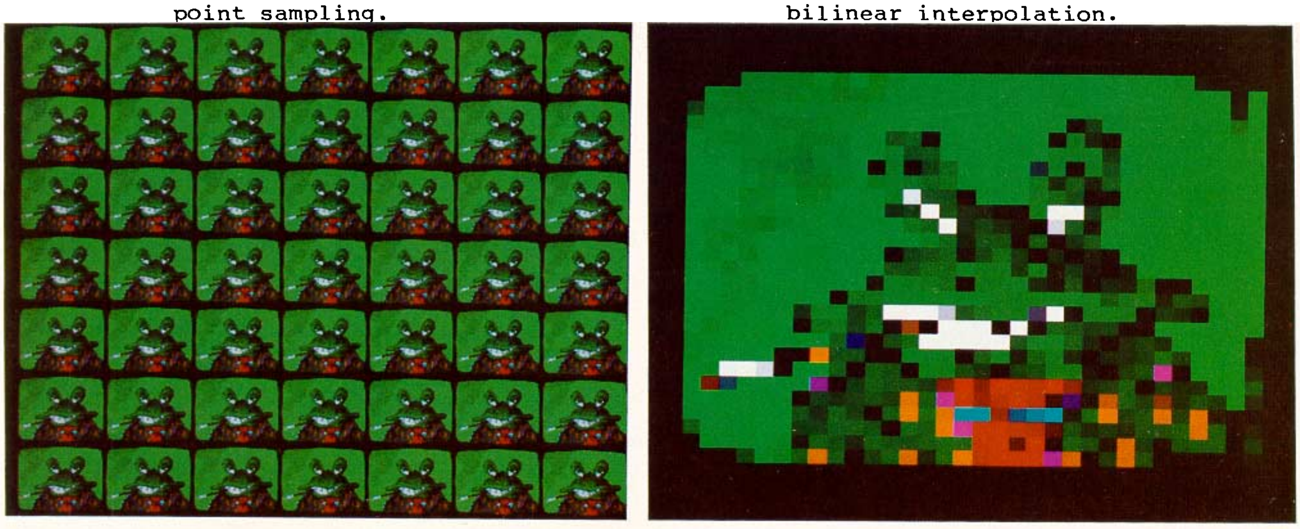


Figure (5)
Downsampling the frog: compression by point sampling (detail, right).

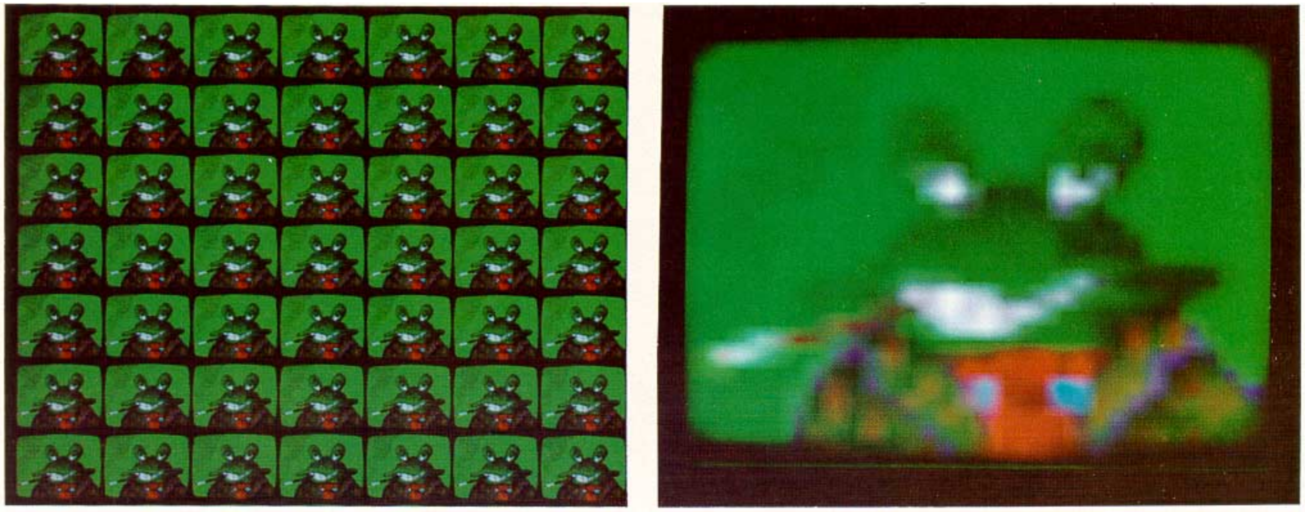
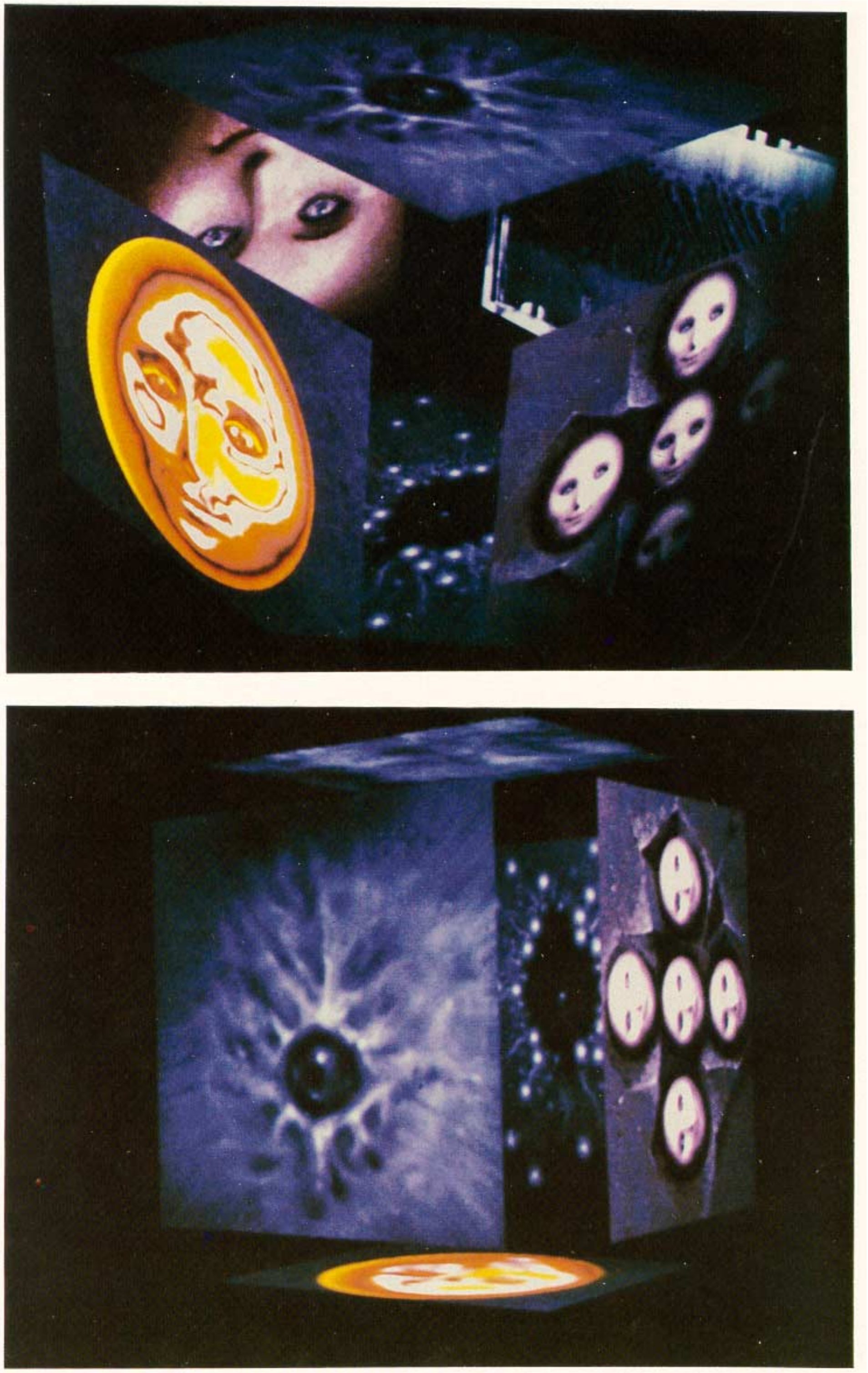


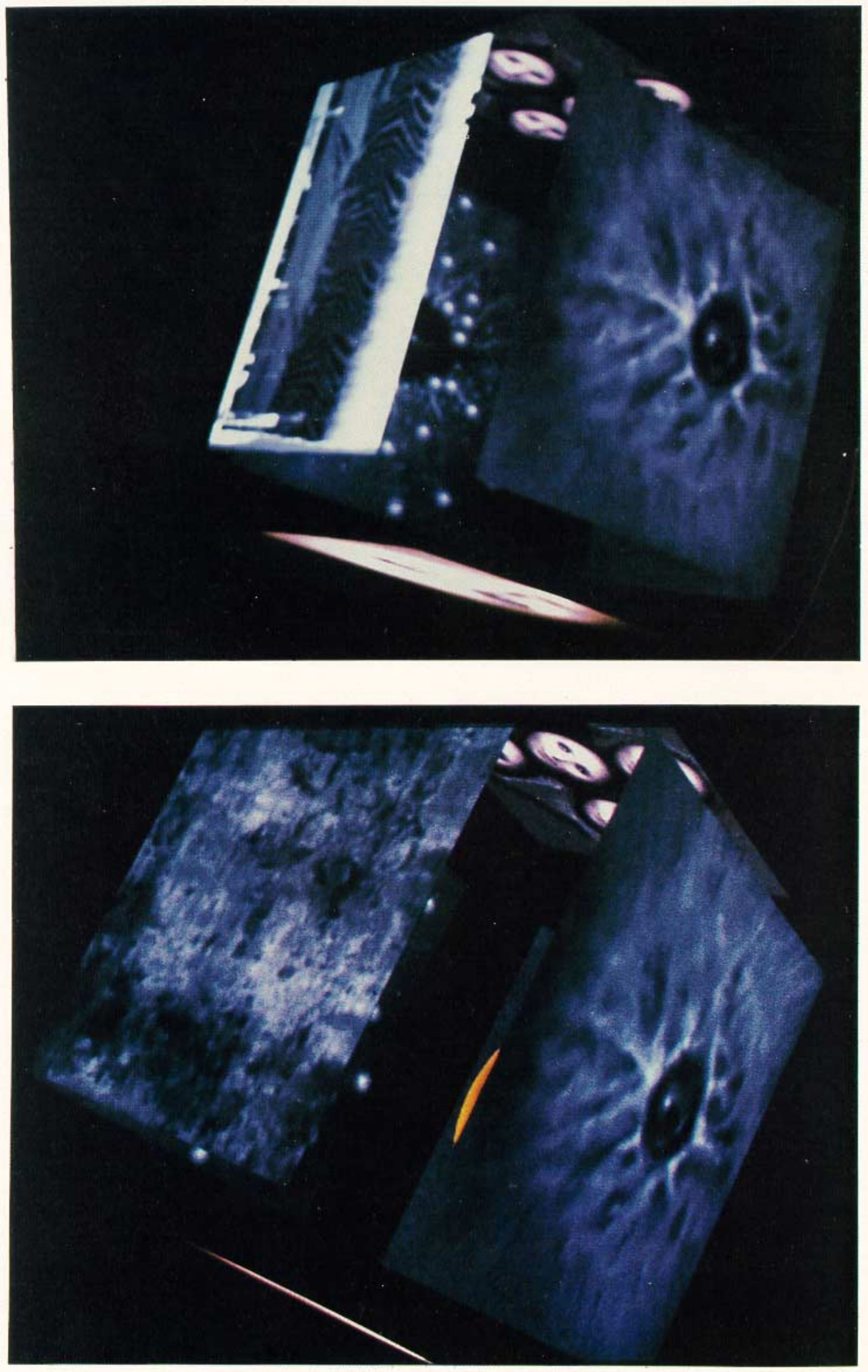
Figure (6)
Downsampling: compression by pyramidal interpolation (detail, right).

APPENDIX L



Figures (8)-(9)
"Sunstone" by Ed Emshwiller, segment animated by Alvy Ray Smith
Pyramidal parametric texture mapping on polygons.

APPENDIX L



Figures (10)-(11)
"Sunstone" by Ed Emshwiller, segment animated by Alvy Ray Smith
Pyramidal parametric texture mapping on polygons.

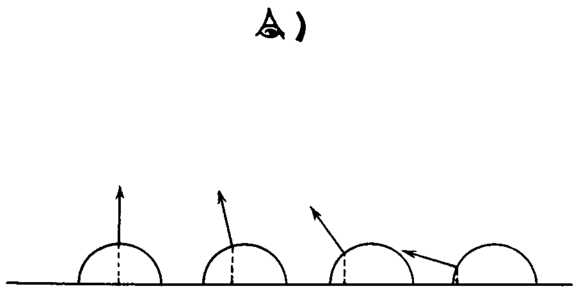
APPENDIX L

## 4. Highlight Antialiasing

As small or highly curved objects move across a raster, their surface normals may beat erratically with the sampling grid. This causes the shading values to flash annoyingly in motion sequences, a symptom of illumination aliasing. The surface normals essentially point-sample the illumination function.

Figure (12) illustrates samples of the surface normals of a set of parallel cylinders. The cylinders in the diagram are depicted as if from the edge of the image plane; the regularly-spaced vertical line segments are the samples along a single axis. The arrows at the sample points indicate the directions of the surface normals. Depending on the shading formula invoked, there may be very high contrast between samples where the normal is nearly parallel to the sample axis, and samples where the normal points directly at the observer's eye.

Figure (12)



The shading function depends not only on the shape of the surface, but its light reflection properties (characterized by the shading formula), the position of the light source, and the position of the observer's eye. Hanrahan [7] expresses it in honest Greek:

$$\int_X\int_Y \varphi(E,N,L)\,\frac{\partial(u,v)}{\partial(x,y)}\;dxdy$$

where the normal, N, the light sources, L, and the eye, E, are vectors which may each be functions of U and V, and the limits of integration are the X, Y boundaries of the pixel.

Figure (13) illustrates highlight aliasing on a perfectly flat surface. The viewing conventions of the diagram are the same as in Figure (12). "L" is the direction vector of the light source; the surface is a polygon at an angle to the image plane; the dotted bump is a graph of the reflected light, characteristic of a specular surface reflection function. The highlight indicated by the bump falls entirely between the samples. (Note that this is only possible on a flat surface if either the eye or the light is local, a point in space rather than simply a direction vector. Some boring shading formulae exclude the possibility of highlight aliasing on polygons by requiring all flat surfaces to be flat in shading.)

Figure (13)

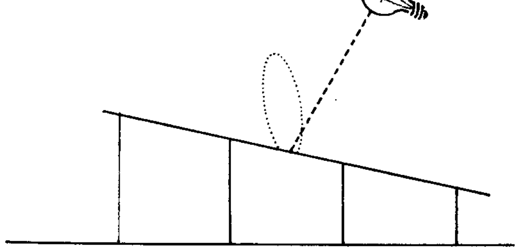
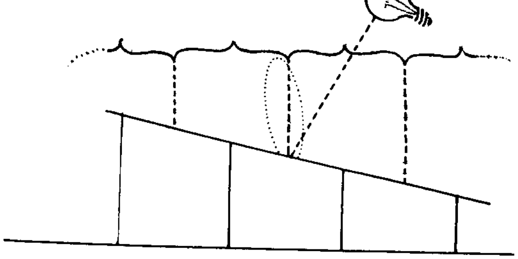


Figure (14)



A first attempt to overcome the limitations of point-sampling the illumination function is to integrate the function over the projected area represented by each sample point. This approach is illustrated in Figure (14). The brackets at each sample represent the area of the surface over which the illumination function is integrated. This procedure is analogous to area-averaging of sampled edges or texture [3].

In order to generalize this approach to curved surfaces, the "sample interval" over which illumination is integrated must be modified according to the local curvature of the surface at a sample. In Figure (15), the area of a surface represented by a pixel has been projected onto a curved surface. The solid angle over which illumination must be integrated is approximated by the volume enclosed by the normals at the pixel corners. The distribution of light within this volume will sum to an estimate of the diffuse reflection over the pixel. If the surface exhibits undulations at the pixel level, however, aliasing will result.

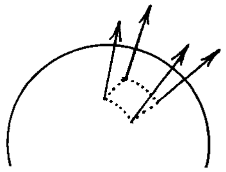


Figure (15)

APPENDIX L

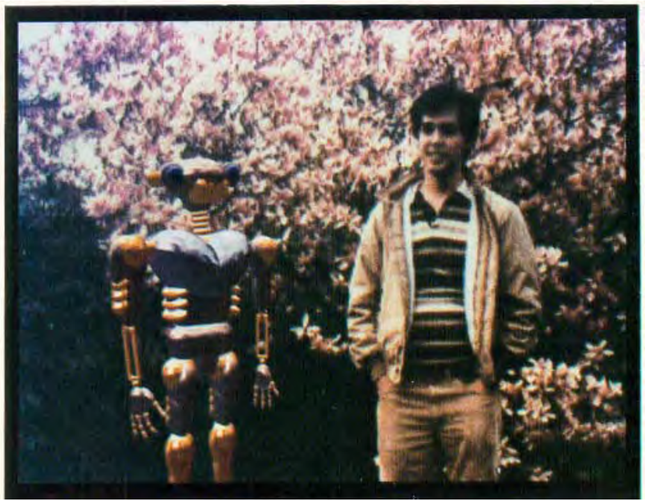


Figure (16)
Michael Chou (right) poses with an imaginary companion. Reflectance maps can enhance the realism of synthetic shading.

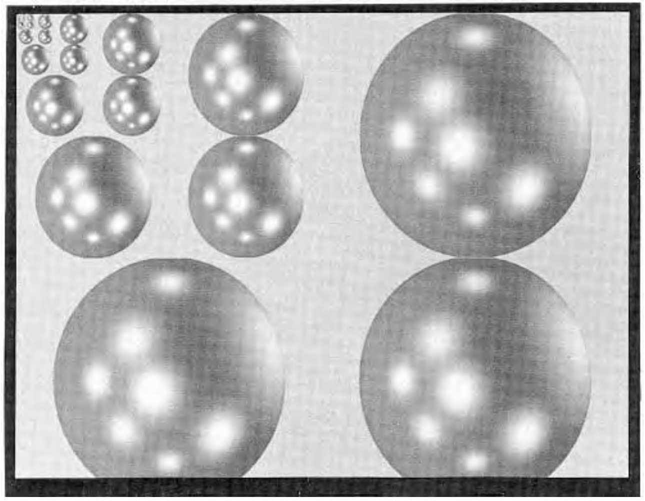


Figure (17)
A pyramidal parametric reflectance map, containing 9 light sources. The region outside the "sphere" is unused.

We might divide the surface up into regions of relatively low curvature (as is done in some patch rendering algorithms), and rely on "edge antialiasing" to integrate the different surfaces within a pixel. Alternatively, we may develop some mechanism for limiting the local curvature of surfaces before rendering. This possibility is explored in the next section.

If we represent the illumination of a scene as a two-dimensional map, highlights can be effectively antialiased in much the same way as textures. Blinn and Newell [1] demonstrated specular reflection using an illumination map. The map was an image of the environment (a spherical projection of the scene, indexed by the X and Y components of the surface normals) which could be used to cast reflections onto specular surfaces. The impression of mirrored facets and chrome objects which can be achieved with this method is striking; Figure (16) provides an illustration. Reflectance mapping is not, however, accurate for local reflections. To achieve similar results with three dimensional accuracy requires ray-tracing.

A pyramidal parametric illumination map permits convenient antialiasing of highlights as long as a good measure of local surface curvature is available. The value of "D" used to index the map is proportional to the solid angle subtended by the surface over the pixel being computed; this may be estimated by the same formula used to compute D for ordinary texture mapping. Nine light sources of varying brightness glint raggedly from the test object in Figure (18); the reflectance map in Figure (17) provided the illumination. In Figure (19), convincing highlight antialiasing results from the full pyramidal parametric treatment.

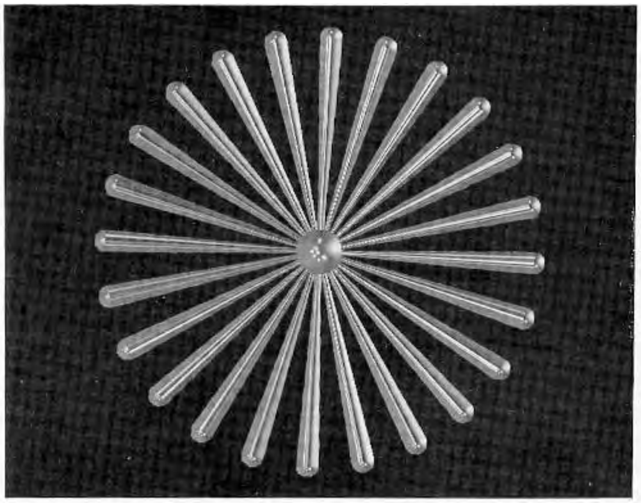


Figure (18) Before

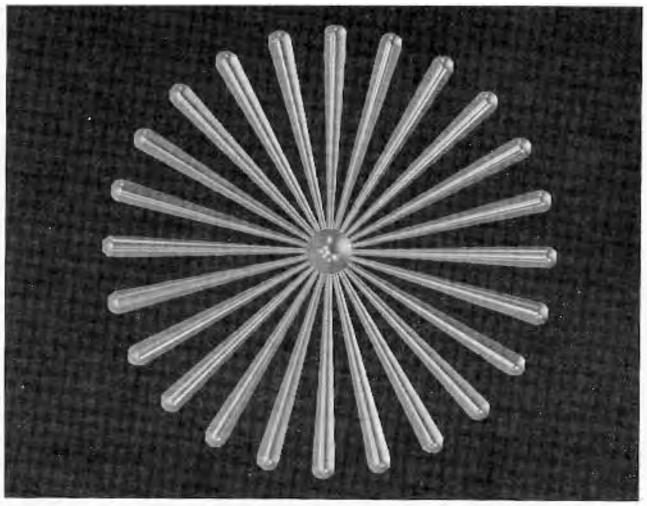
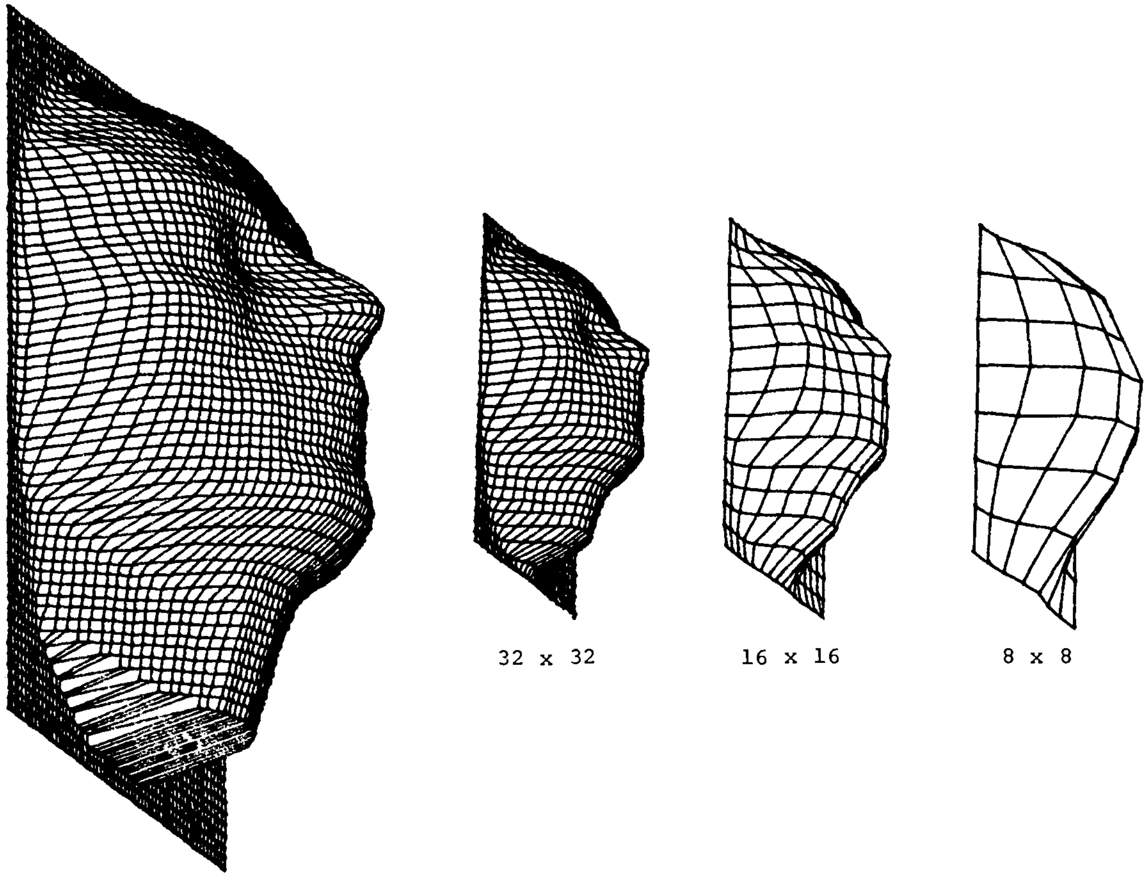


Figure (19) After

APPENDIX L



32 x 32

16 x 16

8 x 8

64 x 64

Figures (20-23) Different resolution meshes.

## 5. Levels of Detail in Surface Representation

In addition to bandlimiting texture and illumination functions for mapping onto a surface, pyramidal parametrics may be used to limit the level of detail with which the surface itself is represented. The goal is to represent an object for graphic display as economically as its projection on the image plane permits, without boiling and sparkling aliasing artifacts as the projection changes.

The expense of computing and shading each pixel dominates the cost of many algorithms for rendering higher-order surfaces. For meshes of polygons or patch control points which project onto a small portion of the image, however, the vertex (or control-point) expense dominates. In these situations it is desirable to reduce the number of points used to represent the object.

A pyramidal parametric data structure the components of which are spatial coordinates (the X-Y-Z of the vertices of a rectangular mesh, for example, as opposed to the R-G-B of a texture or illumination map) provides a continuously-variable filtered instance of the surface for sampling at any desired degree of resolution.

Figures (20) through (23) illustrate a simple surface based on a human face model developed by Fred Parke at the University of Utah. As the sampling density varies, so does the filtering of the surface. These faces are filtered and sampled by the same methods previously discussed for texture and reflectance maps. Pyramidal parametric representations such as these appear promising for reducing aliasing effects as well as systematically sampling very large data bases over a wide range of scales and viewing angles.

## 6. Conclusions

Pyramidal data structures are of proven value in image analysis and have interesting application to image bandwidth compression and transmission. "Pyramidal parametrics," pyramidal data structures with intra- and inter-level interpolation, are here proposed for use in image synthesis. By continuously varying the detail with which data are resolved, pyramidal parametrics provide economical approximate solutions to filtering problems in mapping texture and illumination onto surfaces, and preliminary experiments suggest they may provide flexible surface representations as well.

## 7. Acknowledgments

I would like to acknowledge Ed Catmull, the first (to my knowledge) to apply multiple prefiltered images to texture mapping: the method was applied to the bicubic patches in his thesis, although it was not described. Credit is also due Tom Duff, who wrote both recursive and scan-order routines for creating mip maps which preserved numerical precision over all map instances; Dick Lundin, who wrote the first assembly-coded mip map accessing routines; Ephraim Cohen, who wrote the second; Rick Ace, who translated Ephraim's PDP-11 versions for the VAX assembler; Paul Heckbert, for refining and speeding up both creation and accessing routines, and investigating various estimates of "D"; Michael Chou, for implementing highlight antialiasing and high-resolution reflectance mapping on quadric surfaces.

I owe special thanks to Jules Bloomenthal, Michael Chou, Pat Hanrahan, and Paul Heckbert for critical reading and numerous helpful suggestions in the course of preparing this text. Photographic support was provided by Michael Lehman.

## 8. References

[1] Blinn, J., and Newell, M., "Texture and Reflection on Computer Generated Images," CACM, Vol. 19, #10, Oct. 1976, pp. 542-547.

[2] Bui-Tuong Phong, "Illumination for Computer Generated Pictures," PhD. dissertation, Department of Computer Science, University of Utah, December 1978.

[3] Crow, F.C., "The Aliasing Problem in Computer Synthesized Shaded Images," PhD. dissertation, Department of Computer Science, University of Utah, Tech. Report UTEC-CSc-76-015, March 1976.

[4] Dungan, W., Stenger, A., and Sutty, G., "Texture Tile Considerations for Raster Graphics," SIGGRAPH 1978 Proceedings, Vol. 12, #3, August 1978.

[5] Eastman, Charles M., "Representations for Space Planning," CACM, Vol. 13, #4, April 1970.

[6] Feibush, E.A., Levoy, M., and Cook, R.L., "Synthetic Texturing Using Digital Filters," Computer Graphics, Vol. 14, July, 1980.

[7] Hanrahan, Pat, private communication, 1983.

[8] Heckbert, Paul, "Texture Mapping Polygons in Perspective," NYIT Computer Graphics Lab Tech. Memo #13, April, 1983.

[9] Klinger, A., and Dyer, C.R., "Experiments on Picture Representation Using Regular Decomposition," Computer Graphics and Image Processing, #5, March, 1976.

[10] Knowlton, K., "Progressive Transmission of Gray-Scale and Binary Pictures by Simple, Efficient, and Lossless Encoding Schemes," Proceedings of the IEEE, Vol. 68, #7, July 1980, pp. 885-896.

[11] Meagher, D., "Octree Encoding: A New Technique for the Representation, Manipulation, and Display of Arbitrary 3D Objects by Computer," IPL-TR-80-111, Image Processing Lab, Electrical and Systems Engineering Dept., Rensselaer Polytechnic Institute, October 1980.

[12] Tanimoto, S.L., and Klinger, A., Structured Computer Vision, Academic Press, New York, 1980.

[13] Tanimoto, S.L., and Pavlidis, T., "A Hierarchical Data Structure for Picture Processing," Computer Graphics and Image Processing, Vol. 4, #2, June 1975.

[14] Tanimoto, S.L., "Image Processing with Gross Information First," Computer Graphics and Image Processing 9, 1979.

[15] Warnock, J.E., "A Hidden-Line Algorithm for Halftone Picture Representation," Department of Computer Science, University of Utah, TR 4-15, 1969.

[16] Williams, L., "Pyramidal Parametrics," SIGGRAPH tutorial notes, "Advanced Image Synthesis," 1981.

[17] Yau, M.M., and Srihari, S.N., "Recursive Generation of Hierarchical Data Structures for Multidimensional Digital Images," Proceedings of the IEEE Computer Society Conference on Pattern Recognition and Image Processing, August 1981.

APPENDIX M

Next Up Previous Contents Index
**Next:** 3.8.2 Texture Magnification **Up:** 3.8.1 Texture Minification **Previous:** 3.8.1 Texture Minification

## Mipmapping

`TEXTURE_MIN_FILTER` values `NEAREST_MIPMAP_NEAREST`, `NEAREST_MIPMAP_LINEAR`, `LINEAR_MIPMAP_NEAREST`, and `LINEAR_MIPMAP_LINEAR` each require the use of a *mipmap*. A mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one. If the texture has dimensions $2^n \times 2^m$, then there are $\max\{n, m\} + 1$ mipmap arrays. The first array is the original texture with dimensions $2^n \times 2^m$. Each subsequent array has dimensions $2^{(k-1)} \times 2^{(l-1)}$ where $2^k \times 2^l$ are the dimensions of the previous array. This is the case as long as both **k>0** and **l>0**. Once either **k=0** or **l=0**, each subsequent array has dimension $1 \times 2^{(l-1)}$ or $2^{(k-1)} \times 1$, respectively, until the last array is reached with dimension $1 \times 1$.

Each array in a mipmap is transmitted to the GL using **TexImage2D** or **TexImage1D** ; the array being set is indicated with the *level-of-detail* argument. Level-of-detail numbers proceed from **0** for the original texture array through $p = \max\{n, m\}$ with each unit increase indicating an array of half the dimensions of the previous one as already described. If texturing is enabled (and `TEXTURE_MIN_FILTER` is one that requires a mipmap) at the time a primitive is rasterized and if the set of arrays **0** through **p** is incomplete, based on the dimensions of array **0**, then it is as if texture mapping were disabled. The set of arrays **0** through **p** is incomplete if the internal formats of all the mipmap arrays were not specified with the same symbolic constant, or if the border widths of the mipmap arrays are not the same, or if the dimensions of the mipmap arrays do not follow the sequence described above. Arrays indexed greater than **p** are insignificant.

The mipmap is used in conjunction with the level of detail to approximate the application of an appropriately filtered texture to a fragment. Let $p = \max\{n, m\}$ and let **c** be the value of $\lambda$ at which the transition from minification to magnification occurs (since this discussion pertains to minification, we are concerned only with values of $\lambda$ where $\lambda > c$). For `NEAREST_MIPMAP_NEAREST`, if $c < \lambda \leq 0.5$ then the mipmap array with level-of-detail of 0 is selected. Otherwise, the **d**th mipmap array is selected when $d - \frac{1}{2} < \lambda \leq d + \frac{1}{2}$ as long as $1 \leq d \leq p$. If $\lambda > p + \frac{1}{2}$, then the **p**th mipmap array is selected. The rules for `NEAREST` are then applied to the selected array.

The same mipmap array selection rules apply for `LINEAR_MIPMAP_NEAREST` as for `NEAREST_MIPMAP_NEAREST`, but the rules for `LINEAR` are applied to the selected array.

For `NEAREST_MIPMAP_LINEAR`, the level **d-1** and the level **d** mipmap arrays are selected, where $d - 1 \leq \lambda < d$, unless $\lambda \geq p$, in which case the **p**th mipmap array is used for both arrays. The rules

**APPENDIX M**

for NEAREST are then applied to each of these arrays, yielding two corresponding texture values $\tau_{d-1}$ and $\tau_d$. The final texture value is then found as

$$\tau = [1 - \mathrm{frac}(\lambda)]\tau_{d-1} + \mathrm{frac}(\lambda)\tau_d.$$

LINEAR_MIPMAP_LINEAR has the same effect as NEAREST_MIPMAP_LINEAR except that the rules for LINEAR are applied for each of the two mipmap arrays to generate $\tau_{d-1}$ and $\tau_d$.

Next Up Previous Contents Index
**Next:** 3.8.2 Texture Magnification **Up:** 3.8.1 Texture Minification **Previous:** 3.8.1 Texture Minification

*David Blythe*
*Sat Mar 29 02:23:21 PST 1997*

Microsoft et al. Exhibit 1005

# Progressive Meshes

Hugues Hoppe
Microsoft Research

## ABSTRACT

Highly detailed geometric models are rapidly becoming commonplace in computer graphics. These models, often represented as complex triangle meshes, challenge rendering performance, transmission bandwidth, and storage capacities. This paper introduces the *progressive mesh* (PM) representation, a new scheme for storing and transmitting arbitrary triangle meshes. This efficient, lossless, continuous-resolution representation addresses several practical problems in graphics: smooth geomorphing of level-of-detail approximations, progressive transmission, mesh compression, and selective refinement.

In addition, we present a new mesh simplification procedure for constructing a PM representation from an arbitrary mesh. The goal of this optimization procedure is to preserve not just the geometry of the original mesh, but more importantly its overall appearance as defined by its discrete and scalar appearance attributes such as material identifiers, color values, normals, and texture coordinates. We demonstrate construction of the PM representation and its applications using several practical models.

**CR Categories and Subject Descriptors:** I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - surfaces and object representations.

**Additional Keywords:** mesh simplification, level of detail, shape interpolation, progressive transmission, geometry compression.

## 1 INTRODUCTION

Highly detailed geometric models are necessary to satisfy a growing expectation for realism in computer graphics. Within traditional modeling systems, detailed models are created by applying versatile modeling operations (such as extrusion, constructive solid geometry, and freeform deformations) to a vast array of geometric primitives. For efficient display, these models must usually be tessellated into polygonal approximations—meshes. Detailed meshes are also obtained by scanning physical objects using range scanning systems [5]. In either case, the resulting complex meshes are expensive to store, transmit, and render, thus motivating a number of practical problems:

- *Mesh simplification*: The meshes created by modeling and scanning systems are seldom optimized for rendering efficiency, and can frequently be replaced by nearly indistinguishable approximations with far fewer faces. At present, this process often requires significant user intervention. Mesh simplification tools can hope to automate this painstaking task, and permit the porting of a single model to platforms of varying performance.
- *Level-of-detail (LOD) approximation*: To further improve rendering performance, it is common to define several versions of a model at various levels of detail [3, 8]. A detailed mesh is used when the object is close to the viewer, and coarser approximations are substituted as the object recedes. Since instantaneous switching between LOD meshes may lead to perceptible “popping”, one would like to construct smooth visual transitions, *geomorphs*, between meshes at different resolutions.
- *Progressive transmission*: When a mesh is transmitted over a communication line, one would like to show progressively better approximations to the model as data is incrementally received. One approach is to transmit successive LOD approximations, but this requires additional transmission time.
- *Mesh compression*: The problem of minimizing the storage space for a model can be addressed in two orthogonal ways. One is to use mesh simplification to reduce the number of faces. The other is mesh compression: minimizing the space taken to store a particular mesh.
- *Selective refinement*: Each mesh in a LOD representation captures the model at a uniform (view-independent) level of detail. Sometimes it is desirable to adapt the level of refinement in selected regions. For instance, as a user flies over a terrain, the terrain mesh need be fully detailed only near the viewer, and only within the field of view.

In addressing these problems, this paper makes two major contributions. First, it introduces the *progressive mesh* (PM) representation. In PM form, an arbitrary mesh $\hat{M}$ is stored as a much coarser mesh $M^0$ together with a sequence of *n* detail records that indicate how to incrementally refine $M^0$ exactly back into the original mesh $\hat{M} = M^n$. Each of these records stores the information associated with a *vertex split*, an elementary mesh transformation that adds an additional vertex to the mesh. The PM representation of $\hat{M}$ thus defines a continuous sequence of meshes $M^0, M^1, \ldots, M^n$ of increasing accuracy, from which LOD approximations of any desired complexity can be efficiently retrieved. Moreover, geomorphs can be efficiently constructed between any two such meshes. In addition, we show that the PM representation naturally supports progressive transmission, offers a concise encoding of $\hat{M}$ itself, and permits selective refinement. In short, progressive meshes offer an efficient, lossless, continuous-resolution representation.

The other contribution of this paper is a new simplification procedure for constructing a PM representation from a given mesh $\hat{M}$. Unlike previous simplification methods, our procedure seeks to preserve not just the geometry of the mesh surface, but more importantly its overall appearance, as defined by the discrete and scalar attributes associated with its surface.

Email: hhoppe@microsoft.com
Web: http://www.research.microsoft.com/research/graphics/hoppe/

APPENDIX N

## 2 MESHES IN COMPUTER GRAPHICS

Models in computer graphics are often represented using triangle meshes.[1] Geometrically, a triangle mesh is a piecewise linear surface consisting of triangular faces pasted together along their edges. As described in [9], the mesh geometry can be denoted by a tuple $(K, V)$, where $K$ is a *simplicial complex* specifying the connectivity of the mesh simplices (the adjacency of the vertices, edges, and faces), and $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$ is the set of vertex positions defining the shape of the mesh in $\mathbf{R}^3$. More precisely (cf. [9]), we construct a parametric domain $|K| \subset \mathbf{R}^m$ by identifying each vertex of $K$ with a canonical basis vector of $\mathbf{R}^m$, and define the mesh as the image $\phi_V(|K|)$ where $\phi_V : \mathbf{R}^m \to \mathbf{R}^3$ is a linear map.

Often, surface appearance attributes other than geometry are also associated with the mesh. These attributes can be categorized into two types: *discrete* attributes and *scalar* attributes.

Discrete attributes are usually associated with faces of the mesh. A common discrete attribute, the *material identifier*, determines the shader function used in rendering a face of the mesh [18]. For instance, a trivial shader function may involve simple look-up within a specified texture map.

Many scalar attributes are often associated with a mesh, including diffuse color $(r, g, b)$, normal $(n_x, n_y, n_z)$, and texture coordinates $(u, v)$. More generally, these attributes specify the local parameters of shader functions defined on the mesh faces. In simple cases, these scalar attributes are associated with vertices of the mesh. However, to represent discontinuities in the scalar fields, and because adjacent faces may have different shading functions, it is common to associate scalar attributes not with vertices, but with corners of the mesh [1]. A *corner* is defined as a (vertex,face) tuple. Scalar attributes at a corner $(v, f)$ specify the shading parameters for face $f$ at vertex $v$. For example, along a *crease* (a curve on the surface across which the normal field is not continuous), each vertex has two distinct normals, one associated with the corners on each side of the crease.

We express a mesh as a tuple $M = (K, V, D, S)$ where $V$ specifies its geometry, $D$ is the set of discrete attributes $d_f$ associated with the faces $f = \{j, k, l\} \in K$, and $S$ is the set of scalar attributes $s_{(v,f)}$ associated with the corners $(v, f)$ of $K$.

The attributes $D$ and $S$ give rise to discontinuities in the visual appearance of the mesh. An edge $\{v_j, v_k\}$ of the mesh is said to be *sharp* if either (1) it is a boundary edge, or (2) its two adjacent faces $f_l$ and $f_r$ have different discrete attributes (i.e. $d_{f_l} \neq d_{f_r}$), or (3) its adjacent corners have different scalar attributes (i.e. $s_{(v_j,f_l)} \neq s_{(v_j,f_r)}$ or $s_{(v_k,f_l)} \neq s_{(v_k,f_r)}$). Together, the set of sharp edges define a set of *discontinuity curves* over the mesh (e.g. the yellow curves in Figure 12).

## 3 PROGRESSIVE MESH REPRESENTATION

### 3.1 Overview

Hoppe et al. [9] describe a method, *mesh optimization*, that can be used to approximate an initial mesh $\hat{M}$ by a much simpler one. Their optimization algorithm, reviewed in Section 4.1, traverses the space of possible meshes by successively applying a set of 3 mesh transformations: edge collapse, edge split, and edge swap.

We have discovered that in fact a single one of those transformations, *edge collapse*, is sufficient for effectively simplifying meshes. As shown in Figure 1, an edge collapse transformation $ecol(\{v_s, v_t\})$ unifies 2 adjacent vertices $v_s$ and $v_t$ into a single vertex $v_s$. The vertex $v_t$ and the two adjacent faces $\{v_s, v_t, v_l\}$ and $\{v_t, v_s, v_r\}$ vanish in the process. A position $\mathbf{v}_s$ is specified for the new unified vertex.

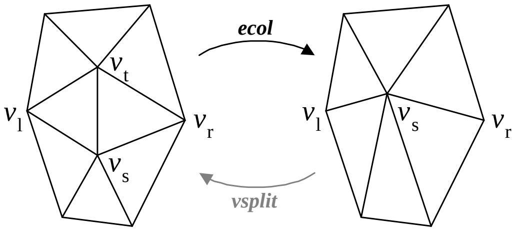


Figure 1: Illustration of the edge collapse transformation.

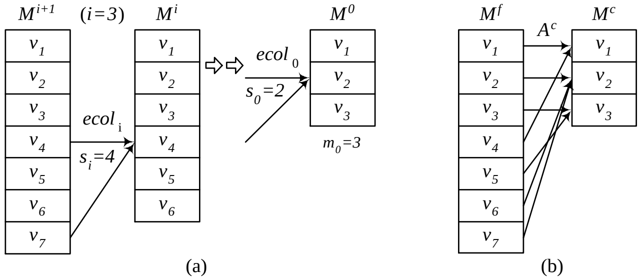


Figure 2: (a) Sequence of edge collapses; (b) Resulting vertex correspondence.

Thus, an initial mesh $\hat{M} = M^n$ can be simplified into a coarser mesh $M^0$ by applying a sequence of $n$ successive edge collapse transformations:

$$(\hat{M}=M^n) \xrightarrow{ecol_{n-1}} \ldots \xrightarrow{ecol_1} M^1 \xrightarrow{ecol_0} M^0 .$$

The particular sequence of edge collapse transformations must be chosen carefully, since it determines the quality of the approximating meshes $M^i, i < n$. A scheme for choosing these edge collapses is presented in Section 4.

Let $m_0$ be the number of vertices in $M^0$, and let us label the vertices of mesh $M^i$ as $V^i = \{v_1, \ldots, v_{m_0+i}\}$, so that edge $\{v_{s_i}, v_{m_0+i+1}\}$ is collapsed by $ecol_i$ as shown in Figure 2a. As vertices may have different positions in the different meshes, we denote the position of $v_j$ in $M^i$ as $\mathbf{v}_j^i$.

A key observation is that an edge collapse transformation is invertible. Let us call that inverse transformation a *vertex split*, shown as *vsplit* in Figure 1. A vertex split transformation $vsplit(s, l, r, t, A)$ adds near vertex $v_s$ a new vertex $v_t$ and two new faces $\{v_s, v_t, v_l\}$ and $\{v_t, v_s, v_r\}$. (If the edge $\{v_s, v_t\}$ is a boundary edge, we let $v_r = 0$ and only one face is added.) The transformation also updates the attributes of the mesh in the neighborhood of the transformation. This attribute information, denoted by $A$, includes the positions $\mathbf{v}_s$ and $\mathbf{v}_t$ of the two affected vertices, the discrete attributes $d_{\{v_s,v_t,v_l\}}$ and $d_{\{v_t,v_s,v_r\}}$ of the two new faces, and the scalar attributes of the affected corners ($s_{(v_s,\cdot)}$, $s_{(v_t,\cdot)}$, $s_{(v_l,\{v_s,v_t,v_l\})}$, and $s_{(v_r,\{v_t,v_s,v_r\})}$).

Because edge collapse transformations are invertible, we can therefore represent an arbitrary triangle mesh $\hat{M}$ as a simple mesh $M^0$ together with a sequence of $n$ *vsplit* records:

$$M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} \ldots \xrightarrow{vsplit_{n-1}} (M^n=\hat{M})$$

where each record is parametrized as $vsplit_i(s_i, l_i, r_i, A_i)$. We call $(M^0, \{vsplit_0, \ldots, vsplit_{n-1}\})$ a *progressive mesh* (PM) representation of $\hat{M}$.

As an example, the mesh $\hat{M}$ of Figure 5d (13,546 faces) was simplified down to the coarse mesh $M^0$ of Figure 5a (150 faces) using

[1] We assume in this paper that more general meshes, such as those containing $n$-sided faces and faces with holes, are first converted into triangle meshes by triangulation. The PM representation could be generalized to handle the more general meshes directly, at the expense of more complex data structures.

APPENDIX N

6,698 edge collapse transformations. Thus its PM representation consists of $M^0$ together with a sequence of $n = 6698$ *vsplit* records. From this PM representation, one can extract approximating meshes with any desired number of faces (actually, within ±1) by applying to $M^0$ a prefix of the *vsplit* sequence. For example, Figure 5 shows approximating meshes with 150, 500, and 1000 faces.

## 3.2 Geomorphs

A nice property of the vertex split transformation (and its inverse, edge collapse) is that a smooth visual transition (a *geomorph*) can be created between the two meshes $M^i$ and $M^{i+1}$ in $M^i \xrightarrow{vsplit_i} M^{i+1}$. For the moment let us assume that the meshes contain no attributes other than vertex positions. With this assumption the vertex split record is encoded as $vsplit_i(s_i, l_i, r_i, A_i = (\mathbf{v}^{i+1}_{s_i}, \mathbf{v}^{i+1}_{m_0+i+1}))$. We construct a geomorph $M^G(\alpha)$ with blend parameter $0 \le \alpha \le 1$ such that $M^G(0)$ looks like $M^i$ and $M^G(1)$ looks like $M^{i+1}$—in fact $M^G(1) = M^{i+1}$—by defining a mesh

$$M^G(\alpha) = (K^{i+1}, V^G(\alpha))$$

whose connectivity is that of $M^{i+1}$ and whose vertex positions linearly interpolate from $v_{s_i} \in M^i$ to the split vertices $v_{s_i}, v_{m_0+i+1} \in M^{i+1}$:

$$\mathbf{v}^G_j(\alpha) = \begin{cases} (\alpha)\mathbf{v}^{i+1}_j + (1-\alpha)\mathbf{v}^i_{s_i} & , j \in \{s_i, m_0+i+1\} \\ \mathbf{v}^{i+1}_j = \mathbf{v}^i_j & , j \notin \{s_i, m_0+i+1\} \end{cases}$$

Using such geomorphs, an application can smoothly transition from a mesh $M^i$ to meshes $M^{i+1}$ or $M^{i-1}$ without any visible “snapping” of the meshes.

Moreover, since individual *ecol* transformations can be transitioned smoothly, so can the composition of any sequence of them. Geomorphs can therefore be constructed between *any* two meshes of a PM representation. Indeed, given a finer mesh $M^f$ and a coarser mesh $M^c$, $0 \le c < f \le n$, there exists a natural correspondence between their vertices: each vertex of $M^f$ is related to a unique ancestor vertex of $M^c$ by a surjective map $A^c$ obtained by composing a sequence of *ecol* transformations (Figure 2b). More precisely, each vertex $v_j$ of $M^f$ corresponds with the vertex $v_{A^c(j)}$ in $M^c$ where

$$A^c(j) = \begin{cases} j & , j \le m_0 + c \\ A^c(s_{j-m_0-1}) & , j > m_0 + c \quad . \end{cases}$$

(In practice, this ancestor information $A^c$ is gathered in a forward fashion as the mesh is refined.) This correspondence allows us to define a geomorph $M^G(\alpha)$ such that $M^G(0)$ looks like $M^c$ and $M^G(1)$ equals $M^f$. We simply define $M^G(\alpha) = (K^f, V^G(\alpha))$ to have the connectivity of $M^f$ and the vertex positions

$$\mathbf{v}^G_j(\alpha) = (\alpha)\mathbf{v}^f_j + (1-\alpha)\mathbf{v}^c_{A^c(j)} \ .$$

So far we have outlined the construction of geomorphs between PM meshes containing only position attributes. We can in fact construct geomorphs for meshes containing both discrete and scalar attributes.

Discrete attributes by their nature cannot be smoothly interpolated. Fortunately, these discrete attributes are associated with faces of the mesh, and the “geometric” geomorphs described above smoothly introduce faces. In particular, observe that the faces of $M^c$ are a proper subset of the faces of $M^f$, and that those faces of $M^f$ missing from $M^c$ are invisible in $M^G(0)$ because they have been collapsed to degenerate (zero area) triangles. Other geomorphing schemes [10, 11, 17] define well-behaved (invertible) parametrizations between meshes at different levels of detail, but these do not permit the construction of geomorphs between meshes with different discrete attributes.

Scalar attributes defined on corners can be smoothly interpolated much like the vertex positions. There is a slight complication in that a corner $(v, f)$ in a mesh $M$ is not naturally associated with any “ancestor corner” in a coarser mesh $M^c$ if $f$ is not a face of $M^c$. We can still attempt to infer what attribute value $(v, f)$ would have in $M^c$ as follows. We examine the mesh $M^{i+1}$ in which $f$ is first introduced, locate a neighboring corner $(v, f')$ in $M^{i+1}$ whose attribute value is the same, and recursively backtrack from it to a corner in $M^c$. If there is no neighboring corner in $M^{i+1}$ with an identical attribute value, then the corner $(v, f)$ has no equivalent in $M^c$ and we therefore keep its attribute value constant through the geomorph.

The interpolating function on the scalar attributes need not be linear; for instance, normals are best interpolated over the unit sphere, and colors may be interpolated in a color space other than RGB.

Figure 6 demonstrates a geomorph between two meshes $M^{175}$ (500 faces) and $M^{425}$ (1000 faces) retrieved from the PM representation of the mesh in Figure 5d.

## 3.3 Progressive transmission

Progressive meshes are a natural representation for progressive transmission. The compact mesh $M^0$ is transmitted first (using a conventional uni-resolution format), followed by the stream of $vsplit_i$ records. The receiving process incrementally rebuilds $\hat{M}$ as the records arrive, and animates the changing mesh. The changes to the mesh can be geomorphed to avoid visual discontinuities. The original mesh $\hat{M}$ is recovered exactly after all $n$ records are received, since PM is a lossless representation.

The computation of the receiving process should be balanced between the reconstruction of $\hat{M}$ and interactive display. With a slow communication line, a simple strategy is to display the current mesh whenever the input buffer is found to be empty. With a fast communication line, we find that a good strategy is to display meshes whose complexities increase exponentially. (Similar issues arise in the display of images transmitted using progressive JPEG.)

## 3.4 Mesh compression

Even though the PM representation encodes both $\hat{M}$ and a continuous family of approximations, it is surprisingly space-efficient, for two reasons. First, the locations of the vertex split transformations can be encoded concisely. Instead of storing all three vertex indices $(s_i, l_i, r_i)$ of $vsplit_i$, one need only store $s_i$ and approximately 5 bits to select the remaining two vertices among those adjacent to $v_{s_i}$.[2] Second, because a vertex split has local effect, one can expect significant coherence in mesh attributes through each transformation. For instance, when vertex $v^i_{s_i}$ is split into $v^{i+1}_{s_i}$ and $v^{i+1}_{m_0+i+1}$, we can predict the positions $\mathbf{v}^{i+1}_{s_i}$ and $\mathbf{v}^{i+1}_{m_0+i+1}$ from $\mathbf{v}^i_{s_i}$, and use delta-encoding to reduce storage. Scalar attributes of corners in $M^{i+1}$ can similarly be predicted from those in $M^i$. Finally, the material identifiers $d_{\{v_s, v_t, v_l\}}$ and $d_{\{v_t, v_s, v_r\}}$ of the new faces in mesh $M^{i+1}$ can often be predicted from those of adjacent faces in $M^i$ using only a few control bits.

As a result, the size of a carefully designed PM representation should be competitive with that obtained from methods for compressing uni-resolution meshes. Our current prototype implementation was not designed with this goal in mind. However, we analyze the compression of the connectivity $K$, and report results on the compression of the geometry $V$. In the following analysis, we assume for simplicity that $m_0 = 0$ since typically $m_0 \ll n$.

A common representation for the mesh connectivity $K$ is to list the three vertex indices for each face. Since the number of vertices is $n$ and the number of faces approximately $2n$, such a list requires $6\lceil \log_2(n) \rceil n$ bits of storage. Using a buffer of 2 vertices, *generalized triangle strip* representations reduce this number to about

[2] On average, $v_{s_i}$ has 6 neighbors, and the number of permutations $P^6_2 = 30$ can be encoded in $\lceil \log_2(P^6_2) \rceil = 5$ bits.

$(\lceil\log_2(n)\rceil + 2k)n$ bits, where vertices are back-referenced once on average and $k \simeq 2$ bits capture the vertex replacement codes [6]. By increasing the vertex buffer size to 16, Deering's *generalized triangle mesh* representation [6] further reduces storage to about $(\frac{1}{8}\lceil\log_2(n)\rceil + 8)n$ bits. Turan [16] shows that planar graphs (and hence the connectivity of closed genus 0 meshes) can be encoded in $12n$ bits. Recent work by Taubin and Rossignac [15] addresses more general meshes. With the PM representation, each $vsplit_i$ requires specification of $s_i$ and its two neighbors, for a total storage of about $(\lceil\log_2(n)\rceil + 5)n$ bits. Although not as concise as [6, 15], this is comparable to generalized triangle strips.

A traditional representation of the mesh geometry $V$ requires storage of $3n$ coordinates, or $96n$ bits with IEEE single-precision floating point. Like Deering [6], we assume that these coordinates can be quantized to 16-bit fixed precision values without significant loss of visual quality, thus reducing storage to $48n$ bits. Deering is able to further compress this storage by delta-encoding the quantized coordinates and Huffman compressing the variable-length deltas. For 16-bit quantization, he reports storage of $35.8n$ bits, which includes both the deltas and the Huffman codes. Using a similar approach with the PM representation, we encode $V$ in $31n$ to $50n$ bits as shown in Table 1. To obtain these results, we exploit a property of our optimization algorithm (Section 4.3): when considering the collapse of an edge $\{v_s, v_t\}$, it considers three starting points for the resulting vertex position $\mathbf{v}_n$: $\{\mathbf{v}_s, \mathbf{v}_t, \frac{\mathbf{v}_s+\mathbf{v}_t}{2}\}$. Depending on the starting point chosen, we delta-encode either $\{\mathbf{v}_s - \mathbf{v}_n, \mathbf{v}_t - \mathbf{v}_n\}$ or $\{\frac{\mathbf{v}_s+\mathbf{v}_t}{2} - \mathbf{v}_n, \frac{\mathbf{v}_t-\mathbf{v}_s}{2}\}$, and use separate Huffman tables for all four quantities.

To further improve compression, we could alter the construction algorithm to forego optimization and let $\mathbf{v}_n \in \{\mathbf{v}_s, \mathbf{v}_t, \frac{\mathbf{v}_s+\mathbf{v}_t}{2}\}$. This would degrade the accuracy of the approximating meshes somewhat, but allows encoding of $V$ in $30n$ to $37n$ bits in our examples. Arithmetic coding [19] of delta lengths does not improve results significantly, reflecting the fact that the Huffman trees are well balanced. Further compression improvements may be achievable by adapting both the quantization level and the delta length models as functions of the *vsplit* record index $i$, since the magnitude of successive changes tends to decrease.

### 3.5 Selective refinement

The PM representation also supports selective refinement, whereby detail is added to the model only in desired areas. Let the application supply a callback function REFINE($v$) that returns a Boolean value indicating whether the neighborhood of the mesh about $v$ should be further refined. An initial mesh $M^c$ is selectively refined by iterating through the list $\{vsplit_c, \ldots, vsplit_{n-1}\}$ as before, but only performing $vsplit_i(s_i, l_i, r_i, A_i)$ if

(1) all three vertices $\{v_{s_i}, v_{l_i}, v_{r_i}\}$ are present in the mesh, and

(2) REFINE($v_{s_i}$) evaluates to TRUE.

(A vertex $v_j$ is absent from the mesh if the prior vertex split that would have introduced it, $vsplit_{j-m_0-1}$, was not performed due to the above conditions.)

As an example, to obtain selective refinement of the model within a view frustum, REFINE($v$) is defined to be TRUE if either $v$ or any of its neighbors lies within the frustum. As seen in Figure 7a, condition (1) described above is suboptimal. The problem is that a vertex $v_{s_i}$ within the frustum may fail to be split because its expected neighbor $v_{l_i}$ lies just outside the frustum and was not previously created. The problem is remedied by using a less stringent version of condition (1). Let us define the *closest living ancestor* of a vertex $v_j$ to be the vertex with index

$$A'(j) = \begin{cases} j & \text{, if } v_j \text{ exists in the mesh} \\ A'(s_{j-m_0-1}) & \text{, otherwise} \end{cases}$$

The new condition becomes:

(1') $v_{s_i}$ is present in the mesh (i.e. $A'(s_i) = s_i$) and the vertices $v_{A'(l_i)}$ and $v_{A'(r_i)}$ are both adjacent to $v_{s_i}$.

As when constructing the geomorphs, the ancestor information $A'$ is carried efficiently as the *vsplit* records are parsed. If conditions (1') and (2) are satisfied, $vsplit(s_i, A'(l_i), A'(r_i), A_i)$ is applied to the mesh. A mesh selectively refined with this new strategy is shown in Figure 7b. This same strategy was also used for Figure 10. Note that it is still possible to create geomorphs between $M^c$ and selectively refined meshes thus created.

An interesting application of selective refinement is the transmission of view-dependent models over low-bandwidth communication lines. As the receiver's view changes over time, the sending process need only transmit those *vsplit* records for which REFINE evaluates to TRUE, and of those only the ones not previously transmitted.

## 4 PROGRESSIVE MESH CONSTRUCTION

The PM representation of an arbitrary mesh $\hat{M}$ requires a sequence of edge collapses transforming $\hat{M} = M^n$ into a base mesh $M^0$. The quality of the intermediate approximations $M^i, i < n$ depends largely on the algorithm for selecting which edges to collapse and what attributes to assign to the affected neighborhoods, for instance the positions $\mathbf{v}^i_{s_i}$.

There are many possible PM construction algorithms with varying trade-offs of speed and accuracy. At one extreme, a crude and fast scheme for selecting edge collapses is to choose them completely at random. (Some local conditions must be satisfied for an edge collapse to be legal, i.e. manifold preserving [9].) More sophisticated schemes can use heuristics to improve the edge selection strategy, for example the "distance to plane" metric of Schroeder et al. [14]. At the other extreme, one can attempt to find approximating meshes that are optimal with respect to some appearance metric, for instance the $E_{dist}$ geometric metric of Hoppe et al. [9].

Since PM construction is a preprocess that can be performed offline, we chose to design a simplification procedure that invests some time in the selection of edge collapses. Our procedure is similar to the mesh optimization method introduced by Hoppe et al. [9], which is outlined briefly in Section 4.1. Section 4.2 presents an overview of our procedure, and Sections 4.3–4.6 present the details of our optimization scheme for preserving both the shape of the mesh and the scalar and discrete attributes which define its appearance.

### 4.1 Background: mesh optimization

The goal of mesh optimization [9] is to find a mesh $M = (K, V)$ that both accurately fits a set $X$ of points $\mathbf{x}_i \in \mathbf{R}^3$ and has a small number of vertices. This problem is cast as minimization of an energy function

$$E(M) = E_{dist}(M) + E_{rep}(M) + E_{spring}(M) \; .$$

The first two terms correspond to the two goals of accuracy and conciseness: the *distance energy* term

$$E_{dist}(M) = \sum_i d^2(\mathbf{x}_i, \phi_V(|K|))$$

measures the total squared distance of the points from the mesh, and the *representation energy* term $E_{rep}(M) = c_{rep}m$ penalizes the number $m$ of vertices in $M$. The third term, the *spring energy* $E_{spring}(M)$ is introduced to regularize the optimization problem. It corresponds to placing on each edge of the mesh a spring of rest length zero and tension $\kappa$:

$$E_{spring}(M) = \sum_{\{j,k\} \in K} \kappa \|\mathbf{v}_j - \mathbf{v}_k\|^2 \; .$$

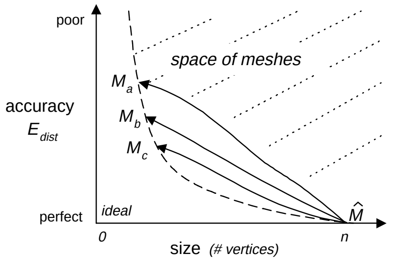


Figure 3: Illustration of the paths taken by mesh optimization using three different settings of $c_{rep}$.

The energy function $E(M)$ is minimized using a nested optimization method:

- *Outer loop*: The algorithm optimizes over $K$, the connectivity of the mesh, by randomly attempting a set of three possible mesh transformations: edge collapse, edge split, and edge swap. This set of transformations is complete, in the sense that any simplicial complex $K$ of the same topological type as $\hat{K}$ can be reached through a sequence of these transformations. For each candidate mesh transformation, $K \to K'$, the continuous optimization described below computes $E_{K'}$, the minimum of $E$ subject to the new connectivity $K'$. If $\Delta E = E_{K'} - E_K$ is found to be negative, the mesh transformation is applied (akin to a zero-temperature simulated annealing method).
- *Inner loop*: For each candidate mesh transformation, the algorithm computes $E_{K'} = \min_V E_{dist}(V) + E_{spring}(V)$ by optimizing over the vertex positions $V$. For the sake of efficiency, the algorithm in fact optimizes only one vertex position $\mathbf{v}_s$, and considers only the subset of points $X$ that project onto the neighborhood affected by $K \to K'$. To avoid surface self-intersections, the edge collapse is disallowed if the maximum dihedral angle of edges in the resulting neighborhood exceeds some threshold.

Hoppe et al. [9] find that the regularizing spring energy term $E_{spring}(M)$ is most important in the early stages of the optimization, and achieve best results by repeatedly invoking the nested optimization method described above with a schedule of decreasing spring constants $\kappa$.

Mesh optimization is demonstrated to be an effective tool for mesh simplification. Given an initial mesh $\hat{M}$ to approximate, a dense set of points $X$ is sampled both at the vertices of $\hat{M}$ and randomly over its faces. The optimization algorithm is then invoked with $\hat{M}$ as the starting mesh. Varying the setting of the representation constant $c_{rep}$ results in optimized meshes with different trade-offs of accuracy and size. The paths taken by these optimizations are shown illustratively in Figure 3.

## 4.2 Overview of the simplification algorithm

As in mesh optimization [9], we also define an explicit energy metric $E(M)$ to measure the accuracy of simplified meshes $M = (K, V, D, S)$ with respect to the original $\hat{M}$, and we also modify the mesh $M$ starting from $\hat{M}$ while minimizing $E(M)$.

Our energy metric has the following form:

$$E(M) = E_{dist}(M) + E_{spring}(M) + E_{scalar}(M) + E_{disc}(M) \ .$$

The first two terms, $E_{dist}(M)$ and $E_{spring}(M)$ are identical to those in [9]. The next two terms of $E(M)$ are added to preserve attributes associated with $M$: $E_{scalar}(M)$ measures the accuracy of its scalar attributes (Section 4.4), and $E_{disc}(M)$ measures the geometric accuracy of its discontinuity curves (Section 4.5). (To achieve scale invariance of the terms, the mesh is uniformly scaled to fit in a unit cube.)

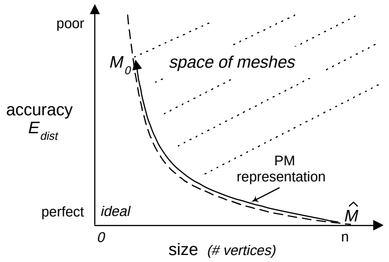


Figure 4: Illustration of the path taken by the new mesh simplification procedure in a graph plotting accuracy vs. mesh size.

Our scheme for optimizing over the connectivity $K$ of the mesh is rather different from [9]. We have discovered that a mesh can be effectively simplified using edge collapse transformations alone. The edge swap and edge split transformations, useful in the context of surface reconstruction (which motivated [9]), are not essential for simplification. Although in principle our simplification algorithm can no longer traverse the entire space of meshes considered by mesh optimization, we find that the meshes generated by our algorithm are just as good. In fact, because of the priority queue approach described below, our meshes are usually better. Moreover, considering only edge collapses simplifies the implementation, improves performance, and most importantly, gives rise to the PM representation (Section 3).

Rather than randomly attempting mesh transformations as in [9], we place all (legal) candidate edge collapse transformations into a priority queue, where the priority of each transformation is its estimated energy cost $\Delta E$. In each iteration, we perform the transformation at the front of the priority queue (with lowest $\Delta E$), and recompute the priorities of edges in the neighborhood of this transformation. As a consequence, we eliminate the need for the awkward parameter $c_{rep}$ as well as the energy term $E_{rep}(M)$. Instead, we can explicitly specify the number of faces desired in an optimized mesh. Also, a single run of the optimization can generate several such meshes. Indeed, it generates a continuous-resolution family of meshes, namely the PM representation of $\hat{M}$ (Figure 4).

For each edge collapse $K \to K'$, we compute its cost $\Delta E = E_{K'} - E_K$ by solving a continuous optimization

$$E_{K'} = \min_{V,S} E_{dist}(V) + E_{spring}(V) + E_{scalar}(V, S) + E_{disc}(V)$$

over both the vertex positions $V$ and the scalar attributes $S$ of the mesh with connectivity $K'$. This minimization is discussed in the next three sections.

## 4.3 Preserving surface geometry ($E_{dist}$+$E_{spring}$)

As in [9], we "record" the geometry of the original mesh $\hat{M}$ by sampling from it a set of points $X$. At a minimum, we sample a point at each vertex of $\hat{M}$. If requested by the user, additional points are sampled randomly over the surface of $\hat{M}$. The energy terms $E_{dist}(M)$ and $E_{spring}(M)$ are defined as in Section 4.1.

For a mesh of fixed connectivity, our method for optimizing the vertex positions to minimize $E_{dist}(V)+E_{spring}(V)$ closely follows that of [9]. Evaluating $E_{dist}(V)$ involves computing the distance of each point $\mathbf{x}_i$ to the mesh. Each of these distances is itself a minimization problem

$$d^2(\mathbf{x}_i, \phi_V(|K|)) = \min_{\mathbf{b}_i \in |K|} \|\mathbf{x}_i - \phi_V(\mathbf{b}_i)\|^2 \qquad (1)$$

where the unknown $\mathbf{b}_i$ is the parametrization of the projection of $\mathbf{x}_i$ on the mesh. The nonlinear minimization of $E_{dist}(V) + E_{spring}(V)$ is performed using an iterative procedure alternating between two steps:

1. For fixed vertex positions $V$, compute the optimal parametrizations $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_{|X|}\}$ by projecting the points $X$ onto the mesh.
2. For fixed parametrizations $B$, compute the optimal vertex positions $V$ by solving a sparse linear least-squares problem.

As in [9], when considering $ecol(\{v_s, v_t\})$, we optimize only one vertex position, $\mathbf{v}_s^i$. We perform three different optimizations with different starting points, $\mathbf{v}_s^i = (1-\alpha)\mathbf{v}_s^{i+1} + (\alpha)\mathbf{v}_t^{i+1}$ for $\alpha = \{0, \frac{1}{2}, 1\}$, and accept the best one.

Instead of defining a global spring constant $\kappa$ for $E_{spring}$ as in [9], we adapt $\kappa$ each time an edge collapse transformation is considered. Intuitively, the spring energy is most important when few points project onto a neighborhood of faces, since in this case finding the vertex positions minimizing $E_{dist}(V)$ may be an under-constrained problem. Thus, for each edge collapse transformation considered, we set $\kappa$ as a function of the ratio of the number of points to the number of faces in the neighborhood.[3] With this adaptive scheme, the influence of $E_{spring}(M)$ decreases gradually and adaptively as the mesh is simplified, and we no longer require the expensive schedule of decreasing spring constants.

## 4.4 Preserving scalar attributes ($E_{scalar}$)

As described in Section 2, we represent piecewise continuous scalar fields by defining scalar attributes $S$ at the mesh corners. We now present our scheme for preserving these scalar fields through the simplification process. For exposition, we find it easier to first present the case of continuous scalar fields, in which the corner attributes at a vertex are identical. The generalization to piecewise continuous fields is discussed shortly.

**Optimizing scalar attributes at vertices** Let the original mesh $\hat{M}$ have at each vertex $v_j$ not only a position $\mathbf{v}_j \in \mathbf{R}^3$ but also a scalar attribute $\underline{\mathbf{v}}_j \in \mathbf{R}^d$. To capture scalar attributes, we sample at each point $\mathbf{x}_i \in X$ the attribute value $\underline{\mathbf{x}}_i \in \mathbf{R}^d$. We would then like to generalize the distance metric $E_{dist}$ to also measure the deviation of the sampled attribute values $\underline{X}$ from those of $M$.

One natural way to achieve this is to redefine the distance metric to measure distance in $\mathbf{R}^{3+d}$:

$$d^2((\mathbf{x}_i\ \ \underline{\mathbf{x}}_i), M(K, V, \underline{V})) = \min_{\mathbf{b}_i \in |K|} \|(\mathbf{x}_i\ \ \underline{\mathbf{x}}_i) - (\phi_V(\mathbf{b}_i)\ \ \phi_{\underline{V}}(\mathbf{b}_i))\|^2 \ .$$

This new distance functional could be minimized using the iterative approach of Section 4.3. However, it would be expensive since finding the optimal parametrization $\mathbf{b}_i$ of each point $\mathbf{x}_i$ would require projection in $\mathbf{R}^{3+d}$, and would be non-intuitive since these parametrizations would not be geometrically based.

Instead we opted to determine the parametrizations $\mathbf{b}_i$ using only geometry with equation (1), and to introduce a separate energy term $E_{scalar}$ to measure attribute deviation based on these parametrizations:

$$E_{scalar}(\underline{V}) = (c_{scalar})^2 \sum_i \|\underline{\mathbf{x}}_i - \phi_{\underline{V}}(\mathbf{b}_i)\|^2$$

where the constant $c_{scalar}$ assigns a relative weight between the scalar attribute errors ($E_{scalar}$) and the geometric errors ($E_{dist}$).

Thus, to minimize $E(V, \underline{V}) = E_{dist}(V) + E_{spring}(V) + E_{scalar}(\underline{V})$, our algorithm first finds the vertex position $\mathbf{v}_s$ minimizing $E_{dist}(V) + E_{spring}(V)$ by alternately projecting the points onto the mesh (obtaining the parametrizations $\mathbf{b}_i$) and solving a linear least-squares problem (Section 4.1). Then, using those same parametrizations $\mathbf{b}_i$, it finds the vertex attribute $\underline{\mathbf{v}}_s$ minimizing $E_{scalar}$ by solving a single linear least-squares problem. Hence introducing $E_{scalar}$ into the optimization causes negligible performance overhead.

Since $\Delta E_{scalar}$ contributes to the estimated cost $\Delta E$ of an edge collapse, we obtain simplified meshes whose faces naturally adapt to the attribute fields, as shown in Figures 8 and 11.

**Optimizing scalar attributes at corners** Our scheme for optimizing the scalar corner attributes $S$ is a straightforward generalization of the scheme just described. Instead of solving for a single unknown attribute value $\underline{\mathbf{v}}_s$, the algorithm partitions the corners around $v_s$ into continuous sets (based on equivalence of corner attributes) and for each continuous set solves independently for its optimal attribute value.

**Range constraints** Some scalar attributes have constrained ranges. For instance, the components $(r, g, b)$ of color are typically constrained to lie between 0 and 1. Least-squares optimization may yield color values outside this range. In these cases we clip the optimized values to the given range. For least-squares minimization of a Euclidean norm at a single vertex, this is in fact optimal.

**Normals** Surface normals $(n_x, n_y, n_z)$ are typically constrained to have unit length, and thus their domain is non-Cartesian. Optimizing over normals would therefore require minimization of a nonlinear functional with nonlinear constraints. We decided to instead simply carry the normals through the simplification process. Specifically, we compute the new normals at vertex $v_{s_i}^i$ by interpolating between the normals at vertices $v_{s_i}^{i+1}$ and $v_{m_0+i+1}^{i+1}$ using the $\alpha$ value that resulted in the best vertex position $\mathbf{v}_{s_i}^i$ in Section 4.3. Fortunately, the absolute directions of normals are less visually important than their discontinuities, and we have a scheme for preserving such discontinuities, as described in the next section.

## 4.5 Preserving discontinuity curves ($E_{disc}$)

Appearance attributes give rise to a set of discontinuity curves on the mesh, both from differences between discrete face attributes (e.g. material boundaries), and from differences between scalar corner attributes (e.g. creases and shadow boundaries). As these discontinuity curves form noticeable features, we have found it useful to preserve them both topologically and geometrically.

We can detect when a candidate edge collapse would modify the topology of the discontinuity curves using some simple tests on the presence of sharp edges in its neighborhood. Let $sharp(v_j, v_k)$ denote that an edge $\{v_j, v_k\}$ is sharp, and let $\#sharp(v_j)$ be the number of sharp edges adjacent to a vertex $v_j$. Then, referring to Figure 1, $ecol(\{v_s, v_t\})$ modifies the topology of discontinuity curves if either:

- $sharp(v_s, v_l)$ and $sharp(v_t, v_l)$, or
- $sharp(v_s, v_r)$ and $sharp(v_t, v_r)$, or
- $\#sharp(v_s) \geq 1$ and $\#sharp(v_t) \geq 1$ and not $sharp(v_s, v_t)$, or
- $\#sharp(v_s) \geq 3$ and $\#sharp(v_t) \geq 3$ and $sharp(v_s, v_t)$, or
- $sharp(v_s, v_t)$ and $\#sharp(v_s) = 1$ and $\#sharp(v_t) \neq 2$, or
- $sharp(v_s, v_t)$ and $\#sharp(v_t) = 1$ and $\#sharp(v_s) \neq 2$.

If an edge collapse would modify the topology of discontinuity curves, we either disallow it, or penalize it as discussed in Section 4.6.

To preserve the geometry of the discontinuity curves, we sample an additional set of points $X_{disc}$ from the sharp edges of $\hat{M}$, and define an additional energy term $E_{disc}$ equal to the total squared distances of each of these points to the discontinuity curve from which it was sampled. Thus $E_{disc}$ is defined just like $E_{dist}$, except that the points $X_{disc}$ are constrained to project onto a set of sharp edges in the mesh. In effect, we are solving a curve fitting problem embedded within the surface fitting problem. Since all boundaries of the surface are defined to be discontinuity curves, our procedure preserves bound-

[3] The neighborhood of an edge collapse transformation is the set of faces shown in Figure 1. Using C notation, we set $\kappa = r < 4 ? 10^{-2} : r < 8 ? 10^{-4} : 10^{-8}$ where $r$ is the ratio of the number of points to faces in the neighborhood.

ary geometry more accurately than [9]. Figure 9 demonstrates the importance of using the $E_{disc}$ energy term in preserving the material boundaries of a mesh with discrete face attributes.

### 4.6 Permitting changes to topology of discontinuity curves

Some meshes contain numerous discontinuity curves, and these curves may delimit features that are too small to be visible when viewed from a distance. In such cases we have found that strictly preserving the topology of the discontinuity curves unnecessarily curtails simplification. We have therefore adopted a hybrid strategy, which is to permit changes to the topology of the discontinuity curves, but to penalize such changes. When a candidate edge collapse $ecol(\{v_s, v_t\})$ changes the topology of the discontinuity curves, we add to its cost $\Delta E$ the value $|X_{disc,\{v_s,v_t\}}| \cdot \|\mathbf{v}_s - \mathbf{v}_t\|^2$ where $|X_{disc,\{v_s,v_t\}}|$ is the number of points of $X_{disc}$ projecting onto $\{v_s, v_t\}$. That simple strategy, although ad hoc, has proven very effective. For example, it allows the dark gray window frames of the "cessna" (visible in Figure 9) to vanish in the simplified meshes (Figures 5a–c).

Table 1: Parameter settings and quantitative results.

| Object | Original $\hat{M}$ | | Base $M^0$ | | User param. | | $\|X_{disc}\|$ | $V$ | Time |
|---|---|---|---|---|---|---|---|---|---|
| | $m_0 + n$ | #faces | $m_0$ | #faces | $\|X\|-(m_0+n)$ | $c_{color}$ | | $\frac{\text{bits}}{n}$ | mins |
| cessna | 6,795 | 13,546 | 97 | 150 | 100,000 | - | 46,811 | 46 | 23 |
| terrain | 33,847 | 66,960 | 3 | 1 | 0 | - | 3,796 | 46 | 16 |
| mandrill | 40,000 | 79,202 | 3 | 1 | 0 | 0.1 | 4,776 | 31 | 19 |
| radiosity | 78,923 | 150,983 | 1,192 | 1,191 | 200,000 | 0.01 | 74,316 | 37 | 106 |
| fandisk | 6,475 | 12,946 | 27 | 50 | 10,000 | - | 5,924 | 50 | 19 |

## 5 RESULTS

Table 1 shows, for the meshes in Figures 5–12, the number of vertices and faces in both $\hat{M}$ and $M^0$. In general, we let the simplification proceed until no more legal edge collapse transformations are possible. For the "cessna", we stopped at 150 faces to obtain a visually aesthetic base mesh. As indicated, the only user-specified parameters are the number of additional points (besides the $m_0 + n$ vertices of $\hat{M}$) sampled to increase fidelity, and the $c_{scalar}$ constants relating the scalar attribute accuracies to the geometric accuracy. The only scalar attribute we optimized is color, and its $c_{scalar}$ constant is denoted as $c_{color}$. The number $|X_{disc}|$ of points sampled from sharp edges is set automatically so that the densities of $X$ and $X_{disc}$ are proportional.[4] Execution times were obtained on a 150MHz Indigo2 with 128MB of memory.

Construction of the PM representation proceeds in three steps. First, as the simplification algorithm applies a sequence $ecol_{n-1} \dots ecol_0$ of transformations to the original mesh, it writes to a file the sequence $vsplit_{n-1} \dots vsplit_0$ of corresponding inverse transformations. When finished, the algorithm also writes the resulting base mesh $M^0$. Next, we reverse the order of the *vsplit* records. Finally, we renumber the vertices and faces of $(M^0, vsplit_0 \dots vsplit_{n-1})$ to match the indexing scheme of Section 3.1 in order to obtain a concise format.

Figure 6 shows a single geomorph between two meshes $M^{175}$ and $M^{425}$ of a PM representation. For interactive LOD, it is useful to select a sequence of meshes from the PM representation, and to construct successive geomorphs between them. We have obtained good results by selecting meshes whose complexities grow exponentially, as in Figure 5. During execution, an application can adjust the granularity of these geomorphs by sampling additional meshes from the PM representation, or freeing some up.

In Figure 10, we selectively refined a terrain (grid of 181 × 187 vertices) using a new REFINE($v$) function that keeps more detail near silhouette edges and near the viewer. More precisely, for the faces $F_v$ adjacent to $v$, we compute the signed projected screen areas $\{a_f : f \in F_v\}$. We let REFINE($v$) return TRUE if

(1) any face $f \in F_v$ lies within the view frustum, and either

(2a) the signs of $a_f$ are not all equal (i.e. $v$ lies near a silhouette edge) or

(2b) $\sum_{f \in F_v} a_f > thresh$ for a screen area threshold $thresh = 0.16^2$ (where total screen area is 1).

## 6 RELATED WORK

**Mesh simplification methods** A number of schemes construct a discrete sequence of approximating meshes by repeated application of a simplification procedure. Turk [17] sprinkles a set of points on a mesh, with density weighted by estimates of local curvature, and then retriangulates based on those points. Both Schroeder et al. [14] and Cohen et al. [4] iteratively remove vertices from the mesh and retriangulate the resulting holes. Cohen et al. are able to bound the maximum error of the approximation by restricting it to lie between two offset surfaces. Hoppe et al. [9] find accurate approximations through a general mesh optimization process (Section 4.1). Rossignac and Borrel [12] merge vertices of a model using spatial binning. A unique aspect of their approach is that the topological type of the model may change in the process. Their method is extremely fast, but since it ignores geometric qualities like curvature, the resulting approximations can be far from optimal. Some of the above methods [12, 17] permit the construction of geomorphs between successive simplified meshes.

**Multiresolution analysis (MRA)** Lounsbery et al. [10, 11] generalize the concept of multiresolution analysis to surfaces of arbitrary topological type. Eck et al. [7] describe how MRA can be applied to the approximation of an arbitrary mesh. Certain et al. [2] extend MRA to capture color, and present a multiresolution Web viewer supporting progressive transmission. MRA has many similarities with the PM scheme, since both store a simple base mesh together with a stream of detail records, and both produce a continuous-resolution representation. It is therefore worthwhile to highlight their differences:

**Advantages of PM over MRA:**

- MRA requires that the detail terms (wavelets) lie on a domain with subdivision connectivity, and as a result an arbitrary initial mesh $\hat{M}$ can only be recovered to within an $\epsilon$ tolerance. In contrast, the PM representation is lossless since $M^n = \hat{M}$.
- Because the approximating meshes $M^i, i < n$ in a PM may have arbitrary connectivity, they can be much better approximations than their MRA counterparts (Figure 12).
- The MRA representation cannot deal effectively with surface creases, unless those creases lie parametrically along edges of the base mesh (Figure 12). PM's can introduce surface creases anywhere and at any level of detail.
- PM's capture continuous, piecewise-continuous, and discrete appearance attributes. MRA schemes can represent discontinuous functions using a piecewise-constant basis (such as the Haar basis as used in [2, 13]), but the resulting approximations have too many discontinuities since none of the basis functions meet continuously. Also, it is not clear how MRA could be extended to capture discrete attributes.

[4] We set $|X_{disc}|$ such that $|X_{disc}|/perim = c(|X|/area)^{\frac{1}{2}}$ where $perim$ is the total length of all sharp edges in $\hat{M}$, $area$ is total area of all faces, and the constant $c = 4.0$ is chosen empirically.

**Advantages of MRA over PM:**

- The MRA framework provides a parametrization between meshes at various levels of detail, thus making possible multiresolution surface editing. PM's also offer such a parametrization, but it is not smooth, and therefore multiresolution editing may be non-intuitive.
- Eck et al. [7] construct MRA approximations with guaranteed maximum error bounds to $\hat{M}$. Our PM construction algorithm does not provide such bounds, although one could envision using simplification envelopes [4] to achieve this.
- MRA allows geometry and color to be compressed independently [2].

**Other related work** There has been relatively little work in simplifying arbitrary surfaces with functions defined over them. One special instance is image compression, since an image can be thought of as a set of scalar color functions defined on a quadrilateral surface. Another instance is the framework of Schröder and Sweldens [13] for simplifying functions defined over the sphere. The PM representation, like the MRA representation, is a generalization in that it supports surfaces of arbitrary topological type.

## 7 SUMMARY AND FUTURE WORK

We have introduced the progressive mesh representation and shown that it naturally supports geomorphs, progressive transmission, compression, and selective refinement. In addition, as a PM construction method, we have presented a new mesh simplification procedure designed to preserve not just the geometry of the original mesh, but also its overall appearance.

There are a number of avenues for future work, including:

- Development of an explicit metric and optimization scheme for preserving surface normals.
- Experimentation with PM editing.
- Representation of articulated or animated models.
- Application of the work to progressive subdivision surfaces.
- Progressive representation of more general simplicial complexes (not just 2-d manifolds).
- Addition of spatial data structures to permit efficient selective refinement.

We envision many practical applications for the PM representation, including streaming of 3D geometry over the Web, efficient storage formats, and continuous LOD in computer graphics applications. The representation may also have applications in finite element methods, as it can be used to generate coarse meshes for multigrid analysis.

## ACKNOWLEDGMENTS

I wish to thank Viewpoint Datalabs for providing the "cessna" mesh, Pratt & Whitney for the gas turbine engine component ("fandisk"), Softimage for the "terrain" mesh, and especially Steve Drucker for creating several radiosity models using Lightscape. Thanks also to Michael Cohen, Steven "Shlomo" Gortler, and Jim Kajiya for their enthusiastic support, and to Rick Szeliski for helpful comments on the paper. Mark Kenworthy first coined the term "geomorph" in '92 to distinguish them from image morphs.

## REFERENCES

[1] APPLE COMPUTER, INC. *3D graphics programming with QuickDraw 3D.* Addison Wesley, 1995.

[2] CERTAIN, A., POPOVIC, J., DUCHAMP, T., SALESIN, D., STUETZLE, W., AND DEROSE, T. Interactive multiresolution surface viewing. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996).

[3] CLARK, J. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19*, 10 (Oct. 1976), 547–554.

[4] COHEN, J., VARSHNEY, A., MANOCHA, D., TURK, G., WEBER, H., AGARWAL, P., BROOKS, F., AND WRIGHT, W. Simplification envelopes. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996).

[5] CURLESS, B., AND LEVOY, M. A volumetric method for building complex models from range images. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996).

[6] DEERING, M. Geometry compression. *Computer Graphics (SIGGRAPH '95 Proceedings)* (1995), 13–20.

[7] ECK, M., DEROSE, T., DUCHAMP, T., HOPPE, H., LOUNSBERY, M., AND STUETZLE, W. Multiresolution analysis of arbitrary meshes. *Computer Graphics (SIGGRAPH '95 Proceedings)* (1995), 173–182.

[8] FUNKHOUSER, T., AND SÉQUIN, C. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (SIGGRAPH '93 Proceedings)* (1995), 247–254.

[9] HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., AND STUETZLE, W. Mesh optimization. *Computer Graphics (SIGGRAPH '93 Proceedings)* (1993), 19–26.

[10] LOUNSBERY, J. M. *Multiresolution analysis for surfaces of arbitrary topological type.* PhD thesis, Dept. of Computer Science and Engineering, U. of Washington, 1994.

[11] LOUNSBERY, M., DEROSE, T., AND WARREN, J. Multiresolution analysis for surfaces of arbitrary topological type. Submitted for publication. (TR 93-10-05b, Dept. of Computer Science and Engineering, U. of Washington, January 1994.).

[12] ROSSIGNAC, J., AND BORREL, P. Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics*, B. Falcidieno and T. L. Kunii, Eds. Springer-Verlag, 1993, pp. 455–465.

[13] SCHRÖDER, P., AND SWELDENS, W. Spherical wavelets: Efficiently representing functions on the sphere. *Computer Graphics (SIGGRAPH '95 Proceedings)* (1995), 161–172.

[14] SCHROEDER, W., ZARGE, J., AND LORENSEN, W. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proceedings) 26*, 2 (1992), 65–70.

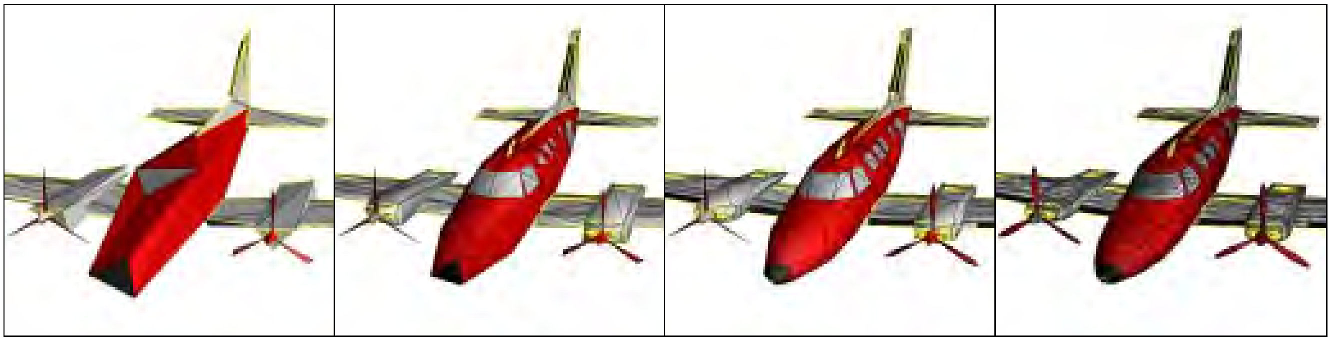
[15] TAUBIN, G., AND ROSSIGNAC, J. Geometry compression through topological surgery. Research Report RC-20340, IBM, January 1996.

[16] TURAN, G. Succinct representations of graphs. *Discrete Applied Mathematics 8* (1984), 289–294.

[17] TURK, G. Re-tiling polygonal surfaces. *Computer Graphics (SIGGRAPH '92 Proceedings) 26*, 2 (1992), 55–64.

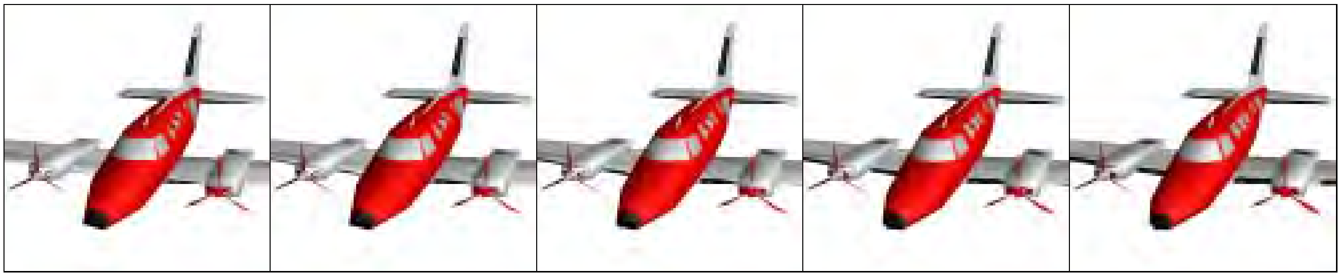
[18] UPSTILL, S. *The RenderMan Companion.* Addison-Wesley, 1990.

[19] WITTEN, I., NEAL, R., AND CLEARY, J. Arithmetic coding for data compression. *Communications of the ACM 30*, 6 (June 1987), 520–540.



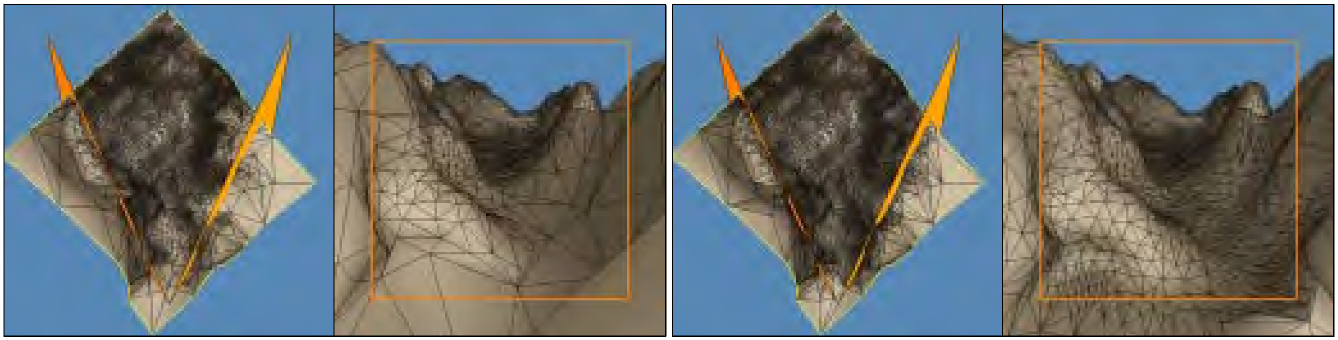
(a) Base mesh $M^0$ (150 faces) (b) Mesh $M^{175}$ (500 faces) (c) Mesh $M^{425}$ (1,000 faces) (d) Original $\hat{M} = M^n$ (13,546 faces)

Figure 5: The PM representation of an arbitrary mesh $\hat{M}$ captures a continuous-resolution family of approximating meshes $M^0 \ldots M^n = \hat{M}$.



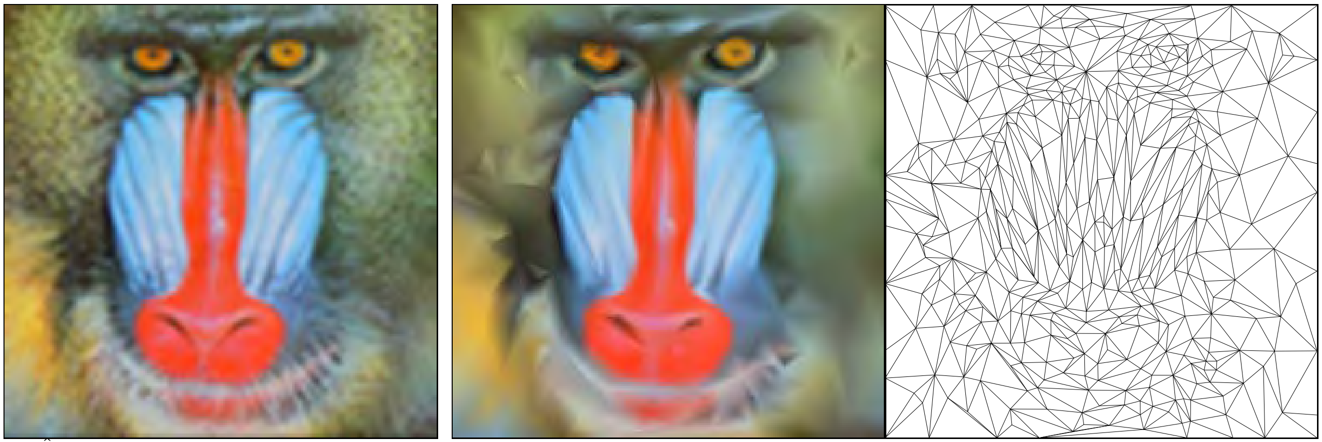
(a) $\alpha$ = 0.00 (b) $\alpha$ = 0.25 (c) $\alpha$ = 0.50 (d) $\alpha$ = 0.75 (e) $\alpha$ = 1.00

Figure 6: Example of a geomorph $M^G(\alpha)$ defined between $M^G(0) \doteq M^{175}$ (with 500 faces) and $M^G(1) = M^{425}$ (with 1,000 faces).



(a) Using conditions (1) and (2); 9,462 faces (b) Using conditions (1') and (2); 12,169 faces

Figure 7: Example of selective refinement within the view frustum (indicated in orange).



(a) $\hat{M}$ ($200 \times 200$ vertices) (b) Simplified mesh (400 vertices)

Figure 8: Demonstration of minimizing $E_{scalar}$: simplification of a mesh with trivial geometry (a square) but complex scalar attribute field. ($\hat{M}$ is a mesh with regular connectivity whose vertex colors correspond to the pixels of an image.)

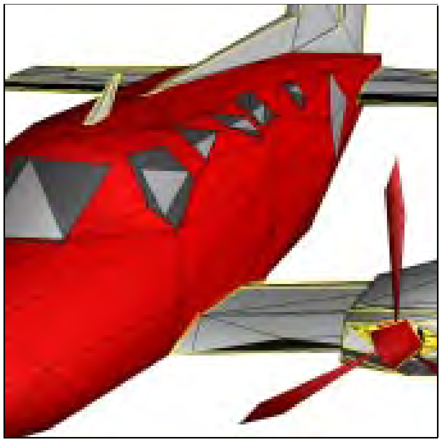


Figure 9: (a) Simplification without $E_{disc}$

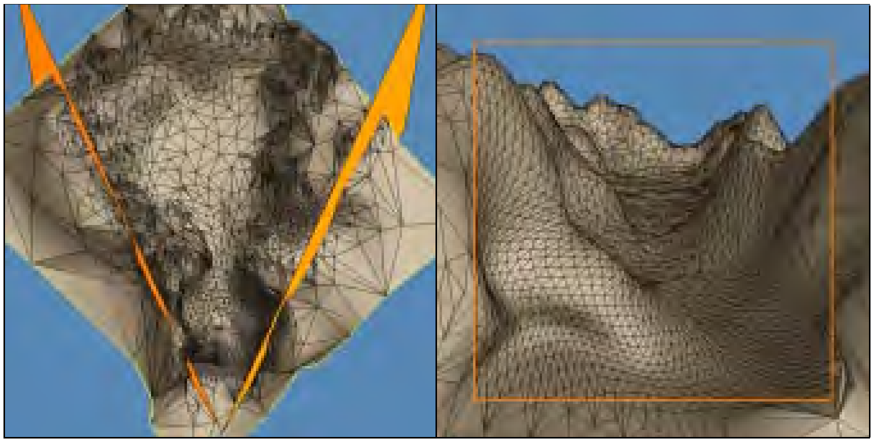


Figure 10: Selective refinement of a terrain mesh taking into account view frustum, silhouette regions, and projected screen size of faces (7,438 faces).

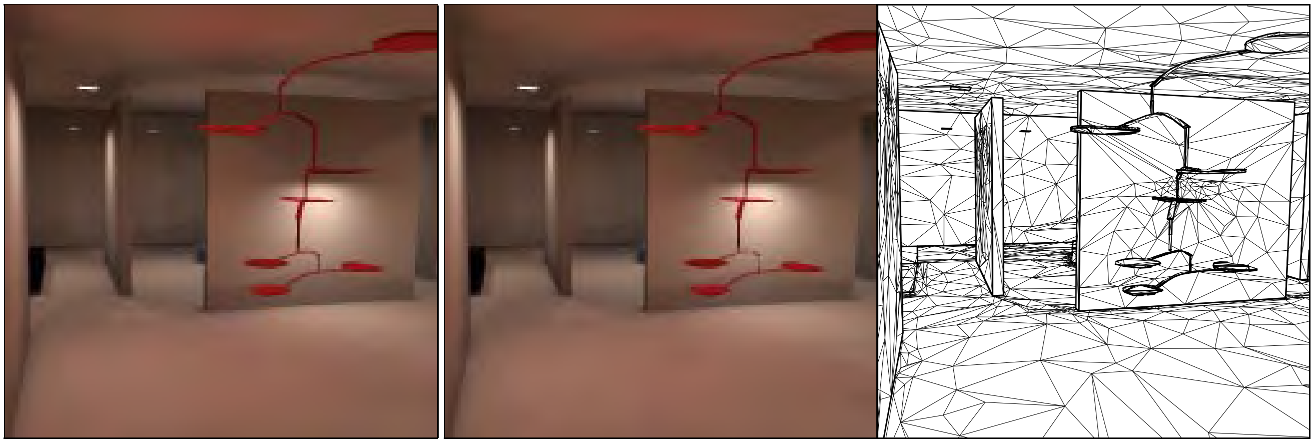
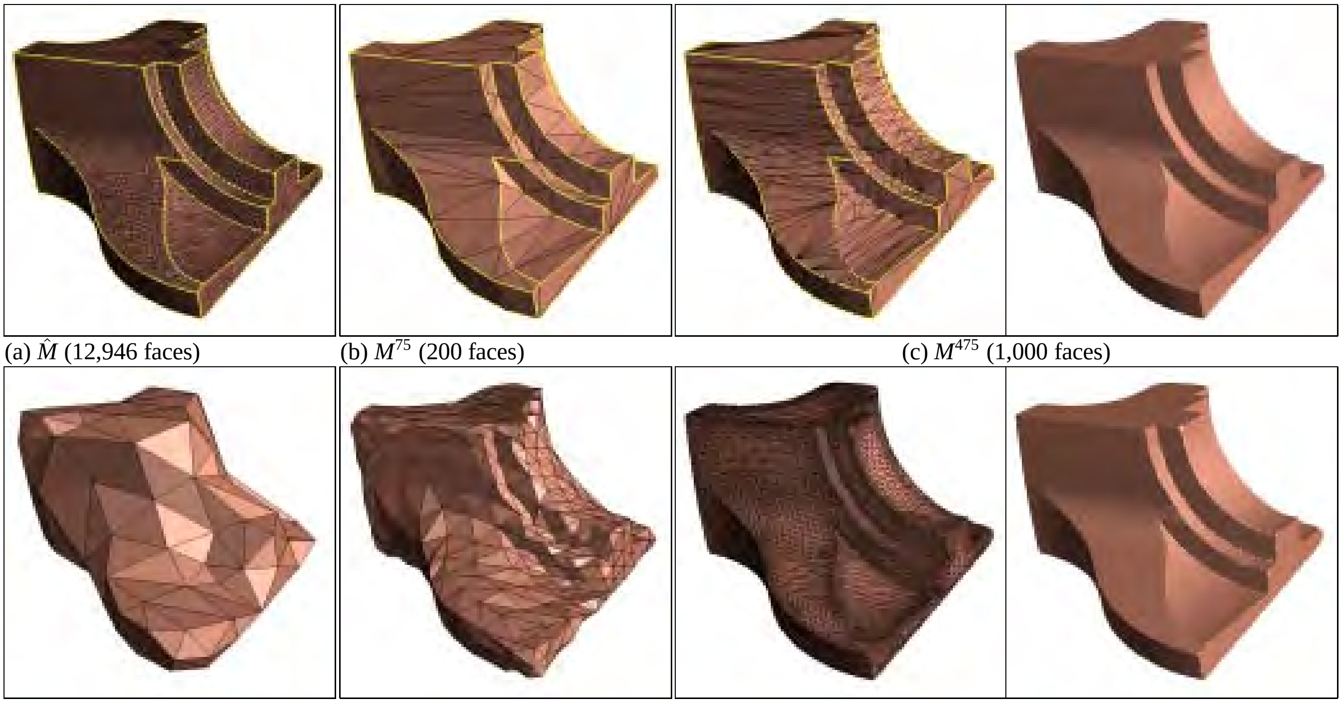


Figure 11: Simplification of a radiosity solution; left: original mesh (150,983 faces); right: simplified mesh (10,000 faces).



(a) $\hat{M}$ (12,946 faces)

(b) $M^{75}$ (200 faces)

(c) $M^{475}$ (1,000 faces)

(d) $\epsilon$ = 9.0 (192 faces)

(e) $\epsilon$ = 2.75 (1,070 faces)

(f) $\epsilon$ = 0.1 (15,842 faces)

Figure 12: Approximations of a mesh $\hat{M}$ using (b–c) the PM representation, and (d–f) the MRA scheme of Eck et al. [7]. As demonstrated, MRA cannot recover $\hat{M}$ exactly, cannot deal effectively with surface creases, and produces approximating meshes of inferior quality.

**APPENDIX O**

US005798770A

# United States Patent [19]

**Baldwin**

[11] **Patent Number: 5,798,770**

[45] **Date of Patent: Aug. 25, 1998**

[54] **GRAPHICS RENDERING SYSTEM WITH RECONFIGURABLE PIPELINE SEQUENCE**

[75] Inventor: **David Robert Baldwin**, Weybridge, United Kingdom

[73] Assignee: **3DLabs Inc. Ltd.**, Hamilton, Bermuda

[21] Appl. No.: **640,620**

[22] Filed: **May 1, 1996**

**Related U.S. Application Data**

[60] Provisional application No. 60/008,803 Dec. 18, 1995.

[63] Continuation-in-part of Ser. No. 410,345, Mar. 24, 1995.

[51] **Int. Cl.$^6$** .................................................... **G06T 1/20**

[52] **U.S. Cl.** ........................ **345/506**; 345/519; 345/509

[58] **Field of Search** .................................. 395/506, 502, 395/507, 509, 519, 122, 130, 132, 125, 503; 345/506, 507, 502, 509, 519, 422, 430–432, 425, 503

[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 4,866,637 | 9/1989 | Gonzalez-Lopez .................... | 395/506 |
| 5,392,391 | 2/1995 | Caulk, Jr. et al. ...................... | 395/503 |

OTHER PUBLICATIONS

Foley *et al.*, "Computer Graphics, Principles and Practice", 2 ed in C.1996, Chapter 18, pp. 855–920.

Kogge, P.M., "The Microprogramming of Pipelined Processors", 1977, Proc. 4th Ann. Conf Parallel Procesing, IEEE, March, pp. 63–69.

Computer Graphics, vol. 22, No. 4, "A display system for the Stellar graphics Supercomputer Model GS1000", Brian Apgar et al., Aug. 1988.

*Primary Examiner*—Kee M. Tung

*Attorney, Agent, or Firm*—Robert Groover; Betty Formby; Matthew S. Anderson

[57] **ABSTRACT**

The preferred embodiment discloses a pipelined graphics processor in which the sequence can be dynamically reconfigured (e.g. between primitives) in a rendering sequence. The pipeline sequence can be configured for compliance with specifications such as OpenGL, but may also be optimized by reconfiguring the pipeline sequence to eliminate unnecessary processing. In a preferred embodiment, pixel elimination sequences such as depth and stencil tests are performed before texturing calculations are performed, so that unneeded pixel data is discarded before said texturing calculations are performed.

**26 Claims, 12 Drawing Sheets**

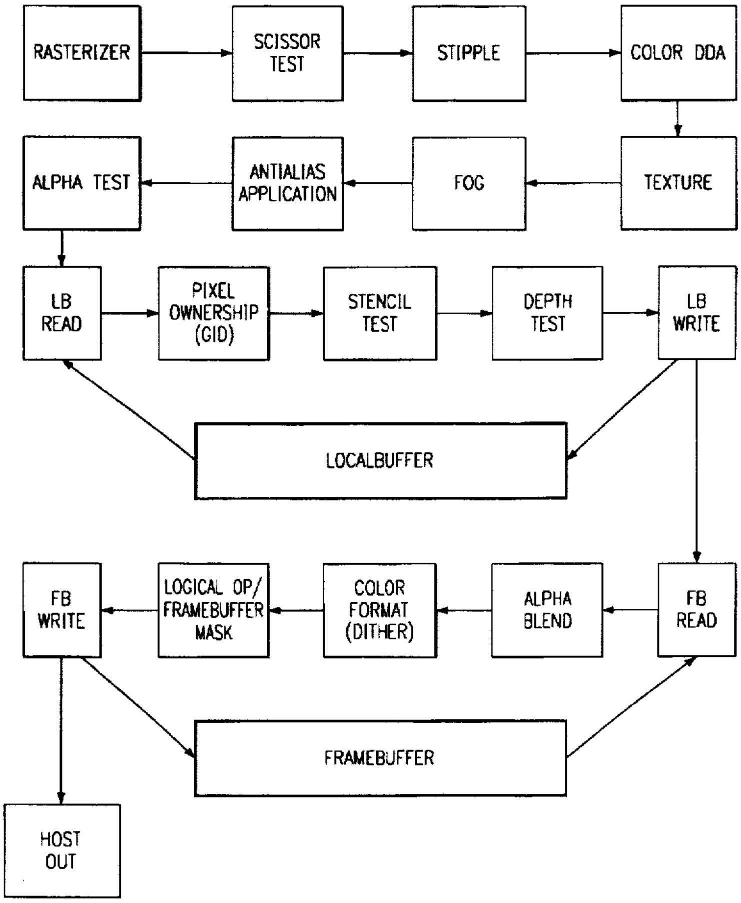
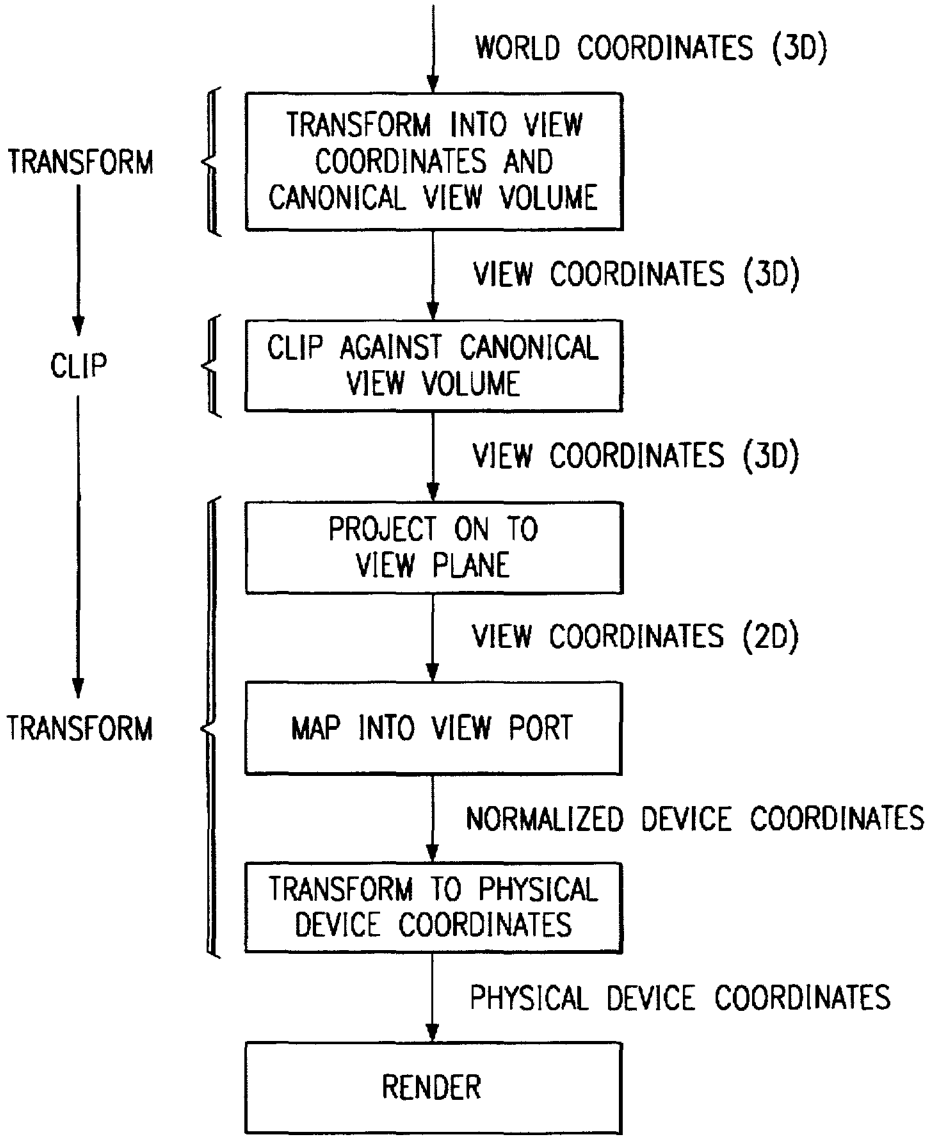
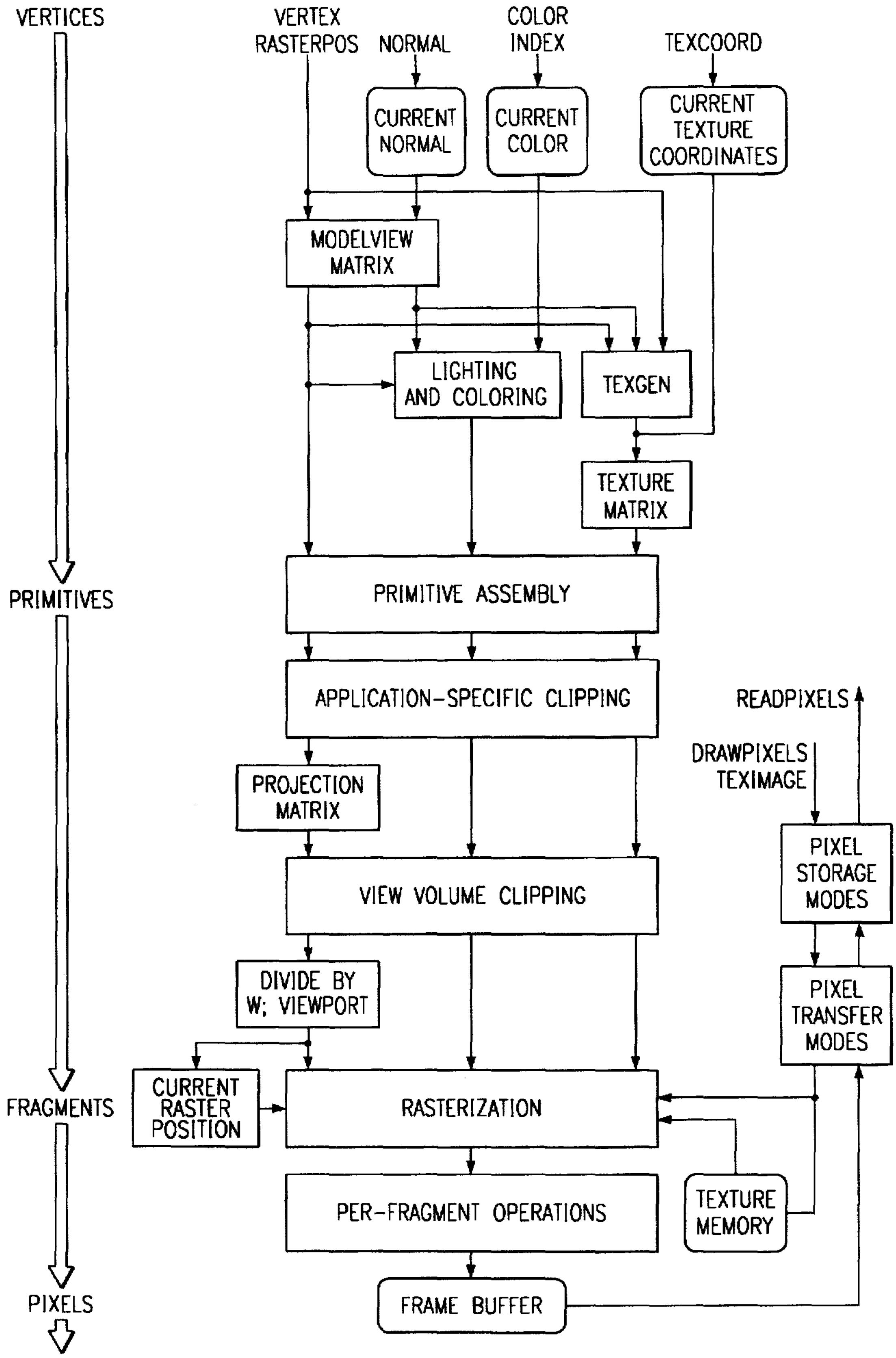


FIG. 1A


WORLD COORDINATES (3D)
TRANSFORM
TRANSFORM INTO VIEW COORDINATES AND CANONICAL VIEW VOLUME
VIEW COORDINATES (3D)
CLIP
CLIP AGAINST CANONICAL VIEW VOLUME
VIEW COORDINATES (3D)
PROJECT ON TO VIEW PLANE
VIEW COORDINATES (2D)
TRANSFORM
MAP INTO VIEW PORT
NORMALIZED DEVICE COORDINATES
TRANSFORM TO PHYSICAL DEVICE COORDINATES
PHYSICAL DEVICE COORDINATES
RENDER

APPENDIX O

FIG. 1B



APPENDIX O

FIG. 2A

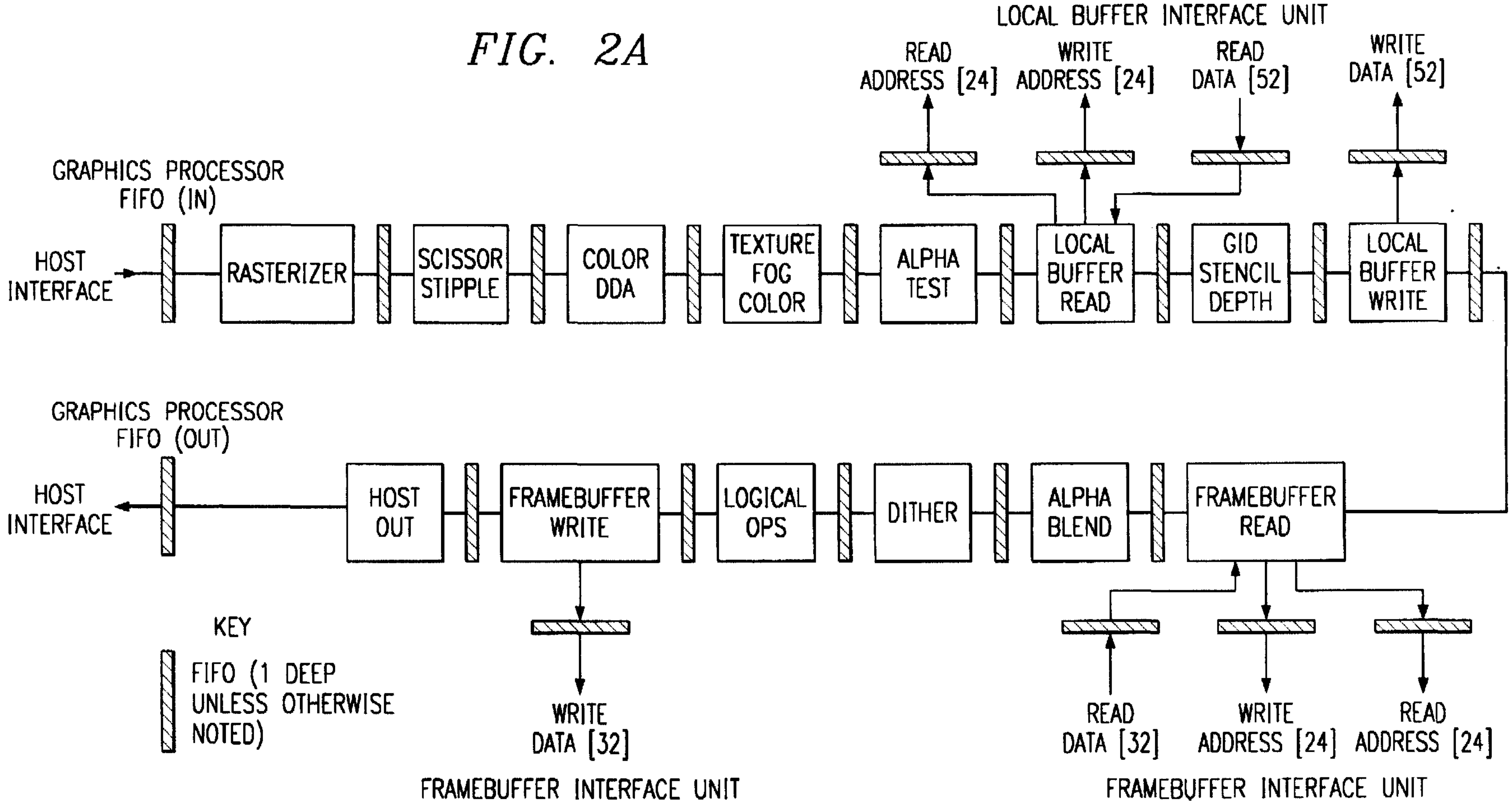


*FIG. 2B*

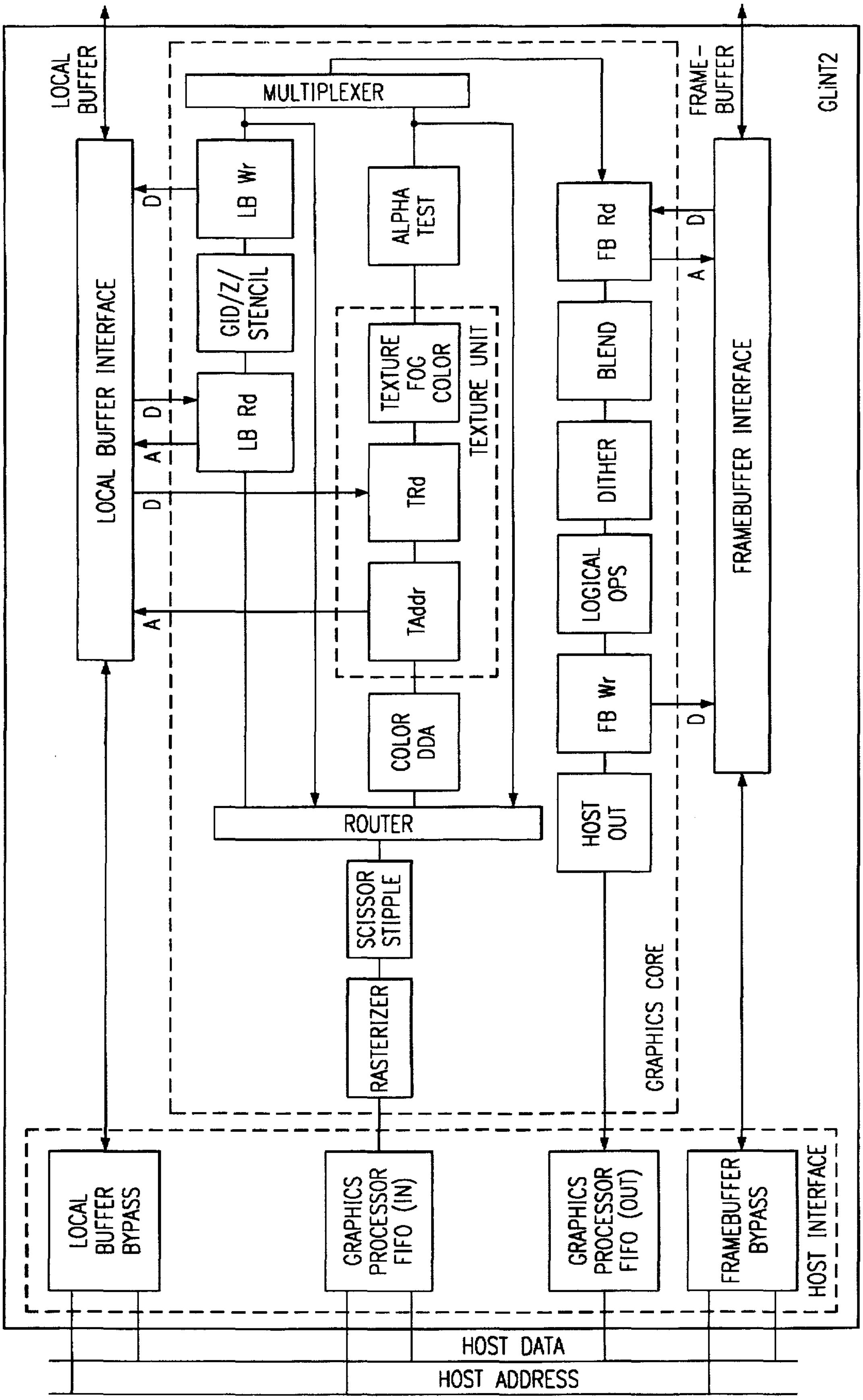
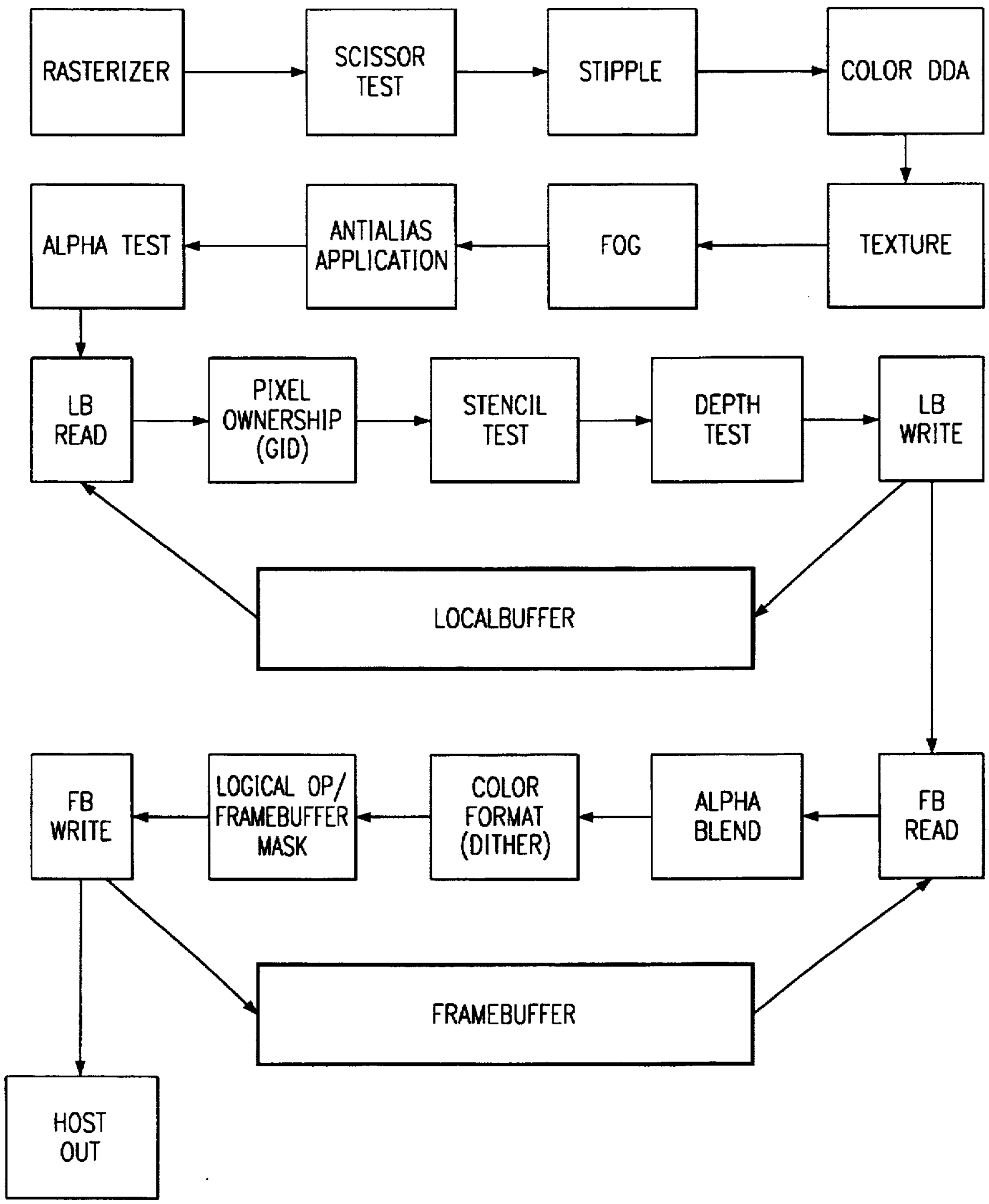

LOCAL BUFFER
FRAME-BUFFER
GLINT2
MULTIPLEXER
LB Wr
GID/Z/STENCIL
LB Rd
ALPHA TEST
TEXTURE FOG COLOR
TRd
TAddr
TEXTURE UNIT
COLOR DDA
FB Rd
BLEND
DITHER
LOGICAL OPS
FB Wr
HOST OUT
LOCAL BUFFER INTERFACE
FRAMEBUFFER INTERFACE
D
A
D
D
A
D
A
D
ROUTER
SCISSOR STIPPLE
RASTERIZER
GRAPHICS CORE
LOCAL BUFFER BYPASS
GRAPHICS PROCESSOR FIFO (IN)
GRAPHICS PROCESSOR FIFO (OUT)
FRAMEBUFFER BYPASS
HOST INTERFACE
HOST DATA
HOST ADDRESS

FIG. 2C


RASTERIZER
SCISSOR TEST
STIPPLE
COLOR DDA
TEXTURE
FOG
ANTIALIAS APPLICATION
ALPHA TEST
LB READ
PIXEL OWNERSHIP (GID)
STENCIL TEST
DEPTH TEST
LB WRITE
LOCALBUFFER
FB READ
ALPHA BLEND
COLOR FORMAT (DITHER)
LOGICAL OP/ FRAMEBUFFER MASK
FB WRITE
FRAMEBUFFER
HOST OUT

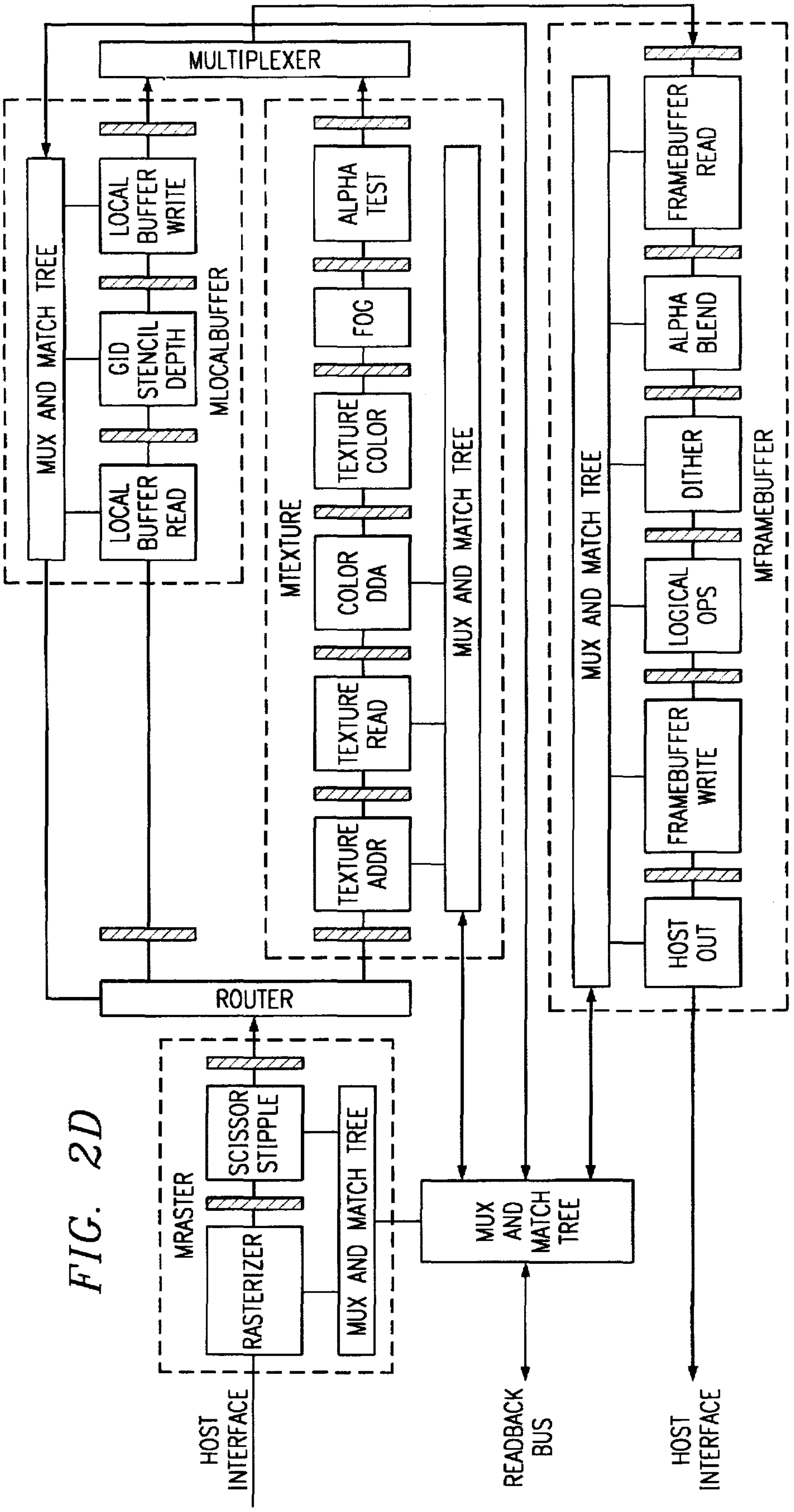

MULTIPLEXER
MUX AND MATCH TREE
LOCAL BUFFER WRITE
GID STENCIL DEPTH
LOCAL BUFFER READ
MLOCALBUFFER
ALPHA TEST
FOG
TEXTURE COLOR
COLOR DDA
TEXTURE READ
TEXTURE ADDR
MUX AND MATCH TREE
MTEXTURE
MUX AND MATCH TREE
FRAMEBUFFER READ
ALPHA BLEND
DITHER
LOGICAL OPS
FRAMEBUFFER WRITE
HOST OUT
MFRAMEBUFFER
ROUTER
SCISSOR STIPPLE
RASTERIZER
MUX AND MATCH TREE
MRASTER
MUX AND MATCH TREE
HOST INTERFACE
READBACK BUS
HOST INTERFACE

*FIG. 2D*

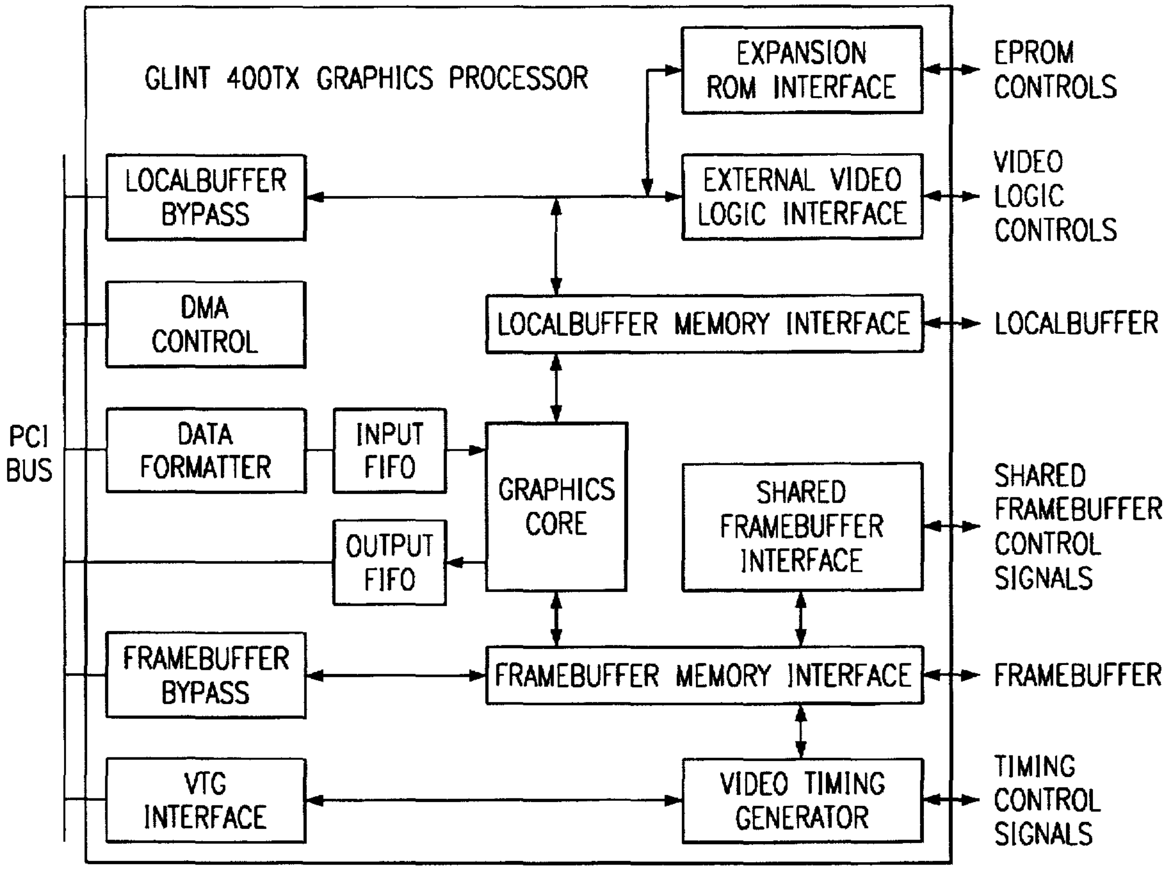

GLINT 400TX GRAPHICS PROCESSOR
EXPANSION ROM INTERFACE
EPROM CONTROLS
LOCALBUFFER BYPASS
EXTERNAL VIDEO LOGIC INTERFACE
VIDEO LOGIC CONTROLS
DMA CONTROL
LOCALBUFFER MEMORY INTERFACE
LOCALBUFFER
PCI BUS
DATA FORMATTER
INPUT FIFO
GRAPHICS CORE
SHARED FRAMEBUFFER INTERFACE
SHARED FRAMEBUFFER CONTROL SIGNALS
OUTPUT FIFO
FRAMEBUFFER BYPASS
FRAMEBUFFER MEMORY INTERFACE
FRAMEBUFFER
VTG INTERFACE
VIDEO TIMING GENERATOR
TIMING CONTROL SIGNALS

*FIG. 2E*

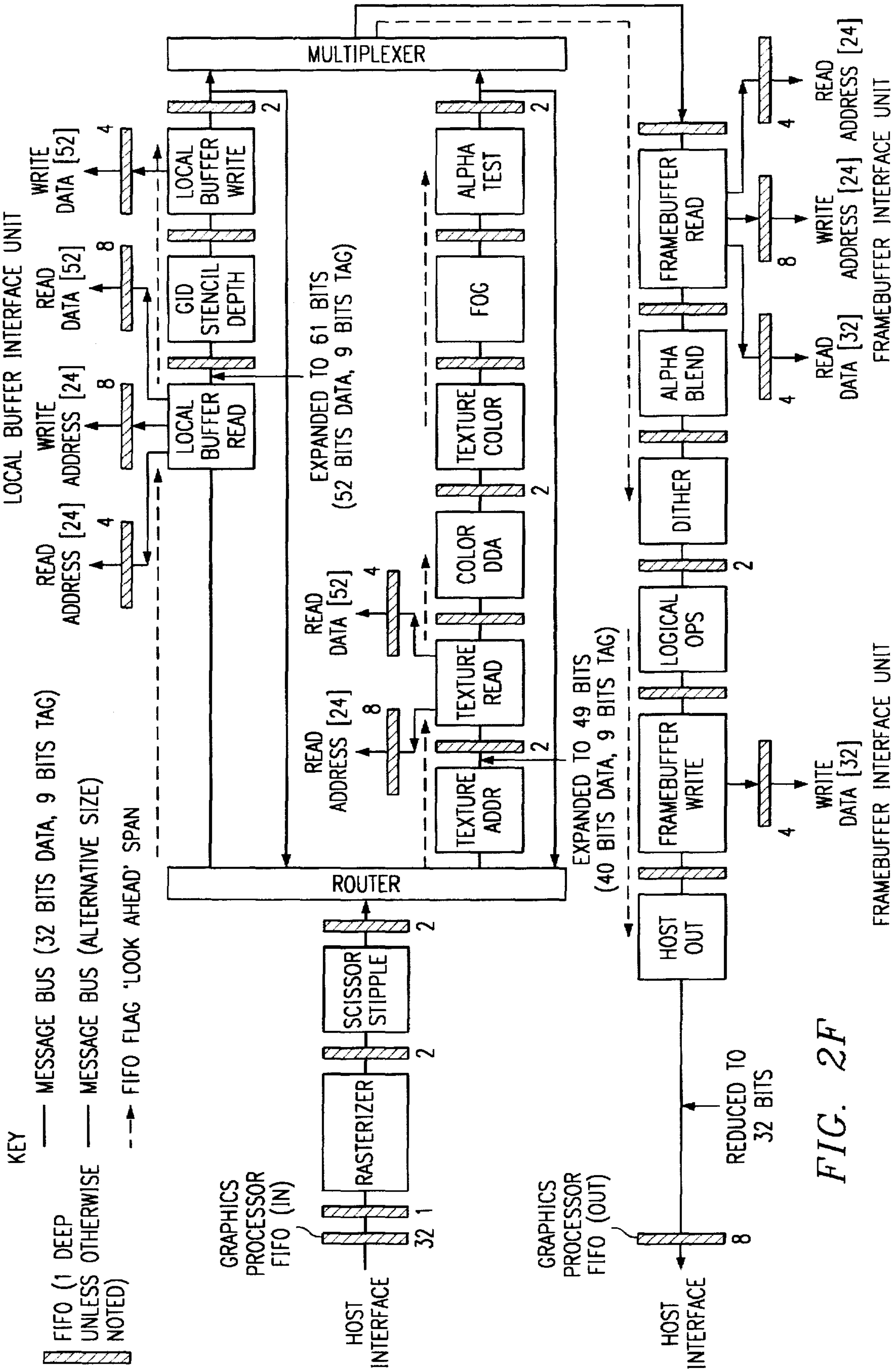

KEY
FIFO (1 DEEP UNLESS OTHERWISE NOTED)
— MESSAGE BUS (32 BITS DATA, 9 BITS TAG)
— MESSAGE BUS (ALTERNATIVE SIZE)
FIFO FLAG 'LOOK AHEAD' SPAN
HOST INTERFACE
GRAPHICS PROCESSOR FIFO (IN)
32
1
RASTERIZER
2
SCISSOR STIPPLE
2
ROUTER
MULTIPLEXER
LOCAL BUFFER INTERFACE UNIT
READ ADDRESS [24]
4
WRITE ADDRESS [24]
8
READ DATA [52]
8
WRITE DATA [52]
4
LOCAL BUFFER READ
GID STENCIL DEPTH
LOCAL BUFFER WRITE
2
EXPANDED TO 61 BITS (52 BITS DATA, 9 BITS TAG)
TEXTURE ADDR
2
TEXTURE READ
READ ADDRESS [24]
8
READ DATA [52]
4
COLOR DDA
2
TEXTURE COLOR
FOG
ALPHA TEST
2
EXPANDED TO 49 BITS (40 BITS DATA, 9 BITS TAG)
HOST OUT
FRAMEBUFFER WRITE
LOGICAL OPS
2
DITHER
ALPHA BLEND
FRAMEBUFFER READ
WRITE DATA [32]
4
FRAMEBUFFER INTERFACE UNIT
READ DATA [32]
4
WRITE ADDRESS [24]
8
READ ADDRESS [24]
4
FRAMEBUFFER INTERFACE UNIT
REDUCED TO 32 BITS
GRAPHICS PROCESSOR FIFO (OUT)
8
HOST INTERFACE

*FIG. 2F*

APPENDIX O

FIG. 3A

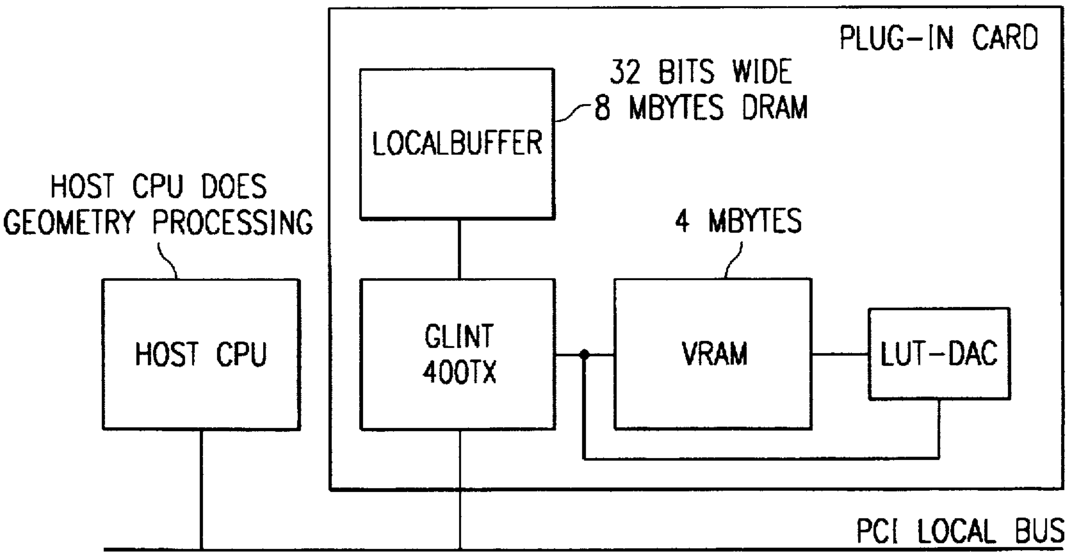
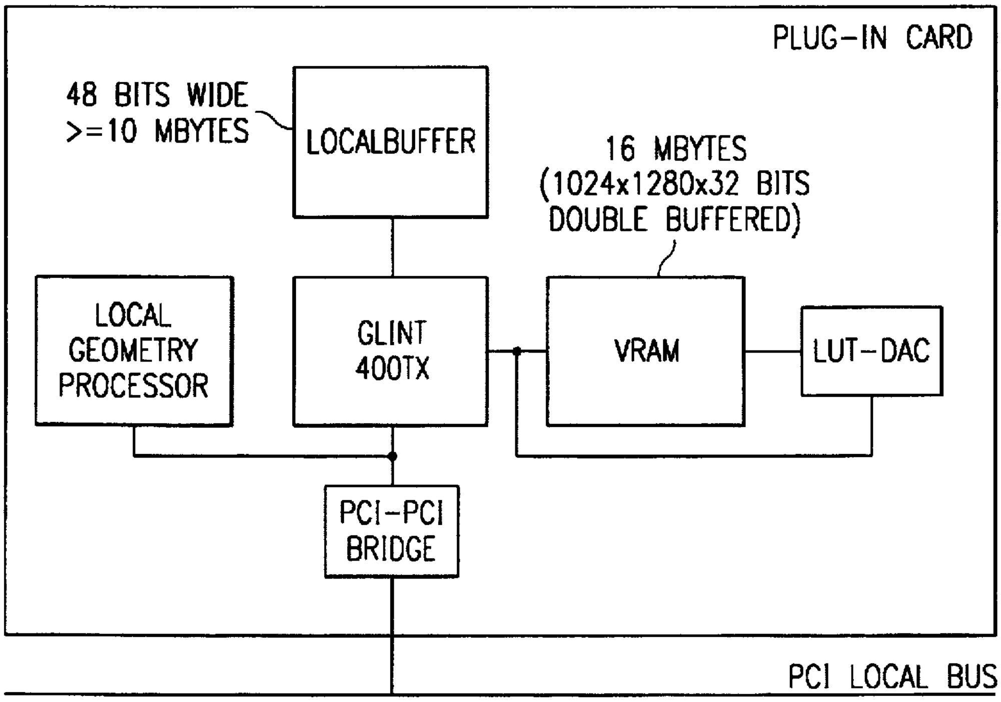


FIG. 3B



APPENDIX O

FIG. 3C

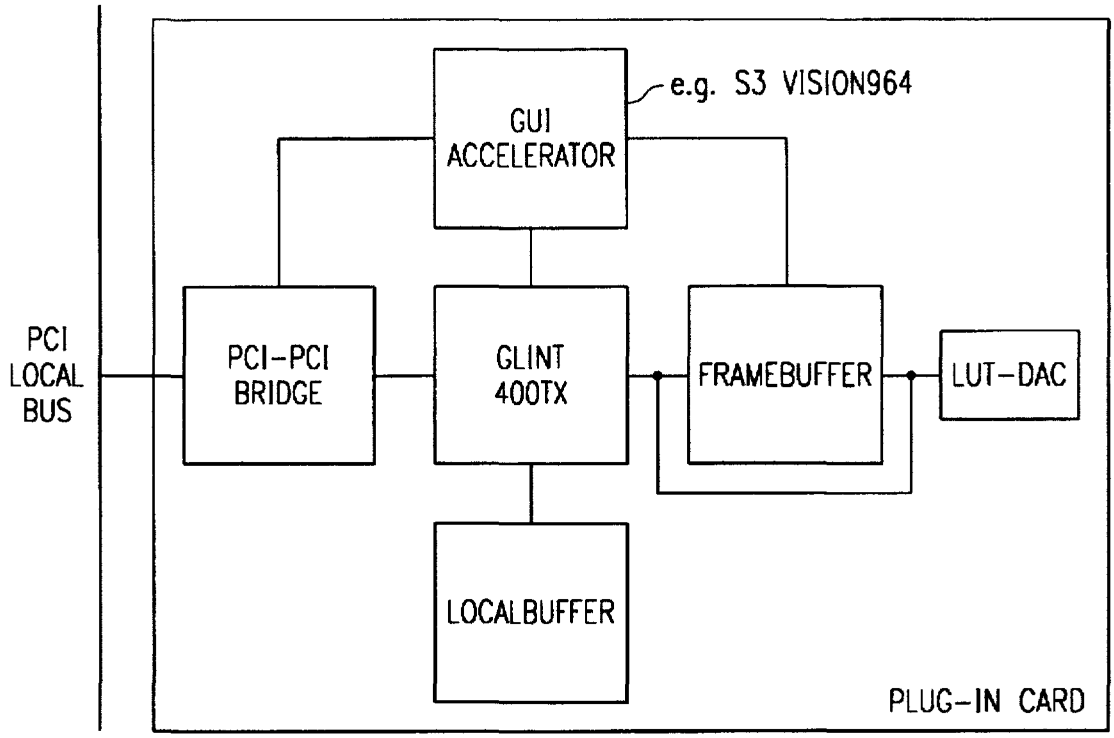


FIG. 3D

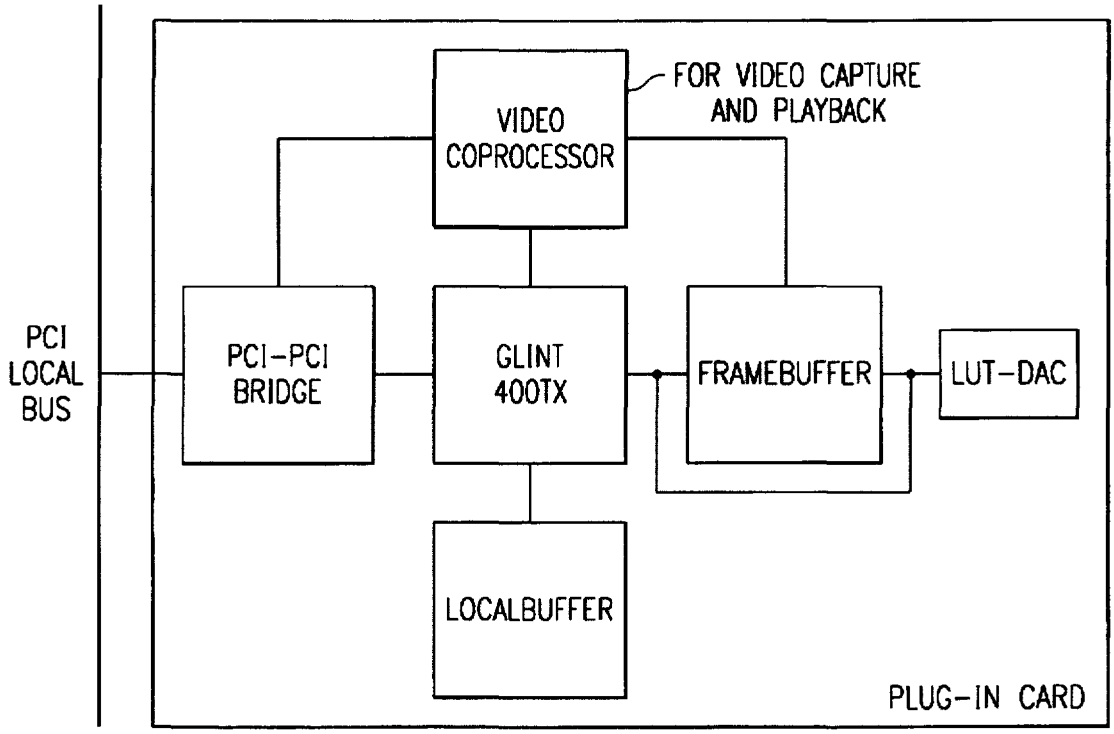


FIG. 4A

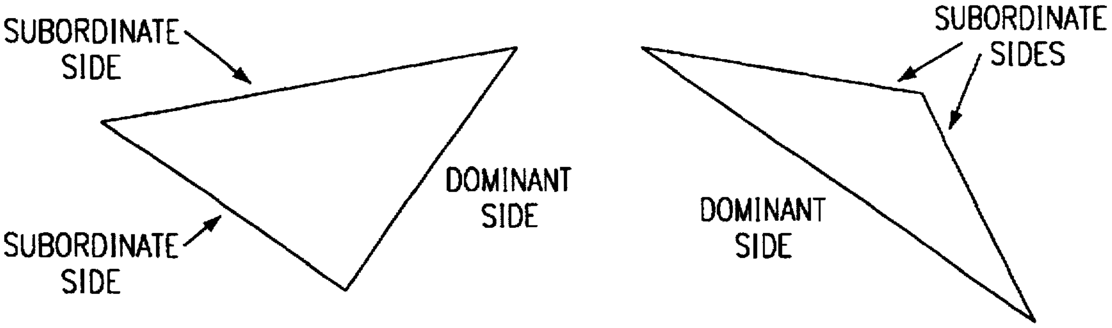
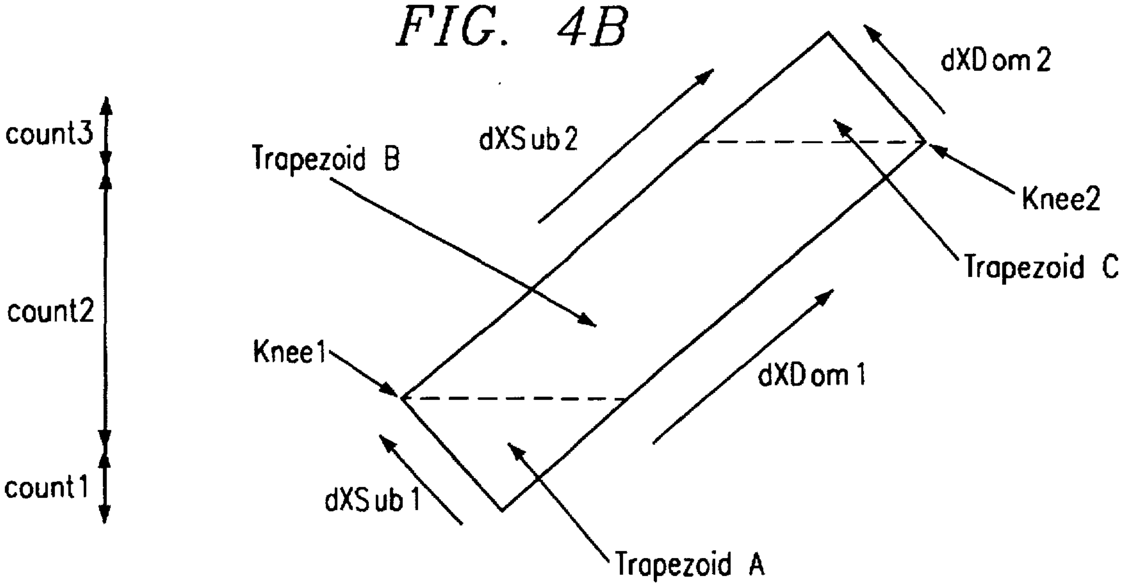

SUBORDINATE SIDE
SUBORDINATE SIDE
DOMINANT SIDE
SUBORDINATE SIDES
DOMINANT SIDE

FIG. 4B


count3
count2
count1
Trapezoid B
dXSub2
dXDom2
Knee2
Trapezoid C
Knee1
dXDom1
dXSub1
Trapezoid A

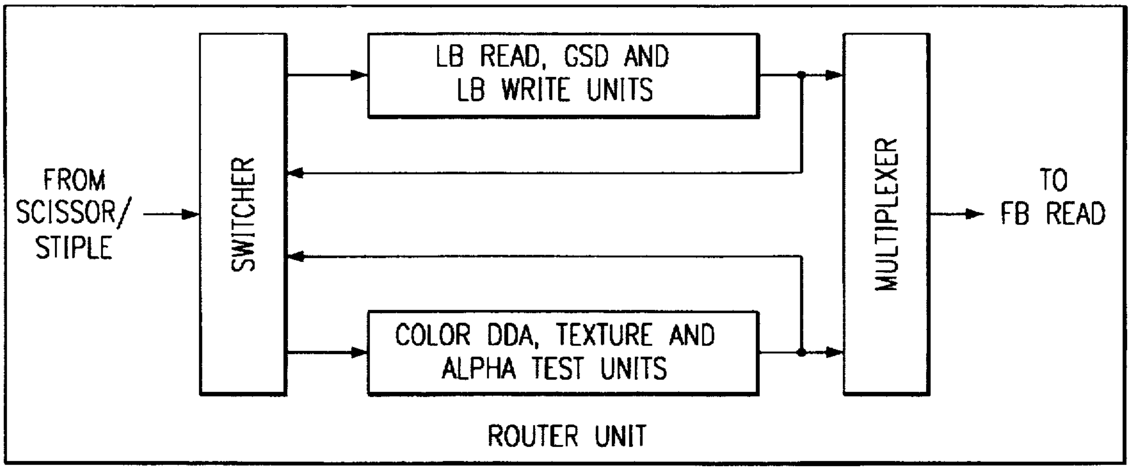

FROM SCISSOR/ STIPLE
SWITCHER
LB READ, GSD AND LB WRITE UNITS
COLOR DDA, TEXTURE AND ALPHA TEST UNITS
MULTIPLEXER
TO FB READ
ROUTER UNIT

FIG. 5A

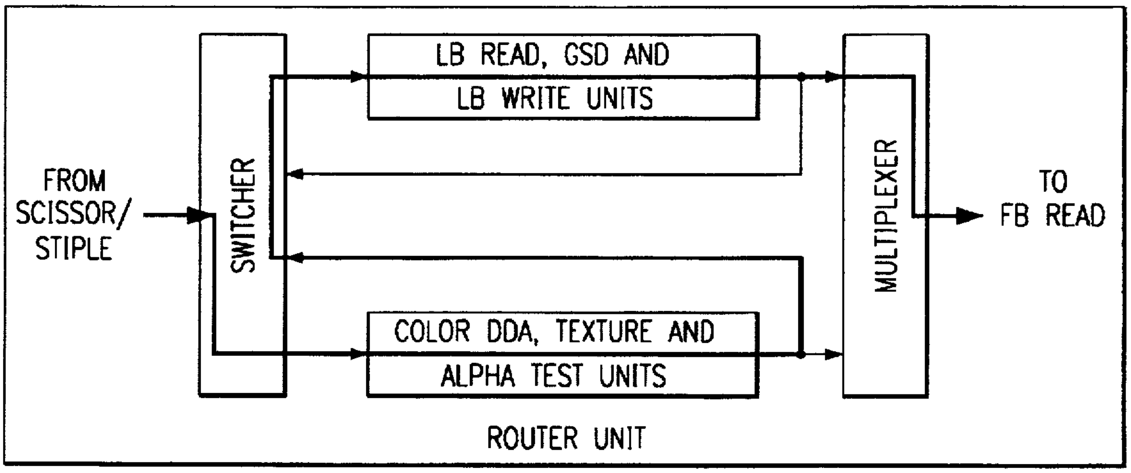

FROM SCISSOR/ STIPLE
SWITCHER
LB READ, GSD AND LB WRITE UNITS
COLOR DDA, TEXTURE AND ALPHA TEST UNITS
MULTIPLEXER
TO FB READ
ROUTER UNIT

FIG. 5B

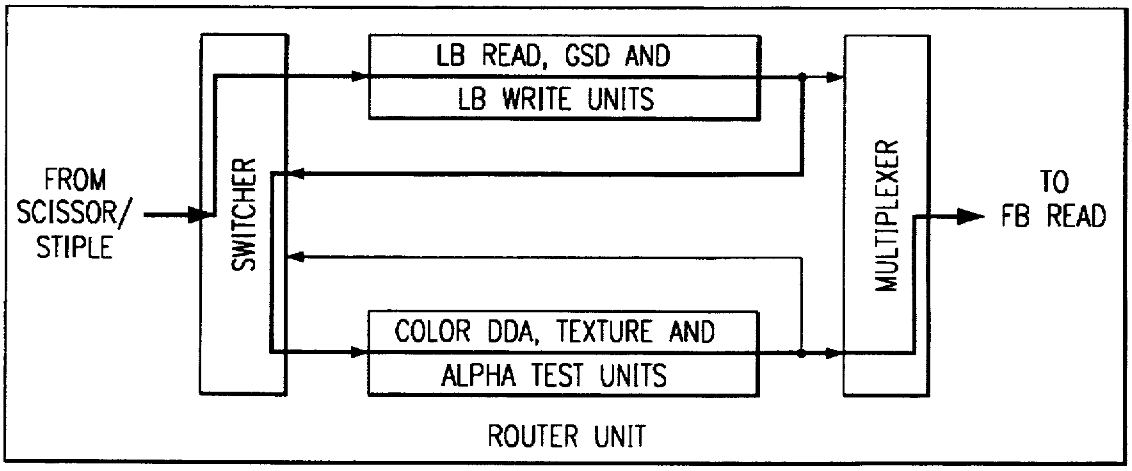

FROM SCISSOR/ STIPLE
SWITCHER
LB READ, GSD AND LB WRITE UNITS
COLOR DDA, TEXTURE AND ALPHA TEST UNITS
MULTIPLEXER
TO FB READ
ROUTER UNIT

FIG. 5C

# GRAPHICS RENDERING SYSTEM WITH RECONFIGURABLE PIPELINE SEQUENCE

## CROSS-REFERENCE TO RELATED APPLICATION

This application is a continuation-in-part of 08/410,345 filed Mar. 24, 1995, and claims priority from provisional 60/008,803 filed Dec. 18, 1995, which is hereby incorporated by reference.

## BACKGROUND AND SUMMARY OF THE INVENTION

The present application relates to computer graphics and animation systems, and particularly to 3D graphics rendering hardware. Background of the art and the prior embodiment, according to the parent application, is described below. Some of the distinctions of the presently preferred embodiment are particularly noted beginning on page **8**.

## COMPUTER GRAPHICS AND RENDERING

Modern computer systems normally manipulate graphical objects as high-level entities. For example, a solid body may be described as a collection of triangles with specified vertices, or a straight line segment may be described by listing its two endpoints with three-dimensional or two-dimensional coordinates. Such high-level descriptions are a necessary basis for high-level geometric manipulations, and also have the advantage of providing a compact format which does not consume memory space unnecessarily.

Such higher-level representations are very convenient for performing the many required computations. For example, ray-tracing or other lighting calculations may be performed, and a projective transformation can be used to reduce a three-dimensional scene to its two-dimensional appearance from a given viewpoint. However, when an image containing graphical objects is to be displayed, a very low-level description is needed. For example, in a conventional CRT display, a "flying spot" is moved across the screen (one line at a time), and the beam from each of three electron guns is switched to a desired level of intensity as the flying spot passes each pixel location. Thus at some point the image model must be translated into a data set which can be used by a conventional display. This operation is known as "rendering."

The graphics-processing system typically interfaces to the display controller through a "frame store" or "frame buffer" of special two-port memory, which can be written to randomly by the graphics processing system, but also provides the synchronous data output needed by the video output driver. (Digital-to-analog conversion is also provided after the frame buffer.) Such a frame buffer is usually implemented using VRAM memory chips (or sometimes with DRAM and special DRAM controllers). This interface relieves the graphics processing system of most of the burden of synchronization for video output. Nevertheless, the amounts of data which must be moved around are very sizable, and the computational and data-transfer burden of placing the correct data into the frame buffer can still be very large.

Even if the computational operations required are quite simple, they must be performed repeatedly on a large number of data points. For example, in a typical 1995 high-end configuration, a display of 1280×1024 elements may need to be refreshed at 72 Hz, with a color resolution of 24 bits per pixel. If blending is desired, additional bits (e.g. another 8 bits per pixel) will be required to store an "alpha" or transparency value for each pixel. This implies manipulation of more than 3 billion bits per second, without allowing for any of the actual computations being performed. Thus it may be seen that this is an environment with unique data manipulation requirements.

If the display is unchanging, no demand is placed on the rendering operations. However, some common operations (such as zooming or rotation) will require every object in the image space to be re-rendered. Slow rendering will make the rotation or zoom appear jerky. This is highly undesirable. Thus efficient rendering is an essential step in translating an image representation into the correct pixel values. This is particularly true in animation applications, where newly rendered updates to a computer graphics display must be generated at regular intervals.

The rendering requirements of three-dimensional graphics are particularly heavy. One reason for this is that, even after the three-dimensional model has been translated to a two-dimensional model, some computational tasks may be bequeathed to the rendering process. (For example, color values will need to be interpolated across a triangle or other primitive.) These computational tasks tend to burden the rendering process. Another reason is that since three-dimensional graphics are much more lifelike, users are more likely to demand a fully rendered image. (By contrast, in the two-dimensional images created e.g. by a GUI or simple game, users will learn not to expect all areas of the scene to be active or filled with information.)

FIG. **1A** is a very high-level view of other processes performed in a 3D graphics computer system. A three dimensional image which is defined in some fixed 3D coordinate system (a "world" coordinate system) is transformed into a viewing volume (determined by a view position and direction), and the parts of the image which fall outside the viewing volume are discarded. The visible portion of the image volume is then projected onto a viewing plane, in accordance with the familiar rules of perspective. This produces a two-dimensional image, which is now mapped into device coordinates. It is important to understand that all of these operations occur prior to the operations performed by the rendering subsystem of the present invention. FIG. **1B** is an expanded version of FIG. **1A**, and shows the flow of operations defined by the OpenGL standard.

A vast amount of engineering effort has been invested in computer graphics systems, and this area is one of increasing activity and demands. Numerous books have discussed the requirements of this area; see, e.g., ADVANCES IN COMPUTER GRAPHICS (ed. Enderle 1990-); Chellappa and Sawchuk, DIGITAL IMAGE PROCESSING AND ANALYSIS (1985); COMPUTER GRAPHICS HARDWARE (ed. Reghbati and Lee 1988); COMPUTER GRAPHICS: IMAGE SYNTHESIS (ed. Joy et al.); Foley et al., FUNDAMENTALS OF INTERACTIVE COMPUTER GRAPHICS (2.ed. 1984); Foley, COMPUTER GRAPHICS PRINCIPLES & PRACTICE (2.ed. 1990); Foley, INTRODUCTION TO COMPUTER GRAPHICS (1994); Giloi, Interactive Computer Graphics (1978); Hearn and Baker, COMPUTER GRAPHICS (2.ed. 1994); Hill, COMPUTER GRAPHICS (1990); Latham, DICTIONARY OF COMPUTER GRAPHICS (1991); Magnenat-Thalma, IMAGE SYNTHESIS THEORY & PRACTICE (1988); Newman and Sproull, PRINCIPLES OF INTERACTIVE COMPUTER GRAPHICS (2.ed. 1979); PICTURE ENGINEERING (ed. Fu and Kunii 1982); PICTURE PROCESSING & DIGITAL FILTERING (2.ed. Huang 1979); Prosise, HOW COMPUTER GRAPHICS WORK (1994); Rimmer, BIT MAPPED GRAPHICS (2.ed. 1993); Salmon, COMPUTER GRAPHICS SYSTEMS & CONCEPTS

(1987); Schachter, COMPUTER IMAGE GENERATION (1990); Watt, THREE-DIMENSIONAL COMPUTER GRAPHICS (2.ed. 1994); Scott Whitman, MULTIPROCESSOR METHODS FOR COMPUTER GRAPHICS RENDERING; the SIGGRAPH PROCEEDINGS for the years 1980–1994; and the *IEEE Computer Graphics and Applications* magazine for the years 1990–1994.

Background: Graphics Animation

In many areas of computer graphics a succession of slowly changing pictures are displayed rapidly one after the other, to give the impression of smooth movement, in much the same way as for cartoon animation. In general the higher the speed of the animation, the smoother (and better) the result.

When an application is generating animation images, it is normally necessary not only to draw each picture into the frame buffer, but also to first clear down the frame buffer, and to clear down auxiliary buffers such as depth (Z) buffers, stencil buffers, alpha buffers and others. A good treatment of the general principles may be found in *Computer Graphics: Principles and Practice*, James D. Foley et al., Reading Mass.: Addison-Wesley. A specific description of the various auxiliary buffers may be found in *The OpenGL Graphics System: A Specification* (*Version* 1.0), Mark Segal and Kurt Akeley, SGI.

In most applications the value written, when clearing any given buffer, is the same at every pixel location, though different values may be used in different auxiliary buffers. Thus the frame buffer is often cleared to the value which corresponds to black, while the depth (Z) buffer is typically cleared to a value corresponding to infinity.

The time taken to clear down the buffers is often a significant portion of the total time taken to draw a frame, so it is important to minimize it.

Background: Parallelism in Graphics Processing

Due to the large number of at least partially independent operations which are performed in rendering, many proposals have been made to use some form of parallel architecture for graphics (and particularly for rendering). See, for example, the special issue of *Computer Graphics* on parallel rendering (September 1994). Other approaches may be found in earlier patent filings by the assignee of the present application and its predecessors, e.g. U.S. Pat. No. 5,195,186, and published PCT applications PCT/GB90/00987, PCT/GB90/01209, PCT/GB90/01210, PCT/GB90/01212, PCT/GB90/01213, PCT/GB90/01214, PCT/GB90/01215, and PCT/GB90/01216.

Background: Pipelined Processing Generally

There are several general approaches to parallel processing. One of the basic approaches to achieving parallelism in computer processing is a technique known as pipelining. In this technique the individual processors are, in effect, connected in series in an assembly-line configuration: one processor performs a first set of operations on one chunk of data, and then passes that chunk along to another processor which performs a second set of operations, while at the same time the first processor performs the first set operations again on another chunk of data. Such architectures are generally discussed in Kogge, THE ARCHITECTURE OF PIPELINED COMPUTERS (1981).

Background: The OpenGL™ Standard

The "OpenGL" standard is a very important software standard for graphics applications. In any computer system which supports this standard, the operating system(s) and application software programs can make calls according to the OpenGL standards, without knowing exactly what the hardware configuration of the system is.

The OpenGL standard provides a complete library of low-level graphics manipulation commands, which can be used to implement three-dimensional graphics operations. This standard was originally based on the proprietary standards of Silicon Graphics, Inc., but was later transformed into an open standard. It is now becoming extremely important, not only in high-end graphics-intensive workstations, but also in high-end PCs. OpenGL is supported by Windows NT™, which makes it accessible to many PC applications.

The OpenGL specification provides some constraints on the sequence of operations. For instance, the color DDA operations must be performed before the texturing operations, which must be performed before the alpha operations. (A "DDA" or digital differential analyzer, is a conventional piece of hardware used to produce linear gradation of color (or other) values over an image area.)

Other graphics interfaces (or "APIs"), such as PHIGS or XGL, are also current as of 1995; but at the lowest level, OpenGL is a superset of most of these.

The OpenGL standard is described in the *OPEN*GL PROGRAMMING *GUIDE* (1993), the OPENGL REFERENCE MANUAL (1993), and a book by Segal and Akeley (of SGI) entitled THE OPENGL GRAPHICS SYSTEM: A SPECIFICATION (Version 1.0).

FIG. 1B is an expanded version of FIG. 1A, and shows the flow of operations defined by the OpenGL standard. Note that the most basic model is carried in terms of vertices, and these vertices are then assembled into primitives (such as triangles, lines, etc.). After all manipulation of the primitives has been completed, the rendering operations will translate each primitive into a set of "fragments." (A fragment is the portion of a primitive which affects a single pixel.) Again, it should be noted that all operations above the block marked "Rasterization" would be performed by a host processor, or possibly by a "geometry engine" (i.e. a dedicated processor which performs rapid matrix multiplies and related data manipulations), but would normally not be performed by a dedicated rendering processor such as that of the presently preferred embodiment.

One disadvantage of standards such as OpenGL is that they require that texturing or other processor-intensive operations be performed on data before pixel elimination tests, e.g. depth testing, is performed, which wastes processor time by performing costly texturing calculations on pixels which will be eliminated later in the pipeline. When the OpenGL specification is not required or when the current OpenGl state vector cannot eliminate pixels as a result of the alpha test, however, it would be much more efficient to eliminate as many pixels as possible before doing these calculations. The present application discloses a method and device for reordering the processing steps in the rendering pipeline to either accommodate order-specific specifications such as OpenGL, or to provide for an optimized throughput by only performing processor-intensive operations on pixels which will actually be displayed.

Background: Texturing

Texture patterns are commonly used as a way to apply realistic visual detail at the sub-polygon level. See Foley et al., COMPUTER GRAPHICS: PRINCIPLES AND PRACTICE (2.ed. 1990, corr. 1995), especially at pages 741–744; Paul S. Heckbert, "Fundamentals of Texture Mapping and Image Warping," Thesis submitted to Dept. of EE and Computer Science, University of California, Berkeley, Jun. 17, 1994; Heckbert, "Survey of Computer Graphics," IEEE Computer Graphics, November 1986, pp.56ff. Since the surfaces are transformed (by the host or geometry engine) to produce a

2D view, the textures will need to be similarly transformed by a linear transform (normally projective or "affine"). (In conventional terminology, the coordinates of the object surface, i.e. the primitive being rendered, are referred to as an (s,t) coordinate space, and the map of the stored texture is referred to a (u,v) coordinate space.) The transformation in the resulting mapping means that a horizontal line in the (x,y) display space is very likely to correspond to a slanted line in the (u,v) space of the texture map, and hence many page breaks will occur, due to the texturing operation, as rendering walks along a horizontal line of pixels.

## Innovative System and Methods

The preferred embodiment discloses a pipelined graphics processor in which the sequence can be dynamically reconfigured (e.g. between primitives) in a rendering sequence. The pipeline sequence can be configured for compliance with specifications such as OpenGL, but may also be optimized by reconfiguring the pipeline sequence to eliminate unnecessary processing. In a preferred embodiment, pixel elimination sequences such as depth and stencil tests are performed before texturing calculations are performed, so that unneeded pixel data is discarded before said texturing calculations are performed.

It is noted that the texturing operations become more computation-intense, early elimination of unneeded pixels becomes even more valuable. For example, Phong shading and bump mapping both require many more operations than more common shading and texture mapping techniques, thus making the system of the present application even more valuable in real-time rendering systems.

An overhead cost is that the reconfigurable portion of the pipeline must be flushed at each reconfiguration—but since reconfiguration is normally done only on a per-primitive basis, or even less frequently, this is a relatively small cost.

## BRIEF DESCRIPTION OF THE DRAWING

The disclosed inventions will be described with reference to the accompanying drawings, which show important sample embodiments of the invention and which are incorporated in the specification hereof by reference, wherein:

FIG. 1A, described above, is an overview of key elements and processes in a 3D graphics computer system.

FIG. 1B is an expanded version of FIG. 1A, and shows the flow of operations defined by the OpenGL standard.

FIG. 2A is an overview of the graphics rendering chip of the preferred embodiment of the parent case.

FIG. 2B is an overview of the graphics rendering chip of the presently preferred embodiment.

FIG. 2C is a more schematic view of the sequence of operations performed in the graphics rendering chip of FIG. 2B, when operating in a first mode.

FIG. 2D is a different view of the graphics rendering chip of FIG. 2B, showing the connections of a readback bus which provides a diagnostic pathway.

FIG. 2E is yet another view of the graphics rendering chip of FIG. 2B, showing how the functions of the core pipeline of FIG. 2C are combined with various external interface functions.

FIG. 2F is yet another view of the graphics rendering chip of FIG. 2B, showing how the details of FIFO depth and lookahead are implemented, in the presently preferred embodiment.

FIG. 3A shows a sample graphics board which incorporates the chip of FIG. 2B.

FIG. 3B shows another sample graphics board implementation, which differs from the board of FIG. 3A in that more memory and an additional component is used to achieve higher performance.

FIG. 3C shows another graphics board, in which the chip of FIG. 2B shares access to a common frame store with GUI accelerator chip.

FIG. 3D shows another graphics board, in which the chip of FIG. 2B shares access to a common frame store with a video coprocessor (which may be used for video capture and playback functions.

FIG. 4A illustrates the definition of the dominant side and the subordinate sides of a triangle.

FIG. 4B illustrates the sequence of rendering an Anti-aliased Line primitive.

FIG. 5A is a detailed view of the router unit of the presently preferred embodiment.

FIG. 5B is a detailed view of the data path through the router unit of the presently preferred embodiment when operating in a first mode.

FIG. 5C is a detailed view of the data path through the router unit of the presently preferred embodiment when operating in a second mode.

## DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The numerous innovative teachings of the present application will be described with particular reference to the presently preferred embodiment (by way of example, and not of limitation). The presently preferred embodiment is a GLINT™ 400TX™ 3D rendering chip. The Hardware Reference Manual and Programmer's Reference Manual for this chip describe further details of this sample embodiment. Both are available, as of the effective filing date of this application, from 3Dlabs Inc. Ltd., 181 Metro Drive, Suite 520, San Jose Calif. 95110.

### Definitions

The following definitions may help in understanding the exact meaning of terms used in the text of this application:

application: a computer program which uses graphics animation.

depth (Z) buffer: A memory buffer containing the depth component of a pixel. Used to, for example, eliminate hidden surfaces.

blt double-buffering: A technique for achieving smooth animation, by rendering only to an undisplayed back buffer, and then copying the back buffer to the front once drawing is complete.

FrameCount Planes: Used to allow higher animation rates by enabling DRAM local buffer pixel data, such as depth (Z), to be cleared down quickly.

frame buffer: An area of memory containing the displayable color buffers (front, back, left, right, overlay, underlay). This memory is typically separate from the local buffer.

local buffer: An area of memory which may be used to store non-displayable pixel information: depth(Z), stencil, FrameCount and GID planes. This memory is typically separate from the framebuffer.

pixel: Picture element. A pixel comprises the bits in all the buffers (whether stored in the local buffer or framebuffer), corresponding to a particular location in the framebuffer.

stencil buffer: A buffer used to store information about a pixel which controls how subsequent stencilled pixels at the same location may be combined with the current value in the framebuffer. Typically used to mask complex two-dimensional shapes.

## Preferred Chip Embodiment—Overview

The GLINT™ high performance graphics processors combine workstation class 3D graphics acceleration, and state-of-the-art 2D performance in a single chip. All 3D rendering operations are accelerated by GLINT, including Gouraud shading, texture mapping, depth buffering, anti-aliasing, and alpha blending.

The scalable memory architecture of GLINT makes it ideal for a wide range of graphics products, from PC boards to high-end workstation accelerators.

There will be several of the GLINT family of graphics processors: the GLINT 300SX™ is the embodiment of the parent case, and the GLINT 400TX™ is a presently preferred embodiment which is which is described herein in great detail. The two devices are generally compatible, with the 400TX adding local texture storage and texel address generation for all texture modes.

FIG. 2B is an overview of the graphics rendering chip of the presently preferred embodiment (i.e. the GLINT 400TX™).

### General Concept

The overall architecture of the GLINT chip is best viewed using the software paradigm of a message passing system. In this system all the processing blocks are connected in a long pipeline with communication with the adjacent blocks being done through message passing. Between each block there is a small amount of buffering, the size being specific to the local communications requirements and speed of the two blocks.

The message rate is variable and depends on the rendering mode. The messages do not propagate through the system at a fixed rate typical of a more traditional pipeline system. If the receiving block can not accept a message, because its input buffer is full, then the sending block stalls until space is available.

The message structure is fundamental to the whole system as the messages are used to control, synchronize and inform each block about the processing it is to undertake. Each message has two fields—a 32 bit data field and a 9 bit tag field. (This is the minimum width guaranteed, but some local block to block connections may be wider to accommodate more data.) The data field will hold color information, coordinate information, local state information, etc. The tag field is used by each block to identify the message type so it knows how to act on it.

Each block, on receiving a message, can do one of several things:

- Not recognize the message so it just passes it on to the next block.
- Recognize it as updating some local state (to the block) so the local state is updated and the message terminated, i.e. not passed on to the next block.
- Recognize it as a processing action, and if appropriate to the unit, the processing work specific to the unit is done. This may entail sending out new messages such as Color and/or modifying the initial message before sending it on. Any new messages are injected into the message stream before the initial message is forwarded on. Some examples will clarify this.

When the Depth Block receives a message 'new fragment', it will calculate the corresponding depth and do the depth test. If the test passes then the 'new fragment' message is passed to the next unit. If the test fails then the message is modified and passed on. The temptation is not to pass the message on when the test fails (because the pixel is not going to be updated), but other units downstream need to keep their local DDA units in step.

(In the present application, the messages are being described in general terms so as not to be bogged down in detail at this stage. The details of what a 'new fragment' message actually specifies (i.e. coordinate, color information) is left till later. In general, the term "pixel" is used to describe the picture element on the screen or in memory. The term "fragment" is used to describe the part of a polygon or other primitive which projects onto a pixel. Note that a fragment may only cover a part of a pixel.) When the Texture Read Unit (if enabled) gets a 'new fragment' message, it will calculate the texture map addresses, and will accordingly provide 1, 2, 4 or 8 texels to the next unit together with the appropriate number of interpolation coefficients.

Each unit and the message passing are conceptually running asynchronous to all the others. However, in the presently preferred embodiment there is considerable synchrony because of the common clock.

How does the host process send messages? The message data field is the 32 bit data written by the host, and the message tag is the bottom 9 bits of the address (excluding the byte resolution address lines). Writing to a specific address causes the message type associated with that address to be inserted into the message queue. Alternatively, the on-chip DMA controller may fetch the messages from the host's memory.

The message throughput, in the presently preferred embodiment, is 50M messages per second and this gives a fragment throughput of up to 50M per second, depending on what is being rendered. Of course, this rate will predictably be further increased over time, with advances in process technology and clock rates.

### Linkage

The block diagram of FIG. 2A shows how the units are connected together in the GLINT 300SX embodiment, and the block diagram of FIG. 2B shows how the units are connected together in the presently preferred embodiment. Some general points are:

- The following functionality is present in the 400TX, but missing from the 300SX: The Texture Address (TAddr) and Texture Read (TRd) Units are missing. Also, the router and multiplexer are missing from this section, so the unit ordering is Scissor/Stipple, Color DDA, Texture Fog Color, Alpha Test, LB Rd, etc.
- In the embodiment of FIG. 2B, the order of the units can be configured in two ways. The most general order (Router, Color DDA, Texture Unit, Alpha Test, LB Rd, GID/Z/Stencil, LB Wr, Multiplexer) and will work in all modes of OpenGL. However, when the alpha test is disabled it is much better to do the Graphics ID, depth and stencil tests before the texture operations rather than after. This is because the texture operations have a high processing cost and this should not be spent on fragments which are later rejected because of window, depth or stencil tests.
- The loop back to the host at the bottom provides a simple synchronization mechanism. The host can insert a Sync command and when all the preceding rendering has finished the sync command will reach the bottom host interface which will notify the host the sync event has occurred.

### Benefits

The very modular nature of this architecture gives great benefits. Each unit lives in isolation from all the others and

APPENDIX O

has a very well defined set of input and output messages. This allows the internal structure of a unit (or group of units) to be changed to make algorithmic/speed/gate count trade-offs.

The isolation and well defined logical and behavioral interface to each unit allows much better testing and verification of the correctness of a unit.

The message passing paradigm is easy to simulate with software, and the hardware design is nicely partitioned. The architecture is self synchronizing for mode or primitive changes.

The host can mimic any block in the chain by inserting messages which that block would normally generate. These message would pass through the earlier blocks to the mimicked block unchanged and from then onwards to the rest of the blocks which cannot tell the message did not originate from the expected block. This allows for an easy work around mechanism to correct any flaws in the chip. It also allows other rasterization paradigms to be implemented outside of the chip, while still using the chip for the low level pixel operations.

"A Day in the Life of a Triangle"

Before we get too detailed in what each unit does it is worth while looking in general terms at how a primitive (e.g. triangle) passes through the pipeline, what messages are generated, and what happens in each unit. Some simplifications have been made in the description to avoid detail which would otherwise complicate what is really a very simple process. The primitive we are going to look at is the familiar Gouraud shaded Z buffered triangle, with dithering. It is assumed any other state (i.e. depth compare mode) has been set up, but (for simplicity) such other states will be mentioned as they become relevant.

The application generates the triangle vertex information and makes the necessary OpenGL calls to draw it.

The OpenGL server/library gets the vertex information, transforms, clips and lights it. It calculates the initial values and derivatives for the values to interpolate ($X_{left}$, $X_{right}$, red, green, blue and depth) for unit change in dx and $dxdy_{left}$. All these values are in fixed point integer and have unique message tags. Some of the values (the depth derivatives) have more than 32 bits to cope with the dynamic range and resolution so are sent in two halves Finally, once the derivatives, start and end values have been sent to GLINT the 'render triangle' message is sent.

On GLINT: The derivative, start and end parameter messages are received and filter down the message stream to the appropriate blocks. The depth parameters and derivatives to the Depth Unit; the RGB parameters and derivative to the Color DDA Unit; the edge values and derivatives to the Rasterizer Unit.

The 'render triangle' message is received by the rasterizer unit and all subsequent messages (from the host) are blocked until the triangle has been rasterized (but not necessarily written to the frame store). A 'prepare to render' message is passed on so any other blocks can prepare themselves.

The Rasterizer Unit walks the left and right edges of the triangle and fills in the spans between. As the walk progresses messages are send to indicate the direction of the next step: StepX or StepYDomEdge. The data field holds the current (x, y) coordinate. One message is sent per pixel within the triangle boundary. The step messages are duplicated into two groups: an active group and a passive group. The messages always start off in the active group but may be changed to the passive group if this pixel fails one of the tests (e.g. depth) on its path down the message stream. The two groups are distinguished by a single bit in the message tag. The step messages (in either form) are always passed throughout the length of the message stream, and are used by all the DDA units to keep their interpolation values in step. The step message effectively identifies the fragment and any other messages pertaining to this fragment will always precede the step message in the message stream.

The Scissor and Stipple Unit. This unit does 4 tests on the fragment (as embodied by the active step message). The screen scissor test takes the coordinates associated with the step message, converts them to be screen relative (if necessary) and compares them against the screen boundaries. The other three tests (user scissor, line stipple and area stipple) are disabled for this example. If the enabled tests pass then the active step is forwarded onto the next unit, otherwise it is changed into a passive step and then forwarded.

The Color DDA unit responds to an active step message by generating a Color message and sending this onto the next unit. The active step message is then forwarded to the next unit. The Color message holds, in the data field, the current RGBA value from the DDA. If the step message is passive then no Color message is generated. After the Color message is sent (or would have been sent) the step message is acted on to increment the DDA in the correct direction, ready for the next pixel.

Texturing, Fog and Alpha Tests Units are disabled so the messages just pass through these blocks.

In general terms the Local Buffer Read Unit reads the Graphic ID, Stencil and Depth information from the Local Buffer and passes it onto the next unit. More specifically it does:

1. If the step message is passive then no further action occurs.
2. On an active step message it calculates the linear address in the local buffer of the required data. This is done using the (X, Y) position recorded in the step message and locally stored information on the 'screen width' and window base address. Separate read and write addresses are calculated.
3. The addresses are passed to the Local Buffer Interface Unit and the identified local buffer location read. The write address is held for use later.
4. Sometime later the local buffer data is returned and is formatted into a consistent internal format and inserted into a 'Local Buffer Data' message and passed on to the next unit.

   The message data field is made wider to accommodate the maximum Local Buffer width of 52 bits (32 depth, 8 stencil, 4 graphic ID, 8 frame count) and this extra width just extends to the Local Buffer Write block.

   The actual data read from the local buffer can be in several formats to allow narrower width memories to be used in cost sensitive systems. The narrower data is formatted into a consistent internal format in this block.

The Graphic ID, Stencil and Depth Unit just passes the Color message through and stores the LBData message until the step message arrives. A passive step message would just pass straight through. When the active step message is received the internal Graphic ID, stencil and depth values are compared with the ones in the LBData message as specified by this unit's mode information. If the enabled tests pass then the new local buffer data is sent in the LBWriteData message to the next unit and the active step message forwarded. If any of the enabled tests fail then an LBCancelWrite message is sent followed by the equivalent passive step message. The depth DDA is stepped to update the local depth value.

The Local Buffer Write Unit performs any writes which are necessary. The LBWriteData message has its data formatted into the external local buffer format and this is posted to the Local Buffer Interface Unit to be written into the memory (the write address is already waiting in the Local Buffer Interface Unit). The LBWriteCancel message just informs the Local Buffer Interface Unit that the pending write address is no longer needed and can be discarded. The step message is just passed through.

In general terms the Framebuffer Read Unit reads the color information from the framebuffer and passes it onto the next unit. More specifically it does:

1. If the step message is passive then no further action occurs.
2. On an active step message it calculates the linear address in the framebuffer of the required data. This is done using the (X, Y) position recorded in the step message and locally stored information on the 'screen width' and window base address. Separate read and write addresses are calculated.
3. The addresses are passed to the Framebuffer Interface Unit and the identified framebuffer location read. The write address is held for use later.
4. Sometime later the color data is returned and inserted into a 'Frame Buffer Data' message and passed on to the next unit.

   The actual data read from the framestore can be in several formats to allow narrower width memories to be used in cost sensitive systems. The formatting of the data is deferred until the Alpha Blend Unit as it is the only unit which needs to match it up with the internal formats. In this example no alpha blending or logical operations are taking place, so reads are disabled and hence no read address is sent to the Framebuffer Interface Unit. The Color and step messages just pass through.

The Alpha Blend Unit is disabled so just passes the messages through.

The Dither Unit stores the Color message internally until an active step is received. On receiving this it uses the least significant bits of the (X, Y) coordinate information to dither the contents of the Color message. Part of the dithering process is to convert from the internal color format into the format of the framebuffer. The new color is inserted into the Color message and passed on, followed by the step message.

The Logical Operations are disabled so the Color message is just converted into the FBWriteData message (just the tag changes) and forwarded on to the next unit. The step message just passes through.

The Framebuffer Write Unit performs any writes which are necessary.

The FBWriteData message has its data posted to the Framebuffer Interface Unit to be written into the memory (the write address is already waiting in the Framebuffer Interface Unit).

The step message is just passed through.

The Host Out Unit is mainly concerned with synchronization with the host so for this example will just consume any messages which reach this point in the message stream.

This description has concentrated on what happens as one fragment flows down the message stream. It is important to

remember that at any instant in time there are many fragments flowing down the message stream and the further down they reach the more processing has occurred.

Interfacing Between Blocks FIG. 2B shows the FIFO buffering and lookahead connections which are used in the presently preferred embodiment. The FIFOs are used to provide an asynchronous interface between blocks, but are expensive in terms of gate count. Note that most of these FIFOs are only one stage deep (except where indicated), which reduces their area. To maintain performance, lookahead connections are used to accelerate the "startup" of the pipeline. For example, when the Local-Buffer-Read block issues a data request, the Texture/Fog/Color blocks also receive this, and begin to transfer data accordingly. Normally a single-entry deep FIFO cannot be read and written in the same cycle, as the writing side doesn't know that the FIFO is going to be read in that cycle (and hence become eligible to be written). The look-ahead feature give the writing side this insight, so that single-cycle transfer can be achieved. This accelerates the throughput of the pipeline.

## Programming Model

The following text describes the programming model for GLINT.

GLINT as a Register file

The simplest way to view the interface to GLINT is as a flat block of memory-mapped registers (i.e. a register file). This register file appears as part of Region 0 of the PCI address map for GLINT. See the GLINT Hardware Reference Manual for details of this address map.

When a GLINT host software driver is initialized it can map the register file into its address space. Each register has an associated address tag, giving its offset from the base of the register file (since all registers reside on a 64-bit boundary, the tag offset is measured in multiples of 8 bytes). The most straightforward way to load a value into a register is to write the data to its mapped address. In reality the chip interface comprises a 16 entry deep FIFO, and each write to a register causes the written value and the register's address tag to be written as a new entry in the FIFO.

Programming GLINT to draw a primitive consists of writing initial values to the appropriate registers followed by a write to a command register. The last write triggers the start of rendering.

GLINT has approximately 200 registers. All registers are 32 bits wide and should be 32-bit addressed. Many registers are split into bit fields, and it should be noted that bit 0 is the least significant bit.

Register Types

GLINT has three main types of register:

Control Registers

Command Registers

Internal Registers

Control Registers are updated only by the host—the chip effectively uses them as read-only registers. Examples of control registers are the Scissor Clip unit min and max registers. Once initialized by the host, the chip only reads these registers to determine the scissor clip extents.

Command Registers are those which, when written to, typically cause the chip to start rendering (some command registers such as ResetPickResult or Sync do not initiate rendering). Normally, the host will initialize the appropriate control registers and then write to a command register to initiate drawing. There are two types of command registers: begin-draw and continue-draw. Begin-draw commands cause rendering to start with those values specified by the control registers. Continue-draw commands cause drawing to continue with internal register values as they were when the previous drawing operation completed. Making use of continue-draw commands can significantly reduce the amount of data that has to be loaded into GLINT when drawing multiple connected objects such as polylines. Examples of command registers include the Render and ContinueNewLine registers.

For convenience this application will usually refer to "sending a Render command to GLINT" rather than saying (more precisely) "the Render Command register is written to, which initiates drawing".

Internal Registers are not accessible to host software. They are used internally by the chip to keep track of changing values. Some control registers have corresponding internal registers. When a begindraw command is sent and before rendering starts, the internal registers are updated with the values in the corresponding control registers. If a continue-draw command is sent then this update does not happen and drawing continues with the current values in the internal registers. For example, if a line is being drawn then the StartXDom and StartY control registers specify the (x, y) coordinates of the first point in the line. When a begin-draw command is sent these values are copied into internal registers. As the line drawing progresses these internal registers are updated to contain the (x, y) coordinates of the pixel being drawn. When drawing has completed the internal registers contain the (x, y) coordinates of the next point that would have been drawn. If a continue-draw command is now given these final (x, y) internal values are not modified and further drawing uses these values. If a begin-draw command had been used the internal registers would have been reloaded from the StartXDom and StartY registers.

For the most part internal registers can be ignored. It is helpful to appreciate that they exist in order to understand the continue-draw commands.

GLINT I/O Interface

There are a number of ways of loading GLINT registers for a given context:

The host writes a value to the mapped address of the register

The host writes address-tag/data pairs into a host memory buffer and uses the on-chip DMA to transfer this data to the FIFO.

The host can perform a Block Command Transfer by writing address and data values to the FIFO interface registers.

In all cases where the host writes data values directly to the chip (via the register file) it has to worry about FIFO overflow. The InFIFOSpace register indicates how many free entries remain in the FIFO. Before writing to any register the host must ensure that there is enough space left in the FIFO. The values in this register can be read at any time. When using DMA, the DMA controller will automatically ensure that there is room in the FIFO before it performs further transfers. Thus a buffer of any size can be passed to the DMA controller.

FIFO Control

The description above considered the GLINT interface to be a register file. More precisely, when a data value is written to a register this value and the address tag for that register are combined and put into the FIFO as a new entry. The actual register is not updated until GLINT processes this entry. In the case where GLINT is busy performing a time consuming operation (e.g. drawing a large texture mapped polygon), and not draining the FIFO very quickly, it is possible for the FIFO to become full. If a write to a register is performed when the FIFO is full no entry is put into the FIFO and that write is effectively lost.

The input FIFO is **16** entries deep and each entry consists of a tag/data pair. The InFIFOSpace register can be read to determine how many entries are free. The value returned by this register will never be greater than **16**.

To check the status of the FIFO before every write is very inefficient, so it is preferably checked before loading the data for each rectangle. Since the FIFO is **16** entries deep, a further optimization is to wait for all **16** entries to be free after every second rectangle. Further optimizations can be made by moving dXDom, dXSub and dY outside the loop (as they are constant for each rectangle) and doing the FIFO wait after every third rectangle.

The InFIFOSpace FIFO control register contains a count of the number of entries currently free in the FIFO. The chip increments this register for each entry it removes from the FIFO and decrements it every time the host puts an entry in the FIFO.

The DMA Interface

Loading registers directly via the FIFO is often an inefficient way to download data to GLINT. Given that the FIFO can accommodate only a small number of entries, GLINT has to be frequently interrogated to determine how much space is left. Also, consider the situation where a given API function requires a large amount of data to be sent to GLINT. If the FIFO is written directly then a return from this function is not possible until almost all the data has been consumed by GLINT. This may take some time depending on the types of primitives being drawn.

To avoid these problems GLINT provides an on-chip DMA controller which can be used to load data from arbitrary sized (<64K 32-bit words) host buffers into the FIFO. In its simplest form the host software has to prepare a host buffer containing register address tag descriptions and data values. It then writes the base address of this buffer to the DMAAddress register and the count of the number of words to transfer to the DMACount register. Writing to the DMACount register starts the DMA transfer and the host can now perform other work. In general, if the complete set of rendering commands required by a given call to a driver function can be loaded into a single DMA buffer then the driver function can return. Meanwhile, in parallel, GLINT is reading data from the host buffer and loading it into its FIFO. FIFO overflow never occurs since the DMA controller automatically waits until there is room in the FIFO before doing any transfers.

The only restriction on the use of DMA control registers is that before attempting to reload the DMACount register the host software must wait until previous DMA has completed. It is valid to load the DMAAddress register while the previous DMA is in progress since the address is latched internally at the start of the DMA transfer.

Using DMA leaves the host free to return to the application, while in parallel, GLINT is performing the DMA and drawing. This can increase performance significantly over loading a FIFO directly. In addition, some algorithms require that data be loaded multiple times (e.g. drawing the same object across multiple clipping rectangles). Since the GLINT DMA only reads the buffer data, it can be downloaded many times simply by restarting the DMA. This can be very beneficial if composing the buffer data is a time consuming task.

The host can use this hardware capability in various ways. For example, a further optional optimization is to use a double buffered mechanism with two DMA buffers. This allows the second buffer to be filled before waiting for the

APPENDIX O

previous DMA to complete, thus further improving the parallelism between host and GLINT processing. Thus, this optimization is dependent on the allocation of the host memory. If there is only one DMA host buffer then either it is being filled or it is being emptied—it cannot be filled and emptied at the same time, since there is no way for the host and DMA to interact once the DMA transfer has started. The host is at liberty to allocate as many DMA buffers as it wants; two is the minimum to do double buffering, but allocating many small buffers is generally better, as it gives the benefits of double buffering together with low latency time, so GLINT is not idle while large buffer is being filled up. However, use of many small buffers is of course more complicated.

In general the DMA buffer format consists of a 32-bit address tag description word followed by one or more data words. The DMA buffer consists of one or more sets of these formats. The following paragraphs describe the different types of tag description words that can be used.

### DMA Tag Description Format

There are 3 different tag addressing modes for DMA: hold, increment and indexed. The different DMA modes are provided to reduce the amount of data which needs to be transferred, hence making better use of the available DMA bandwidth. Each of these is described in the following sections.

### Hold Format

In this format the 32-bit tag description contains a tag value and a count specifying the number of data words following in the buffer. The DMA controller writes each of the data words to the same address tag. For example, this is useful for image download where pixel data is continuously written to the Color register. The bottom 9 bits specify the register to which the data should be written; the high-order 16 bits specify the number of data words (minus 1) which follow in the buffer and which should be written to the address tag (note that the 2 -bit mode field for this format is zero so a given tag value can simply be loaded into the low order 16 bits).

A special case of this format is where the top 16 bits are zero indicating that a single data value follows the tag (i.e. the 32-bit tag description is simply the address tag value itself). This allows simple DMA buffers to be constructed which consist of tag/data pairs.

### Increment Format

This format is similar to the hold format except that as each data value is loaded the address tag is incremented (the value in the DMA buffer is not changed; GLINT updates an internal copy). Thus, this mode allows contiguous GLINT registers to be loaded by specifying a single 32-bit tag value followed by a data word for each register. The low-order 9 bits specify the address tag of the first register to be loaded. The 2 bit mode field is set to 1 and the high-order 16 bits are set to the count (minus 1) of the number of registers to update. To enable use of this format, the GLINT register file has been organized so that registers which are frequently loaded together have adjacent address tags. For example, the 32 AreaStipplePattern registers can be loaded as follows:

```
AreaStipplePattern0, Count=31, Mode=1
row 0 bits
row 1 bits
. . .
row 31 bits
```

### Indexed Format

GLINT address tags are 9 bit values. For the purposes of the Indexed DMA Format they are organized into major groups and within each group there are up to 16 tags. The low-order 4 bits of a tag give its offset within the group. The high-order 5 bits give the major group number.

The following Register Table lists the individual registers with their Major Group and Offset in the presently preferred embodiment:

Register Table

The following table lists registers by group, giving their tag values and indicating their type. The register groups may be used to improve data transfer rates to GLINT when using DMA.

The following types of register are distinguished:

| Unit | Register | Major Group (hex) | Off-set (hex) | Type |
|---|---|---|---|---|
| Rasterizer | StartXDom | 00 | 0 | Control |
| | dXDom | 00 | 1 | Control |
| | StartXSub | 00 | 2 | Control |
| | dXSub | 00 | 3 | Control |
| | StartY | 00 | 4 | Control |
| | dY | 00 | 5 | Control |
| | Count | 00 | 6 | Control |
| | Render | 00 | 7 | Command |
| | ContinueNewLine | 00 | 8 | Command |
| | ContinueNewDom | 00 | 9 | Command |
| | ContinueNewSub | 00 | A | Command |
| | Continue | 00 | B | Command |
| | FlushSpan | 00 | C | Command |
| | BitMaskPattern | 00 | D | Mixed |
| Rasterizer | PointTable[0–3] | 01 | 0–3 | Control |
| | RasterizerMode | 01 | 4 | Control |
| Scissor Stipple | ScissorMode | 03 | 0 | Control |
| | ScissorMinXY | 03 | 1 | Control |
| | ScissorMaxXY | 03 | 2 | Control |
| | ScreenSize | 03 | 3 | Control |
| | AreaStippleMode | 03 | 4 | Control |
| | LineStippleMode | 03 | 5 | Control |
| | LoadLineStipple Counters | 03 | 6 | Control |
| | UpdateLineStipple Counters | 03 | 7 | Command |
| | SaveLineStipple State | 03 | 8 | Command |
| | WindowOrigin | 03 | 9 | Control |
| Scissor Stipple | AreaStipplePattern[0–31] | 04<br>05 | 0-F<br>0-F | Control |
| Texture Color/Fog | Texe10 | 0C | 0 | Control |
| | Texe11 | 0C | 1 | Control |
| | Texe12 | 0C | 2 | Control |
| | Texe13 | 0C | 3 | Control |
| | Texe14 | 0C | 4 | Control |
| | Texe15 | 0C | 5 | Control |
| | Texe16 | 0G | 6 | Control |
| | Texe17 | 0C | 7 | Control |
| | Interp0 | 0C | 8 | Control |
| | Interp1 | 0C | 9 | Control |
| | Interp2 | 0C | A | Control |
| | Interp3 | 0C | B | Control |
| | Interp4 | 0C | C | Control |
| | TextureFilter | 0C | D | Control |
| Texture/Fog Color | TextureColor Mode | 0D | 0 | Control |
| | TextureEnvColor | 0D | 1 | Control |
| | FogMode | 0D | 2 | Control |
| | FogColor | 0D | 3 | Control |
| | FStart | 0D | 4 | Control |
| | dFdx | 0D | 5 | Control |
| | dFdyDom | 0D | 6 | Control |
| Color DDA | RStart | 0F | 0 | Control |
| | dRdx | 0F | 1 | Control |
| | dRdyDom | 0F | 2 | Control |
| | GStart | 0F | 3 | Control |
| | dGdx | 0F | 4 | Control |
| | dGdyDom | 0F | 5 | Control |

APPENDIX O

-continued

| Unit | Register | Major Group (hex) | Off-set (hex) | Type |
|---|---|---|---|---|
| | BStart | 0F | 6 | Control |
| | dBdx | 0F | 7 | Control |
| | dBdyDom | 0F | 8 | Control |
| | AStart | 0F | 9 | Control |
| | dAdx | 0F | A | Control |
| | dAdyDom | 0F | B | Control |
| | ColorDDAMode | 0F | C | Control |
| | ConstantColor | 0F | D | Control |
| | Color | 0F | E | Mixed |
| Alpha Test | AlphaTestMode | 10 | 0 | Control |
| | AntialiasMode | 10 | 1 | Control |
| Alpha Blend | AlphaBlendMode | 10 | 2 | Control |
| Dither | DitherMode | | 3 | Control |
| Logical Ops | FBSoftwareWrite Mask | 10 | 4 | Control |
| | LogicalOpMode | 10 | 5 | Control |
| | FBWriteData | 10 | 6 | Control |
| LB Read | LBReadMode | 11 | 0 | Control |
| | LBReadFormat | 11 | 1 | Control |
| | LBSourceOffset | 11 | 2 | Control |
| | LBStencil | 11 | 5 | Output |
| | LBDepth | 11 | 6 | Output |
| | LBWindowBase | 11 | 7 | Control |
| LB Write | LBWriteMode | 11 | 8 | Control |
| | LBWriteFormat | 11 | 9 | Control |
| GID/Stencil/ Depth | Window | 13 | 0 | Control |
| | StencilMode | 13 | 1 | Control |
| | StencilData | 13 | 2 | Control |
| | Stencil | 13 | 3 | Mixed |
| | DepthMode | 13 | 4 | Control |
| | Depth | 13 | 5 | Mixed |
| | ZStartU | 13 | 6 | Control |
| | ZStartL | 13 | 7 | Control |
| | dZdxU | 13 | 8 | Control |
| | dZdxL | 13 | 9 | Control |
| | dZdyDomU | 13 | A | Control |
| | dZdyDomL | 13 | B | Control |
| | FastClearDepth | 13 | C | Control |
| FB Read | FBReadMode | 15 | 0 | Control |
| | FBSourceOffset | 15 | 1 | Control |
| | FBPixelOffset | 15 | 2 | Control |
| | FBColor | 15 | 3 | Output |
| | FBWindowBase | 15 | 6 | Control |
| FB Write | FBWriteMode | 15 | 7 | Control |
| | FBHardwareWrite Mask | 15 | 8 | Control |
| | FBBlockColor | 15 | 9 | Control |
| Host Out | FilterMode | 18 | 0 | Control |
| | StatisticMode | 18 | 1 | Control |
| | MinRegion | 18 | 2 | Control |
| | MaxRegion | 18 | 3 | Control |
| | ResetPickResult | 18 | 4 | Command |
| | MinHitRegion | 18 | 5 | Command |
| | MaxHitRegion | 18 | 6 | Command |
| | PickResult | 18 | 7 | Command |
| | Sync | 18 | 8 | Command |

This format allows up to 16 registers within a group to be loaded while still only specifying a single address tag description word.

If the Mode of the address tag description word is set to indexed mode, then the high-order 16 bits are used as a mask to indicate which registers within the group are to be used. The bottom 4 bits of the address tag description word are unused. The group is specified by bits 4 to 8. Each bit in the mask is used to represent a unique tag within the group. If a bit is set then the corresponding register will be loaded. The number of bits set in the mask determines the number of data words that should be following the tag description word in the DMA buffer. The data is stored in order of increasing corresponding address tag.

### DMA Buffer Addresses

Host software must generate the correct DMA buffer address for the GLINT DMA controller. Normally, this means that the address passed to GLINT must be the physical address of the DMA buffer in host memory. The buffer must also reside at contiguous physical addresses as accessed by GLINT. On a system which uses virtual memory for the address space of a task, some method of allocating contiguous physical memory, and mapping this into the address space of a task, must be used.

If the virtual memory buffer maps to non-contiguous physical memory, then the buffer must be divided into sets of contiguous physical memory pages and each of these sets transferred separately. In such a situation the whole DMA buffer cannot be transferred in one go; the host software must wait for each set to be transferred. Often the best way to handle these fragmented transfers is via an interrupt handler.

### DMA Interrupts

GLINT provides interrupt support, as an alternative means of determining when a DMA transfer is complete. If enabled, the interrupt is generated whenever the DMACount register changes from having a non-zero to having a zero value. Since the DMACount register is decremented every time a data item is transferred from the DMA buffer this happens when the last data item is transferred from the DMA buffer.

To enable the DMA interrupt, the DMAInterruptEnable bit must be set in the IntEnable register. The interrupt handler should check the DMAFlag bit in the IntFlags register to determine that a DMA interrupt has actually occurred. To clear the interrupt a word should be written to the IntFlags register with the DMAFlag bit set to one.

This scheme frees the processor for other work while DMA is being completed. Since the overhead of handling an interrupt is often quite high for the host processor, the scheme should be tuned to allow a period of polling before sleeping on the interrupt.

### Output FIFO and Graphics Processor FIFO Interface

To read data back from GLINT an output FIFO is provided. Each entry in this FIFO is 32-bits wide and it can hold tag or data values. Thus its format is unlike the input FIFO whose entries are always tag/data pairs (we can think of each entry in the input FIFO as being 41 bits wide: 9 bits for the tag and 32 bits for the data). The type of data written by GLINT to the output FIFO is controlled by the FilterMode register. This register allows filtering of output data in various categories including the following:

Depth: output in this category results from an image upload of the Depth buffer.

Stencil: output in this category results from an image upload of the Stencil buffer.

Color: output in this category results from an image upload of the framebuffer.

Synchronization: synchronization data is sent in response to a Sync command.

The data for the FilterMode register consists of 2 bits per category. If the least significant of these two bits is set (0×1) then output of the register tag for that category is enabled; if the most significant bit is set (0×2) then output of the data for that category is enabled. Both tag and data output can be enabled at the same time. In this case the tag is written first to the FIFO followed by the data.

For example, to perform an image upload from the framebuffer, the FilterMode register should have data output enabled for the Color category. Then, the rectangular area to be uploaded should be described to the rasterizer. Each pixel that is read from the framebuffer will then be placed into the output FIFO. If the output FIFO becomes full, then GLINT will block internally until space becomes available. It is the programmer's responsibility to read all data from the output FIFO. For example, it is important to know how many pixels should result from an image upload and to read exactly this many from the FIFO.

To read data from the output FIFO the OutputFIFOWords register should first be read to determine the number of entries in the FIFO (reading from the FIFO when it is empty returns undefined data). Then this many 32-bit data items are read from the FIFO. This procedure is repeated until all the expected data or tag items have been read. The address of the output FIFO is described below.

Note that all expected data must be read back. GLINT will block if the FIFO becomes full. Programmers must be careful to avoid the deadlock condition that will result if the host is waiting for space to become free in the input FIFO while GLINT is waiting for the host to read data from the output FIFO.

Graphics Processor FIFO Interface

GLINT has a sequence of 1K×32 bit addresses in the PCI Region 0 address map called the Graphics Processor FIFO Interface. To read from the output FIFO any address in this range can be read (normally a program will choose the first address and use this as the address for the output FIFO). All 32-bit addresses in this region perform the same function: the range of addresses is provided for data transfer schemes which force the use of incrementing addresses.

Writing to a location in this address range provides raw access to the input FIFO. Again, the first address is normally chosen. Thus the same address can be used for both input and output FIFOs. Reading gives access to the output FIFO; writing gives access to the input FIFO.

Writing to the input FIFO by this method is different from writing to the memory mapped register file. Since the register file has a unique address for each register, writing to this unique address allows GLINT to determine the register for which the write is intended. This allows a tag/data pair to be constructed and inserted into the input FIFO. When writing to the raw FIFO address an address tag description must first be written followed by the associated data. In fact, the format of the tag descriptions and the data that follows is identical to that described above for DMA buffers. Instead of using the GLINT DMA it is possible to transfer data to GLINT by constructing a DMA-style buffer of data and then copying each item in this buffer to the raw input FIFO address. Based on the tag descriptions and data written GLINT constructs tag/data pairs to enter as real FIFO entries. The DMA mechanism can be thought of as an automatic way of writing to the raw input FIFO address.

Note, that when writing to the raw FIFO address the FIFO full condition must still be checked by reading the InFIFOSpace register. However, writing tag descriptions does not cause any entries to be entered into the FIFO: such a write simply establishes a set of tags to be paired with the subsequent data. Thus, free space need be ensured only for actual data items that are written (not the tag values). For example, in the simplest case where each tag is followed by a single data item, assuming that the FIFO is empty, then 32 writes are possible before checking again for free space.

Other Interrupts

GLINT also provides interrupt facilities for the following:

Sync: If a Sync command is sent and the Sync interrupt has been enabled then once all rendering has been completed, a data value is entered into the Host Out FIFO, and a Sync interrupt is generated when this value reaches the output end of the FIFO. Synchronization is described further in the next section.

External: this provides the capability for external hardware on a GLINT board (such as an external video timing generator) to generate interrupts to the host processor.

Error: if enabled the error interrupt will occur when GLINT detects certain error conditions , such as an attempt to write to a full FIFO.

Vertical Retrace: if enabled a vertical retrace interrupt is generated at the start of the video blank period.

Each of these are enabled and cleared in a similar way to the DMA interrupt.

Synchronization

There are three main cases where the host must synchronize with GLINT:

before reading back from registers

before directly accessing the framebuffer or the localbuffer via the bypass mechanism

framebuffer management tasks such as double buffering

Synchronizing with GLINT implies waiting for any pending DMA to complete and waiting for the chip to complete any processing currently being performed. The following pseudo-code shows the general scheme:

```
GLINTData data;
// wait for DMA to complete
while (*DMACount != 0) {
   poll or wait for interrupt
}
while (*InFIFOSpace < 2) {
   ; // wait for free space in the FIFO
}
// enable sync output and send the Sync command
data.Word = 0;
data.FilterMode.Synchronization = 0x1;
FilterMode(data.Word);
Sync(0x0);
/* wait for the sync output data */
do {
   while (*OutFIFOWords == 0)
      ; // poll waiting for data in output
FIFO
} while (*OutputFIFO != Sync_tag);
```

Initially, we wait for DMA to complete as normal. We then have to wait for space to become free in the FIFO (since the DMA controller actually loads the FIFO). We need space for 2 registers: one to enable generation of an output sync value, and the Sync command itself. The enable flag can be set at initialization time. The output value will be generated only when a Sync command has actually been sent, and GLINT has then completed all processing.

Rather than polling it is possible to use a Sync interrupt as mentioned in the previous section. As well as enabling the interrupt and setting the filter mode, the data sent in the Sync command must have the most significant bit set in order to generate the interrupt. The interrupt is generated when the tag or data reaches the output end of the Host Out FIFO. Use of the Sync interrupt has to be considered carefully as GLINT will generally empty the FIFO more quickly than it takes to set up and handle the interrupt.

Host Framebuffer Bypass

Normally, the host will access the framebuffer indirectly via commands sent to the GLINT FIFO interface. However, GLINT does provide the whole framebuffer as part of its address space so that it can be memory mapped by an application. Access to the framebuffer via this memory mapped route is independent of the GLINT FIFO.

Drivers may choose to use direct access to the framebuffer for algorithms which are not supported by GLINT. The framebuffer bypass supports big-endian, little-endian and GIB-endian formats.

A driver making use of the framebuffer bypass mechanism should synchronize framebuffer accesses made through the FIFO with those made directly through the memory map. If data is written to the FIFO and then an access is made to the framebuffer, it is possible that the framebuffer access will occur before the commands in the FIFO have been fully processed. This lack of temporal ordering is generally not desirable.

Framebuffer Dimensions and Depth

At reset time the hardware stores the size of the framebuffer in the FBMemoryControl register. This register can be read by software to determine the amount of VRAM on the display adapter. For a given amount of VRAM, software can configure different screen resolutions and off-screen memory regions.

The framebuffer width must be set up in the FBReadMode register. The first 9 bits of this register define 3 partial products which determine the offset in pixels from one scanline to the next. Typically, these values will be worked out at initialization time and a copy kept in software. When this register needs to be modified the software copy is retrieved and any other bits modified before writing to the register.

Once the offset from one scanline to the next has been established, determining the visible screen width and height becomes a clipping issue. The visible screen width and height are set up in the ScreenSize register and enabled by setting the ScreenScissorEnable bit in the ScissorMode register.

The framebuffer depth (8, 16 or 32-bit) is controlled by the FBModeSel register. This register provides a 2 bit field to control which of the three pixel depths is being used. The pixel depth can be changed at any time but this should not be attempted without first synchronizing with GLINT. The FBModeSel register is not a FIFO register and is updated immediately it is written. If GLINT is busy performing rendering operations, changing the pixel depth will corrupt that rendering.

Normally, the pixel depth is set at initialization time. To optimize certain 2D rendering operations it may be desirable to change it at other times. For example, if the pixel depth is normally 8 (or 16) bits, changing the pixel depth to 32 bits for the duration of a bitblt can quadruple (or double) the blt speed, when the bit source and destination edges are aligned on 32 bit boundaries. Once such a blt sequence has been set up the host software must wait and synchronize with GLINT and then reset the pixel depth before continuing with further rendering. It is not possible to change the pixel depth via the FIFO, thus explicit synchronization must always be used.

Host Localbuffer Bypass

As with the framebuffer, the localbuffer can be mapped in and accessed directly. The host should synchronize with GLINT before making any direct access to the localbuffer.

At reset time the hardware saves the size of the localbuffer in the LBMemoryControl register (localbuffer visible region size). In bypass mode the number of bits per pixel is either 32 or 64. This information is also set in the LBMemoryControl register (localbuffer bypass packing). This pixel packing defines the memory offset between one pixel and the next. A further set of 3 bits (localbuffer width) in the LBMemoryControl register defines the number of valid bits per pixel. A typical localbuffer configuration might be 48 bits per pixel but in bypass mode the data for each pixel starts on a 64-bit boundary. In this case valid pixel data will be contained in bits 0 to 47. Software must set the LBReadFormat register to tell GLINT how to interpret these valid bits.

Host software must set the width in pixels of each scanline of the localbuffer in the LBReadMode FIFO register. The first 9 bits of this register define 3 partial products which determine the offset in pixels from one scanline to the next. As with the framebuffer partial products, these values will usually be worked out at initialization time and a copy kept in software. When this register needs to be modified the software copy is retrieved and any other bits modified before writing to the register. If the system is set up so that each pixel in the framebuffer has a corresponding pixel in the localbuffer then this width will be the same as that set for the framebuffer.

The localbuffer is accessible via Regions 1 and 3 of the PCI address map for GLINT. The localbuffer bypass supports big-endian and little-endian formats. These are described in a later section.

Register Read Back

Under some operating environments, multiple tasks will want access to the GLINT chip. Sometimes a server task or driver will want to arbitrate access to GLINT on behalf of multiple applications. In these circumstances, the state of the GLINT chip may need to be saved and restored on each context switch. To facilitate this, the GLINT control registers can be read back. (However, internal and command registers cannot be read back.)

To perform a context switch the host must first synchronize with GLINT. This means waiting for outstanding DMA to complete, sending a Sync command and waiting for the sync output data to appear in the output FIFO. After this the registers can be read back.

To read a GLINT register the host reads the same address which would be used for a write, i.e. the base address of the register file plus the offset value for the register.

Note that since internal registers cannot be read back care must be taken when context switching a task which is making use of continue-draw commands. Continue-draw commands rely on the internal registers maintaining previous state. This state will be destroyed by any rendering work done by a new task. To prevent this, continue-draw commands should be performed via DMA since the context switch code has to wait for outstanding DMA to complete. Alternatively, continue-draw commands can be performed in a non-preemptable code segment.

Normally, reading back individual registers should be avoided. The need to synchronize with the chip can adversely affect performance. It is usually more appropriate to keep a software copy of the register which is updated when the actual register is updated.

Byte Swapping

Internally GLINT operates in little-endian mode. However, GLINT is designed to work with both big- and little-endian host processors. Since the PCIBus specification defines that byte ordering is preserved regardless of the size of the transfer operation, GLINT provides facilities to handle byte swapping. Each of the Configuration Space, Control Space, Framebuffer Bypass and Localbuffer Bypass memory areas have both big and little endian mappings available. The mapping to use typically depends on the endian ordering of the host processor.

The Configuration Space may be set by a resistor in the board design to be either little endian or big endian.

The Control Space in PCI address region 0, is 128K bytes in size, and consists of two 64K sized spaces. The first 64K provides little endian access to the control space registers; the second 64K provides big endian access to the same registers.

The framebuffer bypass consists of two PCI address regions: Region 2 and Region 4. Each is independently configurable to by the Aperture0 and Aperture 1 control registers respectively, to one of three modes: no byte swap, 16-bit swap, full byte swap. Note that the 16 bit mode is needed for the following reason. If the framebuffer is configured for 16-bit pixels and the host is big-endian then simply byte swapping is not enough when a 32-bit access is made (to write two pixels). In this case, the required effect is that the bytes are swapped within each 16-bit word, but the two 16-bit halves of the 32-bit word are not swapped. This preserves the order of the pixels that are written as well as the byte ordering within each pixel. The 16 bit mode is referred to as GIB-endian in the PCI Multimedia Design Guide, version 1.0.

The localbuffer bypass consists of two PCI address regions: Region 1 and Region 3. Each is independently configurable to by the Aperture0 and Aperture 1 control registers respectively, to one of two modes: no byte swap, full byte swap.

To save on the size of the address space required for GLINT, board vendors may choose to turn off access to the big endian regions (3 and 4) by the use of resistors on the board.

There is a bit available in the DMAControl control register to enable byte swapping of DMA data. Thus for big-endian hosts, this control bit would normally be enabled.

Red and Blue Swapping

For a given graphics board the RAMDAC and/or API will usually force a given interpretation for true color pixel values. For example, 32-bit pixels will be interpreted as either ARGB (alpha at byte 3, red at byte 2, green at byte 1 and blue at byte 0) or ABGR (blue at byte 2 and red at byte 0). The byte position for red and blue may be important for software which has been written to expect one byte order or the other, in particular when handling image data stored in a file.

GLINT provides two registers to specify the byte positions of blue and red internally. In the Alpha Blend Unit the AlphaBlendMode register contains a 1-bit field called ColorOrder. If this bit is set to zero then the byte ordering is ABGR; if the bit is set to one then the ordering is ARGB. As well as setting this bit in the Alpha Blend unit, it must also be set in the Color Formatting unit. In this unit the DitherMode register contains a Color Order bit with the same interpretation. The order applies to all of the true color pixel formats, regardless of the pixel depth.

Hardware Data Structures

Some of the hardware data structure implementations used in the presently preferred embodiment will now be described in detail. Of course these examples are provided merely to illustrate the presently preferred embodiment in great detail, and do not necessarily delimit any of the claimed inventions.

Localbuffer

The localbuffer holds the per pixel information corresponding to each displayed pixel and any texture maps. The per pixel information held in the localbuffer are Graphic ID (GID), Depth, Stencil and Frame Count Planes (FCP). The possible formats for each of these fields, and their use are covered individually in the following sections.

The maximum width of the localbuffer is 48 bits, but this can be reduced by changing the external memory configuration, albeit at the expense of reducing the functionality or dynamic range of one or more of the fields.

The localbuffer memory can be from 16 bits (assuming a depth buffer is always needed) to 48 bits wide in steps of 4 bits. The four fields supported in the localbuffer, their allowed lengths and positions are shown in the following table:

| Field | Lengths | Start bit positions |
|---|---|---|
| Depth | 16, 24, 32 | 0 |
| Stencil | 0, 4, 8 | 16, 20, 24, 28, 32 |
| FrameCount | 0, 4, 8 | 16, 20, 24, 28, 32, 36, 40 |
| GID | 0, 4 | 16, 20, 24, 28, 32, 36, 40, 44, 48 |

The order of the fields is as shown with the depth field at the least significant end and GID field at the most significant end. The GID is at the most significant end so that various combinations of the Stencil and FrameCount field widths can be used on a per window basis without the position of the GID fields moving. If the GID field is in a different positions in different windows then the ownership tests become impossible to do.

The GID, FrameCount, Stencil and Depth fields in the localbuffer are converted into the internal format by right justification if they are less than their internal widths, i.e. the unused bits are the most significant bits and they are set to 0.

The format of the localbuffer is specified in two places: the LBReadFormat register and the LBWriteFormat register.

It is still possible to part populate the localbuffer so other combinations of the field widths are possible (i.e. depth field width of 0), but this may give problems if texture maps are to be stored in the localbuffer as well.

Any non-bypass read or write to the localbuffer always reads or writes all 48 bits simultaneously.

GID field

The 4 bit GID field is used for pixel ownership tests to allow per pixel window clipping. Each window using this facility is assigned one of the GID values, and the visible pixels in the window have their GID field set to this value. If the test is enabled the current GID (set to correspond with the current window) is compared with the GID in the localbuffer for each fragment. If they are equal this pixel belongs to the window so the localbuffer and framebuffer at this coordinate may be updated.

Using the GID field for pixel ownership tests is optional and other methods of achieving the same result are:

clip the primitive to the window's boundary (or rectangular tiles which make up the window's area) and render only the visible parts of the primitive

use the scissor test to define the rectangular tiles which make up the window's visible area and render the primitive once per tile (This may be limited to only those tiles which the primitive intersects).

Depth Field

The depth field holds the depth (Z) value associated with a pixel and can be 16, 24 or 32 bits wide.

Stencil Field

The stencil field holds the stencil value associated with a pixel and can be 0, 4 or 8 bits wide.

The width of the stencil buffer is also stored in the StencilMode register and is needed for clamping and masking during the update methods. The stencil compare mask should be set up to exclude any absent bits from the stencil compare operation.

FrameCount Field

The Frame Count Field holds the frame count value associated with a pixel and can be 0, 4 or 8 bits wide. It is used during animation to support a fast clear mechanism to aid the rapid clearing of the depth and/or stencil fields needed at the start of each frame.

In addition to the fast clear mechanism the extent of all updates to the localbuffer and framebuffer can be recorded (MinRegion and MaxRegion registers) and read back (MinHitRegion and MaxHitRegion commands) to give the bounding box of the smallest area to clear. For some applications this will be significantly smaller than the whole window or screen, and hence faster.

The fast clear mechanism provides a method where the cost of clearing the depth and stencil buffers can be amortized over a number of clear operations issued by the application. This works as follows:

The window is divided up into n regions, where n is the range of the frame counter (16 or 256 ). Every time the application issues a clear command the reference frame counter is incremented (and allowed to roll over if it exceeds its maximum value) and the $n^{th}$ region is cleared only. The clear updates the depth and/or stencil buffers to the new values and the frame count buffer with the reference value. This region is much smaller than the full window and hence takes less time to clear.

When the localbuffer is subsequently read and the frame count is found to be the same as the reference frame count (held in the Window register) the localbuffer data is used directly. However, if the frame count is found to be different from the reference frame count (held in the Window register) the data which would have been written, if the localbuffer had been cleared properly, is substituted for the stale data returned from the read. Any new writes to the localbuffer will set the frame count to the reference value so the next read on this pixel works normally without the substitution. The depth data to substitute is held in the FastClearDepth register and the stencil data to substitute is held in the StencilData register (along with other stencil information).

The fast clear mechanism does not present a total solution as the user can elect to clear just the stencil planes or just the depth planes, or both. The situation where the stencil planes only are 'cleared' using the fast clear method, then some rendering is done and then the depth planes are 'cleared' using the fast clear will leave ambiguous pixels in the localbuffer. The driver software will need to catch this situation, and fall back to using a per pixel write to do the second clear. Which field(s) the frame count plane refers to is recorded in the Window register.

When clear data is substituted for real memory data (during normal rendering operations) the depth write mask and stencil write masks are ignored to mimic the OpenGL operation when a buffer is cleared.

Localbuffer Coordinates

The coordinates generated by the rasterizer are 16 bit 2's complement numbers, and so have the range +32767 to −32768. The rasterizer will produce values in this range, but any which have a negative coordinate, or exceed the screen width or height (as programmed into the ScreenSize register) are discarded.

Coordinates can be defined window relative or screen relative and this is only relevant when the coordinate gets converted to an actual physical address in the localbuffer. In general it is expected that the windowing system will use absolute coordinates and the graphics system will use relative coordinates (to be independent of where the window really is).

GUI systems (such as Windows, Windows NT and X) usually have the origin of the coordinate system at the top left corner of the screen but this is not true for all graphics systems. For instance OpenGL uses the bottom left corner as its origin. The WindowOrigin bit in the LBReadMode register selects the top left (0) or bottom left (1) as the origin.

The actual equations used to calculate the localbuffer address to read and write are:

Bottom left origin:
Destination address = LBWindowBase − Y * W + X
Source address =
LBWindowBase − Y*W + X + LBSourceOffset
Top left origin:
Destination address = LBWindowBase + Y * W + X
Source address =
LBWindowBase + Y*W + X + LBSourceOffset

where:

x is the pixel's X coordinate.

Y is the pixel's Y coordinate.

LBWindowBase holds the base address in the localbuffer of the current window.

LBSourceOffset is normally zero except during a copy operation where data is read from one address and written to another address. The offset between source and destination is held in the LBSourceOffset register.

W is the screen width. Only a subset of widths are supported and these are encoded into the PP0, PP1 and PP2 fields in the LBReadMode register.

These address calculations translate a 2D address into a linear address.

The Screen width is specified as the sum of selected partial products so a full multiply operation is not needed. The partial products are selected by the fields PP0, PP1 and PP2 in the LBReadMode register.

For arbitrary width screens, for instance bitmaps in 'off screen' memory, the next largest width from the table must be chosen. The difference between the table width and the bitmap width will be an unused strip of pixels down the right hand side of the bitmap.

Note that such bitmaps can be copied to the screen only as a series of scanlines rather than as a rectangular block. However, often windowing systems store offscreen bitmaps in rectangular regions which use the same stride as the screen. In this case normal bitblts can be used.

Texture Memory

The localbuffer is used to hold textures in the GLINT 400TX variant. In the GLINT 300SX variant the texture information is supplied by the host.

Framebuffer

The framebuffer is a region of memory where the information produced during rasterization is written prior to being displayed. This information is not restricted to color but can include window control data for LUT management and double buffering.

The framebuffer region can hold up to 32 MBytes and there are very few restrictions on the format and size of the individual buffers which make up the video stream. Typical buffers include:

True color or color index main planes,

Overlay planes,

Underlay planes,

Window ID planes for LUT and double buffer management,

Cursor planes.

APPENDIX O

Any combination of these planes can be supported up to a maximum of 32 MBytes, but usually it is the video level processing which is the limiting factor. The following text examines the options and choices available from GLINT for rendering, copying, etc. data to these buffers.

To access alternative buffers either the FBPixelOffset register can be loaded, or the base address of the window held in the FBWindow-Base register can be redefined. This is described in more detail below.

Buffer Organization

Each buffer resides at an address in the framebuffer memory map. For rendering and copying operations the actual buffer addresses can be on any pixel boundary. Display hardware will place some restrictions on this as it will need to access the multiple buffers in parallel to mix the buffers together depending on their relative priority, opacity and double buffer selection. For instance, visible buffers (rather than offscreen bitmaps) will typically need to be on a page boundary.

Consider the following highly configured example with a 1280×1024 double buffered system with 32 bit main planes (RGBA), 8 bit overlay and 4 bits of window control information (WID).

Combining the WID and overlay planes in the same 32 bit pixel has the advantage of reducing the amount of data to copy when a window moves, as only two copies are required—one for the main planes and one for the overlay and WID planes.

Note the position of the overlay and WID planes. This was not an arbitrary choice but one imposed by the (presumed) desire to use the color processing capabilities of GLINT (dither and interpolation) in the overlay planes. The conversion of the internal color format to the external one stored in the framebuffer depends on the size and position of the component. Note that GLINT does not support all possible configurations. For example; if the overlay and WID bits were swapped, then eight bit color index starting at bit 4 would be required to render to the overlay, but this is not supported.

Framebuffer Coordinates

Coordinate generation for the framebuffer is similar to that for the localbuffer, but there are some key differences.

As was mentioned before, the coordinates generated by the rasterizer are 16 bit 2's complement numbers. Coordinates can be defined as window relative or screen relative, though this is only relevant when the coordinate gets converted to an actual physical address in the framebuffer. The WindowOrigin bit in the FBReadMode register selects top left (0) or bottom left (1) as the origin for the framebuffer.

The actual equations used to calculate the framebuffer address to read and write are:

Bottom left origin:
Destination address = FBWindowBase − Y*W + X + FBPixelOffset
Source address = FBWindowBase − Y*W + X + FBPixelOffset + FBSourceOffset
Top left Origin:
Destination address = FBWindowBase + Y*W + X + FBPixelOffset
Source address = FBWindowBase + Y*W + X + FBPixelOffset + FBSourceOffset

These address calculations translate a 2D address into a linear address, so non power of two framebuffer widths (i.e. 1280) are economical in memory.

The width is specified as the sum of selected partial products so a full multiply operation is not needed. The partial products are selected by the fields PP0, PP1 and PP2 in the FBReadMode register. This is the same mechanism as is used to set the width of the localbuffer, but the widths may be set independently.

For arbitrary screen sizes, for instance when rendering to 'off screen' memory such as bitmaps the next largest width from the table must be chosen. The difference between the table width and the bitmap width will be an unused strip of pixels down the right hand side of the bitmap.

Note that such bitmaps can be copied to the screen only as a series of scanlines rather than as a rectangular block. However, often windowing systems store offscreen bitmaps in rectangular regions which use the same stride as the screen. In this case normal bitblts can be used.

Color Formats

The contents of the framebuffer can be regarded in two ways:

As a collection of fields of up to 32 bits with no meaning or assumed format as far as GLINT is concerned. Bit planes may be allocated to control cursor, LUT, multi-buffer visibility or priority functions. In this case GLINT will be used to set and clear bit planes quickly but not perform any color processing such as interpolation or dithering. All the color processing can be disabled so that raw reads and writes are done and the only operations are write masking and logical ops. This allows the control planes to be updated and modified as necessary. Obviously this technique can also be used for overlay buffers, etc. providing color processing is not required.

As a collection of one or more color components. All the processing of color components, except for the final write mask and logical ops are done using the internal color format of 8 bits per red, green, blue and alpha color channels. The final stage before write mask and logical ops processing converts the internal color format to that required by the physical configuration of the framebuffer and video logic. The nomenclature n@m means this component is n bits wide and starts at bit position m in the framebuffer. The least significant bit position is 0 and a dash in a column indicates that this component does not exist for this mode. The ColorOrder is specified by a bit in the DitherMode register.

Some important points to note:

The alpha channel is always associated with the RGB color channels rather than being a separate buffer. This allows it to be moved in parallel and to work correctly in multi-buffer updates and double buffering. If the framebuffer is not configured with an alpha channel (e.g. 24 bit framebuffer width with 8:8:8:8 RGB format) then some of the rendering modes which use the retained alpha buffer cannot be used. In these cases the NoAlphaBuffer bit in the AlphaBlendMode register should be set so that an alpha value of 255 is substituted. For the RGB modes where no alpha channel is present (e.g. 3:3:2) then this substitution is done automatically.

For the Front and Back modes the data value is replicated into both buffers.

All writes to the framebuffer try to update all 32 bits irrespective of the color format. This may not matter if the memory planes don't exist, but if they are being used (as overlay planes, for example) then the write masks (FBSoftwareWriteMask or FBHardwareWriteMask) must be set up to protect the alternative planes.

When reading the framebuffer RGBA components are scaled to their internal width of 8 bits, if needed for alpha blending.

CI values are left justified with the unused bits (if any) set to zero and are subsequently processed as the red compo-

APPENDIX O

nent. The result is replicated into each of the streams G,B and A giving four copies for CI8 and eight copies for CI4.

The 4:4:4:4 Front and Back formats are designed to support 12 bit double buffering with 4 bit Alpha, in a 32 bit system.

The 3:3:2 Front and Back formats are designed to support 8 bit double buffering in a 16 bit system.

The 1:2:1 Front and Back formats are designed to support 4 bit double buffering in an 8 bit system.

It is possible to have a color index buffer at other positions as long as reduced functionality is acceptable. For example a 4 bit CI buffer at bit position 16 can be achieved using write masking and 4:4:4:4 Front format with color interpolation, but dithering is lost.

The format information needs to be stored in two places: the DitherMode register and the AlphaBlendMode register.

|  |  |  | Internal Color Channel |  |  |  |
|---|---|---|---|---|---|---|
|  | Format | Name | R | G | B | A |
| Color | 0 | 8:8:8:8 | 8@0 | 8@8 | 8@16 | 8@24 |
| Order: | 1 | 5:5:5:5 | 5@0 | 5@5 | 5@10 | 5@15 |
| RGB | 2 | 4:4:4:4 | 4@0 | 4@4 | 4@8 | 4@12 |
|  | 3 | 4:4:4:4 | 4@0 | 4@8 | 4@16 | 4@24 |
|  |  | Front | 4@4 | 4@12 | 4@20 | 4@28 |
|  | 4 | 4:4:4:4 | 4@0 | 4@8 | 4@16 | 4@24 |
|  |  | Back | 4@4 | 4@12 | 4@20 | 4@28 |
|  | 5 | 3:3:2 | 3@0 | 3@3 | 2@6 | — |
|  |  | Front | 3@8 | 3@11 | 2@14 |  |
|  | 6 | 3:3:2 | 3@0 | 3@3 | 2@6 | — |
|  |  | Back | 3@8 | 3@11 | 2@14 |  |
|  | 7 | 1:2:1 | 1@0 | 2@1 | 1@3 | — |
|  |  | Front | 1@4 | 2@5 | 1@7 |  |
|  | 8 | 1:2:1 | 1@0 | 2@1 | 1@3 | — |
|  |  | Back | 1@4 | 2@5 | 1@7 |  |
| Color | 0 | 8:8:8:8 | 8@16 | 8@8 | 8@0 | 8@24 |
| Order: | 1 | 5:5:5:5 | 5@10 | 5@5 | 5@0 | 5@15 |
| BGR | 2 | 4:4:4:4 | 4@8 | 4@4 | 4@0 | 4@12 |
|  | 3 | 4:4:4:4 | 4@16 | 4@8 | 4@0 | 4@24 |
|  |  | Front | 4@20 | 4@12 | 4@4 | 4@28 |
|  | 4 | 4:4:4:4 | 4@16 | 4@8 | 4@0 | 4@24 |
|  |  | Back | 4@20 | 4@12 | 4@4 | 4@28 |
|  | 5 | 3:3:2 | 3@5 | 3@2 | 2@0 | — |
|  |  | Front | 3@13 | 3@10 | 2@8 |  |
|  | 6 | 3:3:2 | 3@5 | 3@2 | 2@0 | — |
|  |  | Back | 3@13 | 3@10 | 2@8 |  |
|  | 7 | 1:2:1 | 1@3 | 2@1 | 1@0 | — |
|  |  | Front | 1@7 | 2@5 | 1@4 |  |
|  | 8 | 1:2:1 | 1@3 | 2@1 | 1@0 | — |
|  |  | Back | 1@7 | 2@5 | 1@4 |  |
| CI | 14 | CI8 | 8@0 | 0 | 0 | 0 |
|  | 15 | CI4 | 4@0 | 0 | 0 | 0 |

Overlays and Underlays

In a GUI system there are two possible relationships between the overlay planes (or underlay) and the main planes.

The overlay planes are fixed to the main planes, so that if the window is moved then both the data in the main planes and overlay planes move together.

The overlay planes are not fixed to the main planes but floating, so that moving a window only moves the associated main or overlay planes.

In the fixed case both planes can share the same GID. The pixel offset is used to redirect the reads and writes between the main planes and the overlay (underlay) buffer. The pixel ownership tests using the GID field in the localbuffer work as expected.

In the floating case different GIDs are the best choice, because the same GID planes in the localbuffer can not be used for pixel ownership tests. The alternatives are not to use the GID based pixel ownership tests for one of the buffers but rely on the scissor clipping, or to install a second set of GID planes so each buffer has it's own set. GLINT allows either approach.

If rendering operations to the main and overlay planes both need the depth or stencil buffers, and the windows in each overlap then each buffer will need its own exclusive depth and/or stencil buffers. This is easily achieved with GLINT by assigning different regions in the localbuffer to each of the buffers. Typically this would double the localbuffer memory requirements.

One scenario where the above two considerations do not cause problems, is when the overlay planes are used exclusively by the GUI system, and the main planes are used for the 3D graphics.

VRAM Modes

High performance systems will typically use VRAM for the framebuffer and the extended functionality of VRAM over DRAM can be used to enhance performance for many rendering tasks.

Hardware Write Masks.

These allow write masking in the framebuffer without incurring a performance penalty. If hardware write masks are not available, GLINT must be programmed to read the memory, merge the value with the new value using the write mask, and write it back.

To use hardware write masking, the required write mask is written to the FBHardwareWriteMask register, the FBSoftwareWriteMask register should be set to all 1's, and the number of framebuffer reads is set to 0 (for normal rendering). This is achieved by clearing the ReadSource and ReadDestination enables in the FBReadMode register.

To use software write masking, the required write mask is written to the FBSoftwareWriteMask register and the number of framebuffer reads is set to 1 (for normal rendering). This is achieved by setting the ReadDestination enable in the FBReadMode register.

Block Writes Block writes cause consecutive pixels in the framebuffer to be written simultaneously. This is useful when filling large areas but does have some restrictions:

No pixel level clipping is available;

No depth or stencil testing can be done;

All the pixels must be written with the same value so no color interpolation, blending, dithering or logical ops can be done; and

The area is defined in screen relative coordinates.

Block writes are not restricted to rectangular areas and can be used for any trapezoid. Hardware write masking is available during block writes.

The following registers need to be set up before block fills can be used:

FBBlockColor register with the value to write to each pixel; and

FBWriteMode register with the block width field.

Sending a Render command with the PrimitiveType field set to "trapezoid" and the FastFillEnable and FastFillIncrement fields set up will then cause block filling of the area. Note that during a block fill of a trapezoid any inappropriate state is ignored so even if color interpolation, depth testing and logical ops, for example, are enabled they have no effect.

The block sizes supported are 8, 16 and 32 pixels. GLINT takes care of filling any partial blocks at the end of spans.

Graphics Programming

GLINT provides a rich variety of operations for 2D and 3D graphics supported by its Pipelined architecture.

APPENDIX O

The Graphics Pipeline

This section describes each of the units in the graphics Pipeline. FIG. 2C shows a schematic of the pipeline. In this diagram, the localbuffer contains the pixel ownership values (known as Graphic IDs), the FrameCount Planes (FCP), Depth (Z) and Stencil buffer. The framebuffer contains the Red, Green, Blue and Alpha bitplanes. The operations in the Pipeline include:

Rasterizer scan converts the given primitive into a series of fragments for processing by the rest of the pipeline.

Scissor Test clips out fragments that lie outside the bounds of a user defined scissor rectangle and also performs screen clipping to stop illegal access outside the screen memory.

Stipple Test masks out certain fragments according to a specified pattern. Line and area stipples are available.

Color DDA is responsible for generating the color information (True Color RGBA or Color Index(CI)) associated with a fragment.

Texture is concerned with mapping a portion of a specified image (texture) onto a fragment. The process involves filtering to calculate the texture color, and application which applies the texture color to the fragment color.

Fog blends a fog color with a fragment's color according to a given fog factor. Fogging is used for depth cuing images and to simulate atmospheric fogging.

Antialias Application combines the incoming fragment's alpha value with its coverage value when anti aliasing is enabled.

Alpha Test conditionally discards a fragment based on the outcome of a comparison between the fragments alpha value and a reference alpha value.

Pixel Ownership is concerned with ensuring that the location in the framebuffer for the current fragment is owned by the current visual. Comparison occurs between the given fragment and the Graphic ID value in the localbuffer, at the corresponding location, to determine whether the fragment should be discarded.

Stencil Test conditionally discards a fragment based on the outcome of a test between the given fragment and the value in the stencil buffer at the corresponding location. The stencil buffer is updated dependent on the result of the stencil test and the depth test.

Depth Test conditionally discards a fragment based on the outcome of a test between the depth value for the given fragment and the value in the depth buffer at the corresponding location. The result of the depth test can be used to control the updating of the stencil buffer.

Alpha Blending combines the incoming fragment's color with the color in the framebuffer at the corresponding location.

Color Formatting converts the fragment's color into the format in which the color information is stored in the framebuffer.

This may optionally involve dithering.

The Pipeline structure of GLINT is very efficient at processing fragments, for example, texture mapping calculations are not actually performed on fragments that get clipped out by scissor testing. This approach saves substantial computational effort. The pipelined nature does however mean that when programming GLINT one should be aware of what all the pipeline stages are doing at any time. For example, many operations require both a read and/or write to the localbuffer and framebuffer; in this case it is not sufficient to set a logical operation to XOR and enable logical operations, but it is also necessary to enable the reading/writing of data from/to the framebuffer.

A Gouraud Shaded Triangle

We may now revisit the "day in the life of a triangle" example given above, and review the actions taken in greater detail. Again, the primitive being rendered will be a Gouraud shaded, depth buffered triangle. For this example assume that the triangle is to be drawn into a window which has its colormap set for RGB as opposed to color index operation. This means that all three color components; red, green and blue, must be handled. Also, assume the coordinate origin is bottom left of the window and drawing will be from top to bottom. GLINT can draw from top to bottom or bottom to top.

Consider a triangle with vertices, $v_1$, $v_2$ and $v_3$ where each vertex comprises X, Y and Z coordinates. Each vertex has a different color made up of red, green and blue (R, G and B) components. The alpha component will be omitted for this example.

Initialization

GLINT requires many of its registers to be initialized in a particular way, regardless of what is to be drawn, for instance, the screen size and appropriate clipping must be set up. Normally this only needs to be done once and for clarity this example assumes that all initialization has already been done.

Other state will change occasionally, though not usually on a per primitive basis, for instance enabling Gouraud shading and depth buffering.

Dominant and Subordinate Sides of a Triangle

As shown in FIG. 4A, the dominant side of a triangle is that with the greatest range of Y values. The choice of dominant side is optional when the triangle is either flat bottomed or flat topped.

GLINT always draws triangles starting from the dominant edge towards the subordinate edges. This simplifies the calculation of set up parameters as will be seen below.

These values allow the color of each fragment in the triangle to be determined by linear interpolation. For example, the red component color value of a fragment at XN, Ym could be calculated by:

adding $dRdy_{13}$, for each scanline between $Y_1$ and $Y_n$, to $R_1$.

then adding dRdx for each fragment along scanline $Y_n$ from the left edge to $X_n$.

The example chosen has the 'knee,' i.e. vertex 2, on the right hand side, and drawing is from left to right. If the knee were on the left side (or drawing was from right to left), then the Y deltas for both the subordinate sides would be needed to interpolate the start values for each color component (and the depth value) on each scanline. For this reason GLINT always draws triangles starting from the dominant edge and towards the subordinate edges. For the example triangle, this means left to right.

Register Set Up for Color Interpolation

For the example triangle, the GLINT registers must be set as follows, for color interpolation. Note that the format for color values is 24 bit, fixed point 2's complement.

```
// Load the color start and delta values to draw
// a triangle
RStart (R1)
GStart (G1)
BStart (B1)
dRdyDom (dRdy13)          // To walk up the dominant edge
dGdyDom (dGdy13)
dBdyDom (dBdy13)
dRdx (dRdx)               // To walk along the scanline
```

-continued

```
dGdx (dGdx)
dBdx (dBdx)
```

Calculating Depth Gradient Values

To draw from left to right and top to bottom, the depth gradients or deltas) required for interpolation are:

$$dZdy_{13} = \frac{Z_3 - Z_1}{Y_3 - Y_1}$$

And from the plane equation:

$$dZdx = \left\{ (Z_1 - Z_3)\frac{(Y_2 - Y_3)}{c} \right\} - \left\{ (Z_2 - Z_3)\frac{(Y_3 - Y_1)}{c} \right\}$$

where

$$c = |(X_1 - X_3)(Y_2 - Y_3) - (X_2 - X_3)(Y_1 - Y_1)|$$

The divisor, shown here as c, is the same as for color gradient values. The two deltas $dZdyl_{13}$ and dZdx allow the Z value of each fragment in the triangle to be determined by linear interpolation, just as for the color interpolation.

Register Set Up for Depth Testing

Internally GLINT uses fixed point arithmetic. Each depth value must be converted into a 2's complement 32.16 bit fixed point number and then loaded into the appropriate pair of 32 bit registers. The 'Upper' or 'U' registers store the integer portion, whilst the 'Lower' or 'L' registers store the 16 fractional bits, left justified and zero filled.

For the example triangle, GLINT would need its registers set up as follows:

```
// Load the depth start and delta values
// to draw a triangle
ZStartU (Z1_MS)
ZStartL (Z1_LS)
dZdyDomU (dZdy13_MS)
dZdyDomL (dZdy13_LS)
dZdxU (dZdx_MS)
dZdxL (dZdx_LS)
```

Calculating the Slopes for each Side

GLINT draws filled shapes such as triangles as a series of spans with one span per scanline. Therefore it needs to know the start and end X coordinate of each span. These are determined by 'edge walking'. This process involves adding one delta value to the previous span's start X coordinate and another delta value to the previous span's end x coordinate to determine the X coordinates of the new span. These delta values are in effect the slopes of the triangle sides. To draw from left to right and top to bottom, the slopes of the three sides are calculated as:

$$dX_{13} = \frac{X_3 - X_1}{Y_3 - Y_1}$$

$$dX_{12} = \frac{X_2 - X_1}{Y_2 - Y_1}$$

$$dX_{23} = \frac{X_3 - X_2}{Y_3 - Y_2}$$

This triangle will be drawn in two parts, top down to the 'knee' (i.e. vertex 2), and then from there to the bottom. The dominant side is the left side so for the top half:

$dXDom=dX_{13}$

$dXSub=dX_{12}$

The start X,Y, the number of scanlines, and the above deltas give GLINT enough information to edge walk the top half of the triangle. However, to indicate that this is not a flat topped triangle (GLINT is designed to rasterize screen aligned trapezoids and flat topped triangles), the same start position in terms of X must be given twice as StartXDom and StartXSub.

To edge walk the lower half of the triangle, selected additional information is required. The slope of the dominant edge remains unchanged, but the subordinate edge slope needs to be set to:

$dXSub=dX_{23}$

Also the number of scanlines to be covered from $Y_2$ to $Y_3$ needs to be given. Finally to avoid any rounding errors accumulated in edge walking to $X_2$ (which can lead to pixel errors), StartXSub must be set to $X_2$.

Rasterizer Mode

The GLINT rasterizer has a number of modes which have effect from the time they are set until they are modified and can thus affect many primitives. In the case of the Gouraud shaded triangle the default value for these modes are suitable.

Subpixel Correction

GLINT can perform subpixel correction of all interpolated values when rendering aliased trapezoids. This correction ensures that any parameter (color/depth/texture/fog) is correctly sampled at the center of a fragment. Subpixel correction will generally always be enabled when rendering any trapezoid which is smooth shaded, textured, fogged or depth buffered. Control of subpixel correction is in the Render command register described in the next section, and is selectable on a per primitive basis.

Rasterization

GLINT is almost ready to draw the triangle. Setting up the registers as described here and sending the Render command will cause the top half of the example triangle to be drawn.

For drawing the example triangle, all the bit fields within the Render command should be set to 0 except the PrimitiveType which should be set to trapezoid and the SubPixelCorrectionEnable bit which should be set to TRUE.

```
// Draw triangle with knee
// Set deltas
StartXDom ($X_1$<<16)   //   Converted to 16.16 fixed
point
dXDom ((($X_3$ - $X_1$)<<16)/($Y_3$ - $Y_1$))
StartXSub ($X_1$<<16)
dXSub ((($X_2$ - $X_1$)<<16)/($Y_2$ - $Y_1$))
StartY ($Y_1$<<16)
dY (-1<<16)
Count ($Y_1$ - $Y_2$)
// Set the render command mode
render.PrimitiveType = GLINT_TRAPEZOID_PRIMITIVE
render.SubPixelCorrectionEnable = TRUE
// Draw the top half of the triangle
Render(render)
```

After the Render command has been issued, the registers in GLINT can immediately be altered to draw the lower half of the triangle. Note that only two registers need be loaded and the command ContinueNewSub sent. Once GLINT has received ContinueNewSub, drawing of this sub-triangle will begin.

**APPENDIX O**

```
// Setup the delta and start for the new edge
StartXSub (X2<<16)
dXSub (((X3 - X2)<<16)/(Y3 - Y2))
// Draw sub-triangle
ContinueNewSub (Y2 - Y3)   // Draw lower half
```

Rasterizer Unit

The rasterizer decomposes a given primitive into a series of fragments for processing by the rest of the Pipeline.

GLINT can directly rasterize:

aliased screen aligned trapezoids

aliased single pixel wide lines

aliased single pixel points

antialiased screen aligned trapezoids

antialiased circular points

All other primitives are treated as one or more of the above, for example an antialiased line is drawn as a series of antialiased trapezoids.

Trapezoids GLINT's basic area primitives are screen aligned trapezoids. These are characterized by having top and bottom edges parallel to the X axis. The side edges may be vertical (a rectangle), but in general will be diagonal. The top or bottom edges can degenerate into points in which case we are left with either flat topped or flat bottomed triangles. Any polygon can be decomposed into screen aligned trapezoids or triangles. Usually, polygons are decomposed into triangles because the interpolation of values over non-triangular polygons is ill defined. The rasterizer does handle flat topped and flat bottomed 'bow tie' polygons which are a special case of screen aligned trapezoids.

To render a triangle, the approach adopted to determine which fragments are to be drawn is known as 'edge walking'. Suppose the aliased triangle shown in FIG. 4A was to be rendered from top to bottom and the origin was bottom left of the window. Starting at (X1, Y1) then decrementing Y and using the slope equations for edges 1–2 and 1–3, the intersection of each edge on each scanline can be calculated. This results in a span of fragments per scanline for the top trapezoid. The same method can be used for the bottom trapezoid using slopes 2–3 and 1–3.

It is usually required that adjacent triangles or polygons which share an edge or vertex are drawn such that pixels which make up the edge or vertex get drawn exactly once. This may be achieved by omitting the pixels down the left or the right sides and the pixels along the top or lower sides. GLINT has adopted the convention of omitting the pixels down the right hand edge. Control of whether the pixels along the top or lower sides are omitted depends on the start Y value and the number of scanlines to be covered. With the example, if StartY =Y1 and the number of scanlines is set to Y1–Y2, the lower edge of the top half of the triangle will be excluded. This excluded edge will get drawn as part of the lower half of the triangle.

To minimize delta calculations, triangles may be scan converted from left to right or from right to left. The direction depends on the dominant edge, that is the edge which has the maximum range of Y values. Rendering always proceeds from the dominant edge towards the relevant subordinate edge. In the example above, the dominant edge is 1–3 so rendering will be from right to left.

The sequence of actions required to render a triangle (with a 'knee') is:

Load the edge parameters and derivatives for the dominant edge and the first subordinate edges in the first triangle.

Send the Render command. This starts the scan conversion of the first triangle, working from the dominant edge. This means that for triangles where the knee is on the left we are scanning right to left, and vice versa for triangles where the knee is on the right.

Load the edge parameters and derivatives for the remaining subordinate edge in the second triangle.

Send the ContinueNewSub command. This starts the scan conversion of the second triangle.

Pseudocode for the above example is:

```
// Set the rasterizer mode to the default
RasterizerMode (0)
// Setup the start values and the deltas.
// Note that the X and Y coordinates are converted
// to 16.16 format
StartXDom (X1<<16)
dXDom (((X3- X1)<<16)/(Y3 - Y1))
StartXSub (X1<<16)
dXSub (((X2- X1)<<16)/(Y2 - Y1))
StartY (Y1<<16)
dY (-1<16) // Down the screen
Count (Y1 - Y2)
// Set the render mode to aliased primitive with
// subpixel correction.
render.PrimitiveType = GLINT_TRAPEZOID_PRIMITIVE
render.SubpixelCorrectionEnable = GLINT_TRUE
render.AntialiasEnable = GLINT_DISABLE
// Draw top half of the triangle
Render(render)
// Set the start and delta for the second half of
// the triangle.
StartXSub (X2<<16)
dXSub (((X3- X2)<<16)/(Y3 - Y2))
// Draw lower half of triangle
ContinueNewSub (abs(Y2 - Y3))
```

After the Render command has been sent, the registers in GLINT can immediately be altered to draw the second half of the triangle. For this, note that only two registers need be loaded and the command ContinueNewSub be sent. Once drawing of the first triangle is complete and GLINT has received the ContinueNewSub command, drawing of this sub-triangle will start. The ContinueNewSub command register is loaded with the remaining number of scanlines to be rendered.

Lines

Single pixel wide aliased lines are drawn using a DDA algorithm, so all GLINT needs by way of input data is StartX, StartY, dX, dY and length.

For polylines, a ContinueNewLine command (analogous to the Continue command used at the knee of a triangle) is used at vertices.

When a Continue command is issued some error will be propagated along the line. To minimize this, a choice of actions are available as to how the DDA units are restarted on the receipt of a Continue command. It is recommended that for OpenGL rendering the ContinueNewLine command is not used and individual segments are rendered.

Antialiased lines, of any width, are rendered as antialiased screen-aligned trapezoids.

Points

GLINT supports a single pixel aliased point primitive. For points larger than one pixel trapezoids should be used. In this case the PrimitiveType field in the Render command should be set to equal GLINT_POINT_PRIMITIVE.

Anti aliasing

GLINT uses a subpixel point sampling algorithm to antialias primitives. GLINT can directly rasterize antialiased trapezoids and points. Other primitives are composed from these base primitives.

APPENDIX O

The rasterizer associates a coverage value with each fragment produced when antialiasing. This value represents the percentage coverage of the pixel by the fragment. GLINT supports two levels of antialiasing quality:

normal, which represents 4×4 pixel subsampling

high, which represents 8×8 pixel subsampling.

Selection between these two is made by the AntialiasingQuality bit within the Render command register.

When rendering antialiased primitives with GLINT the FlushSpan command is used to terminate rendering of a primitive. This is due to the nature of GLINT antialiasing. When a primitive is rendered which does not happen to complete on a scanline boundary, GLINT retains antialiasing information about the last sub-scanline(s) it has processed, but does not generate fragments for them unless a FlushSpan command is received. The commands ContinueNewSub, ContinueNewDom or Continue can then be used, as appropriate, to maintain continuity between adjacent trapezoids. This allows complex antialiased primitives to be built up from simple trapezoids or points.

To illustrate this consider using screen aligned trapezoids to render an antialiased line. The line will in general consist of three screen aligned trapezoids as shown in FIG. 4B. This FIG. illustrates the sequence of rendering an Antialiased Line primitive. Note that the line has finite width.

The procedure to render the line is as follows:

```
// Setup the blend and coverage application units
// as appropriate - not shown
// In this example only the edge deltas are shown
// loaded into registers for clarity. In reality
// start X and Y values are required
// Render Trapezoid A
dY(1<<16)
dXDom(dXDom1<<16)
dXSub(dXSub1<<16)
Count(count1)
render.PrimitiveType = GLINT_TRAPEZOID
remder.AntialiasEnable = GLINT_TRUE
render.AntialiasQuality = GLINT_MIN_ANTIALIAS
render.CoverageEnable = GLINT_TRUE
Render(render)
// Render Trapezoid B
dXSub(dXSub2<<16)
ContinueNewSub(count2)
// Render Trapezoid C
dXDom(dXDom2<<16)
ContinueNewDom(count3)
// Now we have finished the primitive flush out
// the last scanline
FlushSpan( )
```

Note that when rendering antialiased primitives, any count values should be given in subscanlines, for example if the quality is 4×4 then any scanline count must be multiplied by 4 to convert it into a subscanline count. Similarly, any delta value must be divided by 4.

When rendering, AntialiasEnable must be set in the Antialias-Mode register to scale the fragments color by the coverage value. An appropriate blending function should also be enabled.

Note, when rendering antialiased bow-ties, the coverage value on the cross-over scanline may be incorrect.

GLINT can render small antialiased points. Antialiased points are treated as circles, with the coverage of the boundary fragments ranging from 0% to 100%. GLINT supports:

point radii of 0.5 to 16.0 in steps of 0.25 for 4×4 antialiasing

point radii of 0.25 to 8.0 in steps of 0.125 for 8×8 antialiasing

To scan convert an antialiased point as a circle, GLINT traverses the boundary in sub scanline steps to calculate the coverage value. For this, the sub-scanline intersections are calculated incrementally using a small table. The table holds the change in X for a step in Y. Symmetry is used so the table only holds the delta values for one quadrant.

StartXDom, StartXSub and StartY are set to the top or bottom of the circle and dY set to the subscanline step. In the case of an even diameter, the last of the required entries in the table is set to zero.

Since the table is configurable, point shapes other than circles can be rendered. Also if the StartXDom and StartXSub values are not coincident then horizontal thick lines with rounded ends, can be rendered.

### Block Write Operation

GLINT supports VRAM block writes with block sizes of 8, 16 and 32 pixels. The block write method does have some restrictions: None of the per pixel clipping, stipple, or fragment operations are available with the exception of write masks. One subtle restriction is that the block coordinates will be interpreted as screen relative and not window relative when the pixel mask is calculated in the Framebuffer Units.

Any screen aligned trapezoid can be filled using block writes, not just rectangles.

The use of block writes is enabled by setting the FastFillEnable and FastFillIncrement fields in the Render command register. The framebuffer write unit must also be configured.

Note only the Rasterizer, Framebuffer Read and Framebuffer Write units are involved in block filling. The other units will ignore block write fragments, so it is not necessary to disable them.

### Sub Pixel Precision and Correction

As the rasterizer has 16 bits of fraction precision, and the screen width used is typically less than $2^{16}$ wide a number of bits called subpixel precision bits, are available. Consider a screen width of 4096 pixels. This figure gives a subpixel precision of 4 bits ($4096=2^{12}$). The extra bits are required for a number of reasons:

antialiasing (where vertex start positions can be supplied to subpixel precision)

when using an accumulation buffer (where scans are rendered multiple times with jittered input vertices)

for correct interpolation of parameters to give high quality shading as described below

GLINT supports subpixel correction of interpolated values when rendering aliased trapezoids. Subpixel correction ensures that all interpolated parameters associated with a fragment (color, depth, fog, texture) are correctly sampled at the fragment's center. This correction is required to ensure consistent shading of objects made from many primitives. It should generally be enabled for all aliased rendering which uses interpolated parameters.

Subpixel correction is not applied to antialiased primitives.

### Bitmaps

A Bitmap primitive is a trapezoid or line of ones and zeros which control which fragments are generated by the rasterizer. Only fragments where the corresponding Bitmap bit is set are submitted for drawing. The normal use for this is in drawing characters, although the mechanism is available for all primitives. The Bitmap data is packed contiguously into 32 bit words so that rows are packed adjacent to each other. Bits in the mask word are by default used from the least significant end towards the most significant end and are applied to pixels in the order they are generated in.

APPENDIX O

The rasterizer scans through the bits in each word of the Bitmap data and increments the X,Y coordinates to trace out the rectangle of the given width and height. By default, any set bits (1) in the Bitmap cause a fragment to be generated, any reset bits (0) cause the fragment to be rejected.

The selection of bits from the BitMaskPattern register can be mirrored, that is, the pattern is traversed from MSB to LSB rather than LSB to MSB. Also, the sense of the test can be reversed such that a set bit causes a fragment to be rejected and vice versa. This control is found in the RasterizerMode register.

When one Bitmap word has been exhausted and pixels in the rectangle still remain then rasterization is suspended until the next write to the BitMaskPattern register. Any unused bits in the last Bitmap word are discarded.

Image Copy/Upload/Download

GLINT supports three "pixel rectangle" operations: copy, upload and download. These can apply to the Depth or Stencil Buffers (held within the localbuffer) or the framebuffer.

It should be emphasized that the GLINT copy operation moves RAW blocks of data around buffers. To zoom or re-format data, in the presently preferred embodiment, external software must upload the data, process it and then download it again.

To copy a rectangular area, the rasterizer would be configured to render the destination rectangle, thus generating fragments for the area to be copied. GLINT copy works by adding a linear offset to the destination fragment's address to find the source fragment's address.

Note that the offset is independent of the origin of the buffer or window, as it is added to the destination address. Care must be taken when the source and destination overlap to choose the source scanning direction so that the overlapping area is not overwritten before it has been moved. This may be done by swapping the values written to the StartXDom and StartXSub, or by changing the sign of dY and setting StartY to be the opposite side of the rectangle.

Localbuffer copy operations are correctly tested for pixel ownership. Note that this implies two reads of the localbuffer, one to collect the source data, and one to get the destination GID for the pixel ownership test.

GLINT buffer upload/downloads are very similar to copies in that the region of interest is generated in the rasterizer. However, the localbuffer and framebuffer are generally configured to read or to write only, rather than both read and write. The exception is that an image load may use pixel ownership tests, in which case the localbuffer destination read must be enabled.

Units which can generate fragment values, the color DDA unit for example, should generally be disabled for any copy/upload/download operations.

Warning: During image upload, all the returned fragments must be read from the Host Out FIFO, otherwise the GLINT pipeline will stall. In addition it is strongly recommended that any units which can discard fragments (for instance the following tests: bitmask, alpha, user scissor, screen scissor, stipple, pixel ownership, depth, stencil), are disabled otherwise a shortfall in pixels returned may occur, also leading to deadlock.

Note that because the area of interest in copy/upload/download operations is defined by the rasterizer, it is not limited to rectangular regions.

Color formatting can be used when performing image copies, uploads and downloads. This allows data to be formatted from, or to, any of the supported GLINT color formats.

Rasterizer Mode

A number of long-term modes can be set using the Rasterizer-Mode register, these are:

Mirror BitMask: This is a single bit flag which specifies the direction bits are checked in the BitMask register. If the bit is reset, the direction is from least significant to most significant (bit 0 to bit 31), if the bit is set, it is from most significant to least significant (from bit 31 to bit 0).

Invert BitMask: This is a single bit which controls the sense of the accept/reject test when using a Bitmask. If the bit is reset then when the BitMask bit is set the fragment is accepted and when it is reset the fragment is rejected. When the bit is set the sense of the test is reversed.

Fraction Adjust: These 2 bits control the action taken by the rasterizer on receiving a ContinueNewLine command. As GLINT uses a DDA algorithm to render lines, an error accumulates in the DDA value. GLINT provides for greater control of the error by doing one of the following:

leaving the DDA running, which means errors will be propagated along a line.

or setting the fraction bits to either zero, a half or almost a half (0×7FFF).

Bias Coordinates: Only the integer portion of the values in the DDAs are used to generate fragment addresses. Often the actual action required is a rounding of values, this can be achieved by setting the bias coordinate bit to true which will automatically add almost a half (0×7FFF) to all input coordinates.

Rasterizer Unit Registers

Real coordinates with fractional parts are provided to the rasterizer in 2's complement 16 bit integer, 16 bit fraction format. The following Table lists the command registers which control the rasterizer unit:

| Register Name | Description |
|---|---|
| Render | Starts the rasterization process |
| ContinueNewDom | Allows the rasterization to continue with a new dominant edge. The dominant edge DDA is reloaded with the new parameters. The subordinate edge is carried on from the previous trapezoid. This allows any convex polygon to be broken down into a collection of trapezoids, with continuity maintained across boundaries.<br>The data field holds the number of scanlines (or sub scanlines) to fill. Note this count does not get loaded into the Count register. |
| ContinueNewSub | Allows the rasterization to continue with a new subordinate edge. The subordinate DDA is reloaded with the new |

APPENDIX O

-continued

| Register Name | Description |
|---|---|
| | parameters. The dominant edge is carried on from the previous trapezoid. This is useful when scan converting triangles with a 'knee' (i.e. two subordinate edges).<br>The data field holds the number of scanlines (or sub scanlines) to fill. Note this count does not get loaded into the Count register. |
| Continue | Allows the rasterization to continue after new delta value(s) have been loaded, but does not cause either of the trapezoid's edge DDAs to be reloaded.<br>The data field holds the number of scanlines (or sub scanlines) to fill. Note this count does not get loaded into the Count register. |
| ContinueNewLine | Allows the rasterization to continue for the next segment in a polyline. The XY position is carried on from the previous line, but the fraction bits in the DDAs can be: kept, set to zero, half, or nearly one half, under control of the RasterizerMode.<br>The data field holds the number of scanlines to fill. Note this count does not get loaded into the Count register.<br>The use of ContinueNewLine is not recommended for OpenGL because the DDA units will start with a slight error as compared with the value they would have been loaded with for the second and subsequent segments. |
| FlushSpan | Used when antialiasing to force the last span out when not all sub spans may be defined. |

The following Table shows the control registers of the rasterizer, in the presently preferred embodiment:

| RasterizerMode | Defines the long term mode of operation of the rasterizer. |
|---|---|
| StartXDom | Initial X value for the dominant edge in trapezoid filling, or initial X value in line drawing. |
| dXDom | Value added when moving from one scanline (or sub scanline) to the next for the dominant edge in trapezoid filling.<br>Also holds the change in X when plotting lines so for Y major lines this will be some fraction (dx/dy), otherwise it is normally ± 1.0, depending on the required scanning direction. |
| StartXSub | Initial X value for the subordinate edge. |
| dXSub | Value added when moving from one scanline (or sub scanline) to the next for the subordinate edge in trapezoid filling. |
| StartY | Initial scanline (or sub scanline) in trapezoid filling, or initial Y position for line drawing. |
| dY | Value added to Y to move from one scanline to the next. For X major lines this will be some fraction (dy/dx), otherwise it is normally ± 1.0, depending on the required scanning direction. |
| Count | Number of pixels in a line.<br>Number of scanlines in a trapezoid.<br>Number of sub scanlines in an antialiased trapezoid.<br>Diameter of a point in sub scanlines. |
| BitMaskPattern | Value used to control the BitMask stipple operation (if enabled). |
| PointTable0<br>PointTable1<br>PointTable2<br>PointTable3 | Antialias point data table. There are 4 words in the table and the register tag is decoded to select a word. |

For efficiency, the Render command register has a number of bit fields that can be set or cleared per render operation, and which qualify other state information within GLINT. These bits are AreaStippleEnable, LineStippleEnable, ResetLineStipple, TextureEnable FogEnable, CoverageEnable and SubpixelCorrection.

One use of this feature can occur when a window is cleared to a background color. For normal 3D primitives, stippling and fog operations may have been enabled, but these are to be ignored for window clears. Initially the FogMode, AreaStippleMode and LineStippleMode registers are enabled through the UnitEnable bits. Now bits need only be set or cleared within the Render command to achieve the required result, removing the need for the FogMode, AreaStippleMode and LineStippleMode registers to be loaded for every render operation.

The bitfields of the Render command register, in the presently preferred embodiment, are detailed below:

| Bit | Name | Description |
|---|---|---|
| 0 | Area-Stipple-Enable | This bit, when set, enables area stippling of the fragments produced during rasterization. Note that area stipple in the Stipple Unit must be enabled as well for stippling to occur.<br>When this bit is reset no area stippling occurs irrespective of the setting of the area stipple enable bit in the Stipple Unit.<br>This bit is useful to temporarily force no area stippling for this primitive. |
| 1 | Line-Stipple-Enable | This bit, when set, enables line stippling of the fragments produced during rasterization in the Stipple Unit. Note that line stipple in the Stipple Unit must be enabled as well for stippling to occur.<br>When this bit is reset no line stippling occurs irrespective of the setting if the line stipple enable bit in the Stipple Unit.<br>This bit is useful to temporarily force no line stippling for this primitive. |
| 2 | Reset-Line-Stipple | This bit, when set, causes the line stipple counters in the Stipple Unit to be reset to zero, and would typically be used for the first segment in a polyline. This action is also qualified by the LineStippleEnable bit and also the stipple enable bits in the Stipple Unit.<br>When this bit is reset the stipple counters carry on from where they left off (if line stippling is enabled) |
| 3 | FastFillEnable | This bit, when set, causes fast block filling of primitives.<br>When this bit is reset the normal rasterization process occurs. |
| 4, 5 | Fast-Fill-Increment | This two bit field selects the block size the framebuffer supports. The sizes supported and the corresponding codes are:<br>0 = 8 pixels<br>1 = 16 pixels<br>2 = 32 pixels |
| 6, 7 | Primitive-Type | This two bit field selects the primitive type to rasterize. The primitives are:<br>0 = Line<br>1 = Trapezoid<br>2 = Point |
| 8 | Antialias-Enable | This bit, when set, causes the generation of sub scanline data and the coverage value to be calculated for each fragment. The number of sub pixel samples to use is controlled by the AntialiasingQuality bit.<br>When this bit is reset normal rasterization occurs. |
| 9 | Antialiasing-Quality | This bit, when set, sets the sub pixel resolution to be 8 × 8<br>When this bit is reset the sub pixel resolution is 4 × 4. |
| 10 | UsePoint-Table | When this bit and the AntialiasingEnable are set, the dx values used to remove from one scanline to the next are derived from the Point Table. |
| 11 | SyncOn-BitMask | This bit, when set, causes a number of actions: -<br>The least significant bit or most significant bit (depending on the MirrorBitMask bit) in the Bit Mask register is extracted and optionally inverted (controlled by the InvertMask bit).<br>If this bit is 0 then the corresponding fragment is culled from being drawn.<br>After every fragrant the Bit Mask register is rotated by one bit.<br>If all the bits in the Bit Mask register have been used then rasterization is suspended until a new BitMaskPattern is received. If any other register is written while the rasterization is suspended then the rasterization is aborted. The register write which caused the abort is then processed as normal.<br>Note the behavior is slightly different when the SyncOnHostData bit is set to prevent a deadlock from occurring. In this case the rasterization doesn't suspend when all the bits have been used and if new BitMaskPattern data words are not received in a timely manner then the subsequent fragments will just reuse the bitmask. |
| 12 | SyncOnHostData | When this bit is set a fragment is produced only when one of the following registers has been written by the host: Depth, FBColor, Stencil or Color. If SyncOnBitMask is reset, then if any register other than one of these four is written to, the rasterization is aborted. If SyncOnBitMask is set, then if any register other than one of these four, or BitMaskPattern, is written to, the rasterization is aborted. The register write which caused the abort is then processed as normal. Writing to the BitMaskPattern register doesn't cause any fragments to be generated, but just updates the BitMask register. |
| 13 | TextureEnable | This bit, when set, enables texturing of the fragments produced during rasterization. Note that the Texture Units must be suitably enabled as well for any texturing to occur. |

APPENDIX O

-continued

| Bit | Name | Description |
|---|---|---|
| | | When this bit is reset no texturing occurs irrespective of the setting of the Texture Unit controls.<br>This bit is useful to temporarily force no texturing for this primitive. |
| 14 | Fog-Enable | This bit, When set, enables fogging of the fragments produced during rasterization. Note that the Fog Unit must be suitably enabled as well for any fogging to occur.<br>When this bit is reset no fogging occurs irrespective of the setting of the Fog Unit controls.<br>This bit is useful to temporarily force no fogging for this primitive. |
| 15 | Coverage-Enable | This bit, when set, enables the coverage value produced as part of the antialiasing to weight the alpha value in the alpha test unit. Note that this unit must be suitably enabled as well.<br>When this bit is reset no coverage application. occurs irrespective of the setting of the AntialiasMode in the Alpha. Test unit. |
| 16 | SubPixel-Correction Enable | This bit, when set enables the sub pixel correction of the color, depth, fog and texture values at the start of a scanline. When this bit is reset no correction is done at the start of a scanline. Sub pixel corrections are only applied to aliased trapezoids. |

A number of long-term rasterizer modes are stored in the RasterizerMode register as shown below:

| Bit | Name | Description |
|---|---|---|
| 0 | Mirror-BitMask | When this bit is set the bitmask bits are consumed from the most significant end towards the least significant end. When this bit is reset the bitmask bits are consumed from the least significant end towards the most significant end. |
| 1 | InvertBit-Mask | When this bit is set the bitmask is inverted first before being tested. |
| 2,3 | Fraction-Adjust | These bits control the action of a ContinueNewLine command and specify how the fraction bits in the Y and XDom DDAs are adjusted<br>0: No adjustment is done<br>1: Set the fraction bits to zero<br>2: Set the fraction bits to half<br>3: Set the fraction to nearly half, i.e. 0x7fff |
| 4,5 | BiasCoordinates | These bits control how much is added onto the StartXDom, StartXSub and StartY values, when they are loaded into the DDA units. The original registers are not affected:<br>0: Zero is added<br>1: Half is added<br>2: Nearly half, i.e. 0x7fff is added |

Scissor Unit

Two scissor tests are provided in GLINT, the User Scissor test and the Screen Scissor test. The user scissor checks each fragment against a user supplied scissor region; the screen scissor checks that the fragment lies within the screen.

This test may reject fragments if some part of a window has been moved off the screen. It will not reject fragments if part of a window is simply overlapped by another window (GID testing can be used to detect this).

Stipple Unit

Stippling is a process whereby each fragment is checked against a bit in a defined pattern, and is rejected or accepted depending on the result of the stipple test. If it is rejected it undergoes no further processing; otherwise it proceeds down the pipeline. GLINT supports two types of stippling, line and area.

Area Stippling

A 32×32 bit area stipple pattern can be applied to fragments. The least significant n bits of the fragment's (X,Y) coordinates, index into a 2D stipple pattern. If the selected bit in the pattern is set, then the fragment passes the test, otherwise it is rejected. The number of address bits used, allow regions of 1,2,4,8,16 and 32 pixels to be stippled. The address selection can be controlled independently in the X and Y directions. In addition the bit pattern can be inverted or mirrored. Inverting the bit pattern has the effect of changing the sense of the accept/reject test. If the mirror bit is set the most significant bit of the pattern is towards the left of the window, the default is the converse.

In some situations window relative stippling is required but coordinates are only available screen relative. To allow window relative stippling, an offset is available which is added to the coordinates before indexing the stipple table. X and Y offsets can be controlled independently.

Line Stippling

In this test, fragments are conditionally rejected on the outcome of testing a linear stipple mask. If the bit is zero then the test fails, otherwise it passes. The line stipple pattern is 16 bits in length and is scaled by a repeat factor r (in the range 1 to 512 ). The stipple mask bit b which controls the acceptance or rejection of a fragment is determined using:

b=(floor (s/r)) mod 16

where s is the stipple counter which is incremented for every fragment (normally along the line). This counter may be reset at the start of a polyline, but between segments it continues as if there were no break.

The stipple pattern can be optionally mirrored, that is the bit pattern is traversed from most significant to least significant bits, rather than the default, from least significant to most significant.

Color DDA Unit

The color DDA unit is used to associate a color with a fragment produced by the rasterizer. This unit should be enabled for rendering operations and disabled for pixel rectangle operations (i.e. copies, uploads and downloads). Two color modes are supported by GLINT, true color RGBA and color index (CI).

Gouraud Shading

When in Gouraud shading mode, the color DDA unit performs linear interpolation given a set of start and increment values. Clamping is used to ensure that the interpolated value does not underflow or overflow the permitted color range.

For a Gouraud shaded trapezoid, GLINT interpolates from the dominant edge of a trapezoid to the subordinate edges. This means that two increment values are required per color component, one to move along the dominant edge and one to move across the span to the subordinate edge.

Note that if one is rendering to multiple buffers and has initialized the start and increment values in the color DDA unit, then any subsequent Render command will cause the start values to be reloaded.

If subpixel correction has been enabled for a primitive, then any correction required will be applied to the color components.

Flat Shading

In flat shading mode, a constant color is associated with each fragment. This color is loaded into the ConstantColor register.

Texture Unit

The texture unit combines the incoming fragment's color (generated in the color DDA unit) with a value derived from interpolating texture map values (texels).

Texture application consists of two stages; derivation of the texture color from the texels (a filtering process) and then application of the texture color to the fragment's color, which is dependent on the application mode (Decal, Blend or Modulate).

GLINT 300SX compared with the GLINT 400TX

Both the GLINT 300SX and GLINT 300TX support all the filtering and application modes described in this section. However, when using the GLINT 300SX, texel values, interpolants and texture filter selections are supplied by the host. This implies that texture coordinate interpolation and texel extraction are performed by the host using texture maps resident on the host. The recommended technique for performing texture mapping using the GLINT 300SX is to scan convert primitives on the host and render fragments as GLINT point primitives.

The GLINT 400TX automatically generates all data required for texture application as textures are stored in the localbuffer and texture parameter interpolation with full perspective correction takes place within the processor. Thus the GLINT 400TX is the processor of choice when full texture mapping acceleration is desired, the GLINT 300SX is more suitable in applications where the performance of texture mapping is not critical.

Texture Color Generation.

Texture color generation supports all the filter modes of OpenGL, that is:

Minification:
- Nearest
- Linear
- NearestMipMapNearest
- NearestMipMapLinear
- LinearMipMapNearest
- LinearMipMapLinear

Magnification:
- Nearest
- Linear

Minification is the name given to the filtering process used whereby multiple texels map to a fragment, while magnification is the name given to the filtering process whereby only a portion of a single texel maps to a single fragment.

Nearest is the simplest form of texture mapping where the nearest texel to the sample location is selected with no filtering applied.

Linear is a more sophisticated algorithm which is dependent on the type of primitive. For lines (which are 1D), it involves linear interpolation between the two nearest texels, for polygons and points which are considered to have finite area, linear is in fact bi-linear interpolation which interpolates between the nearest 4 texels.

Mip Mapping is a technique to allow the efficient filtering of texture maps when the projected area of the fragment covers more than one texel (ie. minification). A hierarchy of texture maps is held with each one being half the size (or one quarter the area) of the preceding one. A pair of maps are selected, based on the projected area of the texture. In terms of filtering this means that three filter operations are performed: one on the first map, one on the second map and one between the maps. The first filter name (Nearest or Linear) in the MipMap name specifies the filtering to do on the two maps, and the second filter name specifies the filtering to do between maps. So for instance, linear mapping between two maps, with linear interpolation between the results is supported (LinearMipMapLinear), but linear interpolation on one map, nearest on the other map, and linear interpolation between the two is not supported.

The filtering process takes a number of texels and interpolants, and with the current texture filter mode produces a texture color.

Fog Unit

The fog unit is used to blend the incoming fragment's color (generated by the color DDA unit, and potentially modified by the texture unit) with a predefined fog color. Fogging can be used to simulate atmospheric fogging, and also to depth cue images.

Fog application has two stages; derivation of the fog index for a fragment, and application of the fogging effect. The fog index is a value which is interpolated over the primitive using a DDA in the same way color and depth are interpolated. The fogging effect is applied to each fragment using one of the equations described below.

Note that although the fog values are linearly interpolated over a primitive the fog values can be calculated on the host using a linear fog function (typically for simple fog effects and depth cuing) or a more complex function to model atmospheric attenuation. This would typically be an exponential function.

Fog Index Calculation—The Fog DDA

The fog DDA is used to interpolate the fog index (f) across a primitive. The mechanics are similar to those of the other DDA units, and horizontal scanning proceeds from dominant to subordinate edge as discussed above.

The DDA has an internal range of approximately +511 to −512, so in some cases primitives may exceed these bounds. This problem typically occurs for very large polygons which span the whole depth of a scene. The correct solution is to tessellate the polygon until polygons lie within the acceptable range, but the visual effect is frequently negligible and can often be ignored.

The fog DDA calculates a fog index value which is clamped to lie in the range 0.0 to 1.0 before it is used in the appropriate fogging equation. (Fogging is applied differently depending on the color mode.)

Antialias Application Unit

Antialias application controls the combining of the coverage value generated by the rasterizer with the color generated in the color DDA units. The application depends on the color mode, either RGBA or Color Index (CI).

Antialias Application

When antialiasing is enabled this unit is used to combine the coverage value calculated for each fragment with the fragment's alpha value. In RGBA mode the alpha value is multiplied by the coverage value calculated in the rasterizer (its range is 0% to 100%). The RGB values remain unchanged and these are modified later in the Alpha Blend unit which must be set up appropriately. In CI mode the coverage value is placed in the lower 4 bits of the color field.

The Color Look Up Table is assumed to be set up such that each color has 16 intensities associated with it, one per coverage entry.

Polygon Antialiasing

When using GLINT to render antialiased polygons, depth buffering cannot be used. This is because the order the fragments are combined in is critical in producing the correct final color. Polygons should therefore be depth sorted, and rendered front to back, using the alpha blend modes: SourceAlphaSaturate for the source blend function and One for the destination blend function. In this way the alpha component of a fragment represents the percentage pixel coverage, and the blend function accumulates coverage until the value in the alpha buffer equals one, at which point no further contributions can made to a pixel.

For the antialiasing of general scenes, with no restrictions on rendering order, the accumulation buffer is the preferred choice. This is indirectly supported by GLINT via image uploading and downloading, with the accumulation buffer residing on the host.

When antialiasing, interpolated parameters which are sampled within a fragment (color, fog and texture), will sometimes be unrepresentative of a continuous sampling of a surface, and care should be taken when rendering smooth shaded antialiased primitives. This problem does not occur in aliased rendering, as the sample point is consistently at the center of a pixel.

Alpha Test Unit

The alpha test compares a fragment's alpha value with a reference value. Alpha testing is not available in color index (CI) mode. The alpha test conditionally rejects a fragment based on the comparison between a reference alpha value and one associated with the fragment.

Localbuffer Read/Write Unit

The localbuffer holds the Graphic ID, FrameCount, Stencil and Depth data associated with a fragment. The localbuffer read/write unit controls the operation of GID testing, depth testing and stencil testing.

Localbuffer Read

The LBReadMode register can be configured to make 0, 1 or 2 reads of the localbuffer. The following are the most common modes of access to the localbuffer:

Normal rendering without depth, stencil or GID testing. This requires no localbuffer reads or writes.

Normal rendering without depth or stencil testing and with GID testing. This requires a localbuffer read to get the GID from the localbuffer.

Normal rendering with depth and/or stencil testing required which conditionally requires the localbuffer to be updated. This requires localbuffer reads and writes to be enabled.

Copy operations. Operations which copy all or part of the localbuffer with or without GID testing. This requires reads and writes enabled.

Image upload/download operations. Operations which download depth or stencil information to the local buffer or read depth, stencil fast clear or GID from the localbuffer.

Localbuffer Write

Writes to the localbuffer must be enabled to allow any update of the localbuffer to take place. The LBWriteMode register is a single bit flag which controls updating of the buffer.

## Pixel Ownership (GID) Test Unit

Any fragment generated by the rasterizer may undergo a pixel ownership test. This test establishes the current fragment's write permission to the localbuffer and framebuffer.

Pixel Ownership Test

The ownership of a pixel is established by testing the GID of the current window against the GID of a fragment's destination in the GID buffer. If the test passes, then a write can take place, otherwise the write is discarded. The sense of the test can be set to one of: always pass, always fail, pass if equal, or pass if not equal. Pass if equal is the normal mode. In GLINT the GID planes, if present, are 4 bits deep allowing 16 possible Graphic ID's. The current GID is established by setting the Window register.

If the unit is disabled fragments pass through undisturbed.

Stencil Test Unit

The stencil test conditionally rejects fragments based on the outcome of a comparison between the value in the stencil buffer and a reference value. The stencil buffer is updated according to the current stencil update mode which depends on the result of the stencil test and the depth test.

Stencil Test

This test only occurs if all the preceding tests (bitmask, scissor, stipple, alpha, pixel ownership) have passed. The stencil test is controlled by the stencil function and the stencil operation. The stencil function controls the test between the reference stencil value and the value held in the stencil buffer. The stencil operation controls the updating of the stencil buffer, and is dependent on the result of the stencil and depth tests.

If the stencil test is enabled then the stencil buffer will be updated depending on the outcome of both the stencil and the depth tests (if the depth test is not enabled the depth result is set to pass).

In addition a comparison bit mask is supplied in the StencilData register. This is used to establish which bits of the source and reference value are used in the stencil function test. In addition it should normally be set to exclude the top four bits when the stencil width has been set to 4 bits in the StencilMode register.

The source stencil value can be from a number of places as controlled by a field in the StencilMode register:

| LBWriteData Stencil | Use |
|---|---|
| Test logic | This is the normal mode. |
| Stencil register | This is used, for instance, in the OpenGL draw pixels function where the host supplies the stencil values in the Stencil register.<br>This is used when a constant stencil values is needed, for example, when clearing the stencil buffer when fast clear planes are not available. |
| LBSourceData: (stencil value read from the localbuffer) | This is used, for instance, in the OpenGL copy pixels function when the stencil planes are to be copied to the destination. The source is offset from the destination by the value in LBSourceOffset register. |
| Source stencil value read from the localbuffer | This is used, for instance, in the OpenGL copy pixels function when the stencil planes in the destination are not to be updated. The stencil data will come either from the localbuffer date, or the FCStencil register, depending on whether fast clear operations are enabled. |

Depth Test Unit

The depth (Z) test, if enabled, compares a fragment's depth against the corresponding depth in the depth buffer. The result of the depth test can effect the updating of the stencil buffer if stencil testing is enabled. This test is only performed if all the preceding tests (bitmask, scissor, stipple, alpha, pixel ownership, stencil) have passed. The source value can be obtained from a number of places as controlled by a field in the DepthMode register:

| Source | Use |
|---|---|
| DDA (see below) | This is used for normal Depth buffered 3D rendering. |
| Depth register | This is used, for instance, in the OpenGL draw pixels function where the host supplies the depth values through the Depth register.<br>Alternatively this is used when a constant depth value is needed, for example, when clearing the depth buffer (when fast clear planes are not available) or 2D rendering where the depth is held constant. |
| LBSourceData: Source depth value from the localbuffer | This is used, for instance, in the OpenGL copy pixels function when the depth planes are to be copied to the destination. |
| Source Depth | This is used, for instance, in the OpenGL copy pixels function when the depth planes in the destination are not updated. The depth data will come either from the localbuffer or the FCDepth register depending the state of the Fast Clear modes in operation. |

When using the depth DDA for normal depth buffered rendering operations the depth values required are similar to those required for the color values in the color DDA unit:

ZStart=Start Z Value

dZdYDom=Increment along dominant edge.

dZdX=Increment along the scan line.

The dZdX value is not required for Z-buffered lines.

The depth unit must be enabled to update the depth buffer. If it is disabled then the depth buffer will only be updated if ForceL-BUpdate is set in the Window register.

Framebuffer Read/Write Unit

Before rendering can take place GLINT must be configured to perform the correct framebuffer read and write operations. Framebuffer read and write modes effect the operation of alpha blending, logic ops, write masks, image upload/download operations and the updating of pixels in the framebuffer.

Framebuffer Read

The FBReadMode register allows GLINT to be configured to make 0, 1 or 2 reads of the framebuffer. The following are the most common modes of access to the framebuffer: Note that avoiding unnecessary additional reads will enhance performance.

Rendering operations with no logical operations, software write-masking or alpha blending. In this case no read of the framebuffer is required and framebuffer writes should be enabled.

Rendering operations which use logical ops, software write masks or alpha blending. In these cases the destination pixel must be read from the framebuffer and framebuffer writes must be enabled.

Image copy operations. Here setup varies depending on whether hardware or software write masks are used. For software write masks, the framebuffer needs two reads, one for the source and one for the destination. When hardware write masks are used (or when the software write mask allows updating of all bits in a pixel) then only one read is required.

Image upload. This requires reading of the destination framebuffer reads to be enabled and framebuffer writes to be disabled.

Image download. In this case no framebuffer read is required (as long as software writemasking and logic ops are disabled) and the write must be enabled.

For both the read and the write operations, an offset is added to the calculated address. The source offset (FBSourceOffset) is used for copy operations. The pixel offset (FBPixelOffset) can be used to allow multi-buffer updates. The offsets should be set to zero for normal rendering.

The data read from the framebuffer may be tagged either FBDefault (data which may be written back into the framebuffer or used in some manner to modify the fragment color) or FBColor (data which will be uploaded to the host). The table below summarizes the framebuffer read/write control for common rendering operations:

| Read-Source | ReadDestination | Writes | Read Data Type | Rendering Operation |
|---|---|---|---|---|
| Disabled | Disabled | Enabled | — | Rendering with no logical operations, software write masks or blending. |
| Disabled | Disabled | Enabled | — | Image download. |
| Disabled | Enabled | Disabled | FBColor | Image upload. |
| Enabled | Disabled | Enabled | FBDefault | Image copy with hardware write masks. |
| Disabled | Enabled | Enabled | FBDefault | Rendering using logical operations, software write masks or blending. |
| Enabled | Enabled | Enabled | FBDefault | Image copy with software writemasks. |

Framebuffer Write

Framebuffer writes must be enabled to allow the framebuffer to be updated. A single 1 bit flag controls this operation.

The framebuffer write unit is also used to control the operation of fast block fills, if supported by the framebuffer. Fast fill rendering is enabled via the FastFillEnable bit in the Render command register, the framebuffer fast block size must be configured to the same value as the FastFillIncrement in the Render command register. The FBBlockColor register holds the data written to the framebuffer during a block fill operation and should be formatted to the 'raw' framebuffer format. When using the framebuffer in 8 bit packed mode the data should be replicated into each byte. When using the framebuffer in packed 16 bit mode the data should be replicated into the top 16 bits.

When uploading images the UpLoadData bit can be set to allow color formatting (which takes place in the Alpha Blend unit).

It should be noted that the block write capability provided by the chip of the presently preferred embodiment is itself believed to be novel. According to this new approach, a graphics system can do masked block writes of variable length (e.g. 8, 16, or 32 pixels, in the presently preferred embodiment). The rasterizer defines the limits of the block to be written, and hardware masking logic in the framebuffer interface permits the block to be filled in, with a specified primitive, only up to the limits of the object being rendered. Thus the rasterizer can step by the Block Fill increment. This permits the block-write capabilities of the VRAM chips to be used optimally, to minimize the length which must be written by separate writes per pixel.

Alpha Blend Unit

Alpha blending combines a fragment's color with those of the corresponding pixel in the framebuffer. Blending is supported in RGBA mode only.

Alpha Blending

The alpha blend unit combines the fragment's color value with that stored in the framebuffer, using the blend equation:

$$C_o=C_sS+C_dD$$

where: $C_o$ is the output color; $C_s$ is the source color (calculated internally); $C_d$ is the destination color read from the framebuffer; S is the source blending weight; and D is the destination blending weight. S and D are not limited to linear combinations; lookup functions can be used to implement other combining relations.

If the blend operations require any destination color components then the framebuffer read mode must be set appropriately.

Image Formatting

The alpha blend and color formatting units can be used to format image data into any of the supported GLINT framebuffer formats.

Consider the case where the framebuffer is in RGBA 4:4:4:4 mode, and an area of the screen is to be uploaded and stored in an 8 bit RGB 3:3:2 format. The sequence of operations is:

Set the rasterizer as appropriate

Enable framebuffer reads

Disable framebuffer writes and set the UpLoadData bit in the FBWriteMode register

Enable the alpha blend unit with a blend function which passes the destination value and ignores the source value (source blend Zero, destination blend One) and set the color mode to RGBA 4:4:4:4

Set the color formatting unit to format the color of incoming fragments to an 8 bit RGB 3:3:2 framebuffer format.

The upload now proceeds as normal. This technique can be used to upload data in any supported format.

The same technique can be used to download data which is in any supported framebuffer format, in this case the rasterizer is set to sync with FBColor, rather than Color. In this case framebuffer writes are enabled, and the UpLoadData bit cleared.

Color Formatting Unit

The color formatting unit converts from GLINT's internal color representation to a format suitable to be written into the framebuffer. This process may optionally include dithering of the color values for framebuffers with less than 8 bits width per color component. If the unit is disabled then the color is not modified in any way.

As noted above, the framebuffer may be configured to be RGBA or Color Index (CI).

Color Dithering

GLINT uses an ordered dither algorithm to implement color dithering. Several types of dithering can be selected.

If the color formatting unit is disabled, the color components RGBA are not modified and will be truncated when placed in the framebuffer. In CI mode the value is rounded to the nearest integer. In both cases the result is clamped to a maximum value to prevent overflow.

In some situations only screen coordinates are available, but window relative dithering is required. This can be implemented by adding an optional offset to the coordinates before indexing the dither tables. The offset is a two bit number which is supplied for each coordinate, X and Y. The XOffset, YOffset fields in the DitherMode register control this operation, if window relative coordinates are used they should be set to zero.

Logical Op Unit

The logical op unit performs two functions; logic operations between the fragment color (source color) and a value from the framebuffer (destination color); and, optionally, control of a special GLINT mode which allows high performance flat shaded rendering.

High Speed Flat Shaded Rendering

A special GLINT rendering mode is available which allows high speed rendering of unshaded images. To use the mode the following constraints must be satisfied:

Flat shaded aliased primitive

No dithering required

No logical ops

No stencil, depth or GID testing required

No alpha blending The following are available:

Bit masking in the rasterizer

Area and line stippling

User and Screen Scissor test

If all the conditions are met then high speed rendering can be achieved by setting the FBWriteData register to hold the framebuffer data (formatted appropriately for the framebuffer in use) and setting the UseConstantFBWriteData bit in the LogicalOpMode register. All unused units should be disabled.

This mode is most useful for 2D applications or for clearing the framebuffer when the memory does not support block writes. Note that FBWriteData register should be considered volatile when context switching.

Logical Operations

The logical operations supported by GLINT are:

| Mode | Name | Operation | Mode | Name | Operation |
|---|---|---|---|---|---|
| 0 | Clear | 0 | 8 | Nor | ~(S \| D) |
| 1 | And | S & D | 9 | Equivalent | ~(S ^ D) |
| 2 | And Reverse | S & ~D | 10 | Invert | ~D |
| 3 | Copy | S | 11 | Or Reverse | S \| ~D |
| 4 | And Inverted | ~S & D | 12 | Copy Invert | ~S |
| 5 | Noop | D | 13 | Or Invert | ~S \| D |
| 6 | Xor | S ^ D | 14 | Nand | ~(S & D) |
| 7 | Or | S \| D | 15 | Set | 1 |

Where:

S=Source (fragment) Color, D=Destination (framebuffer) Color.

For correct operation of this unit in a mode which takes the destination color, GLINT must be configured to allow reads from the framebuffer using the FBReadMode register.

GLINT makes no distinction between RGBA and CI modes when performing logical operations. However, logical operations are generally only used in CI mode.

Framebuffer Write Masks

Two types of framebuffer write masking are supported by GLINT, software and hardware. Software write masking requires a read from the framebuffer to combine the fragment color with the framebuffer color, before checking the bits in the mask to see which planes are writeable. Hardware write masking is implemented using VRAM write masks and no framebuffer read is required.

Software Write Masks

Software write masking is controlled by the FBSoftwareWriteMask register. The data field has one bit per framebuffer bit which when set, allows the corresponding framebuffer bit to be updated. When reset it disables writing to that bit. Software write masking is applied to all fragments and is not controlled by an enable/disable bit. However it may effectively be disabled by setting the mask to all 1's. Note that the ReadDestination bit must be enabled in the FBReadMode register when using software write masks, in which some of the bits are zero.

Hardware Write Masks

Hardware write masks, if available, are controlled using the FBHardwareWriteMask register. If the framebuffer supports hardware write masks, and they are to be used, then software write masking should be disabled (by setting all the bits in the FBSoftwareWriteMask register). This will result in fewer framebuffer reads when no logical operations or alpha blending is needed.

If the framebuffer is used in 8 bit packed mode, then an 8 bit hardware write mask must be replicated to all 4 bytes of the FBHardwareWriteMask register. If the framebuffer is in 16 bit packed mode then the 16 bit hardware write mask must be replicated to both halves of the FBHardwareWriteMask register.

Host Out Unit

Host Out Unit controls which registers are available at the output FIFO, gathering statistics about the rendering operations (picking and extent testing) and the synchronization of GLINT via the Sync register. These three functions are as follows:

Message filtering. This unit is the last unit in the core so any message not consumed by a preceding unit will end up here. These messages will fall in to three classifications: Rasterizer messages which are never consumed by the earlier units, messages associated with image uploads, and finally programmer mistakes where an invalid message was written to the input FIFO. Synchronization messages are a special category and are dealt with later. Any messages not filtered out are passed on the output FIFO.

Statistic Collection. Here the active step messages are used to record the extent of the rectangular region where rasterization has been occurring, or if rasterization has occurred inside a specific rectangular region. These facilities are useful for picking and debug activities.

Synchronization. It is often useful for the controlling software to find out when some rendering activity has finished, to allow the timely swapping or sharing of buffers, reading back of state, etc. To achieve this the software would send a Sync message and when this reached this unit any preceding messages or their actions are guaranteed to have finished. On receiving the Sync message it is entered into the FIFO and optionally generates an interrupt.

### Sample Board-Level Embodiment

A sample board incorporating the GLINT chip may include simply:

the GLINT chip itself, which incorporates a PCI interface;

Video RAM (VRAM), to which the chip has read-write access through its frame buffer (FB) port;

DRAM, which provides a local buffer then made for such purposes as Z buffering; and

a RAMDAC, which provides analog color values in accordance with the color values read out from the VRAM.

Thus one of the advantages of the chip of the presently preferred embodiment is that a minimal board implementation is a trivial task.

FIG. 3A shows a sample graphics board which incorporates the chip of FIG. 2B.

FIG. 3B shows another sample graphics board implementation, which differs from the board of FIG. 3A in that more memory and an additional component is used to achieve higher performance.

FIG. 3C shows another graphics board, in which the chip of FIG. 2B shares access to a common frame store with GUI accelerator chip.

FIG. 3D shows another graphics board, in which the chip of FIG. 2B shares access to a common frame store with a video coprocessor (which may be used for video capture and playback functions (e.g. frame grabbing).

Alternative Board Embodiment with Additional Video Processor

In the presently preferred embodiment, the frame buffer interface of the GLINT chip contains additional simple interface logic, so that two chips can both access the same frame buffer memory. This permits the GLINT chip to be combined with an additional chip for management to the graphics produced by the graphical user interface. This provides a migration path for users and applications who need to take advantage of the existing software investment and device drivers for various other graphics chips.

FIG. 3C shows another graphics board, in which the chip of FIG. 2B shares access to a common frame store with a GUI accelerator chip (such as an S3 chip). This provides a path for software migration, and also provides a way to separate 3D rendering tasks from 2D rendering.

In this embodiment, a shared framebuffer is used to enable multiple devices to read or write data to the same physical framebuffer memory. Example applications using the GLINT 300SX:

Using a video device as a coprocessor to GLINT, to grab live video into the framebuffer, for displaying video in a window or acquiring a video sequence;

Using GLINT as a 3D coprocessor to a 2D GUI accelerator, preserving an existing investment in 2D driver software.

In a coprocessor system, the framebuffer is a shared resource, and so access to the resource needs to be arbitrated. There are also other aspects of sharing a framebuffer that need to be considered:

Memory refreshing;

Transfer of data from the memory cells into the shift registers of the VRAM;

Control of writemasks and color registers.

GLINT uses the S3 Shared Frame Buffer Interface (SFBI) to share a framebuffer. This interface is able to handle all of the above aspects for two devices sharing a frame buffer, with the GLINT acting as an arbitration master or slave.

### Timing Considerations in Shared Frame-Buffer Interface

The Control Signals used in the Shared Framebuffer interface, in the presently preferred embodiment, are as follows:

GLINT as Primary Controller

FBReqN is internally re-synchronized to System Clock.

FBSelOEN remains negated.

FBGntN is asserted an unspecified amount of time after FBReqN is asserted.—Framebuffer Address, Data and Control lines are tri-stated by GLINT (the control lines should be held high by external pull-up resistors). The secondary controller is now free to drive the Framebuffer lines and access the memory.

FBGntN remains asserted until GLINT requires a framebuffer access, or a refresh or transfer cycle.

FBReqN must remain asserted while FBGntN is asserted.

When FBGntN is removed, the secondary controller must relinquish the address, data and control bus in a graceful manner i.e. RAS, CAS, WE and OE must all be driven high before being tri-stated.

The secondary controller must relinquish the bus and negate FBReqN within 500 ns of FBGntN being negated.

Once FBReqN has been negated, it must remain inactive for at least 2 system clocks (40 ns at 50 MHz).

GLINT as a Secondary Controller

Framebuffer Refresh and VRAM transfer cycles by GLINT are turned off when GLINT is a secondary framebuffer controller.

GLINT asserts FBReqN whenever is requires a framebuffer access.

FBGntN is internally re-synchronized to system clock.

When FBGntN is asserted, GLINT drives FBselOEN to enable any external buffers used to drive the control signals, and then drives the framebuffer address, data and control lines to perform the memory access. FBReqN remains asserted while FBGntN is asserted.

When FBGntN is negated, GLINT finishes any outstanding memory cycles, drives the control lines inactive, negates FBselOEN and then tri-states the address, data and control lines, then releases FBReqN. GLINT guarantees to release FBReqN within 500 ns of FBGntN being negated.

GLINT will not reassert FBReqN within 4 system clock cycles (80 ns@ 50 MHz).

Considerations for Board-Level Implementations

The following are some points to be noted when implementing a shared framebuffer design with a GLINT 300SX:

Some 2D GUI Accelerators such as the S3 Vision964, and GLINT use configuration resistors on the framebuffer databus at reset. In this case care should be taken with the configuration setup where it effects read only registers inside either device. If conflicts exist that can not be resolved by the board initialization software, then the conflicts should be resolved by isolating the two devices from each other at reset so they can read the correct configuration information. This isolation need only be done for the framebuffer databus lines that cause problems;

GLINT should be configured as the secondary controller when used with an S3 GUI accelerator, as the S3 devices can only be primary controllers;

GLINT cannot be used on the daughter card interface as described in the S3 documentation, because this gives no access to the PCI bus. A suitable PCI bridge should be used in a design with a PCI 2D GUI accelerator and GLINT so they can both have access to the PCI bus;

The use of ribbon cable to carry the framebuffer signals between two PCI boards is not recommended, because of noise problems and the extra buffering required would impact performance;

The GLINT 300SX does not provide a way of sharing its localbuffer.

The 400TX also allows grabbing of live video into the localbuffer and real-time texture mapping of that video into the framebuffer for video manipulation effects.

## Alternative Board Embodiments with Multiple Rendering Accelerator Chips

This technical note describes some system design issues on how multiple GLINT devices can be used in parallel to achieve higher performance. The main driving force for higher performance is the simulation market which, at the low end, demands somewhere between 25–30M texture mapped pixels per second.

There are some key points before we look at different parallel organizations:

To gain any benefit from running multiple GLINTs in parallel, the overall system must be rendering bound. If the system is host bound or geometry bound, then adding in more GLINTs will not improve the systems performance.

The memory systems (i.e. local buffer and framebuffer) are duplicated for each GLINT. Recall that the texture maps are stored in the local buffer. A single GLINT places very high demands on the memory systems, and it would be very difficult to share them between multiple GLINTs. In the presently preferred embodiment there are no provisions for sharing the local buffer, so if this is necessary it would have to be done behind GLINT's back and transparently. The framebuffer can be shared (since GLINT has a SFB interface), but this is likely to be a bottle neck if shared between GLINTs.

Broadcast. In some parallel systems each GLINT will get the same (or mostly the same) primitive data and just render those pixels assigned to it. It is very desirable that this data is written by the host only once, or fetched from the host address space once if DMA is being used. This presents two issues: Firstly the PCI bus does not have any concept of broadcasting to multiple devices, and secondly GLINT does not have a dedicated FIFO status signal pin an external controller can use. Neither of these issues are insurmountable, but will require hardware to solve. However, if the application only uses a 'few' large texture mapped primitives so repeatedly sending or fetching the parameters for each GLINT will not be a problem.

To avoid problems with Antialiasing, Bitmasks for characters, or Line stipple, the area stipple table can be used to reserve scanlines to a processor.

Parallel Configurations

This section looks at some of the common ways of applying parallelism to the rendering operation. The list is not exhaustive and an interested reader is directed to the book by Whitman cited above. No one paradigm is best and the choice is very application or market dependent.

Frame Interleaving

Frame Interleaving is where a GLINT works on frame n, the next GLINT works on frame n+1, etc. Each GLINT does everything for its own frame and the video is sourced from each GLINT's framebuffer in turn. This paradigm is perhaps the simplest one with very little hardware overhead and none of the above complications regarding antialiasing, block copies, bitmasks and line stipples.

This scheme only works when the image is double buffered (normal for simulation systems) and where the increase in transport delay is acceptable. Transport delay is the time it takes for a user to see a visual change after new input stimulus to the system has occurred. With 4 GLINTs this will be 4 frame times attributable to the rendering system, plus whatever else the whole system adds.

The cost of this method is also one of the highest, as ALL the memory has to be duplicated. By contrast, the schemes where the screen is divided up can save depth and color buffer memory (but not texture memory).

Sequential frames will usually have very similar amounts of rendering, unless there is a discontinuity in the viewing position and/or orientation, so load balancing is generally good.

Frame Merging or Primitive Parallelism

Frame merging is a similar technique to frame interleaving where each GLINT has a full local buffer and framebuffer. In this case the primitives are distributed amongst the GLINTs and the resultant partial images composited using the depth information to control which fragment from the multiple buffers is displayed in each pixel position.

GLINT has not been designed to share the local buffer (where the depth information is held) so the compositing is not readily supported. Also the composition frequently needs to be done at video rate so requires some fast hardware.

Alpha blending and Antialiasing presents some problems but the bitmask, block copies and line stipple are easily accommodated. Good load balancing depends on even distribution of primitives. Not all primitives will take the same amount of time to process so a round robin distribution scheme, or a heuristic one with takes into account the expected processing time for each primitive will be needed.

Screen Subdivision—Blocks

Here the screen is divided up into large contiguous regions and a GLINT looks after each region. Primitives which overlap between regions are sent to both regions and scissor clipping used. Primitives contained wholly in one region are ideally just sent to the one GLINT.

The number of regions and the horizontal and/or vertical division of the screen can be chosen as appropriate, but horizontal bands are usually easier for the video hardware to cope with. Each GLINT only needs enough local buffer and frame buffer to cover the pixels in its own region, but texture maps are duplicated in full. Block copies are a problem when the block, or part block is moved between regions. Bit masking and line stipples can be solved with some careful clipping.

Load balancing is very poor in this paradigm, since most of the scene complexity can be concentrated into one region. Dynamically changing the size of the regions based on expected scene complexity (maybe measured from the previous frame) can alleviate the poor load balancing to some extent.

### Screen Subdivision—Interleaved Scanlines

The interleave factor is every other n$^{th}$ scanline where n is the number of GLINTs. Vertical interleaves are possible, but not supported by the GLINT rasterizer. Nearly all primitives will overlap multiple scanlines so are ideally broadcast to all GLINTs. Each GLINT will have different start values for the rasterization and interpolation parameters.

Each GLINT only needs enough local buffer and frame buffer to cover the pixels in its own region, but texture maps are duplicated in full.

Some block copies are a problem when the block is moved between non nth scanlines, but horizontal moves are available with any alignment. Bit masking can be solved with some careful clipping, but line stipples have no easy solution. Antialiasing is not normally a problem but with GLINT 300SX there is no provision for sub scanline steps as well as nth scanline steps. Load balancing is excellent in this paradigm which is the main reason it features prominently in the literature.

Thus the simplest and lowest risk method of using multiple GLINTs is Frame Interleaving, but if this is not an option, e.g. because of the transport delay or the amount of memory needed, then the next best choice is the Interleaved Scanline.

Linkage

FIG. 2B shows how the units are connected together. Some general points are:

The order of the units can be configured in two ways. The most general order (Router, Colour DDA, Texture Units, Fog Unit, Alpha Test, LB Rd, GID/Z/Stencil, LB Wr, Multiplexer) and will work in all modes of OpenGL. However, when the alpha test is disabled it is much better to do the Graphics ID, depth and stencil tests before the texture operations rather than after. This is because the texture operations have a high processing cost and this should not be spent on fragments which are later rejected because of window, depth or stencil tests.

Router Unit Description

The Router Unit allows the order of some of the units to be changed so that texturing can be done before or after the depth test. Any texture operations will cause a loss in performance over the same non-textured rendering, so it is a good idea only to texture those pixels which pass all the depth, stencil and GID tests. OpenGL defines the order in which operations are to be performed on fragments as texture, alpha test, stencil and then depth. It is very likely that in a typical scene many textured fragments will get rejected by the depth test, say, which isn't the most effective use of the texturing capacity. If the alpha test is disabled (or cannot reject fragments) then OpenGL compatible semantics are still maintained if the order is rearranged to be stencil, depth, texture and then alpha test.

The message stream can be re-configured into either of the two orders using the RouterMode message. The reset order is texture, then depth so a to be compatible with OpenGL. Changing the pipeline order is self synchronising so the user doesn't need to wait for the message stream to empty first.

Implementation

This unit is divided into two sub-units: a switcher and a multiplexer. FIG. 5A shows how these are connected together. The basic operation is as follows:

When the Switcher sub-unit receives a RouterMode message it makes a note of the new order, forwards the RouterMode message on and blocks all further messages until it receives a resume signal from the Multiplexer sub unit. When the resume signal is asserted the Switcher re-configures the message paths according to the new order and un-blocks the message stream so it starts to flow again.

When the Multiplexer sub-unit receives the RouterMode message it re-configures the message paths according to the new order and asserts the resume signal to the Switcher. The RouterMode message is consumed. The unit order is controlled using the RouterMode message. It uses the 0-bit of the passed message to indicate if the processing order is:

| | |
|---|---|
| Bit 0=0 | TextureDepth |
| Bit 0=1 | DepthTexture |

When the order is TextureDepth (the default after reset) the message routing is done according to FIG. 5B. When the order is DepthTexture the message routing is done according to FIG. 5C.

### Disclosed Embodiments

Among the disclosed classes of preferred embodiments, there is provided: A method for processing graphics data through a data path comprising the steps of: (a) receiving a routing command from a data bus input; (b) stalling further input from said data bus input until previous data has exited said data path; (c) resuming said input from said data bus input; (d) if said routing command has a first value, then performing a first set of graphics processes on said data, and then performing a second set of graphics processes on said data; (e) if said routing command has a second value, thenperforming said second set of graphics processes on said data, and thenperforming said first set of graphics processes on said data, wherein some portion of said data may be eliminated by said first or second sets of graphics process according to the results of said processes; wherein steps (d) and (e) are repeated until a new routing command is received; wherein said first set of graphics processes requires a longer processing time than said second set of graphics processes.

APPENDIX O

Among the disclosed classes of preferred embodiments, there is also provided: A method for processing graphics data through a data path comprising the steps of: (a) receiving a routing command from a data bus input; (b) stalling further input from said data bus input until previous data has exited said data path; (c) resuming said input from said data bus input; (d) if said routing command has a first value, thenperforming a set of texturing processes on said data, and thenperforming a set of pixel elimination processes on said data; (e) if said routing command has a second value, thenperforming said set of pixel elimination processes on said data, and thenperforming said set of texturing processes on said data, wherein some portion of said data may be eliminated by said set of pixel elimination processes according to the results of said processes; wherein steps (d) and (e) are repeated until a new routing command is received; wherein said first set of graphics processes requires a longer processing time than said second set of graphics processes.

Among the disclosed classes of preferred embodiments, there is also provided: A method for rendering graphics data comprising the steps of: (a) receiving a routing command from a data bus input; (b) stalling further input from said data bus input until previous data has exited said data path; (c) resuming said input from said data bus input; (d) if said routing command has a first value, thenperforming a set of texturing processes on said data, and thenperforming a set of pixel elimination processes on said data; (e) if said routing command has a second value, thenperforming said set of pixel elimination processes on said data, and thenperforming said set of texturing processes on said data, wherein some portion of said data may be eliminated by said set of pixel elimination processes according to the results of said processes; (f) rendering said data and writing the results to a memory; (g) displaying the contents of said memory; wherein steps (d) and (e) are repeated until a new routing command is received;wherein said set of texturing processes requires a longer processing time than said set of pixel elimination processes.

Among the disclosed classes of preferred embodiments, there is also provided: A method for processing graphics data through a data path comprising the steps of: (a) receiving a routing command from a data bus input; (b) stalling further input from said data bus input until previous data has exited said data path; (c) resuming said input from said data bus input; (d) if said routing command has a first value, thenreading said graphics data from said data bus input; performing a color DDA process on said data;performing a texturing process on said data;performing an alpha test on said data; if the data has passed the previous test, then performing a graphics ID test on said data; if the data has passed the previous tests, then performing a stencil test on said data;if the data has passed the previous tests, then performing a depth test on said data; and if the data has passed the previous tests, then writing said data to a local bus; (e) if said routing command has a second value, thenreading said graphics data from said data bus input; performing a graphics ID test on said data;if the data has passed the previous test, then performing a stencil test on said data; if the data has passed the previous tests, then performing a depth test on said data; if the data has passed the previous tests, then performing a color DDA process on said data; if the data has passed the previous tests, then performing a texturing process on said data; if the data has passed the previous tests, then performing an alpha test on said data; if the data has passed the previous tests, then writing said data to a local bus; wherein steps (d) and (e) are repeated until a new routing command is received.

Among the disclosed classes of preferred embodiments, there is also provided: A pipelined graphics processing device, comprising:a switching device connected to a data bus input and configured to route graphics data received on said data bus according to instruction data received on said data bus; a multiplexing device connected to said switching device and to a data bus output; a first processing block connected and configured to receive said graphics data from said switching device and pass processed graphics data to said multiplexing device; anda second processing block connected and configured to receive said graphics data from said switching device and pass processed graphics data to said multiplexing device; wherein said switching device routes said graphics data according to a first data path, wherein said graphics data is processed by said first processing block and then by said second processing block, or a second data path, wherein said graphics data is processed by said second processing block before said first processing block, according to said instruction data.

Among the disclosed classes of preferred embodiments, there is also provided: A pipelined graphics processing device, comprising: a routing device connected to a data bus input and data bus output and configured to route graphics data received on said data bus according to instruction data received on said data bus; a first processing block connected and configured to receive said graphics data from said routing device and pass processed graphics data back to said routing device; anda second processing block connected and configured to receive said graphics data from said routing device and pass processed graphics data back to said routing device; wherein said routing device routes data according to a first data path, wherein said graphics data is processed by said first processing block and then by said second processing block, or a second data path, wherein said graphics data is processed by said second processing block before said first processing block, according to said instruction data.

Among the disclosed classes of preferred embodiments, there is also provided: A graphics processing subsystem, comprising: at least four functionally distinct processing units, each including hardware elements which are customized to perform a rendering operation which is not performed by at least some others of said processing units; at least some ones of said processing units being connected to operate asynchronously to one another; a frame buffer, connected to be accessed by at least one of said processing units;said processing units being mutually interconnected in a pipeline relationship, with at least some successive ones of said processing units being interconnected through a FIFO buffer; and wherein at least one said processing unit is connected to look downstream, in said pipeline relationship, past the immediately succeeding one of said processors; and wherein at least two of said processing units may be dynamically reordered in said pipeline relationship; whereby the duty cycle of said processors is increased while permitting use of a reduced depth for said FIFO.

## Modifications and Variations

As will be recognized by those skilled in the art, the innovative concepts described in the present application can be modified and varied over a tremendous range of applications, and accordingly the scope of patented subject matter is not limited by any of the specific exemplary teachings given.

The foregoing text has indicated a large number of alternative implementations, particularly at the higher levels, but these are merely a few examples of the huge range of possible variations.

For example, the preferred chip context can be combined with other functions, or distributed among other chips, as will be apparent to those of ordinary skill in the art.

For another example, the described graphics systems and subsystems can be used, in various adaptations, not only in high-end PC's, but also in workstations, arcade games, and high-end simulators.

For another example, the described graphics systems and subsystems are not necessarily limited to color displays, but can be used with monochrome systems.

For another example, the described graphics systems and subsystems are not necessarily limited to displays, but also can be used in printer drivers.

What is claimed is:

1. A method for processing graphics data through a data path comprising the steps of:

(a) receiving a routing command from a data bus input;

(b) stalling further input from said data bus input until previous data has exited said data path;

(c) resuming said input from said data bus input;

(d) if said routing command has a first value, then performing a first set of graphics processes on said data, and then performing a second set of graphics processes on said data;

(e) if said routing command has a second value, then performing said second set of graphics processes on said data, and then performing said first set of graphics processes on said data, wherein some portion of said data is selectively eliminated by said first or second sets of graphics process according to the results of said processes;

wherein steps (d) and (e) are repeated until a new routing command is received;

wherein said first set of graphics processes requires a longer processing time than said second set of graphics processes.

2. The method of claim 1, wherein said first set of graphics processes comprises the steps of:

reading said graphics data from said data bus input;

performing a color DDA process on said data;

performing a texturing process on said data; and

performing an alpha test on said data.

3. The method of claim 1, wherein said second set of graphics processes comprises the step of if the data has passed all previous tests, then performing a graphics ID test on said data.

4. The method of claim 1, wherein said second set of graphics processes comprises the step of if the data has passed the previous tests, then performing a stencil test on said data.

5. The method of claim 1, wherein said second set of graphics processes comprises the steps of if the data has passed the previous tests, then performing a depth test on said data.

6. The method of claim 1, wherein step (d) comprises steps according to the OpenGL standard.

7. The method of claim 1, wherein step (b) is performed by a switcher connected at said data bus input.

8. The method of claim 1, wherein a multiplexer at an output of said data path indicates when said data path is clear and step (c) can begin.

9. A method for processing graphics data through a data path comprising the steps of:

(a) receiving a routing command from a data bus input;

(b) stalling further input from said data bus input until previous data has exited said data path;

(c) resuming said input from said data bus input;

(d) if said routing command has a first value, then performing a set of texturing processes on said data, and then performing a set of pixel elimination processes on said data;

(e) if said routing command has a second value, then performing said set of pixel elimination processes on said data, and then performing said set of texturing processes on said data, wherein some portion of said data is selectively eliminated by said set of pixel elimination processes according to the results of said processes;

wherein steps (d) and (e) are repeated until a new routing command is received;

wherein said first set of graphics processes requires a longer processing time than said second set of graphics processes.

10. A method for rendering graphics data comprising the steps of:

(a) receiving a routing command from a data bus input;

(b) stalling further input from said data bus input until previous data has exited said data path;

(c) resuming said input from said data bus input;

(d) if said routing command has a first value, then performing a set of texturing processes on said data, and then performing a set of pixel elimination processes on said data;

(e) if said routing command has a second value, then performing said set of pixel elimination processes on said data, and then performing said set of texturing processes on said data, wherein some portion of said data is selectively eliminated by said set of pixel elimination processes according to the results of said processes;

(f) rendering said data and writing the results to a memory;

(g) displaying the contents of said memory;

wherein steps (d) and (e) are repeated until a new routing command is received;

wherein said set of texturing processes requires a longer processing time than said set of pixel elimination processes.

11. A method for processing graphics data through a data path comprising the steps of:

(a) receiving a routing command from a data bus input;

(b) stalling further input from said data bus input until previous data has exited said data path;

(c) resuming said input from said data bus input;

(d) if said routing command has a first value, then reading said graphics data from said data bus input; performing a color DDA process on said data; performing a texturing process on said data; performing an alpha test on said data; if the data has passed the previous test, then performing a graphics ID test on said data; if the data has passed the previous tests, then performing a stencil test on said data;

if the data has passed the previous tests, then performing a depth test on said data; and

if the data has passed the previous tests, then writing said data to a local bus;

(e) if said routing command has a second value, then reading said graphics data from said data bus input; performing a graphics ID test on said data;

if the data has passed the previous test, then performing a stencil test on said data;

if the data has passed the previous tests, then performing a depth test on said data;

if the data has passed the previous tests, then performing a color DDA process on said data;

if the data has passed the previous tests, then performing a texturing process on said data;

if the data has passed the previous tests, then performing an alpha test on said data;

if the data has passed the previous tests, then writing said data to a local bus;

wherein steps (d) and (e) are repeated until a new routing command is received.

**12**. The method of claim **11**, wherein step (d) comprises steps according to the OpenGL standard.

**13**. The method of claim **11**, wherein step (b) is performed by a switcher connected at said data bus input.

**14**. The method of claim **11**, wherein a multiplexer at said local bus indicates when said data path is clear and step (c) can begin.

**15**. A pipelined graphics processing device, comprising:

a switching device connected to a data bus input and configured to route graphics data received on said data bus according to instruction data received on said data bus;

a multiplexing device connected to said switching device and to a data bus output;

a first processing block connected and configured to receive said graphics data from said switching device and pass processed graphics data to said multiplexing device; and

a second processing block connected and configured to receive said graphics data from said switching device and pass processed graphics data to said multiplexing device;

wherein said switching device routes said graphics data according to a first data path, wherein said graphics data is processed by said first processing block and then by said second processing block, or a second data path, wherein said graphics data is processed by said second processing block before said first processing block, according to said instruction data.

**16**. The device of claim **15**, wherein said first data path processes said graphics data according to the OpenGL standard.

**17**. The device of claim **15**, wherein said switching device halts all input data until the current data path is clear before switching data paths.

**18**. The device of claim **15**, wherein said multiplexing device is configured to determine when the current data path is clear and to allow said switching device to switch data paths.

**19**. A pipelined graphics processing device, comprising:

a routing device connected to a data bus input and data bus output and configured to route graphics data received on said data bus according to instruction data received on said data bus;

a first processing block connected and configured to receive said graphics data from said routing device and pass processed graphics data back to said routing device; and

a second processing block connected and configured to receive said graphics data from said routing device and pass processed graphics data back to said routing device;

wherein said routing device routes data according to a first data path, wherein said graphics data is processed by said first processing block and then by said second processing block, or a second data path, wherein said graphics data is processed by said second processing block before said first processing block, according to said instruction data.

**20**. A graphics processing subsystem, comprising:

at least four functionally distinct processing units, each including hardware elements which are customized to perform a rendering operation which is not performed by at least some others of said processing units; at least some ones of said processing units being connected to operate asynchronously to one another;

a frame buffer, connected to be accessed by at least one of said processing units;

said processing units being mutually interconnected in a pipeline relationship, with at least some successive ones of said processing units being interconnected through a FIFO buffer;

and wherein at least one said processing unit is connected to look downstream, in said pipeline relationship, past the immediately succeeding one of said processors;

and wherein at least two of said processing units are selectively dynamically reordered in said pipeline relationship;

whereby the duty cycle of said processors is increased while permitting use of a reduced depth for said FIFO.

**21**. The graphics processing subsystem of claim **20**, wherein said processing units include a texturing unit.

**22**. The graphics processing subsystem of claim **20**, wherein said processing units include a scissoring unit.

**23**. The graphics processing subsystem of claim **20**, wherein said processing units include a memory access unit which reads and writes a local buffer memory.

**24**. The graphics processing subsystem of claim **20**, wherein at least some ones of said processing units include internally paralleled data paths.

**25**. The graphics processing subsystem of claim **20**, wherein all of said processing units are integrated into a single integrated circuit.

**26**. The graphics processing subsystem of claim **20**, wherein all of said processing units, but not said frame buffer, are integrated into a single integrated circuit.

* * * * *

APPENDIX P

US005987256A

# United States Patent [19]

**Wu et al.**

[11] **Patent Number: 5,987,256**

[45] **Date of Patent: Nov. 16, 1999**

[54] **SYSTEM AND PROCESS FOR OBJECT RENDERING ON THIN CLIENT PLATFORMS**

[75] Inventors: **Bo Wu**; **Ling Lu**, both of San Jose, Calif.

[73] Assignee: **Enreach Technology, Inc.**, San Jose, Calif.

[21] Appl. No.: **08/922,898**

[22] Filed: **Sep. 3, 1997**

[51] **Int. Cl.$^6$** .......... **G06F 9/45**

[52] **U.S. Cl.** .......... **395/707**

[58] **Field of Search** .......... 395/707

[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 5,504,917 | 4/1996 | Austin et al. | 345/522 |
| 5,513,127 | 4/1996 | Gard et al. | 395/200.53 |
| 5,551,015 | 8/1996 | Goettelmann et al. | 395/707 |
| 5,586,020 | 12/1996 | Isozaki | 395/707 |
| 5,826,089 | 10/1998 | Ireton | 395/707 |

OTHER PUBLICATIONS

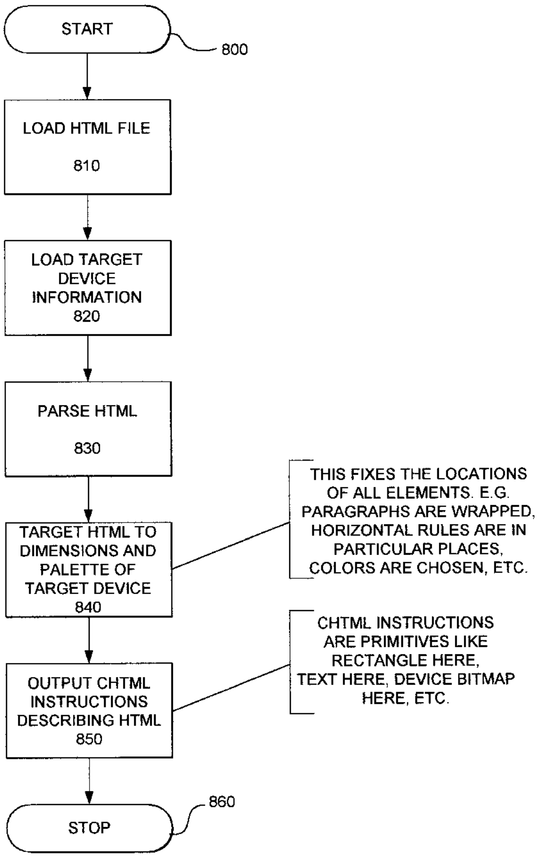
Blickstein et al. The GEM optimizing compiler system; Digital Technical Journal vol. 4 No. 4 Special Issue 1992 pp. 121–136.

*Primary Examiner*—Robert W. Downs
*Assistant Examiner*—Wei Zhen
*Attorney, Agent, or Firm*—Wilson Sonsini Goodrich & Rosati

[57] **ABSTRACT**

A system for processing an object specified by an object specifying language such as HTML, JAVA or other languages relying on relative positioning, that require a rendering program utilizing a minimum set of resources, translates the code for use in a target device that has limited processing resources unsuited for storage and execution of the HTML rendering program, JAVA virtual machine, or other rendering engine for the standard. Data concerning such an object is generated by a process that includes first receiving a data set specifying the object according to the object specifying language, translating the first data set into a second data set in an intermediate object language adapted for a second rendering program suitable for rendering by the target device that utilizes actual target display coordinates. The second data set is stored in a machine readable storage device, for later retrieval and execution by the thin client platform.

**14 Claims, 11 Drawing Sheets**


START
800
LOAD HTML FILE
810
LOAD TARGET DEVICE INFORMATION
820
PARSE HTML
830
TARGET HTML TO DIMENSIONS AND PALETTE OF TARGET DEVICE
840
OUTPUT CHTML INSTRUCTIONS DESCRIBING HTML
850
STOP
860

THIS FIXES THE LOCATIONS OF ALL ELEMENTS. E.G. PARAGRAPHS ARE WRAPPED, HORIZONTAL RULES ARE IN PARTICULAR PLACES, COLORS ARE CHOSEN, ETC.

CHTML INSTRUCTIONS ARE PRIMITIVES LIKE RECTANGLE HERE, TEXT HERE, DEVICE BITMAP HERE, ETC.

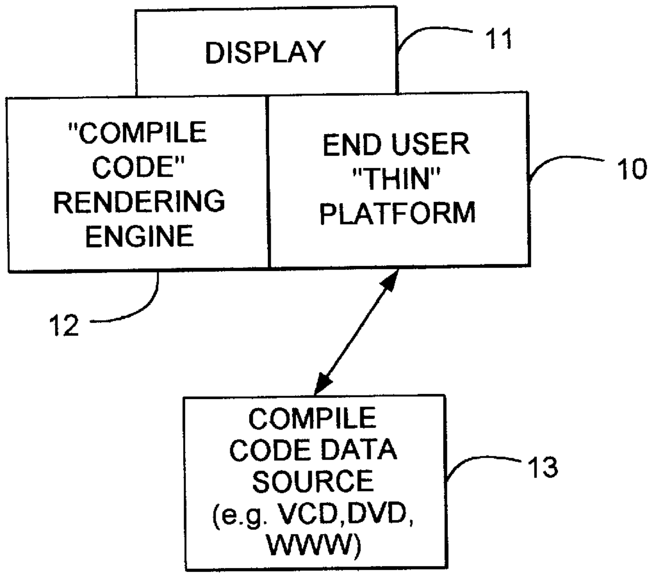

DISPLAY
11
"COMPILE CODE" RENDERING ENGINE
END USER "THIN" PLATFORM
10
12
COMPILE CODE DATA SOURCE (e.g. VCD,DVD, WWW)
13

FIG. 1

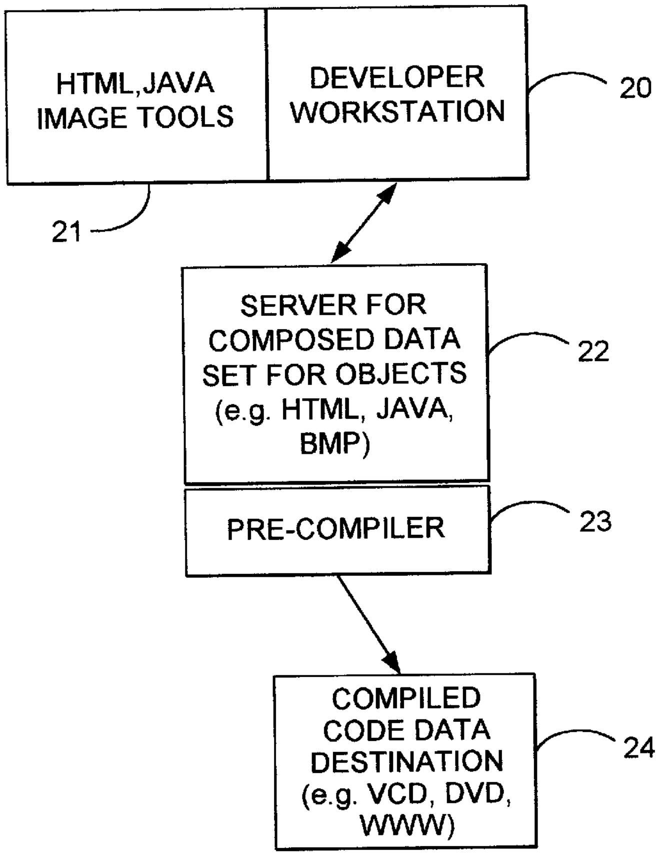

HTML,JAVA IMAGE TOOLS
DEVELOPER WORKSTATION
20
21
SERVER FOR COMPOSED DATA SET FOR OBJECTS (e.g. HTML, JAVA, BMP)
22
PRE-COMPILER
23
COMPILED CODE DATA DESTINATION (e.g. VCD, DVD, WWW)
24

FIG.2

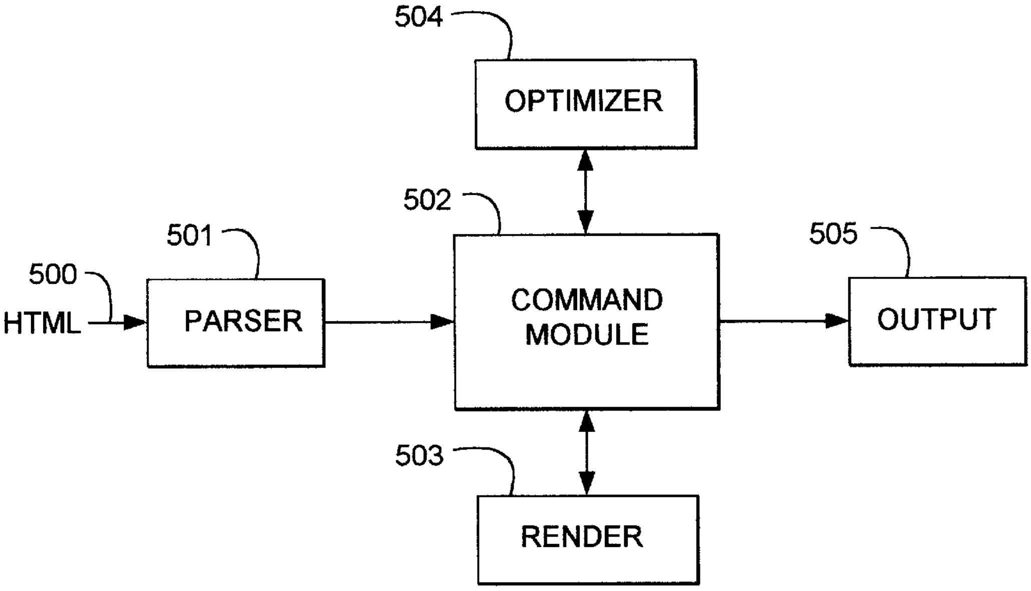

504
OPTIMIZER
502
501
505
500
HTML
PARSER
COMMAND MODULE
OUTPUT
503
RENDER

FIG. 3

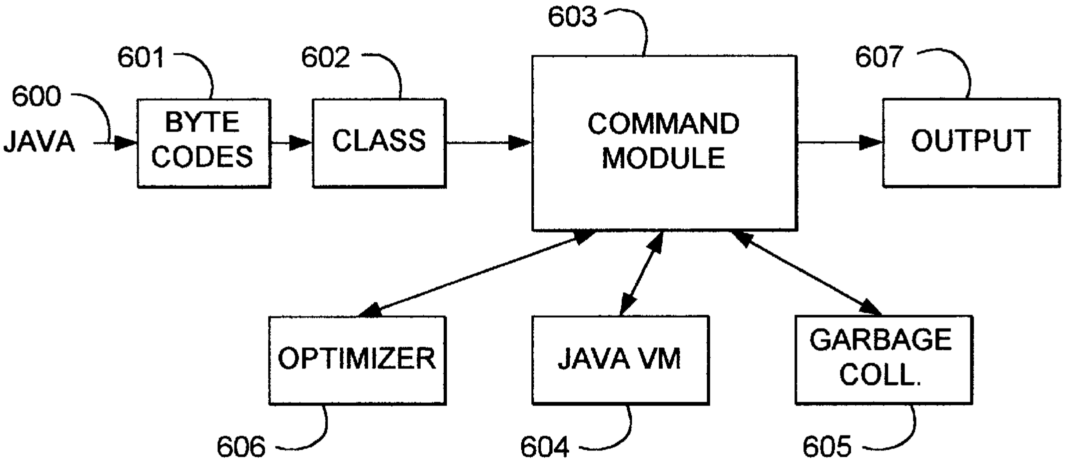

603
601
602
607
600
JAVA
BYTE CODES
CLASS
COMMAND MODULE
OUTPUT
OPTIMIZER
JAVA VM
GARBAGE COLL.
606
604
605

FIG.4

APPENDIX P

```
Class Inheritance Hierachy :
   CObject
         |_ChtmlElement
                     |_CHtmlElementTitle
                     |_CHtmlElementBody
                     |_CHtmlElementText
                     |_CHtmlElementPlainText
                     |_CHtmlElementComments
                     |_CHtmlElementLineBreak
                     |_CHtmlElementHorizontalRule
                     |_CHtmlElementFontElement
                     |_CHtmlElementBaseFont
                     |_CHtmlElementListItem
                     |_CHtmlElementTable
                                |_CHtmlElementBase
                                |_CHtmlElementWhereBreak
                                |_CHtmlElementMap
                                |_CHtmlElementIsIndex
                                |_CHtmlElementStyle
                                |_CHtmlElementApplet
                                |_CHtmlElementSound
                     |_CHtmlElementSpacer
                     |
                     |_. . .   . . .
                     |
                     |_CHtmlRectangleElement
                     |          |_CHtmlElementImage
                     |          |_CHtmlControlElement
                     |                     |_CHtmlElementInput
                     |                     |_CHtmlElementTextarea
                     |                     |_CHtmlElementSelect
                     |                     |_CHtmlElementOption
                     |
                     |                . . .  . . .
                     |
                     |_CHtmlContainerElement
                                |_CHtmlElementAnchor
                                |_CHtmlElementAlign
                                |_CHtmlElementHeader
                                |_CHtmlElementParagraph
                                |_CHtmlElementList
                                |_CHtmlElementFont
                                |_CHtmlElementDivision
                                |_CHtmlElementBlockquote
                                |_CHtmlElementAddress
                                |_CHtmlElementTableCell
                                |_CHtmlElementTableCaption
                                |_CHtmlElementPreformat
                                |_CHtmlElementForm

                                |_CHtmlElementSupSub
                                |_CHtmlElementNoBreak
                                                   |_CHtmlElementSpan
                                         |_CHtmlElementFrameSet
                                                 |_CHtmlElementFrame
```

FIG. 5

```
Class Containership Hierachy :
   CIEditorDoc                                    CHtmlContainerElement
         |_ChtmlElementArray                               |_ChtmlElementArray
                     |_ChtmlElement1                                  |_ChtmlElement1
                     |_ChtmlElement2                                  |_ChtmlElement2
                         . . .  . . .                                     . . .  . . .
                     |_ChtmlElementn                                  |_ChtmlElementn
```

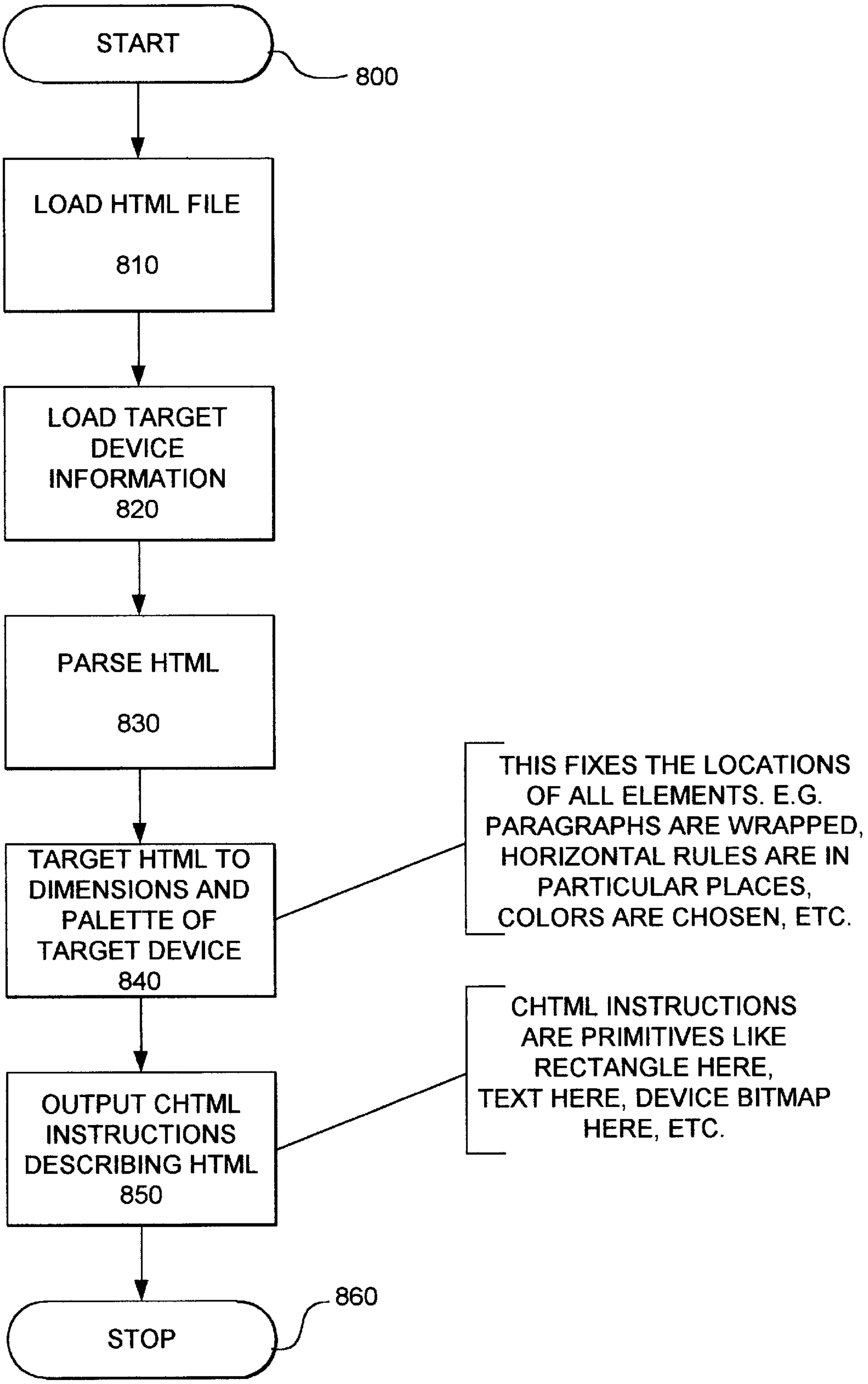

START
800
LOAD HTML FILE
810
LOAD TARGET DEVICE INFORMATION
820
PARSE HTML
830
TARGET HTML TO DIMENSIONS AND PALETTE OF TARGET DEVICE
840
THIS FIXES THE LOCATIONS OF ALL ELEMENTS. E.G. PARAGRAPHS ARE WRAPPED, HORIZONTAL RULES ARE IN PARTICULAR PLACES, COLORS ARE CHOSEN, ETC.
OUTPUT CHTML INSTRUCTIONS DESCRIBING HTML
850
CHTML INSTRUCTIONS ARE PRIMITIVES LIKE RECTANGLE HERE, TEXT HERE, DEVICE BITMAP HERE, ETC.
STOP
860

FIG. 6

APPENDIX P

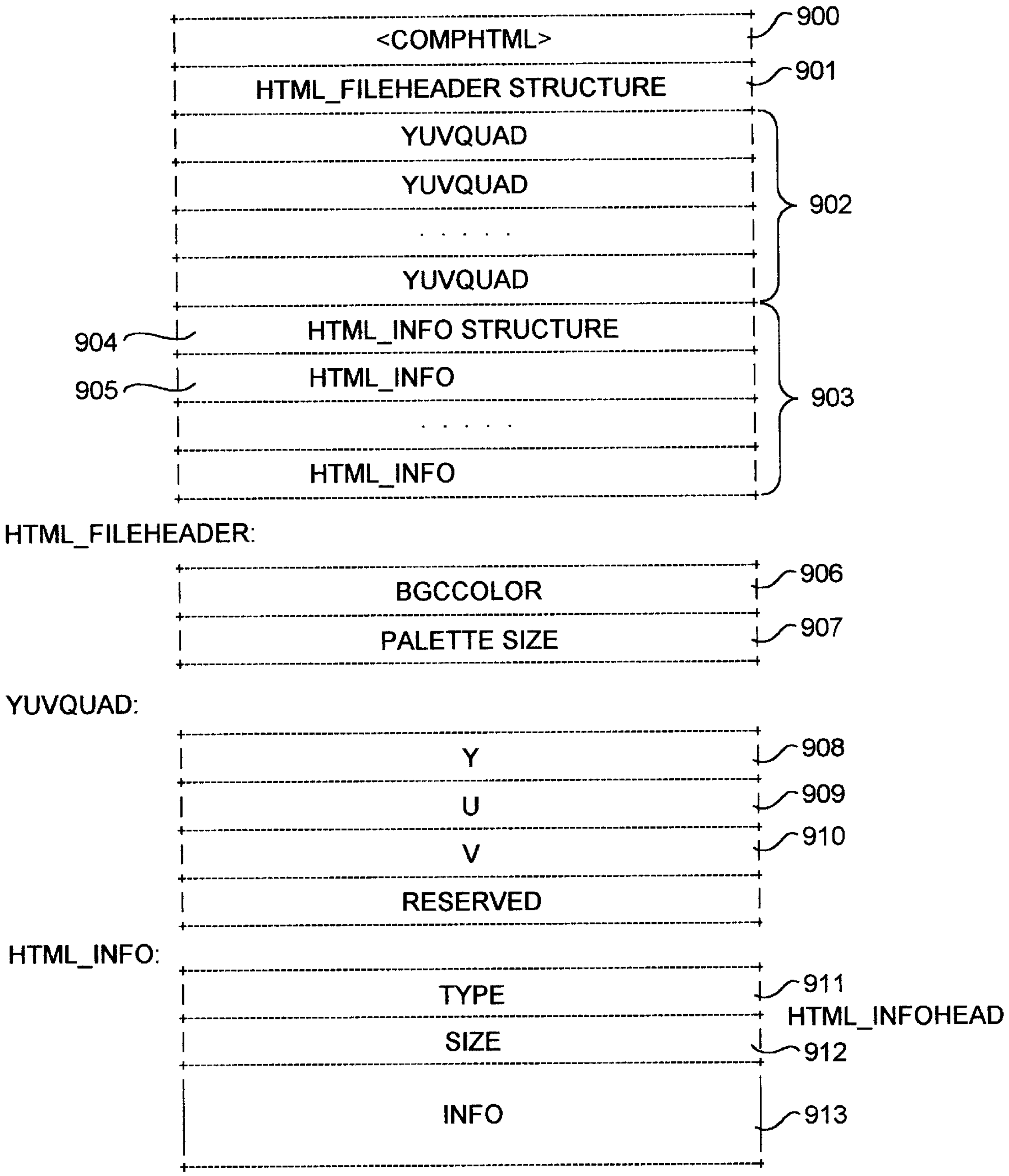


FIG. 7

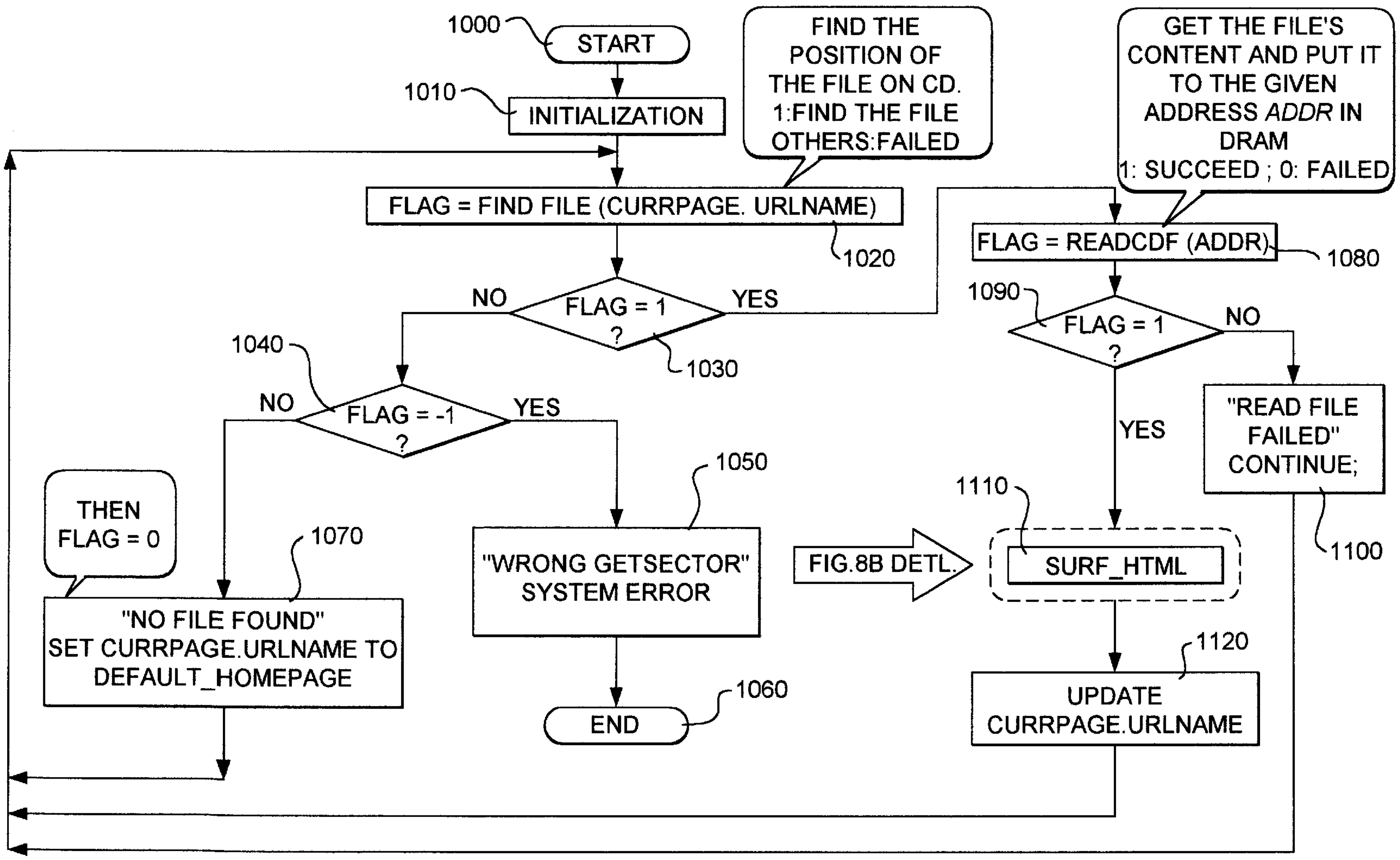

1000
START
1010
INITIALIZATION
FIND THE POSITION OF THE FILE ON CD. 1:FIND THE FILE OTHERS:FAILED
FLAG = FIND FILE (CURRPAGE. URLNAME)
1020
NO
FLAG = 1 ?
YES
1030
1040
NO
FLAG = -1 ?
YES
THEN FLAG = 0
1070
"NO FILE FOUND" SET CURRPAGE.URLNAME TO DEFAULT_HOMEPAGE
1050
"WRONG GETSECTOR" SYSTEM ERROR
1060
END
GET THE FILE'S CONTENT AND PUT IT TO THE GIVEN ADDRESS *ADDR* IN DRAM 1: SUCCEED ; 0: FAILED
FLAG = READCDF (ADDR)
1080
1090
FLAG = 1 ?
NO
YES
"READ FILE FAILED" CONTINUE;
1100
1110
FIG.8B DETL.
SURF_HTML
1120
UPDATE CURRPAGE.URLNAME

FIG. 8A

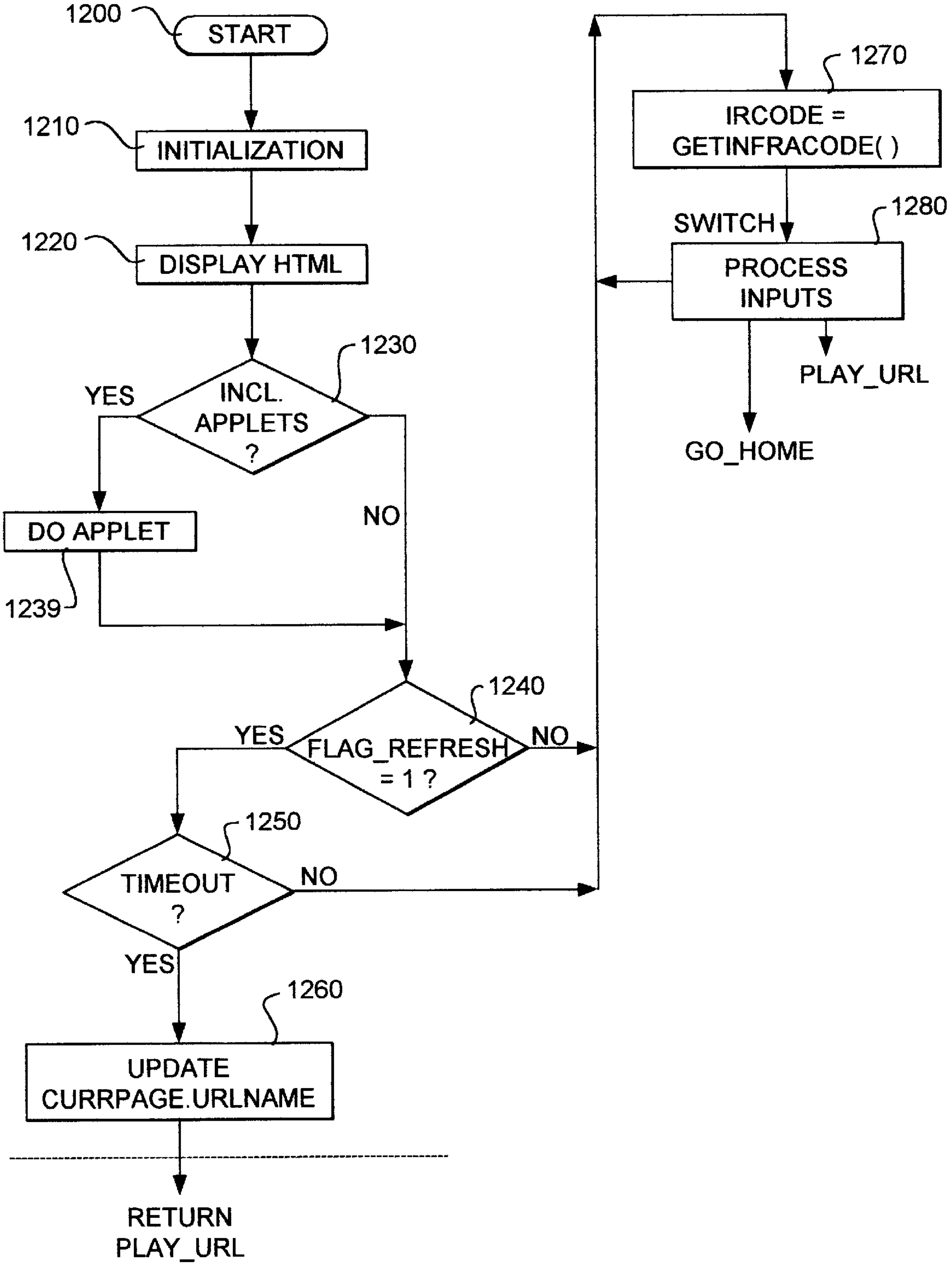

1200
START
1210
INITIALIZATION
1220
DISPLAY HTML
1230
INCL. APPLETS ?
YES
NO
DO APPLET
1239
1240
FLAG_REFRESH = 1 ?
YES
NO
1250
TIMEOUT ?
NO
YES
1260
UPDATE CURRPAGE.URLNAME
RETURN PLAY_URL
1270
IRCODE = GETINFRACODE( )
SWITCH
1280
PROCESS INPUTS
PLAY_URL
GO_HOME

FIG.8B

APPENDIX P

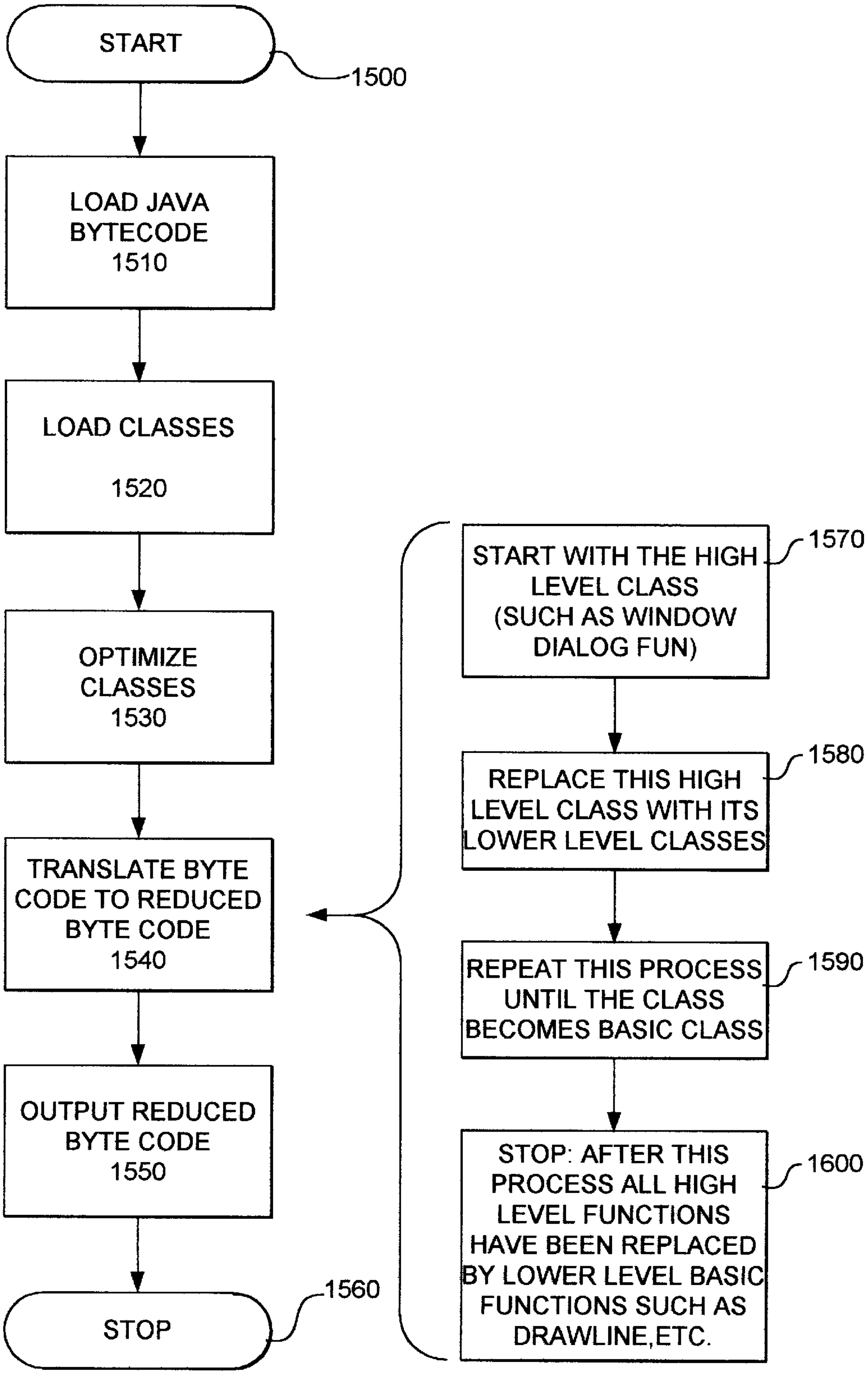

START
1500
LOAD JAVA BYTECODE 1510
LOAD CLASSES 1520
OPTIMIZE CLASSES 1530
TRANSLATE BYTE CODE TO REDUCED BYTE CODE 1540
OUTPUT REDUCED BYTE CODE 1550
STOP
1560
START WITH THE HIGH LEVEL CLASS (SUCH AS WINDOW DIALOG FUN)
1570
REPLACE THIS HIGH LEVEL CLASS WITH ITS LOWER LEVEL CLASSES
1580
REPEAT THIS PROCESS UNTIL THE CLASS BECOMES BASIC CLASS
1590
STOP: AFTER THIS PROCESS ALL HIGH LEVEL FUNCTIONS HAVE BEEN REPLACED BY LOWER LEVEL BASIC FUNCTIONS SUCH AS DRAWLINE,ETC.
1600

FIG. 9

FIG. 9A

APPENDIX P

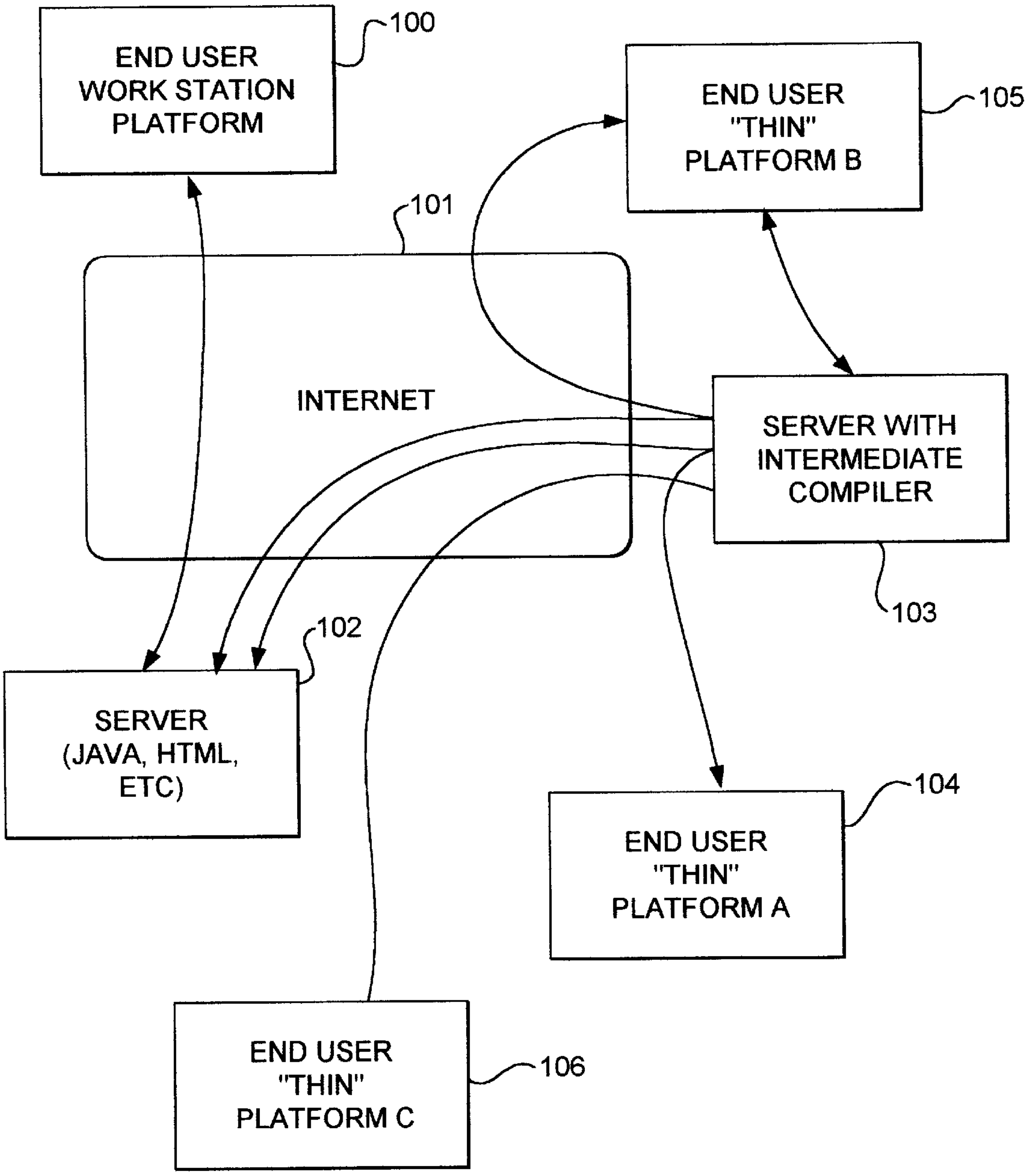

END USER WORK STATION PLATFORM
100
END USER "THIN" PLATFORM B
105
101
INTERNET
SERVER WITH INTERMEDIATE COMPILER
103
102
SERVER (JAVA, HTML, ETC)
104
END USER "THIN" PLATFORM A
END USER "THIN" PLATFORM C
106

FIG. 10

APPENDIX P

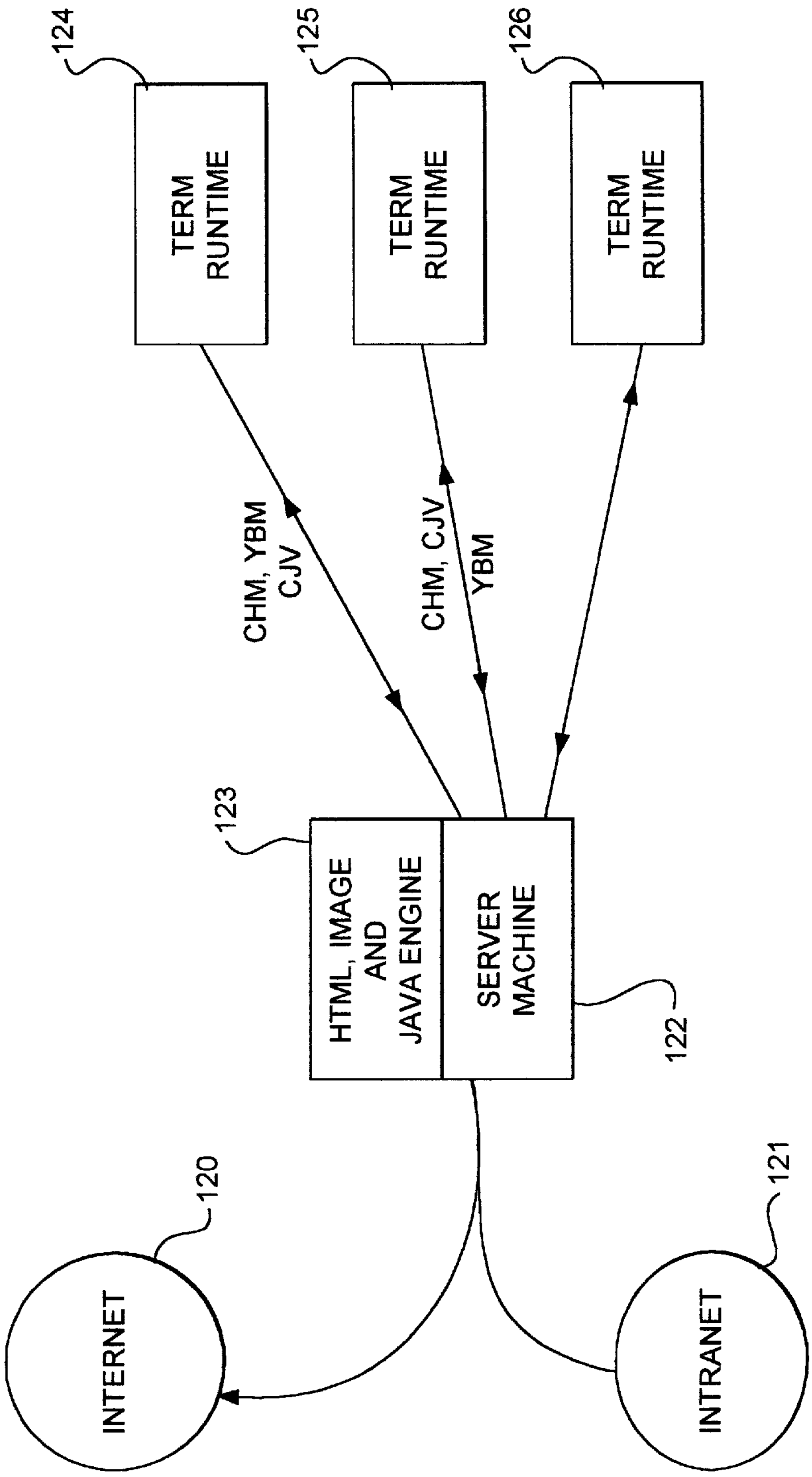


FIG. 11

APPENDIX P

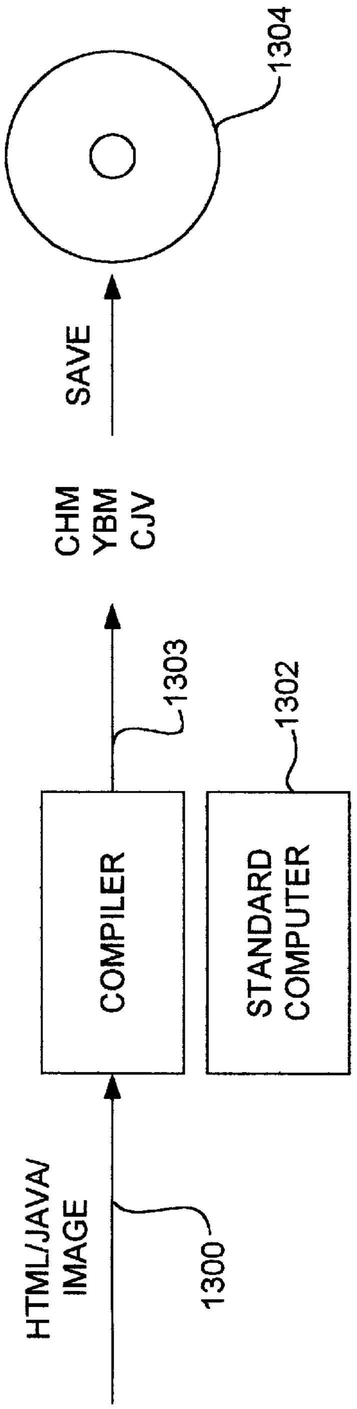


FIG. 12A

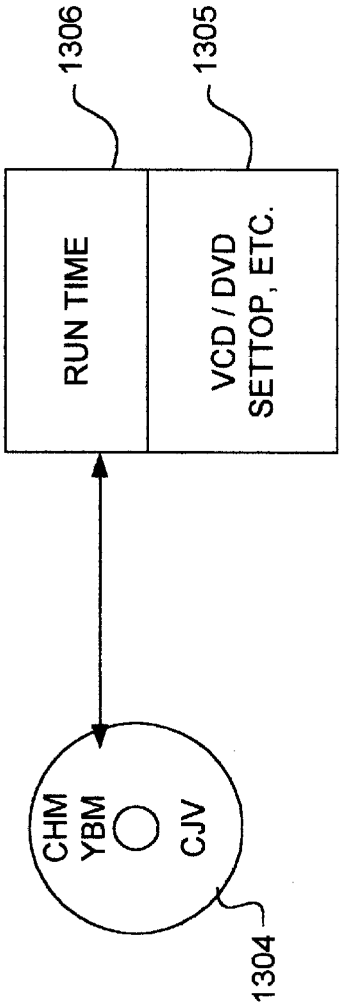


FIG. 12B

# SYSTEM AND PROCESS FOR OBJECT RENDERING ON THIN CLIENT PLATFORMS

## COPYRIGHT DISCLAIMER

## BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention generally relates to a method of providing full feature program processing according to a variety of standard language codes such as HTML, JAVA and other standard languages, for execution on a thin client platform. More particularly the invention relates to methods for compiling and rendering full feature standard HTML and JAVA programs into a format which is efficient for a limited processing resource platforms.

2. Description of Related Art

Standard HTML and JAVA programs, and other hypertext languages, are designed for computers having a significant amount of data processing resources, such as CPU speed and memory bandwidth, to run well. One feature of these object specifying languages is the ability to specify a graphic object for display using relative positioning. Relative positioning enables the display of the graphic object on displays having a wide range of dimensions, resolutions, and other display characteristics. However, relative positioning of graphic objects requires that the target device have computational resources to place the graphic object on the display at specific coordinates. Thus, there are a number of environments, such as TV set top boxes, hand held devices, digital video disk DVD players, compact video disk VCD players or thin network computer environments in which these standard object specifying languages are inefficient or impractical. The original HTML and JAVA programs run very slowly, or not at all, in these types of thin client environments. To solve these problems, simpler versions of HTML and JAVA have been proposed, which have resulted in scripting out some of the features. This trades off some of the nice functionality of HTML and JAVA, which have contributed to their wide acceptance. Furthermore, use in thin client environments of the huge number of files that are already specified according to these standards, is substantially limited.

## SUMMARY OF THE INVENTION

The present invention provides a system and method for processing an Display object specified by an object specifying language such as HTML, JAVA or other languages relying on relative positioning, that require a rendering program utilizing a minimum set of resources, for use in a target device that has limited processing resources unsuited for storage and execution of the HTML rendering program, JAVA virtual machine, or other rendering engine for the standard. Thus, the invention can be characterized as a method for storing data concerning such an object that includes first receiving a data set specifying the object according to the object specifying language, translating the first data set into a second data set in an intermediate object language adapted for a second rendering program suitable for rendering by the target device that utilizes actual target display coordinates. The second data set is stored in a machine readable storage device, for later retrieval and execution by the thin client platform.

The object specifying language according to alternative embodiments comprises a HTML standard language or other hypertext mark up language, a JAVA standard language or other object oriented language that includes object specifying tools.

The invention also can be characterized as a method for sending data concerning such an object to a target device having limited processing resources. This method includes receiving the first data set specifying the object according to the first object specifying language, translating the first data set to a second data set in an intermediate object language, and then sending the second data set to the target device. The target device then renders the object by a rendering engine adapted for the intermediate object language. The step of sending the second data set includes sending the second data set across a packet switched network such as the Internet or the World Wide Web to the target device. Also, the step of translating according to one aspect of the invention includes sending the first data set across a packet switched network to a translation device, and executing a translation process on the translation device to generate the second data set. The second data set is then transferred from the translation device, to the target device, or alternatively from the translation device back to the source of the data, from which it is then forwarded to the target device.

According to other aspects of the invention, the step of translating the first data set includes first identifying the object specifying language of the first data set from among a set of object specifying languages, such as HTML and JAVA. Then, a translation process is selected according to the identified object specifying language.

According to yet another aspect of the invention, before the step of translating the steps of identifying the target device from among a set of target devices, and selecting a translation process according to the identified target device, are executed.

In yet another alternative of the present invention, a method for providing data to a target device is provided. This method includes requesting for the target device a first data set from a source of data, the first data set specifying the object according to the object specifying language; translating the first data set to a second data set in an intermediate language adapted for execution according to a second rendering program by the target device. The second data set is then sent to this target device. This allows a thin platform target device to request objects specified by full function HTML, JAVA and other object specifying languages, and have them automatically translated to a format suitable for rendering in the thin environment.

Thus, the present invention provides a method which uses a computer to automatically compile standard HTML, JAVA and other programs so that such programs can run both CPU and memory efficiently on a thin client platform such as a TV set top box, a VCD/DVD player, a hand held device, a network computer or an embedded computer. The automatic compilation maintains all the benefits of full feature HTML and JAVA or other language.

The significance of the invention is evident when it is considered that in the prior art, standard HTML and JAVA were reduced in features or special standards are created for the thin client environment. Thus according to the prior art approaches, the standard programs and image files on the Internet need to be specially modified to meet the needs of special thin client devices. This is almost impossible considering the amount of HTML and JAVA formatted files on the Web. According to the invention each HTML file, compiled JAVA class file or other object specifying language data set is processed by a standard full feature HTML browser JAVA virtual machine, or other complementary rendering engine, optimized for a target platform on the fly, and then output into a set of display oriented language codes which can be easily executed and displayed on a thin client platform. Furthermore, the technique can use in general to speed up the HTML and JAVA computing in standard platforms.

Other aspects and advantages of the present invention can be seen upon review of the figures, the detailed description and the claims which follow.

## BRIEF DESCRIPTION OF THE FIGURES

FIG. **1** is a simplified diagram of a end user thin platform for execution of a compiled code data source according to the present invention.

FIG. **2** is a simplified diagram of a user workstation and server for precompiling a composed data set according to the present invention.

FIG. **3** is a simplified diagram of a precompiler for a HTML formatted file.

FIG. **4** is a simplified diagram of a precompiler for a JAVA coded program.

FIG. **5** is a class inheritance hierarchy for a precompiler for HTML.

FIG. **6** is a flow chart for the HTML precompiler process.

FIG. **7** illustrates the compiled HTML structure according to one embodiment of the present invention.

FIGS. **8**A–**8**B illustrate a compiled HTML run time engine for execution on the thin platform according to the present invention.

FIG. **9** is a flow chart of the process for precompiling a JAVA program according to the present invention.

FIG. **9**A is a flow chart of one example process for translating the byte codes into a reduced byte code in the sequence of FIG. **9**.

FIG. **10** is a schematic diagram illustrating use of the present invention in the Internet environment.

FIG. **11** is a schematic diagram illustrating use of the present invention in a "network computer" environment.

FIG. **12**A is a schematic diagram illustrating use of the present invention in an off-line environment for producing a compiled format of the present invention and saving it to a storage medium.

FIG. **12**B illustrates the off-line environment in which the stored data is executed by thin platform.

## DETAILED DESCRIPTION

A detailed description of preferred embodiments of the present invention is provided with respect to FIGS. **1**–**12**A and **12**B. FIGS. **1**–**2** illustrated simplified implementation of the present invention. FIGS. **3**–**9** and **9**A illustrate processes executed according to the present invention. FIGS. **10**–**12**A and **12**B illustrate the use of the present invention in the Internet environment or other packet switched network environment.

FIG. **1** illustrates a "thin" platform which includes a limited set of data processing resources represented by box **10**, a display **11**, and a "compiled code" rendering engine **12** for a display oriented language which relies on the data processing resources **10**. The end user platform **10** is coupled to a compiled code data source **13**. A compiled code data sources comprises, for example a VCD, a DVD, or other computer readable data storage device. Alternatively, the compiled code data source **13** consists of a connection to the World Wide Web or other packet switched or point-to-point network environment from which compiled code data is retrieved.

The limited data processing resources of the thin platform **10** include for example a microcontroller and limited memory. For example, 512k of RAM associated with a 8051 microcontroller, or a 66 MHz MIPS RISC CPU and 512k of dynamic RAM may be used in a representative thin platform. Other thin user platforms use low cost microprocessors with limited memory. In addition, other thin platforms may comprise high performance processors which have little resources available for use in rendering the compiled code data source. Coupled with the thin platform is a compiled code rendering engine **12**. This rendering engine **12** is a relatively compact program which runs efficiently on the thin platform data processing resources. The rendering engine translates the compiled code data source data set into a stream of data suitable for the display **11**. In this environment, the present invention is utilized by having the standard HTML or JAVA code preprocessed and compiled into a compiled HTML/JAVA format according to the present invention using the compiler engine described in more detail below on a more powerful computer. The compiled HTML/JAVA codes are saved on the storage media. A small compiled HTML/JAVA run time engine **12** is embedded or loaded into the thin client device. The run time engine **12** is used to play the compiled HTML/JAVA files on the thin platform **10**. This enables the use of a very small client to run full feature HTML or JAVA programs. The machine can be used both online, offline or in a hybrid mode.

FIG. **2** illustrates the environment in which the compiled code data is generated according to the present invention. Thus for example, a developer workstation **20** is coupled with image rendering tools such as HTML, JAVA, or other image tools **21**. The workstation **20** is coupled to a server for the composed data **22**. The server includes a precompiler **23** which takes the composed data and translates it into the compiled code data. Compiled code data is then sent to a destination **24** where it is stored or rendered as suits the needs of a particular environment. Thus for example, the destination may be a VCD, DVD or the World Wide Web.

According to the environment of FIG. **2** compiled HTML and JAVA "middleware" is implemented on an Internet server. Thus the thin set top box or other compiled code data destination **24** is coupled to the Internet/Intranet through the compiled HTML/JAVA middleware **22**, **23**. A small compiled HTML/JAVA run time engine is embedded in the thin destination device. All the HTML/JAVA files created in the workstation **20** go through the middleware server **22** to reach the thin client devices. The HTML/JAVA files are converted to the compiled format on the fly by the precompiler **23** on the middleware server **22**. The server **22** passes the compiled code onto the destination device. This allows for most software updates of precompiler techniques to be made in the server environment without the need to update the destination devices. Also, any changes in the run time engine that need to be executed in the destination device **24** can be provided through the link to the server **22**.

FIGS. **3** and **4** illustrate simplified diagrams of the precompilers for HTML and JAVA respectively. In FIG. **3**,

APPENDIX P

standard HTML files are received at input **500** and applied to a HTML parser **501**. The output of the parser is applied to a command module **502** which includes a HTML rendering engine **503**, and memory resident HTML objects optimizing engine **504**. The output consists of the compiled HTML output engine **505** generates the output with simplified graphics primatives.

The basic class inheritance hierarchy for the HTML precompiling is shown in FIG. **5**. The process of translating a HTML file to the compiled HTML structure of the present invention is illustrated in FIG. **6**. The process begins at point **800** in FIG. **6**. The first step involves loading the HTML file into the rendering device. Next information concerning the target device is loaded (step **820**). The HTML file is then parsed by searching for HTML tags, and based on such tags creating the class structure of FIG. **5** (step **830**).

Using the parameters of the target device, and the parsing class structure set up after the parsing process, the algorithm does HTML rendering based on a class hierarchy adapted to the dimensions and palette of the target device (step **840**). This fixes the coordinates of all the graphic objects specified by the HTML code on the screen of the target device. For example, the paragraphs are word wrapped, horizontal rules are placed in particular places, the colors are chosen, and other device specific processes are executed.

After the rendering, all the display information is saved back into the class structure of FIG. **5**. Finally the process goes through the class hierarchy and outputs the rendering information in compiled HTML format (step **850**). The compiled HTML instructions are primitives that define rectangles, text, bitmaps and the like and their respective locations. After outputting the compiled instructions, the process is finished (step **860**).

A simplified pseudo code for the HTML compilation process is provided in Table 1.

TABLE 1

Copyright EnReach 1997

```
function convert_html (input : pointer) : chtmlfile;
        // this takes a pointer to an HTML file and translates it into a CHTML binary file
begin
        deviceInfo := LoadDeviceInfo( );     // Loads size and colors of target device
        Parse HTML file                      // use a parser to break the HTML file up into
                                             // tags represented in a fashion suitable for display
        For each HTML tag (<IMG . . . > = 1 tag, <P> a paragraph </P> = 1 tag),
select a sequence of CHTML instructions to render the tag on the output device.
As instructions are selected, colors and positioning are optimized based on the
device size and palette.
                CHTML instructions include:
                        TITLE           string
                        TEXT            formatted text at a specific position,
                                        complex formatting will
                                        require multiple CHTML TEXT instructions
                        IMAGE           image information including image-map,
                                        animation info, image data
                        ANCHOR          HTML reference
                Basic geometric instructions such as: SQUARE, FILLEDSQUARE, CIRCLE,
                FILLEDCIRCLE, and LINE, permit the complex rendering required by some
                HTML instructions to be decomposed into basic drawing instructions. For
                example, the bullets in front of lists can be described in CHTML instructions
                as squares and circles at specific locations.
                CHTML instructions including TEXT and IMAGE instructions can be
                contained within anchors. The CHTML compiler must properly code all
                instructions to indicate if an instruction is contained in an anchor.
        The CHTML instructions can then be written to the output file along with some header
        information.
end;
```

Table 2 sets forth the data structure for the precompiling process.

TABLE 2

Copyright EnReach 1997

```
/* HTML font structure */
typedef struct tagHTMLFont
{
     char name[64];
     int size;
     int bold;
     int italic;
     int underline;
     int strikeout;
} HTMLFont;
/* FG point structure */
typedef struct tagFGPoint
```

APPENDIX P

TABLE 2-continued

Copyright EnReach 1997

```
{
    int fX;
    int fY;
} FGPoint;
/* FG rectangle structure */
typedef struct tagFGRect
{
    int fLeft;
    int fTop;
    int fRight;
    int fBottom;
} FGRect;
/* html node types, used by hType attribute in HTML_InfoHead structure */
#define HTML_TYPE_TITLE     0            /* title of the html page */
#define HTML_TYPE_TEXT    1              /* text node */
#define HTML_TYPE_CHINESE    2           /* chinese text node */
#define HTML_TYPE_IMAGE    3             /* image node */
#define HTML_TYPE_SQUARE    4            /* square frame */
#define HTML_TYPE_FILLEDSQUARE 5         /* filled square */
#define HTML_TYPE_CIRCLE    6            /* circle frame */
#define HTML_TYPE_FILLEDCIRCLE 7         /* filled circle */
#define HTML_TYPE_LINE    8              /* line */
#define HTML_TYPE_ANCHOR    9            /* anchor node */
#define HTML_TYPE_ANIMATION    10        /* animation node */
#define HTML_TYPE_MAPAREA    11          /* client side image map area node */
/* header info of compiled html file */
typedef struct tagHTML_FileHead
{
    unsigned int fBgColor;                      /* background color index */
    unsigned int fPaletteSize;              /* size of palette */
} HTML_FileHead;
/* header info of each html node */
typedef struct tagHTML_InfoHead
{
    unsigned int hType;                         /* type of the node */
    unsigned int hSize;                         /* size of htmlInfo */
} HTML_InfoHead;
/* html info structure */
typedef struct tagHTML_Info
{
    HTML_InfoHead htmlHead;                     /* header info */
    unsigned char htmlInfo[1];              /* info of the html node */
} HTML_Info;
/* html title structure */
typedef struct tagHTML_Title
{
    unsigned int textLen;                   /* length of text buffer */
    char textBuffer[1];                         /* content of text buffer */
} HTML_Title;
/* html text structure */
typedef struct tagHTML_Text
{
    FGPoint dispPos;                            /* display coordinates */
    int anchorID;                       /* anchor id if it's inside an anchor, -1 if not */
    HTMLFont textFont;                          /* font of the text */
    unsigned int textColor;                 /* color index of the text */
    unsigned int textLen;                   /* length of text buffer */
    char textBuffer[1];                         /* content of text buffer */
} HTML_Text
/* html chinese structure */
typedef struct tagHTML_Chinese
{
    FGPNT dispPos;                              /* display coordinates */
    int anchorID;                       /* anchor id if it's inside an anchor, -1 if not */
    unsigned int textColor;                 /* color index of the text */
    unsigned int bufLen;                    /* length of the bitmap buffer (16* 16) */
    char textBuffer[1];                         /* content of text buffer */
} HTML_Chinese;
/* html image structure */
typedef struct tagHTML_Image
{
    FGRect dispPos;                             /* display coordinates */
    int anchorID;                       /* anchor id if it's inside an anchor, -1 if not */
```

TABLE 2-continued

Copyright EnReach 1997

```
    int animationID; /* animation id if it supports animation, -1 if not */
    int animationDelay;                     /* delay time for animation */
    char mapName[64];              /* name of client side image map, empty if no
image map */
    void *data;                                  /* used to store image
data */
    unsigned int fnameLen;              /* length of the image file name */
    char fname[1];                          /* image filename */
} HTML_Image;
/* square structure */
typedef strnct tagHTML_Square
{
    FGRect dispPos;                         /* display coordinates */
    unsigned int borderColor; /* border color index */
} HTML_Square;
/* filled square structure */
typedef struct tagHTML_FilledSquare
{
    FGRect dispPos;                         /* display coordinates */
    unsigned int brushColor; /* the inside color index */
} HTML_FilledSquare;
/* circle structure */
typedef struct tagHTML_Circle
{
    FGRect dispPos;                         /* display coordinates */
    unsigned int borderColor; /* border color index */
} HTML_Circle
/* circle structure */
typedef struct tagHTML_FilledCircle
{
    FGRect dispPos;                         /* display coordinates */
    unsigned int brushColor; /* the inside color index */
} HTML_FilledCircle;
/* line structure */
typedef struct tagHTML_Line
{
    FGPoint startPos;                     /* line starting position */
    FGPoint endPos;                       /* line end position */
    int style;                            /* style of the line (solid, dashed, dotted,
etc.) */
    unsigned int penColor;                  /* pen color index */
} HTML_Line;
/* anchor structure */
typedef struct tagHTML_Anchor
{
    int anchorID;                           /* id of the anchor */
    unsigned int hrefLen;.              /* length of href */
    char href[1];                           /* url of the anchor */
} HTML_Anchor;
/* animation structure */
typedef struct tagHTML_Animation
{
    int animationID;                    /* id of the animation */
    unsigned int frameTotal; /* total number of animation frames */
    long runtime;                           /* animation runtime */
} HTML_Animation;
#define SHAPE_RECTANGLE 0
#define SHAPE_CIRCLE 1
#define SHAPE_POLY 2
/* image map area structure */
typedef struct tagHTML_MapArea
{
    char mapName[64];                       /* name of client side image map
*/
    intshape;                                    /* shape of the area */
    int numVer;                                  /* number of vertix */
    int coords[6][2];              /* coordinates */
    unsigned int hrefLen;          /* length of href */
    char href[1];                           /* url the area pointed to */
} HTML_MapArea;
```

An example routine for reading this file into the thin platform memory follows in Table 3.

TABLE 3

Copyright EnReach 1997

```
reading this file:
#define BLOCK_SIZE 256
/* returns number of nodes */
long read_chm(const char *filename,           /* input: .chm file name */
    HTML_Info ***ppNodeList, /* output: array of (HTML_Info *)
                                            including anchors. */
    YUVQUAD **ppPalette,                              /* output: page palette */
    unsigned int *palette_size)                       /* output: palette size */
{
    int fd;
    char head[12];
    long total_nodes = 0;
    long max_nodes = 0;
    HTML_FileHead myFileHead;
    HTML_InfoHead myInfoHead;
    HTML_Info *pNodeInfo;
    void *pNodeData;
    long i;
    HTML_InfoHead *pHead;
    if (!ppNodeList || !ppPalette || !palette_size)
            return 0;
    (*ppNodeList) = NULL;
    (*ppPalette) = NULL;
    (*palette_size) = 0
    /* open file */
    fd = _open(filename,_O_BINARY|_O_RDONLY);
    if(fd < 0)
            return 0;
    /* read header and check for file type */
    if (_read(fd, head, 10) != 10)
    {
            _close(fd);
            return 0;
    }
    if (strncmp(head, "<COMPHTML>", 10))
    {
            _close(fd);
            return 0;
    }
    /* read file header */
    if(_read(fd, &myFileHead, sizeof(HTML_FileHead)) !=
sizeof(HTML_FileHead))
    {
            _close(fd);
            return 0;
    }
    (*palette_size) = myFileHead.fpaletteSize;
    /* read the palette */
    if ((*palette_size) > 0)
    {
            (*ppPalette) = (YUVQUAD *)malloc(sizeof(YUVQUAD)*
(*palette_size));
            if (_read(fd, (*ppPalette), sizeof(YUVQUAD) * (*palette_size))
                    != (int) (sizeof(YUVQUAD) * (*palette_size)))
            {
                    _close(fd);
                    return 0;
            }
    }
    /* read anchors along with other html nodes */
    while (1)
    {
            if (_read(fd, &myInfoHead, sizeof(HTML_InfoHead))
                    != sizeof(HTML_InfoHead))
            {
                    break;
            }
            if (myInfoHead.hSize > 0)
            {
                    pNodeInfo = (HTML_Info *) malloc(myInfoHead.hSize +
sizeof(HTML_InfoHead));
                    if (!pNodeInfo)
                            break;
                    memcpy(pNodeInfo, &myInfoHead,
sizeof(HTML_InfoHead));
                    if (_read(fd, &pNodeInfo[sizeof(HTML_InfoHead)],
myInfoHead.hSize)
```

APPENDIX P

TABLE 3-continued

Copyright EnReach 1997

```
                != (int)myInfoHead.hSize)
            {
                break;
            }
            /* check if we need to do memory allocation */
            if (total_nodes >= max_nodes)
            {
                if(!max_nodes)
                {
                    /* no node in the list yet */
                    (*ppNodeList) = (HTML_Info **)
malloc(
                        sizeof(HTML_Info *)*
BLOCK_SIZE);
                }
                else
                {
                        (*ppNodeList) = (HTML_Info **)
realloc((*ppNodeList),
                            max_nodes + sizeof(HTML_Info
*) * BLOCK_SIZE);
                    }
                    if (!(*ppNodeList))
                        break;
                    max_nodes += BLOCK_SIZE;
                }
                (*ppNodeList)[total_nodes] = pNodeInfo;
                total_nodes++;
        }
    }
    _close(fd);
    /* test our data */
    for (i = 0; i < total_nodes; i++)
    {
        pNodeInfo = (*ppNodeList)[i];
        pHead = (HTML_InfoHead *) pNodeInfo;
        pNodeData = pNodeInfo + sizeof(HTML_InfoHead);
        if(pHead->hType == HTML_TYPE_TEXT)
        {
            HTML_Text *pText = (HTML_Text *) pNodeData;
        }
        else if(pHead->hType == HTML_TYPE_IMAGE)
        {
            HTML_Image *pImage = (HTML_Image *) pNodeData;
            if (pImage->fnameLen > 0)
            {
                /* load the image file */
                pImage->data = load_ybm(pImage->fname);
            }
        }
        else if (pHead->hType == HTML_TYPE_ANCHOR)
        {
            HTML_Anchor *pAnchor = (HTML_Anchor *)
pNodeData;
        }
        else if(pHead->hType == HTML_TYPE_ANIMATION)
        {
            HTML_Animation *pAnimation = (HTML_Animation *)
pNodeData;
        }
        else if(pHead->hType == HTML_TYPE_MAPAREA)
        {
            HTML_MapArea *pMapArea = (HTML_MapArea *)
pNodeData;
        }
        else if (pHead->hType == HTML_TYPE_LINE)
        {
            HTML_Line *pLine = (HTML_Line *) pNodeData;
        }
        else if(1)Head->hType == HTML_TYPE_SQUARE)
        {
            HTML_Square *pSquare = (HTML_Square *) pNodeData;
        }
        else if(pHead->hType == HTML_TYPE_CIRCLE)
        {
            HTML_Circle *pCircle = (HTML_Circle *) pNodeData;
        }
```

APPENDIX P

TABLE 3-continued

Copyright EnReach 1997

```
            else if (pHead->hType == HTML_TYPE_FILLEDSQUARE)
            {
                HTML_FilledSquare *pFilledSquare =
(HTML_FilledSquare *) pNodeData;
            }
            else if (pHead->hType == HTML_TYPE_FILLEDCIRCLE)
            {
                HTML_FilledCircle *pFilledCircle = (HTML_FilledCircle
*) pNodeData;
            }
            else if(pHead->hType == HTML_TYPE_TITLE)
            {
                HTML_Title *pTitle = (HTML_Title *) pNodeData;
            }
      }
      return total_nodes;
}
```

The compiled HTML file structure is set forth in FIG. **7** as described in Table 2. The file structure begins with a ten character string COMPHTML **900**. This string is followed by a HTML file header structure **901**. After the file header structure, a YUV color palette is set forth in the structure **902** this consists of an array of YUVQUAD values for the target device. After the palette array, a list **903** of HTML information structures follows. Usually the first HTML information structure **904** consists of a title. Next, a refresh element typically follows at point **905**. This is optional. Next in the line is a background color and background images if they are used in this image. After that, a list of display elements is provided in proper order. The anchor node for the HTML file is always in front of the nodes that it contains. An animation node is always right before the animation image frames start. The image area nodes usually appear at the head of the list.

The HTML file header structure includes a first value BgColor at point **906** followed by palette size parameters for the target device at point **907**. The YUVQUAD values in the color palette consist of a four word structure specifying the Y, U, and V values for the particular pixel at points **908–910**. The HTML information structures in the list **903** consist of a type field **911**, a size field **912**, and the information which supports the type at field **913**. The type structures can be a HTML_Title, HTML_Text, HTML_Chinese, HTML_Xxge, HTML_Square, HTML_FilledSquare, HTML_Circle, HTML_FilledCircle, HTML_Line, HTML_Author, HTML_Animation, . . .

Functions that would enable a thin platform to support viewing of HTML-based content pre-compiled according to the present invention includes the following:

General graphics functions

int DrawPoint (int x, int y, COLOR color, MODE mode);

int DrawLine (int x1, int y1, int x2, int y2, COLOR color, MODE mode);

int DrawRectangle(int x1, int y1, int x2, int y2, COLOR color, MODE mode);

int FillRectangle(int x1, int y1, int x2, int y2, COLOR color, MODE mode);

int ClearScreen(COLOR color);

Color palette

int ChangeYUVColorPalette( );

Bitmap function

int BitBlt(int dst_x1, int dst_y1, int dst_x2, int dst_y2, unsigned char *bitmap, MODE mode);

String drawing functions

int GetStringWidth(char *str, int len);

int GetStringHeight(char *str, int len);

int DrawStringOnScreen(int x, int y, char *str, int len, COLOR color, MODE mode);

Explanation

All (x, y) coordinates are based on the screen resolution of the target display device (e.g. 320×240 pixels).

COLOR is specified as an index to a palette.

MODE defines how new pixels replace currently displayed pixels (COPY, XOR, OR, AND).

Minimum support for DrawLine is a horizontal or vertical straight line, although it would be nice to have support for diagonal lines.

The ChangeYUVColorPalette function is used for every page.

BitBlt uses (x1, y1) and (x2, y2) for scaling but it is not a requirement to have this scaling functionality.

String functions are used for English text output only. Bitmaps are used for Chinese characters.

FIGS. **8**A and **8**B set forth the run time engine suitable for execution on a thin client platform for display of the compiled HTML material which includes the function outlined above in the "display" step **1220** of FIG. **8**B.

The process of FIG. **8**A starts at block **1000**. The ran time engine is initialized on the client platform by loading the appropriate elements of the run time engine and other processes known in the art (step **1010**). The next step involves identifying the position of the file, such as on the source CD or other location from which the file is to be retrieved and setting a flag (step **1020**). The flag is tested at step **1030**. If the flag is not set, then the algorithm branches to block **1040** at which the flag is tested to determine whether it is −1 or not. If the flag is −1, then the algorithm determines that a system error has occurred (step **1050**) and the process ends at step **1060**. If the flag at step **1040** is not −1, then the file has not been found (step **1070**). Thus after step **1070** the algorithm returns to step **1020** to find the next file or retry.

If at step **1030**, the flag is set to 1 indicating that the file was found, then the content of the file is retrieved using a program like that in Table 3, and it is stored at a specified address. A flag is returned if this process succeeds set equal to 1 otherwise it is set equal to 0 (step **1080**). Next the flag is tested (step **1090**). If the flag is not equal to 1 then reading of the file failed (step **1100**). The process then returns to step **1020** to find the next file or retry.

If the flag is set to 1, indicating that the file has been successfully loaded into the dynamic RAM of the target device, then the "Surf_HTML" process is executed (step **1110**). The details of this process are illustrated in FIG. **8**B. Next the current page URL name is updated according to the HTML process (step **1120**). After updating the current URL name, the process returns to step **1020** to find the next file.

FIG. **8**B illustrates the "Surf_HTML" process of step **1110** in FIG. **8**A. This process starts at point **1200**. The first part is initialization step **1210**. A display routine is executed at step **1220** having the fixed coordinate functions of the precompiled HTML data set. First, the process determines whether applets are included in the file (step **1230**). If they are included, then the applet is executed (step **1240**). If no applets are included or after execution of the applet, then a refresh flag is tested (step **1240**). If the flag is equal to 1, then it is tested whether a timeout has occurred (step **1250**). If a timeout has occurred, then the current page is updated (step **1260**) and the process returns set **1210** of FIG. **8**B, for example.

If at block **1240** the refresh flag was not equal to 1, or at block **1250** the timeout had not expired, then the process proceeds to step **1270** to get a user supplied input code such as an infrared input signal provided by a remote control at the target device code. In response to the code, a variety of process are executed as suits a particular target platform to handle the user inputs (step **1280**). The process returns a GO_HOME, or a PLAY_URL command, for example, which result in returning the user to a home web page or to a current URL, respectively. Alternatively the process loops to step **1270** for a next input code.

As mentioned above, FIG. **4** illustrates the JAVA precompiler according to the present invention. The JAVA precompiler receives standard full feature JAVA byte codes as input on line **600**. Byte codes are parsed at block **601**. A JAVA class loader is then executed at block **602**. The classes are loaded into a command module **603** which coordinates operations of a JAVA virtual machine **604**, a JAVA garbage collection module **605**, and a JAVA objects memory mapping optimizing engine **606**. The output is applied by block **607** which consists of a compiled JAVA bytecode format according to the present invention.

The process is illustrated in FIG. **9** beginning at block **1500**. First the JAVA bytecode file is loaded (block **1510**). Next, the JAVA classes are loaded based on the interpretation of the bytecode (step **1520**). Next the classes are optimized at step **1530**. After optimizing the classes, the byte codes are translated to a reduced bytecode (step **1540**). Finally the reduced bytecode is supplied (step **1550**) and the algorithm stops at step **1560**. Basically the process receives a JAVA source code file which usually has the format of a text file with the extension JAVA. The JAVA compiler includes a JAVA virtual machine plus compiler classes such as SUN.TOOLS.JAVAC which are commercially available from Sun Micro Systems. The JAVA class file is parsed which typically consists of byte codes with the extension .CLASS. A class loader consists of a parser and bytecode verifier and processes other class files. The class structures are processed according to the JAVA virtual machine specification, such as the constant pool, the method tables, and the like. An interpreter and compiler are then executed. The JAVA virtual machine executes byte codes in methods and outputs compiled JAVA class files starting with "Main". The process of loading and verifying classes involves first finding a class. If the class is already loaded a read pointer to the class is returned, if not, the class is found from the user specified class path or directory, in this case a flash memory chunk. After finding the class, the next step is executed. This involves loading the bytes from the class file. Next, class file bytes are put into a class structure suitable for run time use, as defined by the JAVA virtual machine specification. The process recursively loads and links the class to its super classes. Various checks and initializations are executed to verify and prepare the routine for execution. Next, initialization is executed for the method of the class. First the process ensures that all the super classes are initialized, and then cause the initialization method for the class. Finally, the class is resolved by resolving a constant pool entry the first time it is encountered. A method is executed with the interpreter and compiler by finding the method. The method may be in the current class, its super class or other classes as specified. A frame is created for the method, including a stack, local variables and a program counter. The process starts executing the bytecode instructions. The instructions can be stack operations, branch statements, loading/storing values, from/to the local variables or constant pool items, or invoking other methods. When an invoked method is a native function, the implemented platform dependent function is executed.

In FIG. **9**A, the process of translating JAVA byte codes into compiled byte codes (step **1504** of FIG. **9**) is illustrated. According to the process FIG. **9**A, the high level class byte codes are parsed from the sequence. For example, Windows dialog functions are found (**1570**). The high level class is replaced with its lower level classes (**1580**). This process is repeated until all the classes in the file become basic classes (**1590**). After this process, all the high level functions have been replaced by lower level level basic functions, such as draw a line, etc. (**1600**).

JAVA byte codes in classes include a number of high level object specifying functions such as a window drawing function and other tool sets. According to the present invention, these classes are rendered by the precompiler into a set of specific coordinate functions such as those outlined above in connection with the HTML precompiler. By precompiling the object specifying functions of the JAVA byte code data set, significant processing resources are freed up on the thin client platform for executing the other programs carried in a JAVA byte code file. Furthermore, the amount of memory required to store the run time engine and JAVA class file for the thin client platform according to the present invention which is suitable for running a JAVA byte code file is substantially reduced.

FIG. **10** illustrates one environment in which use of the present invention is advantageous. In particular, in the Internet environment a wide variety of platforms are implemented. For example, an end user workstation platform **100** is coupled to the Internet **101**. An Internet server platform **102** is also coupled to the Internet **101** and includes storage for JAVA data sets, HTML data sets, and other image files. A server **103** with an intermediate compiler according to the present invention for one or more of the data sets available in the Internet is coupled to the Internet **101** as well. A variety of "thin" platforms are also coupled to the Internet and/or the server **103**. For example, an end user thin platform A **104** is coupled to the server **103**. End user thin platform B **105** is coupled to the server **103** and to the Internet **101**. End user thin platform C **106** is coupled to the Internet **101** and via the Internet all the other platforms in the network. A variety of scenarios are thus instituted. The source of data sets for end user platform C **106** consists of the World Wide Web. When it requests a file from server

APPENDIX P

**102**, the file is first transferred to the intermediate compiler at server **103**, and from server **103** to the end user platform **106**. End user platform A **104** is coupled directly to the server **103**. When it makes a request for a file, the request is transmitted to the server **103**, which retrieves the file from its source at server **102**, translates it to the compiled version and sends it to platform A **104**. End user platform B is coupled to both the server **103** and to the Internet **101**. Thus, it is capable of requesting files directly from server **102**. The server **102** transmits the file to server **103** from which the translated compiled version is sent to platform B **105**. Alternatively, platform B may request a file directly from server **103** which performs all retrieval and processing functions on behalf of platform B.

FIG. **11** illustrates an alternative environment for the present invention. For example, the Internet **120** and an Intranet **121** are connected together. A server **122** is coupled to the Intranet **121** and the Internet **120**. The server **122** includes the HTML and JAVA intermediate compiling engines according to the present invention as represented by block **123**. The server **122** acts as a source of precompiled data sets for thin client platforms **124**, **125** and **126** each of which has a simplified run time engine suitable for the compiled data sets. Thus the powerful HTML/JAVA engine resides on the network server **122**. The thin network computers **124**, **125**, **126** are connected to the server have only the simplified run time engine for the compiled image set. Thus, very small computing power is required for executing the display. Thus computing tasks are done using the network server, but displayed on a thin network computer terminals **124**–**126**.

FIGS. **12**A and **12**B illustrate the off-line environment for use of the present invention. In FIG. **12**A, the production of the compiled files is illustrated. Thus, a standard object file, such as an HTML or JAVA image, is input online **1300** to a compiler **1301** which runs on a standard computer **1302**. The output of the compiler on line **1303** is the compiled bitmap, compiled HTML or compiled JAVA formatted file. This file is then saved on a non-volatile storage medium such as a compact disk, video compact disk or other storage medium represented by the disk **1304**.

FIG. **12**B illustrates the reading of the data from the disk **1304** and a thin client such as a VCD box, a DVD box or a set top box **1305**. The run time engine **1306** for the compiled data is provided on the thin platform **1305**.

Thus, off-line fill feature HTML and JAVA processing is provided for a run time environment on a very thin client such as a VCD/DVD player. The standard HTML/JAVA objects are pre-processed and compiled into the compiled format using the compiler engine **1301** on a more powerful computer **1302**. The compiled files are saved on a storage medium such as a floppy disk, hard drive, a CD-ROM, a VCD, or a DVD disk. A small compiled run time engine is embedded or loaded into the thin client device. The run time engine is used to play the compiled files. This enables use of a very small client for running full feature HTML and JAVA programs. Thus, the machine can be used in both online, and off-line modes, or in a hybrid mode.

The foregoing description of a preferred embodiment of the invention has been presented for purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Obviously, many modifications and variations will be apparent to practitioners skilled in this art. It is intended that the scope of the invention be defined by the following claims and their equivalents.

What is claimed is:

**1**. A method of translating a document on a first device for use on a second device, the document being in a standard HTML language, the method comprising:

reading the document;

reading a profile describing characteristics of the second device, the profile including a display resolution and a supported image format; and

translating the document on the first device according to the profile, the translating including

retrieving a plurality of images referenced by the document,

generating a color palette for the second platform using the plurality of images and the document,

executing the document according to the standard HTML language using the profile to generate a plurality of drawing instructions for displaying the document on the second device,

translating the plurality of images from respective formats to the supported image format, and

outputting a translated document, the translated document including at least a reference to the color palette, the plurality of images in the supported image format, and the plurality of drawing instructions.

**2**. The method of claim **1** wherein the reading the document further comprises retrieving the document from a world wide web (WWW) site based on a uniform resource locator (URL).

**3**. The method of claim **1** wherein the profile includes a maximum number of colors for the color palette.

**4**. The method of claim **3** wherein the generating the color palette using the plurality of images and the document comprises:

creating a set of colors comprised of all colors used in the plurality of images and all colors used in the document;

reducing the set of colors to contain no more than the maximum number of colors for the color palette.

**5**. The method of claim **1** wherein the document includes a plurality of references to a plurality of images, each of the plurality of references comprising a URL, and the retrieving a plurality of images referenced by the document further comprises retrieving respective images using the plurality of references.

**6**. The method of claim **1** wherein the executing the document according to the standard HTML language using the profile to generate a plurality of drawing instructions for displaying the document on the second device further comprises:

executing the document for display on the second device according to the display resolution;

positioning HTML elements in the document according to the display resolution;

word wrapping HTML text elements in the document according to the display resolution; and

generating a plurality of text drawing elements.

**7**. The method of claim **6** wherein the generating a plurality of text drawing elements further comprises generating a text element for each text segment, a text segment comprised of one or more characters from the document, the one or more characters sharing a font, a size, a style, and a color, and the text segment occupying not more than one line in the font at the size in the style, each text element including an absolute position at which the text segment should be displayed on the second device.

**8**. The method of claim **6** further comprising generating a plurality of graphics drawing elements including:

APPENDIX P

generating a plurality of line elements;

generating a plurality of rectangle elements; and

generating a plurality of circle elements.

**9**. The method of claim **6** further comprising generating a plurality of link elements, each link element including a URL of a corresponding linked item.

**10**. The method of claim **1** wherein the supported image format includes a color palette indexed bitmap format and the translating the plurality of images from respective formats to the supported image format comprises:

decoding each of the plurality of images into a red-green-blue bitmap format;

selecting a color in the color palette for pixels in each of the plurality of images; and

outputting a color palette indexed bitmap format for each of the plurality of images.

**11**. The method of claim **10** wherein the document comprises a plurality of Java classes, and wherein the executing the document according to the standard HTML language using the profile to generate a plurality of drawing instructions for displaying the document on the second device comprises:

loading and verifying the plurality of Java classes;

initializing methods associated with the plurality of Java classes; and

replacing calls to complex drawing operations with a plurality of graphics drawing elements and a plurality of text drawing elements.

**12**. The method of claim **1** wherein the translated document includes a plurality of text elements and a plurality of graphics drawing elements.

**13**. The method of claim **1** wherein the standard HTML language comprises a Java language program.

**14**. The method of claim **1** wherein the translating the document on the first device for use on the second device further comprises:

receiving a request at the first device over a packet switched network from the second device, the request including a URL;

retrieving the document using the URL responsive to the request; and

providing the translated document to the second device over the packet switched network.

* * * * *

# Visualization of Large Terrains in Resource-Limited Computing Environments

Boris Rabinovich Craig Gotsman

Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000, Israel
[borisr|gotsman]@cs.technion.ac.il

## Abstract

We describe a software system supporting interactive visualization of large terrains in a resource-limited environment, i.e. a low-end client computer accessing a large terrain database server through a low-bandwidth network. By “large”, we mean that the size of the terrain database is orders of magnitude larger than the computer RAM. Superior performance is achieved by manipulating both geometric and texture data at a continuum of resolutions, and, at any given moment, using the best resolution dictated by the CPU and bandwidth constraints. The geometry is maintained as a Delaunay triangulation of a dynamic subset of the terrain data points, and the texture compressed by a progressive wavelet scheme.

A careful blend of algorithmic techniques enables our system to achieve superior rendering performance on a low-end computer by optimizing the number of polygons and texture pixels sent to the graphics pipeline. It guarantees a frame rate depending *only* on the size and quality of the rendered image, independent of the viewing parameters and scene database size. An efficient paging scheme minimizes data I/O, thus enabling the use of our system in a low-bandwidth client/server data-streaming scenario, such as on the Internet.

**CR Categories:** I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation; D.4.4 [Operating Systems]: Communications Management—Network Communication.
**Keywords:** Terrain rendering, level-of-detail, interactive graphics

## 1 Introduction

Terrain visualization is an important component of many civilian and military applications [10, 3]. The input to the terrain visualization problem is usually a large Digital Terrain Map (DTM), consisting of elevation data sampled on a regular grid, and corresponding aerial and/or satellite texture data, which is mapped onto the reconstructed terrain surface. The output is rendered images of the terrain surface, usually as part of a “flythrough” sequence.

The advent of the World-Wide-Web suggests the running of this type of application over the Internet, in a client/server scenario. The server is a very large remote database, accessed by the client, usually a low-end computer, over a narrow-bandwidth line (3 KByte/sec is typical for the contemporary Internet). The two bottlenecks that have to be overcome are the bandwidth in delivering relevant terrain data from the server to the client, and the CPU power required at the client for rendering this data.

The key to efficient terrain *rendering* is efficient online manipulation of both the geometric and texture data, especially when the scene database at the server is orders of magnitude larger that the size of client system RAM. Naive terrain rendering algorithms convert each DTM cell (bounded by four adjacent grid points) into two 3D triangles, and render (send through the graphics pipeline) all such triangles in a region determined by the viewing frustum. They also map the texture data at its highest resolution onto these polygons. This is a very inefficient procedure, as for low pitch angles, the number of these triangles and texture pixels (*texels*) may be extremely large. Each individual triangle projection to image space is very small, and many texels may be condensed to one image pixel, contributing negligibly to the image. One remedy to this problem, adopted in a number of works over the past few years (e.g. [8]) is to maintain the scene data at a number of discrete *levels-of-detail*. Since terrain areas at large viewing distances project to small image areas, there is no point rendering them in full detail. At any given moment during the animation, the appropriate level-of-detail is used to render the image. To do this effectively, pieces of the scene must be taken from multiple levels (foreground areas from a high-detail version, and background areas from a low-detail version), requiring methods to “stitch” together pieces of different models in a continuous fashion, so that there are no holes or breaks along the seams. This has proven to be a major problem for the geometric data, since there usually is no topological correlation between the different levels of detail. De Berg and Dobrint [1], Cohen-Or and Levanoni [5], and Lindstrom et al. [12] have provided partial solutions to the stitching problem.

In this paper we use a different approach to maintaining the terrain geometry, proposed independently by Klein and Huttner [11] and Delepine [6]. The geometry is treated in a *continuous-resolution* fashion. We do not maintain multiple geometric models (at different levels of detail), rather continuously update one model online to represent in an optimal way the projection of the terrain contained in the viewing frustum. As a result, the number of polygons in the approximation is more or less constant, independent of the viewing parameters (for a fixed frame rate). For the texture,we employ a progressive wavelet compression scheme [2], which enables the extraction of texture at a continuum of resolutions from arbitrary prefixes of the encoded bit stream.

Our ultimate goal is to render any terrain image in time proportional to the *image* resolution (in pixels), and not to the scene complexity, number of DTM points in the viewing frustrum, texture resolution, etc. We are motivated by the (simple) observation that an image of fixed resolution can contain only a bounded amount of information, therefore any algorithm rendering such an image should not use more than a bounded number of polygons and texels. Such algorithms are called *output-sensitive*. Most algorithms are not output-sensitive, and in order that they be such, require careful design. Our system contains a careful blend of techniques, some borrowed from computational geometry, which together achieve a high degree of output sensitivity, enabling adequate performance in a limited-resource environment.

Since one server may be accessed simultaneously by a large number of clients, is is crucial to minimize the amount of work the server performs per client. If this load is minimized, the server will be scalable, able to support a virtually unlimited number of clients. We adhere to this principle throughout our implementation.

Using these methods, we have developed a client application achieving terrain visualization at interactive rates on a low-end SGI ($O_2$)workstation, accessing a server database over a network with bandwidth comparable to the Internet. This paper describes the architecture and algorithms incorporated into our system.

## 2 System Overview

The large terrain scene resides on the server disk, partitioned into geometry and texture *tiles* of fixed size. A raw geometry tile contains a matrix of elevation heights, and a texture tile a matrix of texels. Tiling schemes are standard in terrain visualization applications (e.g. [4]). The server processes requests for geometry and texture data received from remote clients. In a preprocessing step at the server, applied independently to each tile (thus enabling a scene consisting of an unlimited number of tiles), the DTM points are assigned “grades” related to their importance in approximating the terrain surface. These grades are obtained from the *simplification* algorithm of Heckbert and Garland [9]. Using these grades as a third dimension, the DTM points in each tile are organized into a 3D octree, which will enable efficient answers to future geometric queries. The client maintains online a geometry cache containing DTM points from a small subset of the server’s geometry tiles. Even from these tiles, only the relevant upper levels of the corresponding octrees are imported to the client. Which levels are relevant is determined on the fly by the client.

At any given moment, a subset of the geometry cache points are maintained at the client in a dynamic Delaunay triangulation, our primary geometric data structure. To maintain the triangulation, we use the algorithms of Devillers, Meiser and Teillaud [7] for efficient insertion and deletion into a 2D Delaunay triangulation. Delaunay triangulations are commonly considered to be suitable for terrain visualization purposes. A DTM point deserves to be in the triangulation if its grade is greater than a threshold, which is proportional to the distance of the point from the viewpoint. Section 3 elaborates on the details of how we handle the geometry.

The texture data is maintained at the server in tiles, compressed using the progressive wavelet scheme of Buccigrossi and Simoncelli [2]. This scheme compresses the data to approximately 30% of its raw size with negligble loss, and, more important, allows the decoding of the texture data from any prefix of the bit stream. Naturally, using more bits will result in a higher quality result. Client requests for texture data at a given resolution result in the streaming of the prefix of minimal length sufficing for the required resolution. Section 4 describes our handling of the texture in more detail.

The client graphics pipeline, sometimes supported in hardware, is fed relevant triangles and texels. This pipeline takes care of the basic rendering operations, e.g. perspective projection, hidden surface elimination, and texture mapping. The main issues we address in our implementation are the minimization of data transmitted from the server to the client caches and subsequently fed to the graphics pipeline.

Typical triangulations and rendered images generated by our client system are shown in Fig. 2.

## 3 Geometry Processing

### 3.1 Data Reduction

A typical DTM is supplied on a regular grid, and this data is usually highly redundant. If the surface is to be approximated by a piecewise-linear 2D function (a collection of planar polygons), a small number of large polygons suffice to approximate the surface well in planar regions. On the other hand, terrain areas with high curvature, such as ridges and ravines, require a large number of small polygons to achieve a satisfactory approximation (see Fig. 2). By this argument, is it obvious that some DTM points are more important than others. Heckbert and Garland [9] have described a procedure which starts off with a small number of DTM points (usually the four corners of the DTM coverage), and incrementally adds points whose contribution to the surface approximation is most significant. The contribution of a point to the approximation is quantified by its vertical distance from the piecewise-linear approximation built with all previous points. The larger this distance - the more important the point is. The incremental procedure is done efficiently using a priority queue mechanism.

We use the Heckbert and Garland procedure at the server as a preprocessing operation on each tile to assign each DTM point a numeric “grade” - precisely the vertical distance described in the previous paragraph. This grade is stored with the point, and used later to determine online whether the point is required for the terrain approximation. This decision is based on the grade and the point’s distance from the viewpoint. To facilitate efficient decision-making, we build a 3D octree of the DTM points, the grade serving as the third dimension. The grid structure of the points in the XY plane facilitates a fixed quadtree structure in this plane, which, in turn, facilitates the organization of the data stored in the tile in a record of fixed length. This hierarchical spatial data structure will enable efficient range reporting of points.

### 3.2 View Frustum Culling

The first step in frame generation is to determine which DTM tiles are relevant to the current view. In principle, if the terrain surface were planar, the intersection of the viewing frustum with the terrain surface (the view *footprint*) would be a trapezoid, whose four vertex positions could be easily computed (see Fig. 3). Since the terrain surface is not planar, the footprint terrain is bounded by a region which is the union of *two* trapezoids, formed on horizontal planes whose elevations coincide with the minimal and maximal elevations in the projection area, repectively.

The footprint is "scan-converted" by the client to determine which DTM tiles intersect it, and what resolution data (which levels of the octree) are required. This data is requested from the server. For every tile received, the octree structure of its points enables to efficiently determine which tile points are actually contained in the footprint. Efficiency is achieved by pruning off large sets of the points corresponding to branches of the octree close to its root. The remaining points are then tested, as described in Section 3.3, to determine if they are required for the terrain approximation and rendering.

### 3.3 Continuous Resolution

Each DTM point has a grade quantifying its importance in the terrain approximation. This grade is traded off with distance from the viewpoint. In other words, more distant points are considered less significant. In practice, the client considers a virtual cone centered at the viewpoint, and calculates which DTM points in the geometry cache have a grade positioning them *inside* the cone (see Fig. 3). We would like to be able to determine this set of points in time proportional mainly to their number (and not to the total number of points in the viewing frustum). In computational-geometric terminology, this is called *output-sensitive range reporting*. We achieve this again using the tile octree. The complexity of the range reporting procedure is $O(\sqrt{N}+k)$, where $N$ is the number of points in the viewing frustum, and $k$ the number of points in the answer to the query ([13], p.79). Using this virtual cone also implies that a small change in the viewpoint induces a small change in the DTM points used for the rendering, thus ensuring the temporal continuity of the rendered images.

### 3.4 Caching

Portions of geometry tiles are imported from the server on demand and stored in the client cache. Only the neccesary upper levels of the tile octree are imported, possible due to the fixed structure of the octree. Hence a typical snapshot of the client cache contents would reveal a few (foreground) tiles from which almost the entire data content has been read, and many (background) tiles with a very sparse content. A prediction mechanism, based on the viewpoint trajectory, enables the loading of tiles in advance, resulting in smooth streaming of geometry from server to client.

### 3.5 Dynamic Delaunay Triangulation

The piecewise linear surface induced by the *Delaunay triangulation* of the 2D projection of the DTM points is generally considered the most suitable for surface approximation. This is because the minimal angle in the triangulation is maximized, eliminating long "slivery" triangles. Hence, the client constantly maintains a Delaunay triangulation of the DTM points contributing to the approximation of the terrain in the footprint. Many $O(n \log n)$ time algorithms exist for the Delaunay triangulation of $n$ points, but not many are able to efficiently support update of the triangulation upon insertion or deletion of points. We use the algorithm of DeVillers et al [7], which inserts points in $O(\log n)$ and deletes points in $O(\log\log n)$ average time using a hierarchical data structure. Care must be taken to slightly perturb the spatial positions of the DTM points, otherwise degeneracies in the Delaunay triangulation and unstable numerics may occur.

At the client, points which were in the footprint corresponding to the previous frame, and are no longer in the current footprint, are removed from the triangulation - the main geometric data structure maintained online by the client. New points which were previously not in the footprint, and now are, are inserted into the triangulation. The turnover of points in the triangulation depends on the viewpoint velocity. Theoretically, very large velocities could cause successive frames to see totally different regions of the terrain, requiring the formation of an entirely different triangulation between frames. In practice, however, this does not occur. Typically, 99% of the footprint areas overlap between successive frames.

Pseudo-code of the flow of control in the client while rendering a single frame appears in Fig. 1.

## 4 Texture Processing

The texture data must also be manipulated at multiple resolutions, since image foreground pixels contain high resolution texels, and image background pixels contain low resolution texels. The resolution of the texels contributing to any given image pixel is essentially a function of the viewing distance to that scene point. The server texture database is also organized in tiles, storing the texels compressed to approximately 30% of their original volume, using a progressive wavelet scheme. This results in a bit stream sorted by importance.

A typical low-end client computer contains a texture buffer of limited capacity (e.g. 1024x1024 pixels) with a pyramid structure on top of it. By supplying appropriate texture coordinates for the rendered triangle vertices, the graphics hardware/software maps texels from the texture buffer to the image pixels in the interior of the projected triangles. Each level of the texture pyramid contains texels representing the same terrain area, at decreasing resolutions. However, since not all texels, especially not at *all* resolutions, will contribute to the terrain image (see Fig. 4), there is no need to import them from the server. We optimize network bandwidth by loading *only* those texture tiles which intersect the view footprint, at the appropriate resolution, if they are not yet loaded. By this we mean we calculate the number of encoded bits of the texture stream required to reconstruct the texture tile at the appropriate resolution (the lower the required resolution, the less bits required). In any case, we use any bits available at rendering time, even though there might be less than required (if the network temporarily slows down). Which tiles are relevant can be easily determined from the geometry of the footprint. Occasionally, it is neccesary to shift the contents of the texture buffer, due to the movement of the viewpoint.

## 5 Experimental Results

We have implemented the procedures described in Sections 2 - 4 as a prototype client/server system, the client running on a R5000 SGI $O_2$, at 180MHz with 64MB RAM, based on the OpenGL API, and an X/Motif GUI. This client accesses the scene database server over a 3 KByte/sec network. The main parameters influencing the overall performance of the system are the size of the visualization window, i.e. the number of rendered image pixels, and the flight velocity. This performance is measured in the client frame rate, and the quality of the imagery delivered at that frame rate. There is an obvious tradeoff between the two, which is controlled by two independent "resolution" parameters, one for geometry, and one for texture. Increasing these parameters increases the number of triangles and/or texture bytes used for the rendering process, thus increasing the image quality, but decreasing the frame rate, due to higher rendering and bandwidth overhead. There is, however, a point beyond which the resolution parameter saturates, i.e. the marginal increase in image quality is insignificant.

The geometric resolution parameter, namely, the average number of triangles rendered per image pixel, is controlled by the angle of the cone used for culling DTM points, as described in Section 3.3. The smaller the angle, the narrower the cone, admitting less DTM points into the Delaunay triangulation, in turn implying less triangles for the same number of image pixels (see also Fig. 3). The texture resolution is controlled by specifying the fraction of the texture tile bit stream imported and decoded to texels for the foreground image pixels. The resolution of the background image pixels is derived from this.

Keeping the resolution parameters and velocity fixed causes the system to maintain a fixed frame rate. Increasing the velocity would slow down the system, as the turnover of points in the Delaunay triangulation and turnover of texture tiles in the texture buffer increases, incurring more CPU and bandwidth overhead. By trial and error, it seems that reasonable image quality is obtained at a geometric resolution of 0.06 triangles and 0.5 texture bytes per output image pixel. Any more than that imposes an unneccesary load on the system, slowing it down, and any less than that results in poor quality images (see Fig. 2). A telltale sign of insufficient geometric resolution (triangles per image pixel) is if there are "jumps" (also known as "popping") in the terrain surface during animation, due to the triangles being too large and crude. A telltale sign of insufficient texture resolution (texels per image pixel) are blurred images.

Fig. 5 shows the speed/quality tradeoffs we are able to achieve with our system at different "flight" velocity parameters, when one of the geometric/texture resolution parameters is fixed, and the other varied. Velocity is measured as the percentage of non-overlapping area between footprints corresponding to successive frames. The figure shows that approximately 3 frames/sec are achievable with reasonable quality, when the image size is fixed at 300x400 pixels, and flying at an average (3%) velocity. Higher velocities result in a larger turnover of geometry and texture, slowing down the system frame rate. Our system accesses a scene database server covering the northern part of Israel, containing a total of $10^7$ DTM points and $10^8$ texels. The client uses a geometry cache of size 2MB RAM, and texture buffer of 1024x1024 texels.

## 6 Conclusion

In the long-term, our techniques will support client/server terrain visualization applications over the Internet. A large scene database resides at a central server site, and is accessed (perhaps simultaneously) by a number of low-end clients over the Internet for visualization purposes. This application requires tight optimization of the available network bandwidth and client rendering power.

The ever-increasing user appetite for larger and richer geometric scenes has forced computer graphics practitioners to develop output-sensitive rendering algorithms whose computational complexity is not sensitive to the complexity of the input scene, rather to the complexity of the output image. We have implemented this for the terrain visualization application by rendering at geometric and texture level-of-detail which changes continuously along the spatial and temporal dimensions. Our algorithm satisfies almost all of the five requirements from such an algorithm, as formulated in [12].

Use of other sophisticated data optimization techniques, such as *occlusion culling* [14], in which large portions of the geometry inside the view frustrum are efficiently culled because they are invisible, can further reduce the rendering load.

Temporal aliasing sometimes occurs in our implementation. The use of *morphing* techniques to alleviate this, such as that of Cohen-Or and Levanoni [5], are not directly applicable, again due to the dynamic nature of our Delaunay triangulation. Alternatives are being investigated.

## Acknowledgements

We thank Olivier DeVillers for providing code implementing the algorithm of [7], Paul Heckbert for code implementing the algorithm of [9], and R. Buccigrossi for code implementing the algorithm of [2].

This research was supported by the Technion V.P.R. Fund - Promotion of Sponsored Research.

APPENDIX R

## References

[1] M. De Berg and K. Dobrindt. On levels of detail in terrains. In *11th Annual ACM Symposium on Computational Geometry*. ACM, 1994.

[2] R.W. Buccigrossi and E.P. Simoncelli. Progressive wavelet image coding based on a conditional probability model. In *Proceedings of Int'l Conf. Acoustics Speech and Signal Processing*. IEEE, 1997.

[3] D. Cohen and C. Gotsman. Photorealistic terrain imaging and flight simulation. *IEEE Computer Graphics and Applications*, 14(2):10–12, March 1994.

[4] D. Cohen-Or, U. Lerner, E. Rich, and V. Shenkar. A real-time photo-realistic visual flythrough. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):255–265, September 1996.

[5] D. Cohen-Or and Y. Levanoni. Temporal continuity of levels of detail in Delaunay triangulated terrain. In *Proceedings of Visualization '96*. IEEE Computer Society Press, 1996.

[6] T. Delepine. Online terrain level-of-detail. In *Proceedings of ITECH*, 1997.

[7] O. Devillers, S. Meiser, and M.Teillaud. Fully dynamic Delaunay triangulation in logarithmic expected time per operation. *Computational Geometry: Theory and Applications*, 2:55–80, 1992.

[8] L. De Floriani. A pyramidal data structure for triangle-based surface representation. *IEEE Computer Graphics and Applications*, 9(2):67–78, 1989.

[9] P. Heckbert and M. Garland. Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, School of Computer Science,Carnegie Mellon University,Pittsburg ,PA , 15213, 1995.

[10] K. Kaneda, F. Kato, E. Nakamae, T. Nishita, Tanaka, and Nogushi. Three-dimensional terrain modeling and display for environmental assessment. *Computer Graphics (Proceedings of SIGGRAPH'89)*, 23(3):207–214, 1989.

[11] R. Klein and T. Huttner. Simple camera-dependent approximation of terrain surfaces for fast visualization and animation. In *Proceedings of Visualization '96 (late breaking topics)*. IEEE Computer Society Press, 1996.

[12] P. Lindstrom, D. Koller, L.F. Hodges W. Ribarsky, N. Faust, and G. Turner. Real-time, continuous level of detail rendering of height fields. In *Proceedings of SIGGRAPH '96*, 1996.

[13] M. Shamos and F. Preparata. *Computational Geometry*. Springer, 1989.

[14] O. Sudarsky and C. Gotsman. Output-sensitive visibility algorithms for dynamic scenes with applications to virtual reality. *Computer Graphics Forum*, 15(3):249–258, 1996 (Proceedings of Eurographics, Poitiers, France, August 1996).

```
1. Calculate view frustum and bound terrain footprint by rectangle.
2. Scan-convert the rectangle and for each geometry tile in it:
   (a) If the tile is not in the footprint, but was in it in the previous frame, then:
       • Remove all its points from the Delaunay triangulation.
   (b) If the tile is in the footprint, but was not in the previous frame, then:
       • Request tile from server at appropriate resolution.
       • Search in tile octree for appropriate voxels.
       • Insert the points from these voxels in Delaunay triangulation.
   (c) If tile is in the footprint and was also in the previous frame, then:
       • Search in tile octree for appropriate voxels.
       • Find difference from previous frame.
       • Insert (Delete) difference points in (from) Delaunay triangulation.
3. For each texture tile in the bounding rectangle:
   (a) If the texture tile is in the footprint, but was not in the previous frame, then:
       • Calculate required resolution.
       • Request the appropriate bit stream prefix from the server.
   (b) If texture tile is in the footprint, and was also in the previous frame, then:
       • Calculate its resolution.
       • If this resolution is higher than that of the previous frame, then request more of the bit stream from the server.
4. For every tenth frame check the actual performance (frames/sec) against the required performance and adjust the geometric and/or texture resolution parameters to achieve that performance.
5. Render image.
```

Figure 1: Pseudo-code of the client algorithm.

APPENDIX R

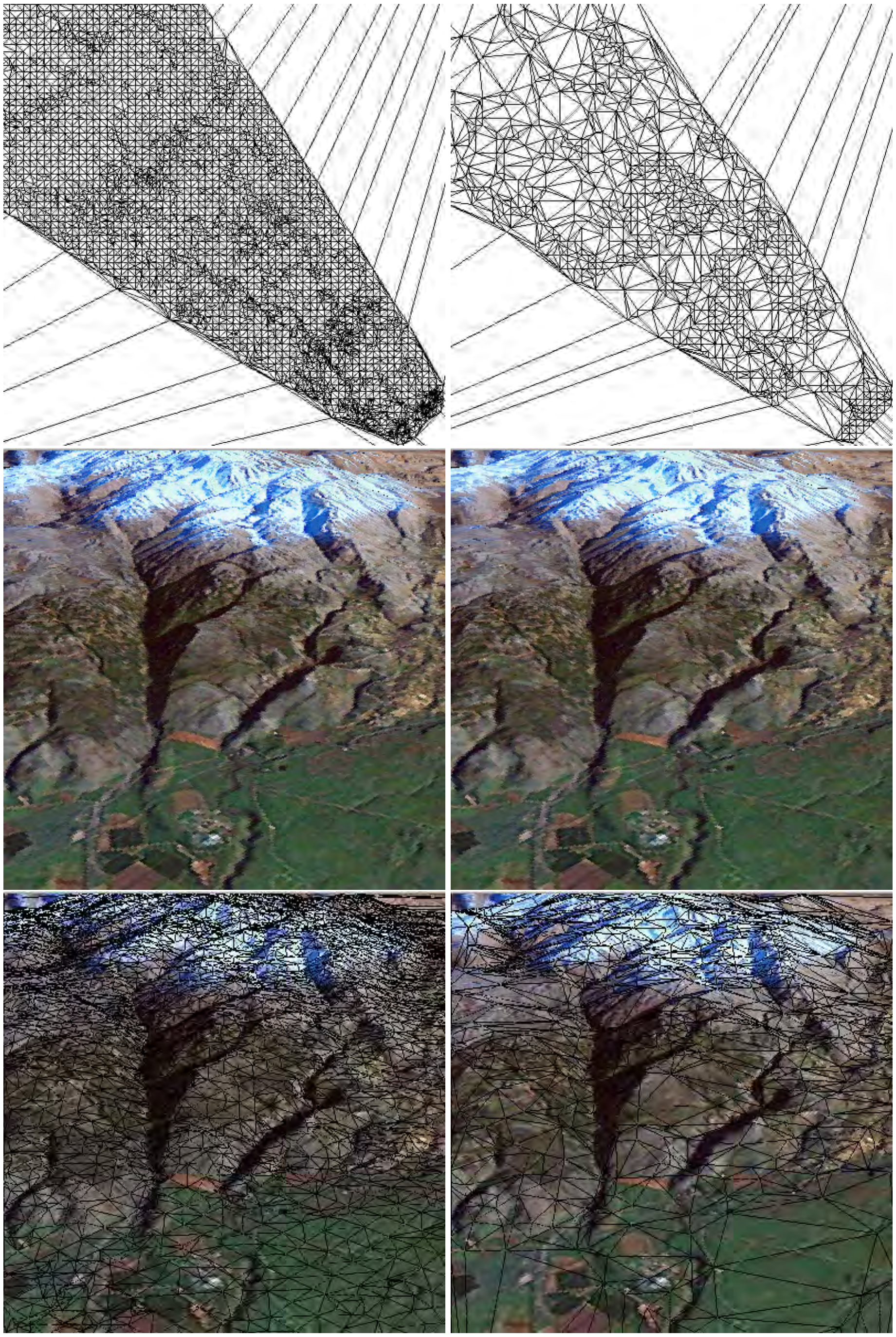


Figure 2: Terrain meshes (Delaunay triangulated) and views rendered at different data resolutions. (a) High resolution: 0.08 triangles/pixel and 1 texels/pixel. (b) Equivalent quality at lower resolution: 0.02 triangles and 0.8 texels/pixels. Note how more DTM points are used in foreground areas or areas of high curvature.

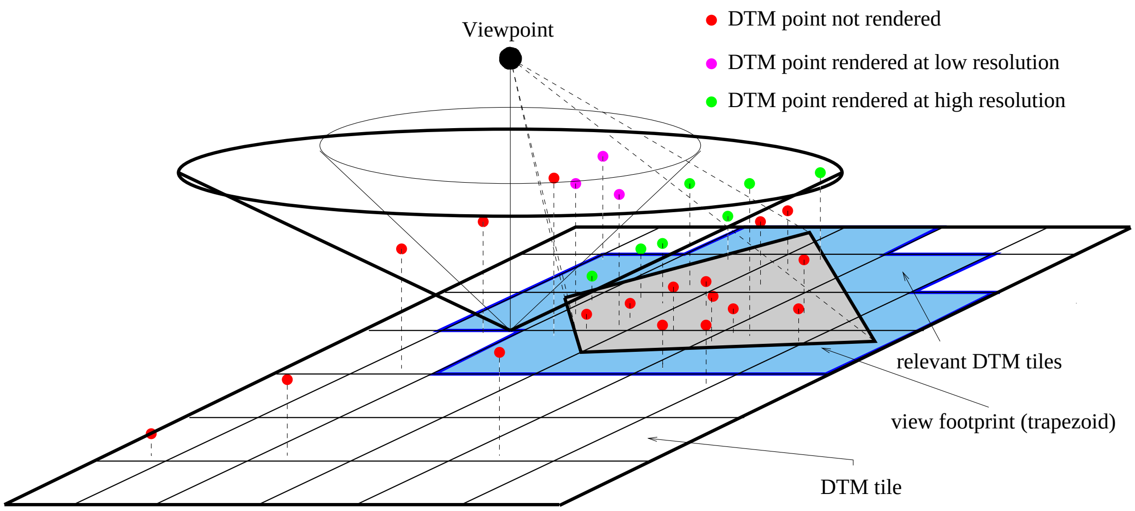

Viewpoint
DTM point not rendered
DTM point rendered at low resolution
DTM point rendered at high resolution
relevant DTM tiles
view footprint (trapezoid)
DTM tile

Figure 3: Determining the DTM points of the rendered Delaunay triangulation for a given view at different geometric resolutions. The narrow cone represents a low-resolution view, and the wide one a high resolution. The “elevations” of the DTM points are their precalculated grades. All points within the footprint with grade above the relevant cone are included in the triangulation. This range-reporting operation is performed efficiently using an octree structure on the points in each tile. Note that more points are admitted in the view foreground than in its background.

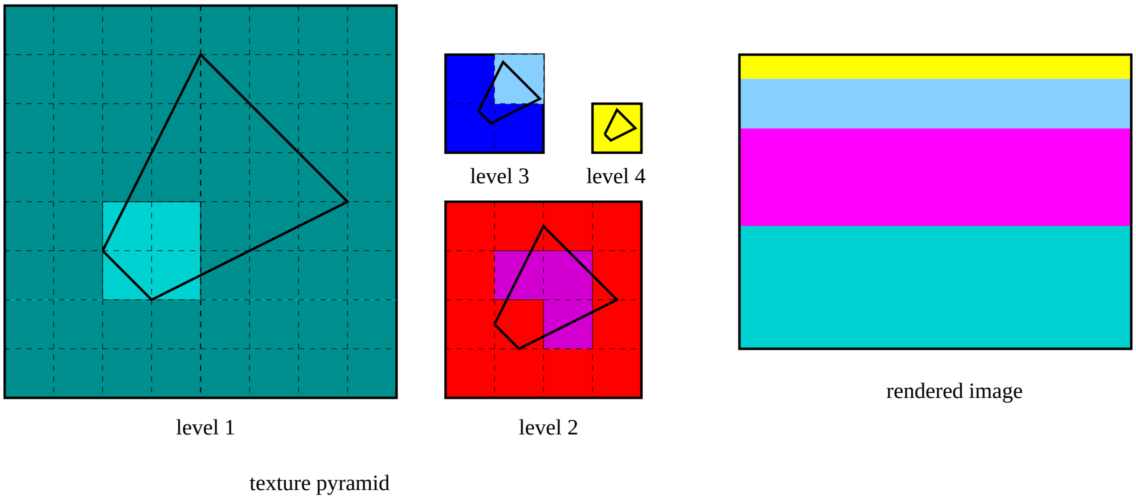
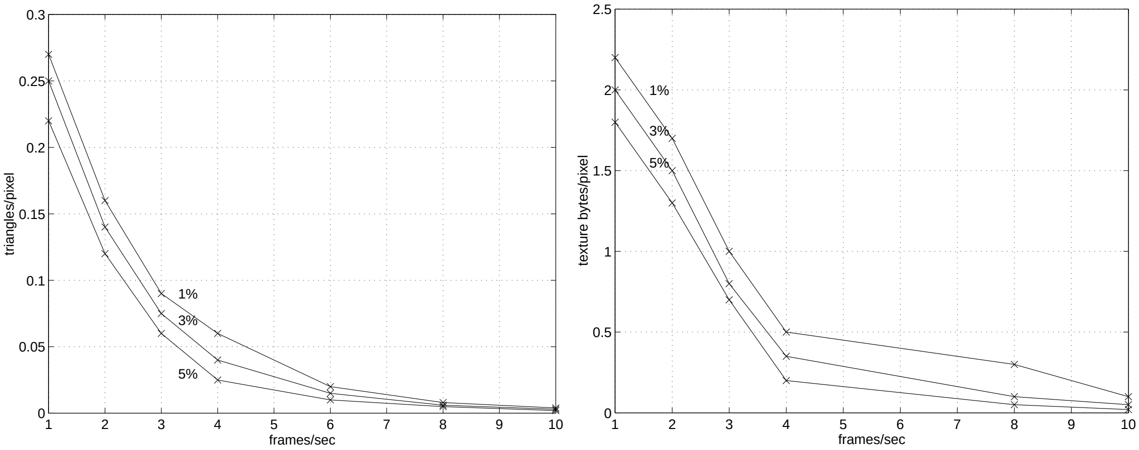


Figure 4: The contribution of individual tiles in the texture buffer to the rendered image corresponding to the marked footprint. Those tiles not contributing need not reside in the texture buffer at all, and are not streamed and decoded from the server.



(a) (b)

Figure 5: Speed/resolution tradeoff in our prototype visualization client while rendering 300x400 pixel images on a R5000 SGI $O_2$, accessing the scene database server over a 3 KByte/sec network. (a) Varying only geometric resolution. The texture resolution is fixed to 0.5 compressed texture bytes per pixel. (b) Varying only texture resolution. The geometric resolution is fixed to 0.06 triangles/pixel. The individual curves correspond to different flight velocities, which influence the turnover of data in system caches and bandwidth overhead.

APPENDIX R

# PROCEEDINGS
# Visualization '97

October 19 – 24, 1997

Phoenix, Arizona

*Sponsored by*
IEEE Computer Society Technical Committee on Computer Graphics

*In cooperation with*
ACM SIGGRAPH

APPENDIX R

**The Association for Computing Machinery**
**1515 Broadway**
**New York, NY 10036**

*The papers in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and, in the interests of timely dissemination, are published as presented and without change. Their inclusion in this publication does not necessarily constitute endorsement by the editors, the IEEE Computer Society Press, or the Institute of Electrical and Electronics Engineers, Inc.*

ACM ISBN: 1-58113-011-2
ACM Order Number: 428978

IEEE Computer Society Press Order Number: PR08262
IEEE Catalog Number: 97CB36155
IEEE ISBN: 0-8186-8262-0
IEEE ISBN - Library Binding: 0-8186-8263-9
IEEE ISBN - Microfiche: 0-8186-8264-7
ISSN: 1070-2385

Additional copies may be ordered from:

ACM Order Department
P.O. Box 12114
Church Street Station
New York, NY 10257 USA
Tel: +1-212-626-0500
Fax: +1-212-944-1318
E-mail: orders@acm.org

ACM European Service Center
108 Cowley Road
Oxford OX4 1JF
United Kingdom
E-mail: acm_europe@acm.org

IEEE Computer Society Press
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264 USA
Tel: +1-714-821-8380
Fax: +1-714-821-4641
E-mail: cs.books@computer.org

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331 USA
Tel: +1-908-981-1393
Fax: +1-908-981-9667
E-mail: mis.custserv@computer.org

IEEE Computer Society
13, Avenue de l'Aquilon
B-1200 Brussels
Belgium
Tel: +32-2-770-2198
Fax: +32-2-770-8505
E-mail: euro.ofc@computer.org

IEEE Computer Society
Ooshima Building
2-19-1 Minami-Aoyama
Minato-ku, Tokyo 107
Japan
Tel: +81-3-3408-3118
Fax: +81-3-3408-3553
E-mail: tokyo.ofc@computer.org

# Table of Contents

APPENDIX R

## Papers

### *Session 2B: Volume Rendering I*

### *Session 3A: Vector Fields*

### *Session 3B: Terrain Visualization*

APPENDIX R

### *Session 4A: Information Visualization*

### *Session 4B: MultiResolution*

### *Session 5A: User Interfaces & Interaction*

### *Session 5B: Volume Rendering II*

*Session 6A: Isosurfaces*

APPENDIX R

*Session 6B: Visualization Systems*

*Session 7A: Data Extraction*

APPENDIX R

### *Session 7B: Flow Visualization*

### *Session 8A: Compression*

### *Session 9A: Polygonal Surfaces*

### *Session 10A: Surface Simplification*

## Case Studies

### *Session 2C: Flow Visualization*

### *Session 3C: Medical Visualization*

### *Session 5C: Educational Visualization*

APPENDIX R

*Session 6C: Web & Virtual Reality*

*Session 7C: Engineering and Computational Geometry*

*Session 9B: Math & Statistics*

## Panels

**APPENDIX S**

# User Datagram Protocol (UDP) (Windows CE 5.0)

**Windows CE 5.0**

Send Feedback

UDP provides a connectionless, unreliable transport service. Connectionless means that a communication session between hosts is not established before exchanging data. UDP is often used for one-to-many communications that use broadcast or multicast IP datagrams. The UDP connectionless datagram delivery service is unreliable because it does not guarantee data packet delivery and no notification is sent if a packet is not delivered. Also, UDP does not guarantee that packets are delivered in the same order in which they were sent.

Because delivery of UDP datagrams is not guaranteed, applications using UDP must supply their own mechanisms for reliability, if needed. Although UDP appears to have some limitations, it is useful in certain situations. For example, Winsock IP multicasting is implemented with UDP datagram type sockets. UDP is very efficient because of low overhead. Microsoft networking uses UDP for logon, browsing, and name resolution. UDP can also be used to carry IP multicast streams for applications such as Microsoft® Windows Media®.

**See Also**

Core Protocol Stack for IPv4 | User Datagram Protocol (UDP) and Name Resolution for IPv4

Send Feedback on this topic to the authors

Feedback FAQs

Microsoft et al. Exhibit 1005

APPENDIX T

# The OpenGL® Graphics System: A Specification (Version 1.2.1)

Mark Segal
Kurt Akeley

*Editor (version 1.1): Chris Frazier*
*Editor (versions 1.2, 1.2.1): Jon Leech*

Microsoft et al. Exhibit 1005

APPENDIX T

Microsoft et al. Exhibit 1005

# Contents

**APPENDIX T**

APPENDIX T

APPENDIX T

APPENDIX T

APPENDIX T

# List of Figures

APPENDIX T

APPENDIX T

# List of Tables

APPENDIX T

APPENDIX T

APPENDIX T

# Chapter 1

# Introduction

This document describes the OpenGL graphics system: what it is, how it acts, and what is required to implement it. We assume that the reader has at least a rudimentary understanding of computer graphics. This means familiarity with the essentials of computer graphics algorithms as well as familiarity with basic graphics hardware and associated terms.

## 1.1 Formatting of Optional Features

Starting with version 1.2 of OpenGL, some features in the specification are considered optional; an OpenGL implementation may or may not choose to provide them (see section 3.6.2).

Portions of the specification which are optional are so labelled where they are defined. Additionally, those portions are typeset in gray, and state table entries which are optional are typeset against a gray background .

## 1.2 What is the OpenGL Graphics System?

OpenGL (for "Open Graphics Library") is a software interface to graphics hardware. The interface consists of a set of several hundred procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects.

Most of OpenGL requires that the graphics hardware contain a framebuffer. Many OpenGL calls pertain to drawing objects such as points, lines, polygons, and bitmaps, but the way that some of this drawing occurs (such as when antialiasing or texturing is enabled) relies on the existence of a

APPENDIX T

framebuffer. Further, some of OpenGL is specifically concerned with framebuffer manipulation.

## 1.3 Programmer's View of OpenGL

To the programmer, OpenGL is a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the framebuffer. For the most part, OpenGL provides an immediate-mode interface, meaning that specifying an object causes it to be drawn.

A typical program that uses OpenGL begins with calls to open a window into the framebuffer into which the program will draw. Then, calls are made to allocate a GL context and associate it with the window. Once a GL context is allocated, the programmer is free to issue OpenGL commands. Some calls are used to draw simple geometric objects (i.e. points, line segments, and polygons), while others affect the rendering of these primitives including how they are lit or colored and how they are mapped from the user's two- or three-dimensional model space to the two-dimensional screen. There are also calls to effect direct control of the framebuffer, such as reading and writing pixels.

## 1.4 Implementor's View of OpenGL

To the implementor, OpenGL is a set of commands that affect the operation of graphics hardware. If the hardware consists only of an addressable framebuffer, then OpenGL must be implemented almost entirely on the host CPU. More typically, the graphics hardware may comprise varying degrees of graphics acceleration, from a raster subsystem capable of rendering two-dimensional lines and polygons to sophisticated floating-point processors capable of transforming and computing on geometric data. The OpenGL implementor's task is to provide the CPU software interface while dividing the work for each OpenGL command between the CPU and the graphics hardware. This division must be tailored to the available graphics hardware to obtain optimum performance in carrying out OpenGL calls.

OpenGL maintains a considerable amount of state information. This state controls how objects are drawn into the framebuffer. Some of this state is directly available to the user: he or she can make calls to obtain its value. Some of it, however, is visible only by the effect it has on what is drawn. One of the main goals of this specification is to make OpenGL state

APPENDIX T

information explicit, to elucidate how it changes, and to indicate what its effects are.

## 1.5 Our View

We view OpenGL as a state machine that controls a set of specific drawing operations. This model should engender a specification that satisfies the needs of both programmers and implementors. It does not, however, necessarily provide a model for implementation. An implementation must produce results conforming to those produced by the specified methods, but there may be ways to carry out a particular computation that are more efficient than the one specified.

Microsoft et al. Exhibit 1005

APPENDIX T

# Chapter 2

# OpenGL Operation

## 2.1 OpenGL Fundamentals

OpenGL (henceforth, the "GL") is concerned only with rendering into a framebuffer (and reading values stored in that framebuffer). There is no support for other peripherals sometimes associated with graphics hardware, such as mice and keyboards. Programmers must rely on other mechanisms to obtain user input.

The GL draws *primitives* subject to a number of selectable modes. Each primitive is a point, line segment, polygon, or pixel rectangle. Each mode may be changed independently; the setting of one does not affect the settings of others (although many modes may interact to determine what eventually ends up in the framebuffer). Modes are set, primitives specified, and other GL operations described by sending *commands* in the form of function or procedure calls.

Primitives are defined by a group of one or more *vertices*. A vertex defines a point, an endpoint of an edge, or a corner of a polygon where two edges meet. Data (consisting of positional coordinates, colors, normals, and texture coordinates) are associated with a vertex and each vertex is processed independently, in order, and in the same way. The only exception to this rule is if the group of vertices must be *clipped* so that the indicated primitive fits within a specified region; in this case vertex data may be modified and new vertices created. The type of clipping depends on which primitive the group of vertices represents.

Commands are always processed in the order in which they are received, although there may be an indeterminate delay before the effects of a command are realized. This means, for example, that one primitive must be

APPENDIX T

drawn completely before any subsequent one can affect the framebuffer. It also means that queries and pixel read operations return state consistent with complete execution of all previously invoked GL commands. In general, the effects of a GL command on either GL modes or the framebuffer must be complete before any subsequent command can have any such effects.

In the GL, data binding occurs on call. This means that data passed to a command are interpreted when that command is received. Even if the command requires a pointer to data, those data are interpreted when the call is made, and any subsequent changes to the data have no effect on the GL (unless the same pointer is used in a subsequent command).

The GL provides direct control over the fundamental operations of 3D and 2D graphics. This includes specification of such parameters as transformation matrices, lighting equation coefficients, antialiasing methods, and pixel update operators. It does not provide a means for describing or modeling complex geometric objects. Another way to describe this situation is to say that the GL provides mechanisms to describe how complex geometric objects are to be rendered rather than mechanisms to describe the complex objects themselves.

The model for interpretation of GL commands is client-server. That is, a program (the client) issues commands, and these commands are interpreted and processed by the GL (the server). The server may or may not operate on the same computer as the client. In this sense, the GL is "network-transparent." A server may maintain a number of GL *contexts*, each of which is an encapsulation of current GL state. A client may choose to *connect* to any one of these contexts. Issuing GL commands when the program is not *connected* to a *context* results in undefined behavior.

The effects of GL commands on the framebuffer are ultimately controlled by the window system that allocates framebuffer resources. It is the window system that determines which portions of the framebuffer the GL may access at any given time and that communicates to the GL how those portions are structured. Therefore, there are no GL commands to configure the framebuffer or initialize the GL. Similarly, display of framebuffer contents on a CRT monitor (including the transformation of individual framebuffer values by such techniques as gamma correction) is not addressed by the GL. Framebuffer configuration occurs outside of the GL in conjunction with the window system; the initialization of a GL context occurs when the window system allocates a window for GL rendering.

The GL is designed to be run on a range of graphics platforms with varying graphics capabilities and performance. To accommodate this variety, we specify ideal behavior instead of actual behavior for certain GL operations.

APPENDIX T

In cases where deviation from the ideal is allowed, we also specify the rules that an implementation must obey if it is to approximate the ideal behavior usefully. This allowed variation in GL behavior implies that two distinct GL implementations may not agree pixel for pixel when presented with the same input even when run on identical framebuffer configurations.

Finally, command names, constants, and types are prefixed in the GL (by **gl**, `GL_`, and `GL`, respectively in `C`) to reduce name clashes with other packages. The prefixes are omitted in this document for clarity.

### 2.1.1 Floating-Point Computation

The GL must perform a number of floating-point operations during the course of its operation. We do not specify how floating-point numbers are to be represented or how operations on them are to be performed. We require simply that numbers' floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1 part in $10^5$. The maximum representable magnitude of a floating-point number used to represent positional or normal coordinates must be at least $2^{32}$; the maximum representable magnitude for colors or texture coordinates must be at least $2^{10}$. The maximum representable magnitude for all other floating-point values must be at least $2^{32}$. $x \cdot 0 = 0 \cdot x = 0$ for any non-infinite and non-NaN $x$. $1 \cdot x = x \cdot 1 = x$. $x + 0 = 0 + x = x$. $0^0 = 1$. (Occasionally further requirements will be specified.) Most single-precision floating-point formats meet these requirements.

Any representable floating-point value is legal as input to a GL command that requires floating-point data. The result of providing a value that is not a floating-point number to such a command is unspecified, but must not lead to GL interruption or termination. In IEEE arithmetic, for example, providing a negative zero or a denormalized number to a GL command yields predictable results, while providing a NaN or an infinity yields unspecified results.

Some calculations require division. In such cases (including implied divisions required by vector normalizations), a division by zero produces an unspecified result but must not lead to GL interruption or termination.

## 2.2 GL State

The GL maintains considerable state. This document enumerates each state variable and describes how each variable can be changed. For purposes of discussion, state variables are categorized somewhat arbitrarily by their function. Although we describe the operations that the GL performs on the framebuffer, the framebuffer is not a part of GL state.

We distinguish two types of state. The first type of state, called GL *server state*, resides in the GL server. The majority of GL state falls into this category. The second type of state, called GL *client state*, resides in the GL client. Unless otherwise specified, all state referred to in this document is GL server state; GL client state is specifically identified. Each instance of a GL context implies one complete set of GL server state; each connection from a client to a server implies a set of both GL client state and GL server state.

While an implementation of the GL may be hardware dependent, this discussion is independent of the specific hardware on which a GL is implemented. We are therefore concerned with the state of graphics hardware only when it corresponds precisely to GL state.

## 2.3 GL Command Syntax

GL commands are functions or procedures. Various groups of commands perform the same operation but differ in how arguments are supplied to them. To conveniently accommodate this variation, we adopt a notation for describing commands and their arguments.

GL commands are formed from a *name* followed, depending on the particular command, by up to 4 characters. The first character indicates the number of values of the indicated type that must be presented to the command. The second character or character pair indicates the specific type of the arguments: 8-bit integer, 16-bit integer, 32-bit integer, single-precision floating-point, or double-precision floating-point. The final character, if present, is `v`, indicating that the command takes a pointer to an array (a vector) of values rather than a series of individual arguments. Two specific examples come from the **Vertex** command:

`void` **Vertex3f**( `float` *x*, `float` *y*, `float` *z* );

and

`void` **Vertex2sv**( `short` *v[2]* );

These examples show the ANSI `C` declarations for these commands. In general, a command declaration has the form[1]

[1] The declarations shown in this document apply to ANSI `C`. Languages such as `C++`

APPENDIX T

| Letter | Corresponding `GL` Type |
|---|---|
| **b** | `byte` |
| **s** | `short` |
| **i** | `int` |
| **f** | `float` |
| **d** | `double` |
| **ub** | `ubyte` |
| **us** | `ushort` |
| **ui** | `uint` |

Table 2.1: Correspondence of command suffix letters to GL argument types. Refer to Table 2.2 for definitions of the GL types.

*rtype* **Name**{$\epsilon$**1234**}{$\epsilon$ **b s i f d ub us ui**}{$\epsilon$**v**}
( *[args ,] T arg1 , . . . , T argN [, args]* ) ;

*rtype* is the return type of the function. The braces ({}) enclose a series of characters (or character pairs) of which one is selected. $\epsilon$ indicates no character. The arguments enclosed in brackets (*[args ,]* and *[, args]*) may or may not be present. The $N$ arguments *arg1* through *argN* have type $T$, which corresponds to one of the type letters or letter pairs as indicated in Table 2.1 (if there are no letters, then the arguments' type is given explicitly). If the final character is not **v**, then $N$ is given by the digit **1**, **2**, **3**, or **4** (if there is no digit, then the number of arguments is fixed). If the final character is **v**, then only *arg1* is present and it is an array of $N$ values of the indicated type. Finally, we indicate an `unsigned` type by the shorthand of prepending a `u` to the beginning of the type name (so that, for instance, `unsigned char` is abbreviated `uchar`).

For example,

`void` **Normal3**{**fd**}( `T` *arg* );

indicates the two declarations

`void` **Normal3f**( `float` *arg1*, `float` *arg2*, `float` *arg3* );
`void` **Normal3d**( `double` *arg1*, `double` *arg2*, `double` *arg3* );

while

and Ada that allow passing of argument type information admit simpler declarations and fewer entry points.

APPENDIX T

```
void Normal3{fd}v( T arg );
```

means the two declarations

```
void Normal3fv( float arg[3] );
void Normal3dv( double arg[3] );
```

Arguments whose type is fixed (i.e. not indicated by a suffix on the command) are of one of 14 types (or pointers to one of these). These types are summarized in Table 2.2.

## 2.4 Basic GL Operation

Figure 2.1 shows a schematic diagram of the GL. Commands enter the GL on the left. Some commands specify geometric objects to be drawn while others control how the objects are handled by the various stages. Most commands may be accumulated in a *display list* for processing by the GL at a later time. Otherwise, commands are effectively sent through a processing pipeline.

The first stage provides an efficient means for approximating curve and surface geometry by evaluating polynomial functions of input values. The next stage operates on geometric primitives described by vertices: points, line segments, and polygons. In this stage vertices are transformed and lit, and primitives are clipped to a viewing volume in preparation for the next stage, rasterization. The rasterizer produces a series of framebuffer addresses and values using a two-dimensional description of a point, line segment, or polygon. Each *fragment* so produced is fed to the next stage that performs operations on individual fragments before they finally alter the framebuffer. These operations include conditional updates into the framebuffer based on incoming and previously stored depth values (to effect depth buffering), blending of incoming fragment colors with stored colors, as well as masking and other logical operations on fragment values.

Finally, there is a way to bypass the vertex processing portion of the pipeline to send a block of fragments directly to the individual fragment operations, eventually causing a block of pixels to be written to the framebuffer; values may also be read back from the framebuffer or copied from one portion of the framebuffer to another. These transfers may include some type of decoding or encoding.

This ordering is meant only as a tool for describing the GL, not as a strict rule of how the GL is implemented, and we present it only as a means to

| GL Type | Minimum Number of Bits | Description |
|---|---|---|
| `boolean` | 1 | Boolean |
| `byte` | 8 | signed 2's complement binary integer |
| `ubyte` | 8 | unsigned binary integer |
| `short` | 16 | signed 2's complement binary integer |
| `ushort` | 16 | unsigned binary integer |
| `int` | 32 | signed 2's complement binary integer |
| `uint` | 32 | unsigned binary integer |
| `sizei` | 32 | Non-negative binary integer size |
| `enum` | 32 | Enumerated binary integer value |
| `bitfield` | 32 | Bit field |
| `float` | 32 | Floating-point value |
| `clampf` | 32 | Floating-point value clamped to $[0, 1]$ |
| `double` | 64 | Floating-point value |
| `clampd` | 64 | Floating-point value clamped to $[0, 1]$ |

Table 2.2: GL data types. GL types are not C types. Thus, for example, GL type `int` is referred to as `GLint` outside this document, and is not necessarily equivalent to the C type `int`. An implementation may use more bits than the number indicated in the table to represent a GL type. Correct interpretation of integer values outside the minimum range is not required, however.

APPENDIX T

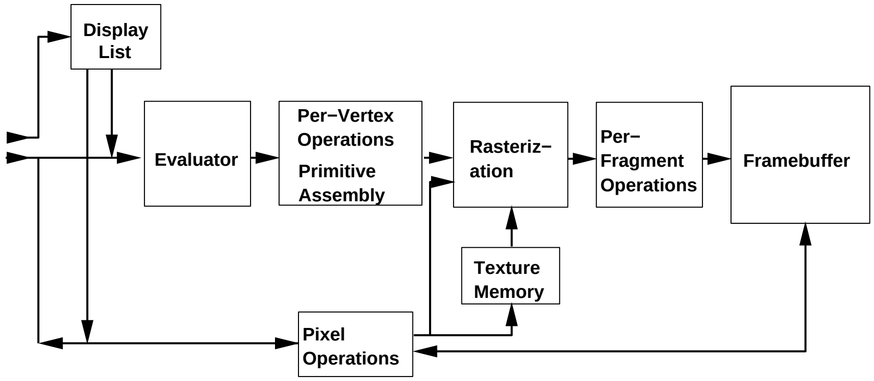


Figure 2.1. Block diagram of the GL.

organize the various operations of the GL. Objects such as curved surfaces, for instance, may be transformed before they are converted to polygons.

## 2.5 GL Errors

The GL detects only a subset of those conditions that could be considered errors. This is because in many cases error checking would adversely impact the performance of an error-free program.

The command

enum **GetError**( void );

is used to obtain error information. Each detectable error is assigned a numeric code. When an error is detected, a flag is set and the code is recorded. Further errors, if they occur, do not affect this recorded code. When **GetError** is called, the code is returned and the flag is cleared, so that a further error will again record its code. If a call to **GetError** returns NO_ERROR, then there has been no detectable error since the last call to **GetError** (or since the GL was initialized).

To allow for distributed implementations, there may be several flag-code pairs. In this case, after a call to **GetError** returns a value other than NO_ERROR each subsequent call returns the non-zero code of a distinct flag-code pair (in unspecified order), until all non-NO_ERROR codes have been returned. When there are no more non-NO_ERROR error codes, all flags are reset. This scheme requires some positive number of pairs of a flag bit and an integer. The initial state of all flags is cleared and the initial value of all codes is NO_ERROR.

Table 2.3 summarizes GL errors. Currently, when an error flag is set, results of GL operation are undefined only if OUT_OF_MEMORY has occurred. In other cases, the command generating the error is ignored so that it has no effect on GL state or framebuffer contents. If the generating command returns a value, it returns zero. If the generating command modifies values through a pointer argument, no change is made to these values. These error semantics apply only to GL errors, not to system errors such as memory access errors. This behavior is the current behavior; the action of the GL in the presence of errors is subject to change.

Three error generation conditions are implicit in the description of every GL command. First, if a command that requires an enumerated value is passed a symbolic constant that is not one of those specified as allowable for that command, the error INVALID_ENUM results. This is the case even if the argument is a pointer to a symbolic constant if that value is not allowable for the given command. Second, if a negative number is provided where an argument of type sizei is specified, the error INVALID_VALUE results. Finally, if memory is exhausted as a side effect of the execution of a command, the error OUT_OF_MEMORY may be generated. Otherwise errors are generated only for conditions that are explicitly described in this specification.

## 2.6 Begin/End Paradigm

In the GL, most geometric objects are drawn by enclosing a series of coordinate sets that specify vertices and optionally normals, texture coordinates, and colors between **Begin**/**End** pairs. There are ten geometric objects that are drawn this way: points, line segments, line segment loops, separated line segments, polygons, triangle strips, triangle fans, separated triangles, quadrilateral strips, and separated quadrilaterals.

Each vertex is specified with two, three, or four coordinates. In addition, a *current normal*, *current texture coordinates*, and *current color* may be used in processing each vertex. Normals are used by the GL in lighting calculations; the current normal is a three-dimensional vector that may be set by sending three coordinates that specify it. Texture coordinates determine how a texture image is mapped onto a primitive.

Primary and secondary colors are associated with each vertex (see sec-

| Error | Description | Offending command ignored? |
|---|---|---|
| INVALID_ENUM | enum argument out of range | Yes |
| INVALID_VALUE | Numeric argument out of range | Yes |
| INVALID_OPERATION | Operation illegal in current state | Yes |
| STACK_OVERFLOW | Command would cause a stack overflow | Yes |
| STACK_UNDERFLOW | Command would cause a stack underflow | Yes |
| OUT_OF_MEMORY | Not enough memory left to execute command | Unknown |
| TABLE_TOO_LARGE | The specified table is too large | Yes |

Table 2.3: Summary of GL errors

tion 3.9). These *associated* colors are either based on the current color or produced by lighting, depending on whether or not lighting is enabled. Texture coordinates are similarly associated with each vertex. Figure 2.2 summarizes the association of auxiliary data with a transformed vertex to produce a *processed vertex.*

The current values are part of GL state. Vertices and normals are transformed, colors may be affected or replaced by lighting, and texture coordinates are transformed and possibly affected by a texture coordinate generation function. The processing indicated for each current value is applied for each vertex that is sent to the GL.

The methods by which vertices, normals, texture coordinates, and colors are sent to the GL, as well as how normals are transformed and how vertices are mapped to the two-dimensional screen, are discussed later.

Before colors have been assigned to a vertex, the state required by a vertex is the vertex's coordinates, the current normal, the current edge flag (see section 2.6.2), the current material properties (see section 2.13.2), and the current texture coordinates. Because color assignment is done vertex-by-vertex, a processed vertex comprises the vertex's coordinates, its edge flag, its assigned colors, and its texture coordinates.

Figure 2.3 shows the sequence of operations that builds a *primitive* (point, line segment, or polygon) from a sequence of vertices. After a primi-

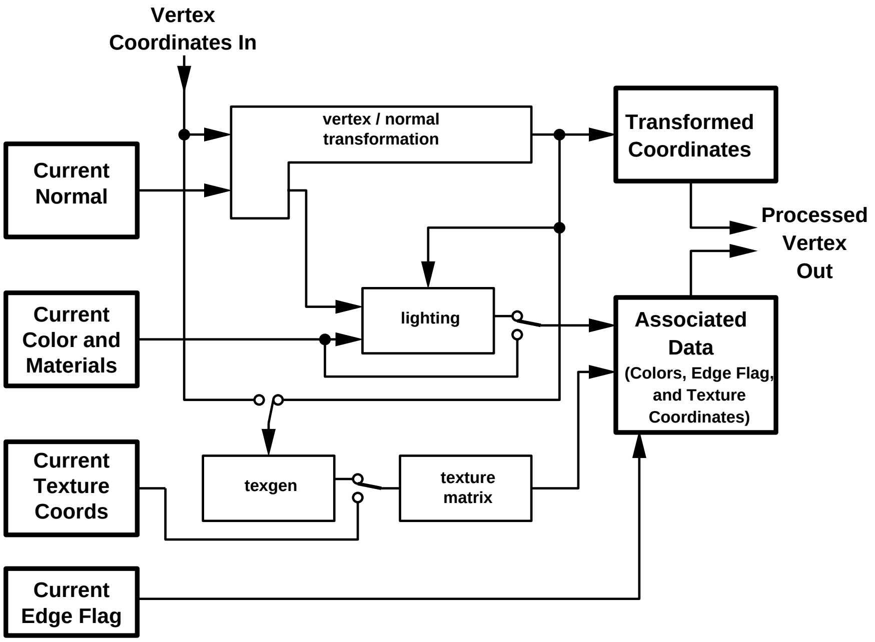


Figure 2.2. Association of current values with a vertex. The heavy lined boxes represent GL state.

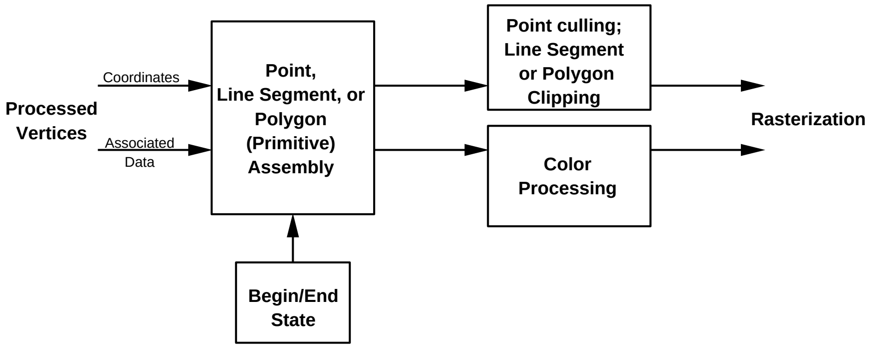


Figure 2.3. Primitive assembly and processing.

**APPENDIX T**

tive is formed, it is clipped to a viewing volume. This may alter the primitive by altering vertex coordinates, texture coordinates, and colors. In the case of a polygon primitive, clipping may insert new vertices into the primitive. The vertices defining a primitive to be rasterized have texture coordinates and colors associated with them.

### 2.6.1 Begin and End Objects

**Begin** and **End** require one state variable with eleven values: one value for each of the ten possible **Begin**/**End** objects, and one other value indicating that no **Begin**/**End** object is being processed. The two relevant commands are

```
void Begin( enum mode );
void End( void );
```

There is no limit on the number of vertices that may be specified between a **Begin** and an **End**.

**Points.** A series of individual points may be specified by calling **Begin** with an argument value of `POINTS`. No special state need be kept between **Begin** and **End** in this case, since each point is independent of previous and following points.

**Line Strips.** A series of one or more connected line segments is specified by enclosing a series of two or more endpoints within a **Begin**/**End** pair when **Begin** is called with `LINE_STRIP`. In this case, the first vertex specifies the first segment's start point while the second vertex specifies the first segment's endpoint and the second segment's start point. In general, the $i$th vertex (for $i > 1$) specifies the beginning of the $i$th segment and the end of the $i - 1$st. The last vertex specifies the end of the last segment. If only one vertex is specified between the **Begin**/**End** pair, then no primitive is generated.

The required state consists of the processed vertex produced from the last vertex that was sent (so that a line segment can be generated from it to the current vertex), and a boolean flag indicating if the current vertex is the first vertex.

**Line Loops.** Line loops, specified with the `LINE_LOOP` argument value to **Begin**, are the same as line strips except that a final segment is added from the final specified vertex to the first vertex. The additional state consists of the processed first vertex.

**Separate Lines.** Individual line segments, each specified by a pair of vertices, are generated by surrounding vertex pairs with **Begin** and **End**

APPENDIX T

when the value of the argument to **Begin** is LINES. In this case, the first two vertices between a **Begin** and **End** pair define the first segment, with subsequent pairs of vertices each defining one more segment. If the number of specified vertices is odd, then the last one is ignored. The state required is the same as for lines but it is used differently: a vertex holding the first vertex of the current segment, and a boolean flag indicating whether the current vertex is odd or even (a segment start or end).

**Polygons.** A polygon is described by specifying its boundary as a series of line segments. When **Begin** is called with POLYGON, the bounding line segments are specified in the same way as line loops. Depending on the current state of the GL, a polygon may be rendered in one of several ways such as outlining its border or filling its interior. A polygon described with fewer than three vertices does not generate a primitive.

Only convex polygons are guaranteed to be drawn correctly by the GL. If a specified polygon is nonconvex when projected onto the window, then the rendered polygon need only lie within the convex hull of the projected vertices defining its boundary.

The state required to support polygons consists of at least two processed vertices (more than two are never required, although an implementation may use more); this is because a convex polygon can be rasterized as its vertices arrive, before all of them have been specified. The order of the vertices is significant in lighting and polygon rasterization (see sections 2.13.1 and 3.5.1).

**Triangle strips.** A triangle strip is a series of triangles connected along shared edges. A triangle strip is specified by giving a series of defining vertices between a **Begin**/**End** pair when **Begin** is called with TRIANGLE_STRIP. In this case, the first three vertices define the first triangle (and their order is significant, just as for polygons). Each subsequent vertex defines a new triangle using that point along with two vertices from the previous triangle. A **Begin**/**End** pair enclosing fewer than three vertices, when TRIANGLE_STRIP has been supplied to **Begin**, produces no primitive. See Figure 2.4.

The state required to support triangle strips consists of a flag indicating if the first triangle has been completed, two stored processed vertices, (called vertex A and vertex B), and a one bit pointer indicating which stored vertex will be replaced with the next vertex. After a **Begin**(TRIANGLE_STRIP), the pointer is initialized to point to vertex A. Each vertex sent between a **Begin**/**End** pair toggles the pointer. Therefore, the first vertex is stored as vertex A, the second stored as vertex B, the third stored as vertex A, and so on. Any vertex after the second one sent forms a triangle from vertex A, vertex B, and the current vertex (in that order).

**Triangle fans.** A triangle fan is the same as a triangle strip with one

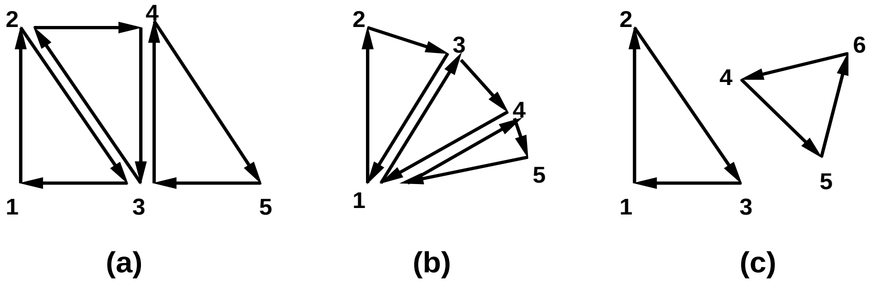


Figure 2.4. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices between **Begin** and **End**. Note that in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle's edges is independent of the other triangles.

exception: each vertex after the first always replaces vertex B of the two stored vertices. The vertices of a triangle fan are enclosed between **Begin** and **End** when the value of the argument to **Begin** is `TRIANGLE_FAN`.

**Separate Triangles.** Separate triangles are specified by placing vertices between **Begin** and **End** when the value of the argument to **Begin** is `TRIANGLES`. In this case, The $3i + 1$st, $3i + 2$nd, and $3i + 3$rd vertices (in that order) determine a triangle for each $i = 0, 1, \ldots, n-1$, where there are $3n+k$ vertices between the **Begin** and **End**. $k$ is either 0, 1, or 2; if $k$ is not zero, the final $k$ vertices are ignored. For each triangle, vertex A is vertex $3i$ and vertex B is vertex $3i+1$. Otherwise, separate triangles are the same as a triangle strip.

The rules given for polygons also apply to each triangle generated from a triangle strip, triangle fan or from separate triangles.

**Quadrilateral (quad) strips.** Quad strips generate a series of edge-sharing quadrilaterals from vertices appearing between **Begin** and **End**, when **Begin** is called with `QUAD_STRIP`. If the $m$ vertices between the **Begin** and **End** are $v_1, \ldots, v_m$, where $v_j$ is the $j$th specified vertex, then quad $i$ has vertices (in order) $v_{2i}$, $v_{2i+1}$, $v_{2i+3}$, and $v_{2i+2}$ with $i = 0, \ldots, \lfloor m/2 \rfloor$. The state required is thus three processed vertices, to store the last two vertices of the previous quad along with the third vertex (the first new vertex) of the current quad, a flag to indicate when the first quad has been completed, and a one-bit counter to count members of a vertex pair. See Figure 2.5.

APPENDIX T

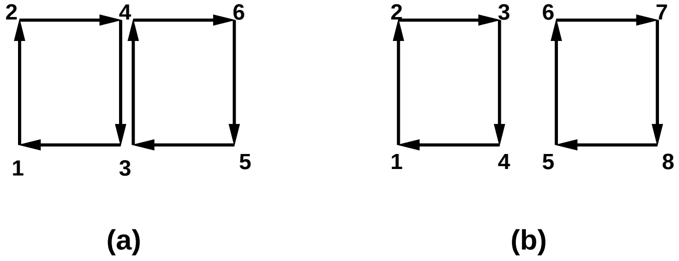


Figure 2.5. (a) A quad strip. (b) Independent quads. The numbers give the sequencing of the vertices between **Begin** and **End**.

A quad strip with fewer than four vertices generates no primitive. If the number of vertices specified for a quadrilateral strip between **Begin** and **End** is odd, the final vertex is ignored.

**Separate Quadrilaterals** Separate quads are just like quad strips except that each group of four vertices, the $4j+1$st, the $4j+2$nd, the $4j+3$rd, and the $4j+4$th, generate a single quad, for $j = 0, 1, \ldots, n-1$. The total number of vertices between **Begin** and **End** is $4n+k$, where $0 \leq k \leq 3$; if $k$ is not zero, the final $k$ vertices are ignored. Separate quads are generated by calling **Begin** with the argument value `QUADS`.

The rules given for polygons also apply to each quad generated in a quad strip or from separate quads.

### 2.6.2 Polygon Edges

Each edge of each primitive generated from a polygon, triangle strip, triangle fan, separate triangle set, quadrilateral strip, or separate quadrilateral set, is flagged as either *boundary* or *non-boundary.* These classifications are used during polygon rasterization; some modes affect the interpretation of polygon boundary edges (see section 3.5.4). By default, all edges are boundary edges, but the flagging of polygons, separate triangles, or separate quadrilaterals may be altered by calling

```
void EdgeFlag( boolean flag );
void EdgeFlagv( boolean *flag );
```

to change the value of a flag bit. If *flag* is zero, then the flag bit is set to `FALSE`; if *flag* is non-zero, then the flag bit is set to `TRUE`.

Microsoft et al. Exhibit 1005

APPENDIX T

When **Begin** is supplied with one of the argument values `POLYGON`, `TRIANGLES`, or `QUADS`, each vertex specified within a **Begin** and **End** pair begins an edge. If the edge flag bit is `TRUE`, then each specified vertex begins an edge that is flagged as boundary. If the bit is `FALSE`, then induced edges are flagged as non-boundary.

The state required for edge flagging consists of one current flag bit. Initially, the bit is `TRUE`. In addition, each processed vertex of an assembled polygonal primitive must be augmented with a bit indicating whether or not the edge beginning on that vertex is boundary or non-boundary.

### 2.6.3 GL Commands within Begin/End

The only GL commands that are allowed within any **Begin**/**End** pairs are the commands for specifying vertex coordinates, vertex color, normal coordinates, and texture coordinates (**Vertex**, **Color**, **Index**, **Normal**, **TexCoord**), the **ArrayElement** command (see section 2.8), the **EvalCoord** and **EvalPoint** commands (see section 5.1), commands for specifying lighting material parameters (**Material** commands; see section 2.13.2), display list invocation commands (**CallList** and **CallLists**; see section 5.4), and the **EdgeFlag** command. Executing any other GL command between the execution of **Begin** and the corresponding execution of **End** results in the error `INVALID_OPERATION`. Executing **Begin** after **Begin** has already been executed but before an **End** is executed generates the `INVALID_OPERATION` error, as does executing **End** without a previous corresponding **Begin**.

Execution of the commands **EnableClientState**, **DisableClientState**, **PushClientAttrib**, **PopClientAttrib**, **EdgeFlagPointer**, **TexCoordPointer**, **ColorPointer**, **IndexPointer**, **NormalPointer**, **VertexPointer**, **InterleavedArrays**, and **PixelStore**, is not allowed within any **Begin**/**End** pair, but an error may or may not be generated if such execution occurs. If an error is not generated, GL operation is undefined. (These commands are described in sections 2.8, 3.6.1, and Chapter 6.)

## 2.7 Vertex Specification

Vertices are specified by giving their coordinates in two, three, or four dimensions. This is done using one of several versions of the **Vertex** command:

```
void Vertex{234}{sifd}( T coords );
void Vertex{234}{sifd}v( T coords );
```

A call to any **Vertex** command specifies four coordinates: $x$, $y$, $z$, and $w$. The $x$ coordinate is the first coordinate, $y$ is second, $z$ is third, and $w$ is fourth. A call to **Vertex2** sets the $x$ and $y$ coordinates; the $z$ coordinate is implicitly set to zero and the $w$ coordinate to one. **Vertex3** sets $x$, $y$, and $z$ to the provided values and $w$ to one. **Vertex4** sets all four coordinates, allowing the specification of an arbitrary point in projective three-space. Invoking a **Vertex** command outside of a **Begin**/**End** pair results in undefined behavior.

Current values are used in associating auxiliary data with a vertex as described in section 2.6. A current value may be changed at any time by issuing an appropriate command. The commands

```
void TexCoord{1234}{sifd}( T coords );
void TexCoord{1234}{sifd}v( T coords );
```

specify the current homogeneous texture coordinates, named $s$, $t$, $r$, and $q$. The **TexCoord1** family of commands set the $s$ coordinate to the provided single argument while setting $t$ and $r$ to 0 and $q$ to 1. Similarly, **TexCoord2** sets $s$ and $t$ to the specified values, $r$ to 0 and $q$ to 1; **TexCoord3** sets $s$, $t$, and $r$, with $q$ set to 1, and **TexCoord4** sets all four texture coordinates.

The current normal is set using

```
void Normal3{bsifd}( T coords );
void Normal3{bsifd}v( T coords );
```

Byte, short, or integer values passed to **Normal** are converted to floating-point values as indicated for the corresponding (signed) type in Table 2.6.

Finally, there are several ways to set the current color. The GL stores both a current single-valued *color index*, and a current four-valued RGBA color. One or the other of these is significant depending as the GL is in *color index mode* or *RGBA mode*. The mode selection is made when the GL is initialized.

The command to set RGBA colors is

```
void Color{34}{bsifd ubusui}( T components );
void Color{34}{bsifd ubusui}v( T components );
```

The **Color** command has two major variants: **Color3** and **Color4**. The four value versions set all four values. The three value versions set R, G, and B to the provided values; A is set to 1.0. (The conversion of integer color components (R, G, B, and A) to floating-point values is discussed in section 2.13.)

APPENDIX T

Versions of the **Color** command that take floating-point values accept values nominally between 0.0 and 1.0. 0.0 corresponds to the minimum while 1.0 corresponds to the maximum (machine dependent) value that a component may take on in the framebuffer (see section 2.13 on colors and coloring). Values outside $[0, 1]$ are not clamped.

The command

```
void Index{sifd ub}( T index );
void Index{sifd ub}v( T index );
```

updates the current (single-valued) color index. It takes one argument, the value to which the current color index should be set. Values outside the (machine-dependent) representable range of color indices are not clamped.

The state required to support vertex specification consists of four floating-point numbers to store the current texture coordinates $s$, $t$, $r$, and $q$, three floating-point numbers to store the three coordinates of the current normal, four floating-point values to store the current RGBA color, and one floating-point value to store the current color index. There is no notion of a current vertex, so no state is devoted to vertex coordinates. The initial values of $s$, $t$, and $r$ of the current texture coordinates are zero; the initial value of $q$ is one. The initial current normal has coordinates $(0, 0, 1)$. The initial RGBA color is $(\mathrm{R}, \mathrm{G}, \mathrm{B}, \mathrm{A}) = (1, 1, 1, 1)$. The initial color index is 1.

## 2.8 Vertex Arrays

The vertex specification commands described in section 2.7 accept data in almost any format, but their use requires many command executions to specify even simple geometry. Vertex data may also be placed into arrays that are stored in the client's address space. Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command. The client may specify up to six arrays: one each to store edge flags, texture coordinates, colors, color indices, normals, and vertices. The commands

```
void EdgeFlagPointer( sizei stride, void *pointer );

void TexCoordPointer( int size, enum type, sizei stride,
   void *pointer );

void ColorPointer( int size, enum type, sizei stride,
   void *pointer );
```

APPENDIX T

| Command | Sizes | Types |
|---|---|---|
| **VertexPointer** | 2,3,4 | `short`, `int`, `float`, `double` |
| **NormalPointer** | 3 | `byte`, `short`, `int`, `float`, `double` |
| **ColorPointer** | 3,4 | `byte`, `ubyte`, `short`, `ushort`, `int`, `uint`, `float`, `double` |
| **IndexPointer** | 1 | `ubyte`, `short`, `int`, `float`, `double` |
| **TexCoordPointer** | 1,2,3,4 | `short`, `int`, `float`, `double` |
| **EdgeFlagPointer** | 1 | `boolean` |

Table 2.4: Vertex array sizes (values per vertex) and data types.

```
void IndexPointer( enum type, sizei stride,
   void *pointer );

void NormalPointer( enum type, sizei stride,
   void *pointer );

void VertexPointer( int size, enum type, sizei stride,
   void *pointer );
```

describe the locations and organizations of these arrays. For each command, *type* specifies the data type of the values stored in the array. Because edge flags are always type `boolean`, **EdgeFlagPointer** has no *type* argument. *size*, when present, indicates the number of values per vertex that are stored in the array. Because normals are always specified with three values, **NormalPointer** has no *size* argument. Likewise, because color indices and edge flags are always specified with a single value, **IndexPointer** and **EdgeFlagPointer** also have no *size* argument. Table 2.4 indicates the allowable values for *size* and *type* (when present). For *type* the values `BYTE`, `SHORT`, `INT`, `FLOAT`, and `DOUBLE` indicate types `byte`, `short`, `int`, `float`, and `double`, respectively; and the values `UNSIGNED_BYTE`, `UNSIGNED_SHORT`, and `UNSIGNED_INT` indicate types `ubyte`, `ushort`, and `uint`, respectively. The error `INVALID_VALUE` is generated if *size* is specified with a value other than that indicated in the table.

The one, two, three, or four values in an array that correspond to a single vertex comprise an array *element*. The values within each array element are stored sequentially in memory. If *stride* is specified as zero, then array elements are stored sequentially as well. Otherwise pointers to the $i$th and $(i + 1)$st elements of an array differ by *stride* basic machine units (typically unsigned bytes), the pointer to the $(i+1)$st element being greater. For each command, *pointer* specifies the location in memory of the first value of the first element of the array being specified.

An individual array is enabled or disabled by calling one of

```
void EnableClientState( enum array );
void DisableClientState( enum array );
```

with *array* set to `EDGE_FLAG_ARRAY`, `TEXTURE_COORD_ARRAY`, `COLOR_ARRAY`, `INDEX_ARRAY`, `NORMAL_ARRAY`, or `VERTEX_ARRAY`, for the edge flag, texture coordinate, color, color index, normal, or vertex array, respectively.

The $i$th element of every enabled array is transferred to the GL by calling

```
void ArrayElement( int i );
```

For each enabled array, it is as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element $i$. For the vertex array, the corresponding command is **Vertex**[*size*][*type*]**v**, where *size* is one of [2,3,4], and *type* is one of [s,i,f,d], corresponding to array types `short`, `int`, `float`, and `double` respectively. The corresponding commands for the edge flag, texture coordinate, color, color index, and normal arrays are **EdgeFlagv**, **TexCoord**[*size*][*type*]**v**, **Color**[*size*][*type*]**v**, **Index**[*type*]**v**, and **Normal**[*type*]**v**, respectively. If the vertex array is enabled, it is as though **Vertex**[*size*][*type*]**v** is executed last, after the executions of the other corresponding commands.

Changes made to array data between the execution of **Begin** and the corresponding execution of **End** may affect calls to **ArrayElement** that are made within the same **Begin**/**End** period in non-sequential ways. That is, a call to **ArrayElement** that precedes a change to array data may access the changed data, and a call that follows a change to array data may access original data.

The command

```
void DrawArrays( enum mode, int first, sizei count );
```

constructs a sequence of geometric primitives using elements $first$ through $first+count-1$ of each enabled array. *mode* specifies what kind of primitives are constructed; it accepts the same token values as the *mode* parameter of the **Begin** command. The effect of

**DrawArrays** $(mode, first, count)$;

**APPENDIX T**

is the same as the effect of the command sequence

```
if (mode or count is invalid )
   generate appropriate error
else {
   int i;
   Begin(mode);
   for (i=0; i < count ; i++)
      ArrayElement(first+ i);
   End();
}
```

with one exception: the current edge flag, texture coordinates, color, color index, and normal coordinates are each indeterminate after the execution of **DrawArrays**, if the corresponding array is enabled. Current values corresponding to disabled arrays are not modified by the execution of **DrawArrays**.

The command

```
void DrawElements( enum mode, sizei count, enum type,
   void *indices );
```

constructs a sequence of geometric primitives using the *count* elements whose indices are stored in *indices*. *type* must be one of `UNSIGNED_BYTE`, `UNSIGNED_SHORT`, or `UNSIGNED_INT`, indicating that the values in *indices* are indices of GL type `ubyte`, `ushort`, or `uint` respectively. *mode* specifies what kind of primitives are constructed; it accepts the same token values as the *mode* parameter of the **Begin** command. The effect of

```
DrawElements (mode, count, type, indices);
```

is the same as the effect of the command sequence

```
if (mode, count, or type is invalid )
   generate appropriate error
else {
   int i;
   Begin(mode);
   for (i=0; i < count ; i++)
      ArrayElement(indices[i]);
   End();
}
```

with one exception: the current edge flag, texture coordinates, color, color index, and normal coordinates are each indeterminate after the execution of **DrawElements**, if the corresponding array is enabled. Current values corresponding to disabled arrays are not modified by the execution of **DrawElements**.

The command

> `void` **DrawRangeElements**( `enum` *mode*, `uint` *start*,
> `uint` *end*, `sizei` *count*, `enum` *type*, `void` **indices* );

is a restricted form of **DrawElements**. *mode*, *count*, *type*, and *indices* match the corresponding arguments to **DrawElements**, with the additional constraint that all values in the array *indices* must lie between *start* and *end* inclusive.

Implementations denote recommended maximum amounts of vertex and index data, which may be queried by calling **GetIntegerv** with the symbolic constants `MAX_ELEMENTS_VERTICES` and `MAX_ELEMENTS_INDICES`. If $end - start + 1$ is greater than the value of `MAX_ELEMENTS_VERTICES`, or if *count* is greater than the value of `MAX_ELEMENTS_INDICES`, then the call may operate at reduced performance. There is no requirement that all vertices in the range $[start, end]$ be referenced. However, the implementation may partially process unused vertices, reducing performance from what could be achieved with an optimal index set.

The error `INVALID_VALUE` is generated if $end < start$. Invalid *mode*, *count*, or *type* parameters generate the same errors as would the corresponding call to **DrawElements**. It is an error for indices to lie outside the range $[start, end]$, but implementations may not check for this. Such indices will cause implementation-dependent behavior.

The command

> `void` **InterleavedArrays**( `enum` *format*, `sizei` *stride*,
> `void` **pointer* );

efficiently initializes the six arrays and their enables to one of 14 configurations. *format* must be one of 14 symbolic constants: `V2F`, `V3F`, `C4UB_V2F`, `C4UB_V3F`, `C3F_V3F`, `N3F_V3F`, `C4F_N3F_V3F`, `T2F_V3F`, `T4F_V4F`, `T2F_C4UB_V3F`, `T2F_C3F_V3F`, `T2F_N3F_V3F`, `T2F_C4F_N3F_V3F`, or `T4F_C4F_N3F_V4F`.

The effect of

> **InterleavedArrays**(*format*, *stride*, *pointer*);

is the same as the effect of the command sequence

APPENDIX T

| *format* | $e_t$ | $e_c$ | $e_n$ | $s_t$ | $s_c$ | $s_v$ | $t_c$ |
|---|---|---|---|---|---|---|---|
| V2F | *False* | *False* | *False* | | | 2 | |
| V3F | *False* | *False* | *False* | | | 3 | |
| C4UB_V2F | *False* | *True* | *False* | | 4 | 2 | UNSIGNED_BYTE |
| C4UB_V3F | *False* | *True* | *False* | | 4 | 3 | UNSIGNED_BYTE |
| C3F_V3F | *False* | *True* | *False* | | 3 | 3 | FLOAT |
| N3F_V3F | *False* | *False* | *True* | | | 3 | |
| C4F_N3F_V3F | *False* | *True* | *True* | | 4 | 3 | FLOAT |
| T2F_V3F | *True* | *False* | *False* | 2 | | 3 | |
| T4F_V4F | *True* | *False* | *False* | 4 | | 4 | |
| T2F_C4UB_V3F | *True* | *True* | *False* | 2 | 4 | 3 | UNSIGNED_BYTE |
| T2F_C3F_V3F | *True* | *True* | *False* | 2 | 3 | 3 | FLOAT |
| T2F_N3F_V3F | *True* | *False* | *True* | 2 | | 3 | |
| T2F_C4F_N3F_V3F | *True* | *True* | *True* | 2 | 4 | 3 | FLOAT |
| T4F_C4F_N3F_V4F | *True* | *True* | *True* | 4 | 4 | 4 | FLOAT |

| *format* | $p_c$ | $p_n$ | $p_v$ | $s$ |
|---|---|---|---|---|
| V2F | | | $0$ | $2f$ |
| V3F | | | $0$ | $3f$ |
| C4UB_V2F | $0$ | | $c$ | $c + 2f$ |
| C4UB_V3F | $0$ | | $c$ | $c + 3f$ |
| C3F_V3F | $0$ | | $3f$ | $6f$ |
| N3F_V3F | | $0$ | $3f$ | $6f$ |
| C4F_N3F_V3F | $0$ | $4f$ | $7f$ | $10f$ |
| T2F_V3F | | | $2f$ | $5f$ |
| T4F_V4F | | | $4f$ | $8f$ |
| T2F_C4UB_V3F | $2f$ | | $c + 2f$ | $c + 5f$ |
| T2F_C3F_V3F | $2f$ | | $5f$ | $8f$ |
| T2F_N3F_V3F | | $2f$ | $5f$ | $8f$ |
| T2F_C4F_N3F_V3F | $2f$ | $6f$ | $9f$ | $12f$ |
| T4F_C4F_N3F_V4F | $4f$ | $8f$ | $11f$ | $15f$ |

Table 2.5: Variables that direct the execution of **InterleavedArrays**. $f$ is sizeof(FLOAT). $c$ is 4 times sizeof(UNSIGNED_BYTE), rounded up to the nearest multiple of $f$. All pointer arithmetic is performed in units of sizeof(UNSIGNED_BYTE).

APPENDIX T

```
if ($format$ or $stride$ is invalid)
   generate appropriate error
else {
   int str;
   set $e_t, e_c, e_n, s_t, s_c, s_v, t_c, p_c, p_n, p_v$, and $s$ as a function
      of Table 2.5 and the value of $format$.
   str = $stride$;
   if (str is zero)
      str = $s$;
   DisableClientState(EDGE_FLAG_ARRAY);
   DisableClientState(INDEX_ARRAY);
   if ($e_t$) {
      EnableClientState(TEXTURE_COORD_ARRAY);
      TexCoordPointer($s_t$, FLOAT, str, $pointer$);
   } else {
      DisableClientState(TEXTURE_COORD_ARRAY);
   }
   if ($e_c$) {
      EnableClientState(COLOR_ARRAY);
      ColorPointer($s_c$, $t_c$, str, $pointer + p_c$);
   } else {
      DisableClientState(COLOR_ARRAY);
   }
   if ($e_n$) {
      EnableClientState(NORMAL_ARRAY);
      NormalPointer(FLOAT, str, $pointer + p_n$);
   } else {
      DisableClientState(NORMAL_ARRAY);
   }
   EnableClientState(VERTEX_ARRAY);
   VertexPointer($s_v$, FLOAT, str, $pointer + p_v$);
}
```

The client state required to implement vertex arrays consists of six boolean values, six memory pointers, six integer stride values, five symbolic constants representing array types, and three integers representing values per element. In the initial state the boolean values are each disabled, the memory pointers are each null, the strides are each zero, the array types are each FLOAT, and the integers representing values per element are each four.

**APPENDIX T**

## 2.9 Rectangles

There is a set of GL commands to support efficient specification of rectangles as two corner vertices.

```
void Rect{sifd}( T x1, T y1, T x2, T y2 );
void Rect{sifd}v( T v1[2], T v2[2] );
```

Each command takes either four arguments organized as two consecutive pairs of $(x, y)$ coordinates, or two pointers to arrays each of which contains an $x$ value followed by a $y$ value. The effect of the **Rect** command

**Rect** ($x_1, y_1, x_2, y_2$);

is exactly the same as the following sequence of commands:

**Begin**(POLYGON);
  **Vertex2**($x_1, y_1$);
  **Vertex2**($x_2, y_1$);
  **Vertex2**($x_2, y_2$);
  **Vertex2**($x_1, y_2$);
**End**();

The appropriate **Vertex2** command would be invoked depending on which of the **Rect** commands is issued.

## 2.10 Coordinate Transformations

Vertices, normals, and texture coordinates are transformed before their coordinates are used to produce an image in the framebuffer. We begin with a description of how vertex coordinates are transformed and how this transformation is controlled.

Figure 2.6 diagrams the sequence of transformations that are applied to vertices. The vertex coordinates that are presented to the GL are termed *object coordinates.* The *model-view* matrix is applied to these coordinates to yield *eye* coordinates. Then another matrix, called the *projection* matrix, is applied to eye coordinates to yield *clip* coordinates. A perspective division is carried out on clip coordinates to yield *normalized device* coordinates. A final *viewport* transformation is applied to convert these coordinates into *window coordinates.*

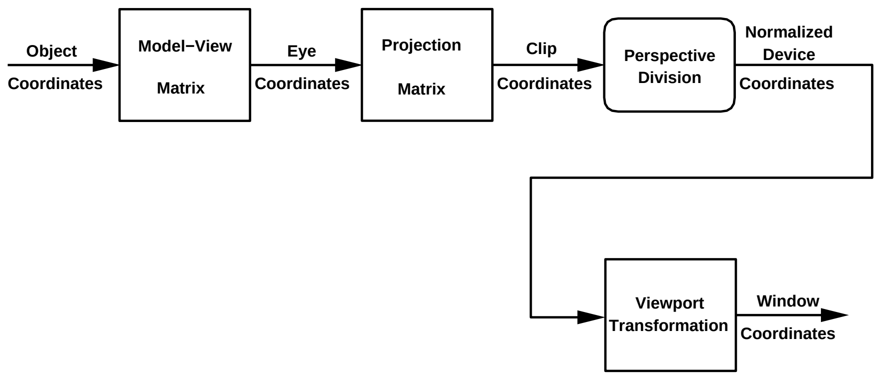


Figure 2.6. Vertex transformation sequence.

Object coordinates, eye coordinates, and clip coordinates are four-dimensional, consisting of $x$, $y$, $z$, and $w$ coordinates (in that order). The model-view and perspective matrices are thus 4 × 4.

If a vertex in object coordinates is given by $\begin{pmatrix} x_o \\ y_o \\ z_o \\ w_o \end{pmatrix}$ and the model-view matrix is $M$, then the vertex's eye coordinates are found as

$$\begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} = M \begin{pmatrix} x_o \\ y_o \\ z_o \\ w_o \end{pmatrix}.$$

Similarly, if $P$ is the projection matrix, then the vertex's clip coordinates are

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = P \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}.$$

The vertex's normalized device coordinates are then

$$\begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \end{pmatrix}.$$

### 2.10.1 Controlling the Viewport

The viewport transformation is determined by the viewport's width and height in pixels, $p_x$ and $p_y$, respectively, and its center $(o_x, o_y)$ (also in pixels). The vertex's window coordinates, $\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix}$, are given by

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} (p_x/2)x_d + o_x \\ (p_y/2)y_d + o_y \\ [(f-n)/2]z_d + (n+f)/2 \end{pmatrix}.$$

The factor and offset applied to $z_d$ encoded by $n$ and $f$ are set using

void **DepthRange**( clampd *n*, clampd *f* );

Each of $n$ and $f$ are clamped to lie within $[0, 1]$, as are all arguments of type `clampd` or `clampf`. $z_w$ is taken to be represented in fixed-point with at least as many bits as there are in the depth buffer of the framebuffer. We assume that the fixed-point representation used represents each value $k/(2^m - 1)$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as $k$ (e.g. 1.0 is represented in binary as a string of all ones).

Viewport transformation parameters are specified using

void **Viewport**( int *x*, int *y*, sizei *w*, sizei *h* );

where $x$ and $y$ give the $x$ and $y$ window coordinates of the viewport's lower-left corner and $w$ and $h$ give the viewport's width and height, respectively. The viewport parameters shown in the above equations are found from these values as $o_x = x + w/2$ and $o_y = y + h/2$; $p_x = w$, $p_y = h$.

Viewport width and height are clamped to implementation-dependent maximums when specified. The maximum width and height may be found by issuing an appropriate **Get** command (see Chapter 6). The maximum viewport dimensions must be greater than or equal to the visible dimensions of the display being rendered to. `INVALID_VALUE` is generated if either $w$ or $h$ is negative.

The state required to implement the viewport transformation is 6 integers. In the initial state, $w$ and $h$ are set to the width and height, respectively, of the window into which the GL is to do its rendering. $o_x$ and $o_y$ are set to $w/2$ and $h/2$, respectively. $n$ and $f$ are set to 0.0 and 1.0, respectively.

APPENDIX T

### 2.10.2 Matrices

The projection matrix and model-view matrix are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

```
void MatrixMode( enum mode );
```

which takes one of the pre-defined constants `TEXTURE`, `MODELVIEW`, `COLOR`, or `PROJECTION` as the argument value. `TEXTURE` is described later in section 2.10.2, and `COLOR`is described in section 3.6.3. If the current matrix mode is `MODELVIEW`, then matrix operations apply to the model-view matrix; if `PROJECTION`, then they apply to the projection matrix.

The two basic commands for affecting the current matrix are

```
void LoadMatrix{fd}( T m[16] );
void MultMatrix{fd}( T m[16] );
```

**LoadMatrix** takes a pointer to a 4 × 4 matrix stored in column-major order as 16 consecutive floating-point values, i.e. as

$$\begin{pmatrix} a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \\ a_4 & a_8 & a_{12} & a_{16} \end{pmatrix}.$$

(This differs from the standard row-major `C` ordering for matrix elements. If the standard ordering is used, all of the subsequent transformation equations are transposed, and the columns representing vectors become rows.)

The specified matrix replaces the current matrix with the one pointed to. **MultMatrix** takes the same type argument as **LoadMatrix**, but multiplies the current matrix by the one pointed to and replaces the current matrix with the product. If $C$ is the current matrix and $M$ is the matrix pointed to by **MultMatrix**'s argument, then the resulting current matrix, $C'$, is

$$C' = C \cdot M.$$

The command

```
void LoadIdentity( void );
```

effectively calls **LoadMatrix** with the identity matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

There are a variety of other commands that manipulate matrices. **Rotate**, **Translate**, **Scale**, **Frustum**, and **Ortho** manipulate the current matrix. Each computes a matrix and then invokes **MultMatrix** with this matrix. In the case of

void **Rotate{fd}**( T $\theta$, T $x$, T $y$, T $z$ );

$\theta$ gives an angle of rotation in degrees; the coordinates of a vector $\mathbf{v}$ are given by $\mathbf{v} = (x\ y\ z)^T$. The computed matrix is a counter-clockwise rotation about the line through the origin with the specified axis when that axis is pointing up (i.e. the right-hand rule determines the sense of the rotation angle). The matrix is thus

$$\begin{pmatrix} & & & 0 \\ & R & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Let $\mathbf{u} = \mathbf{v}/||\mathbf{v}|| = (\,x' \quad y' \quad z'\,)^T$. If

$$S = \begin{pmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{pmatrix}$$

then

$$R = \mathbf{u}\mathbf{u}^T + \cos\theta(I - \mathbf{u}\mathbf{u}^T) + \sin\theta S.$$

The arguments to

void **Translate{fd}**( T $x$, T $y$, T $z$ );

give the coordinates of a translation vector as $(x\ y\ z)^T$. The resulting matrix is a translation by the specified vector:

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

```
void Scale{fd}( T x, T y, T z );
```

produces a general scaling along the $x$-, $y$-, and $z$- axes. The corresponding matrix is

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

For

```
void Frustum( double l, double r, double b, double t,
   double n, double f );
```

the coordinates $(l \ b \ -n)^T$ and $(r \ t \ -n)^T$ specify the points on the near clipping plane that are mapped to the lower-left and upper-right corners of the window, respectively (assuming that the eye is located at $(0\ 0\ 0)^T$). $f$ gives the distance from the eye to the far clipping plane. If either $n$ or $f$ is less than or equal to zero, $l$ is equal to $r$, $b$ is equal to $t$, or $n$ is equal to $f$, the error `INVALID_VALUE` results. The corresponding matrix is

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

```
void Ortho( double l, double r, double b, double t,
   double n, double f );
```

describes a matrix that produces parallel projection. $(l \ b \ -n)^T$ and $(r \ t \ -n)^T$ specify the points on the near clipping plane that are mapped to the lower-left and upper-right corners of the window, respectively. $f$ gives the distance from the eye to the far clipping plane. If $l$ is equal to $r$, $b$ is equal to $t$, or $n$ is equal to $f$, the error `INVALID_VALUE` results. The corresponding matrix is

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

There is another $4 \times 4$ matrix that is applied to texture coordinates. This matrix is applied as

$$\begin{pmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{pmatrix} \begin{pmatrix} s \\ t \\ r \\ q \end{pmatrix},$$

where the left matrix is the current texture matrix. The matrix is applied to the coordinates resulting from texture coordinate generation (which may simply be the current texture coordinates), and the resulting transformed coordinates become the texture coordinates associated with a vertex. Setting the matrix mode to `TEXTURE` causes the already described matrix operations to apply to the texture matrix.

There is a stack of matrices for each of the matrix modes. For `MODELVIEW` mode, the stack depth is at least 32 (that is, there is a stack of at least 32 model-view matrices). For the other modes, the depth is at least 2. The current matrix in any mode is the matrix on the top of the stack for that mode.

void **PushMatrix**( void );

pushes the stack down by one, duplicating the current matrix in both the top of the stack and the entry below it.

void **PopMatrix**( void );

pops the top entry off of the stack, replacing the current matrix with the matrix that was the second entry in the stack. The pushing or popping takes place on the stack corresponding to the current matrix mode. Popping a matrix off a stack with only one entry generates the error `STACK_UNDERFLOW`; pushing a matrix onto a full stack generates `STACK_OVERFLOW`.

The state required to implement transformations consists of a four-valued integer indicating the current matrix mode, a stack of at least two 4 × 4 matrices for each of `COLOR`, `PROJECTION`, and `TEXTURE` with associated stack pointers, and a stack of at least 32 4 × 4 matrices with an associated stack pointer for `MODELVIEW`. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial matrix mode is `MODELVIEW`.

### 2.10.3 Normal Transformation

Finally, we consider how the model-view matrix and transformation state affect normals. Before use in lighting, normals are transformed to eye coordinates by a matrix derived from the model-view matrix. Rescaling and normalization operations are performed on the transformed normals to make

APPENDIX T

them unit length prior to use in lighting. Rescaling and normalization are controlled by

```
void Enable( enum target );
```

and

```
void Disable( enum target );
```

with *target* equal to RESCALE_NORMAL or NORMALIZE. This requires two bits of state. The initial state is for normals not to be rescaled or normalized.

If the model-view matrix is $M$, then the normal is transformed to eye coordinates by:

$$( n_x{}' \quad n_y{}' \quad n_z{}' \quad q' ) = ( n_x \quad n_y \quad n_z \quad q ) \cdot M^{-1}$$

where, if $\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$ are the associated vertex coordinates, then

$$q = \begin{cases} 0, & w = 0, \\ \dfrac{-( n_x \quad n_y \quad n_z ) \begin{pmatrix} x \\ y \\ z \end{pmatrix}}{w}, & w \neq 0 \end{cases} \tag{2.1}$$

Implementations may choose instead to transform $( n_x \quad n_y \quad n_z )$ to eye coordinates using

$$( n_x{}' \quad n_y{}' \quad n_z{}' ) = ( n_x \quad n_y \quad n_z ) \cdot M_u{}^{-1}$$

where $M_u$ is the upper leftmost 3x3 matrix taken from $M$.

Rescale multiplies the transformed normals by a scale factor

$$( n_x{}'' \quad n_y{}'' \quad n_z{}'' ) = f \, ( n_x{}' \quad n_y{}' \quad n_z{}' )$$

If rescaling is disabled, then $f = 1$. If rescaling is enabled, then $f$ is computed as ($m_{ij}$ denotes the matrix element in row $i$ and column $j$ of $M^{-1}$, numbering the topmost row of the matrix as row 1 and the leftmost column as column 1)

$$f = \frac{1}{\sqrt{m_{31}{}^2 + m_{32}{}^2 + m_{33}{}^2}}$$

Note that if the normals sent to GL were unit length and the model-view matrix uniformly scales space, then rescale makes the transformed normals unit length.

Alternatively, an implementation may chose f as

$$f = \frac{1}{\sqrt{{n_x}'^2 + {n_y}'^2 + {n_z}'^2}}$$

recomputing $f$ for each normal. This makes all non-zero length normals unit length regardless of their input length and the nature of the model-view matrix.

After rescaling, the final transformed normal used in lighting, $n_f$, is computed as

$$n_f = m \, ( \, {n_x}'' \quad {n_y}'' \quad {n_z}'' \, )$$

If normalization is disabled, then $m = 1$. Otherwise

$$m = \frac{1}{\sqrt{{n_x}''^2 + {n_y}''^2 + {n_z}''^2}}$$

Because we specify neither the floating-point format nor the means for matrix inversion, we cannot specify behavior in the case of a poorly-conditioned (nearly singular) model-view matrix $M$. In case of an exactly singular matrix, the transformed normal is undefined. If the GL implementation determines that the model-view matrix is uninvertible, then the entries in the inverted matrix are arbitrary. In any case, neither normal transformation nor use of the transformed normal may lead to GL interruption or termination.

### 2.10.4 Generating Texture Coordinates

Texture coordinates associated with a vertex may either be taken from the current texture coordinates or generated according to a function dependent on vertex coordinates. The command

```
void TexGen{ifd}( enum coord, enum pname, T param );
void TexGen{ifd}v( enum coord, enum pname, T params );
```

controls texture coordinate generation. *coord* must be one of the constants S, T, R, or Q, indicating that the pertinent coordinate is the $s$, $t$, $r$, or $q$

coordinate, respectively. In the first form of the command, *param* is a symbolic constant specifying a single-valued texture generation parameter; in the second form, *params* is a pointer to an array of values that specify texture generation parameters. *pname* must be one of the three symbolic constants TEXTURE_GEN_MODE, OBJECT_PLANE, or EYE_PLANE. If *pname* is TEXTURE_GEN_MODE, then either *params* points to or *param* is an integer that is one of the symbolic constants OBJECT_LINEAR, EYE_LINEAR, or SPHERE_MAP.

If TEXTURE_GEN_MODE indicates OBJECT_LINEAR, then the generation function for the coordinate indicated by *coord* is

$$g = p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o.$$

$x_o$, $y_o$, $z_o$, and $w_o$ are the object coordinates of the vertex. $p_1, \ldots, p_4$ are specified by calling **TexGen** with *pname* set to OBJECT_PLANE in which case *params* points to an array containing $p_1, \ldots, p_4$. There is a distinct group of plane equation coefficients for each texture coordinate; *coord* indicates the coordinate to which the specified coefficients pertain.

If TEXTURE_GEN_MODE indicates EYE_LINEAR, then the function is

$$g = p'_1 x_e + p'_2 y_e + p'_3 z_e + p'_4 w_e$$

where

$$\begin{pmatrix} p'_1 & p'_2 & p'_3 & p'_4 \end{pmatrix} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 \end{pmatrix} M^{-1}$$

$x_e$, $y_e$, $z_e$, and $w_e$ are the eye coordinates of the vertex. $p_1, \ldots, p_4$ are set by calling **TexGen** with *pname* set to EYE_PLANE in correspondence with setting the coefficients in the OBJECT_PLANE case. $M$ is the model-view matrix in effect when $p_1, \ldots, p_4$ are specified. Computed texture coordinates may be inaccurate or undefined if $M$ is poorly conditioned or singular.

When used with a suitably constructed texture image, calling **TexGen** with TEXTURE_GEN_MODE indicating SPHERE_MAP can simulate the reflected image of a spherical environment on a polygon. SPHERE_MAP texture coordinates are generated as follows. Denote the unit vector pointing from the origin to the vertex (in eye coordinates) by $\mathbf{u}$. Denote the current normal, after transformation to eye coordinates, by $\mathbf{n}'$. Let $\mathbf{r} = \begin{pmatrix} r_x & r_y & r_z \end{pmatrix}^T$, the reflection vector, be given by

$$\mathbf{r} = \mathbf{u} - 2\mathbf{n}'^T \left(\mathbf{n}'\mathbf{u}\right),$$

and let $m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$. Then the value assigned to an $s$ coordinate (the first **TexGen** argument value is S) is $s = r_x/m + \frac{1}{2}$; the value assigned to a $t$ coordinate is $t = r_y/m + \frac{1}{2}$. Calling **TexGen** with a *coord* of either R or Q when *pname* indicates SPHERE_MAP generates the error INVALID_ENUM.

A texture coordinate generation function is enabled or disabled using **Enable** and **Disable** with an argument of TEXTURE_GEN_S, TEXTURE_GEN_T, TEXTURE_GEN_R, or TEXTURE_GEN_Q (each indicates the corresponding texture coordinate). When enabled, the specified texture coordinate is computed according to the current EYE_LINEAR, OBJECT_LINEAR or SPHERE_MAP specification, depending on the current setting of TEXTURE_GEN_MODE for that coordinate. When disabled, subsequent vertices will take the indicated texture coordinate from the current texture coordinates.

The state required for texture coordinate generation comprises a three-valued integer for each coordinate indicating coordinate generation mode, and a bit for each coordinate to indicate whether texture coordinate generation is enabled or disabled. In addition, four coefficients are required for the four coordinates for each of EYE_LINEAR and OBJECT_LINEAR. The initial state has the texture generation function disabled for all texture coordinates. The initial values of $p_i$ for $s$ are all 0 except $p_1$ which is one; for $t$ all the $p_i$ are zero except $p_2$, which is 1. The values of $p_i$ for $r$ and $q$ are all 0. These values of $p_i$ apply for both the EYE_LINEAR and OBJECT_LINEAR versions. Initially all texture generation modes are EYE_LINEAR.

## 2.11 Clipping

Primitives are clipped to the *clip volume.* In clip coordinates, the *view volume* is defined by

$$\begin{array}{c} -w_c \le x_c \le w_c \\ -w_c \le y_c \le w_c \\ -w_c \le z_c \le w_c \end{array} .$$

This view volume may be further restricted by as many as $n$ client-defined clip planes to generate the clip volume. ($n$ is an implementation dependent maximum that must be at least 6.) Each client-defined plane specifies a half-space. The clip volume is the intersection of all such half-spaces with the view volume (if there no client-defined clip planes are enabled, the clip volume is the view volume).

A client-defined clip plane is specified with

void **ClipPlane**( enum *p*, double *eqn[4]* );

The value of the first argument, $p$, is a symbolic constant, CLIP_PLANE$i$, where $i$ is an integer between 0 and $n-1$, indicating one of $n$ client-defined clip planes. *eqn* is an array of four double-precision floating-point values. These are the coefficients of a plane equation in object coordinates: $p_1$, $p_2$, $p_3$, and $p_4$ (in that order). The inverse of the current model-view matrix is applied to these coefficients, at the time they are specified, yielding

$$(\, p_1' \quad p_2' \quad p_3' \quad p_4' \,) = (\, p_1 \quad p_2 \quad p_3 \quad p_4 \,)\, M^{-1}$$

(where $M$ is the current model-view matrix; the resulting plane equation is undefined if $M$ is singular and may be inaccurate if $M$ is poorly-conditioned) to obtain the plane equation coefficients in eye coordinates. All points with eye coordinates $(\, x_e \quad y_e \quad z_e \quad w_e \,)^T$ that satisfy

$$(\, p_1' \quad p_2' \quad p_3' \quad p_4' \,) \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} \geq 0$$

lie in the half-space defined by the plane; points that do not satisfy this condition do not lie in the half-space.

Client-defined clip planes are enabled with the generic **Enable** command and disabled with the **Disable** command. The value of the argument to either command is CLIP_PLANE$i$ where $i$ is an integer between 0 and $n$; specifying a value of $i$ enables or disables the plane equation with index $i$. The constants obey CLIP_PLANE$i$ $=$ CLIP_PLANE0 $+\, i$.

If the primitive under consideration is a point, then clipping passes it unchanged if it lies within the clip volume; otherwise, it is discarded. If the primitive is a line segment, then clipping does nothing to it if it lies entirely within the clip volume and discards it if it lies entirely outside the volume. If part of the line segment lies in the volume and part lies outside, then the line segment is clipped and new vertex coordinates are computed for one or both vertices. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.

This clipping produces a value, $0 \leq t \leq 1$, for each clipped vertex. If the coordinates of a clipped vertex are $\mathbf{P}$ and the original vertices' coordinates are $\mathbf{P}_1$ and $\mathbf{P}_2$, then $t$ is given by

$$\mathbf{P} = t\mathbf{P}_1 + (1-t)\mathbf{P}_2.$$

The value of $t$ is used in color and texture coordinate clipping (section 2.13.8).

APPENDIX T

If the primitive is a polygon, then it is passed if every one of its edges lies entirely inside the clip volume and either clipped or discarded otherwise. Polygon clipping may cause polygon edges to be clipped, but because polygon connectivity must be maintained, these clipped edges are connected by new edges that lie along the clip volume's boundary. Thus, clipping may require the introduction of new vertices into a polygon. Edge flags are associated with these vertices so that edges introduced by clipping are flagged as boundary (edge flag TRUE), and so that original edges of the polygon that become cut off at these vertices retain their original flags.

If it happens that a polygon intersects an edge of the clip volume's boundary, then the clipped polygon must include a point on this boundary edge. This point must lie in the intersection of the boundary edge and the convex hull of the vertices of the original polygon. We impose this requirement because the polygon may not be exactly planar.

A line segment or polygon whose vertices have $w_c$ values of differing signs may generate multiple connected components after clipping. GL implementations are not required to handle this situation. That is, only the portion of the primitive that lies in the region of $w_c > 0$ need be produced by clipping.

Primitives rendered with clip planes must satisfy a complementarity criterion. Suppose a single clip plane with coefficients $(\, p'_1 \quad p'_2 \quad p'_3 \quad p'_4 \,)$ (or a number of similarly specified clip planes) is enabled and a series of primitives are drawn. Next, suppose that the original clip plane is respecified with coefficients $(\, -p'_1 \quad -p'_2 \quad -p'_3 \quad -p'_4 \,)$ (and correspondingly for any other clip planes) and the primitives are drawn again (and the GL is otherwise in the same state). In this case, primitives must not be missing any pixels, nor may any pixels be drawn twice in regions where those primitives are cut by the clip planes.

The state required for clipping is at least 6 sets of plane equations (each consisting of four double-precision floating-point coefficients) and at least 6 corresponding bits indicating which of these client-defined plane equations are enabled. In the initial state, all client-defined plane equation coefficients are zero and all planes are disabled.

## 2.12 Current Raster Position

The *current raster position* is used by commands that directly affect pixels in the framebuffer. These commands, which bypass vertex transformation and primitive assembly, are described in the next chapter. The current raster position, however, shares some of the characteristics of a vertex.

APPENDIX T

The state required for the current raster position consists of three window coordinates $x_w$, $y_w$, and $z_w$, a clip coordinate $w_c$ value, an eye coordinate distance, a valid bit, and associated data consisting of a color and texture coordinates. It is set using one of the **RasterPos** commands:

```
void RasterPos{234}{sifd}( T coords );
void RasterPos{234}{sifd}v( T coords );
```

**RasterPos4** takes four values indicating $x$, $y$, $z$, and $w$. **RasterPos3** (or **RasterPos2**) is analogous, but sets only $x$, $y$, and $z$ with $w$ implicitly set to 1 (or only $x$ and $y$ with $z$ implicitly set to 0 and $w$ implicitly set to 1).

The coordinates are treated as if they were specified in a **Vertex** command. The $x$, $y$, $z$, and $w$ coordinates are transformed by the current model-view and perspective matrices. These coordinates, along with current values, are used to generate a color and texture coordinates just as is done for a vertex. The color and texture coordinates so produced replace the color and texture coordinates stored in the current raster position's associated data. The distance from the origin of the eye coordinate system to the vertex as transformed by only the current model-view matrix replaces the current raster distance. This distance can be approximated (see section 3.10).

The transformed coordinates are passed to clipping as if they represented a point. If the "point" is not culled, then the projection to window coordinates is computed (section 2.10) and saved as the current raster position, and the valid bit is set. If the "point" is culled, the current raster position and its associated data become indeterminate and the valid bit is cleared. Figure 2.7 summarizes the behavior of the current raster position.

The current raster position requires five single-precision floating-point values for its $x_w$, $y_w$, and $z_w$ window coordinates, its $w_c$ clip coordinate, and its eye coordinate distance, a single valid bit, a color (RGBA and color index), and texture coordinates for associated data. In the initial state, the coordinates and texture coordinates are both $(0, 0, 0, 1)$, the eye coordinate distance is 0, the valid bit is set, the associated RGBA color is $(1, 1, 1, 1)$ and the associated color index color is 1. In RGBA mode, the associated color index always has its initial value; in color index mode, the RGBA color always maintains its initial value.

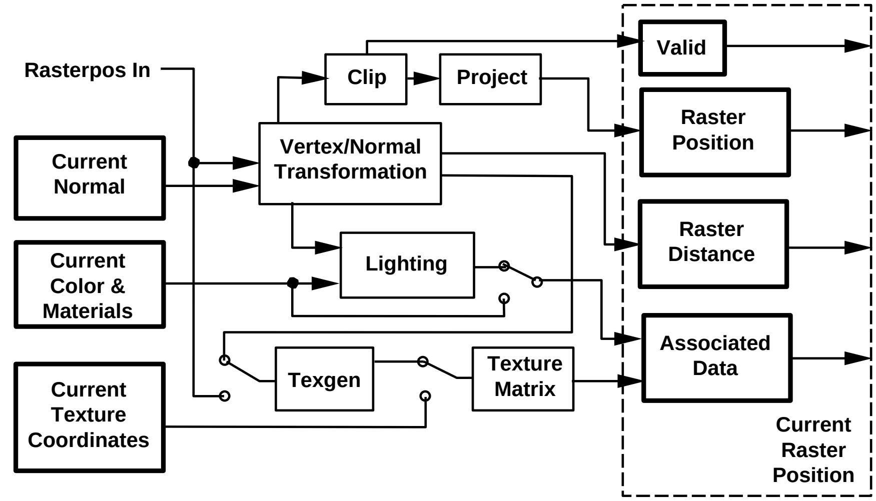


Figure 2.7. The current raster position and how it is set.

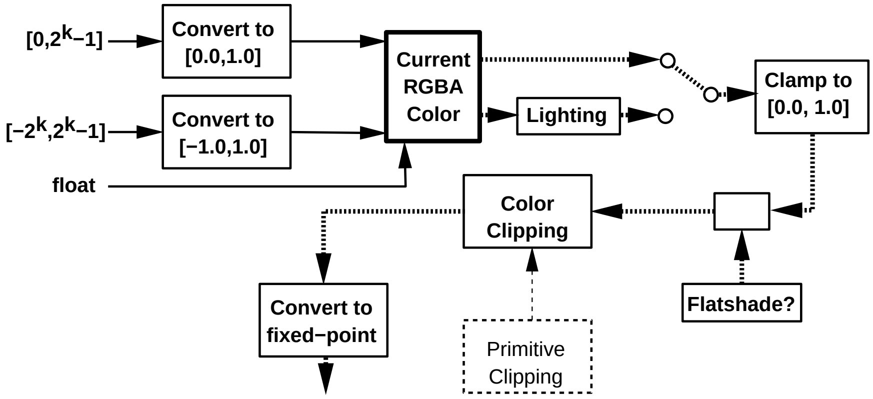


Figure 2.8. Processing of RGBA colors. The heavy dotted lines indicate both primary and secondary vertex colors, which are processed in the same fashion. See Table 2.6 for the interpretation of $k$.

APPENDIX T

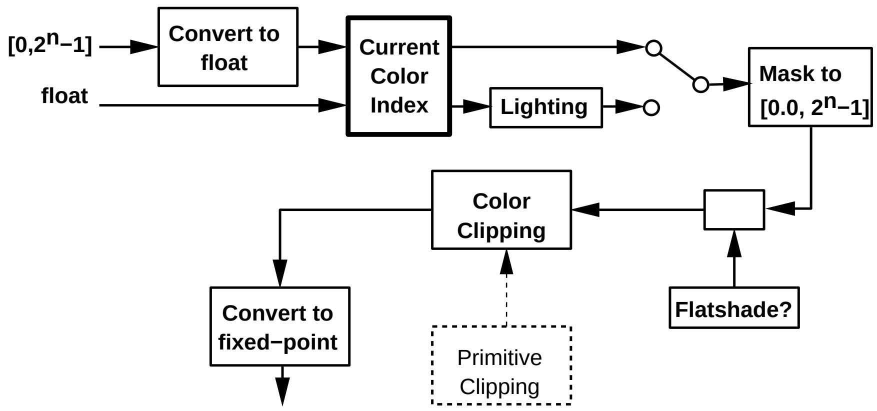


Figure 2.9. Processing of color indices. $n$ is the number of bits in a color index.

## 2.13 Colors and Coloring

Figures 2.8 and 2.9 diagram the processing of RGBA colors and color indices before rasterization. Incoming colors arrive in one of several formats. Table 2.6 summarizes the conversions that take place on R, G, B, and A components depending on which version of the **Color** command was invoked to specify the components. As a result of limited precision, some converted values will not be represented exactly. In color index mode, a single-valued color index is not mapped.

Next, lighting, if enabled, produces either a color index or primary and secondary colors. If lighting is disabled, the current color index or color is used in further processing (the current color is the primary color, and the secondary color is $(0,0,0,0)$). After lighting, RGBA colors are clamped to the range $[0,1]$. A color index is converted to fixed-point and then its integer portion is masked (see section 2.13.6). After clamping or masking, a primitive may be *flatshaded*, indicating that all vertices of the primitive are to have the same color. Finally, if a primitive is clipped, then colors (and texture coordinates) must be computed at the vertices introduced or modified by clipping.

APPENDIX T

| GL Type | Conversion |
|---|---|
| ubyte | $c/(2^8 - 1)$ |
| byte | $(2c + 1)/(2^8 - 1)$ |
| ushort | $c/(2^{16} - 1)$ |
| short | $(2c + 1)/(2^{16} - 1)$ |
| uint | $c/(2^{32} - 1)$ |
| int | $(2c + 1)/(2^{32} - 1)$ |
| float | $c$ |
| double | $c$ |

Table 2.6: Component conversions. Color, normal, and depth components, ($c$), are converted to an internal floating-point representation, ($f$), using the equations in this table. All arithmetic is done in the internal floating point format. These conversions apply to components specified as parameters to GL commands and to components in pixel data. The equations remain the same even if the implemented ranges of the GL data types are greater than the minimum required ranges. (Refer to table 2.2)

### 2.13.1 Lighting

GL lighting computes colors for each vertex sent to the GL. This is accomplished by applying an equation defined by a client-specified lighting model to a collection of parameters that can include the vertex coordinates, the coordinates of one or more light sources, the current normal, and parameters defining the characteristics of the light sources and a current material. The following discussion assumes that the GL is in RGBA mode. (Color index lighting is described in section 2.13.5.)

Lighting may be in one of two states:

1. Lighting Off. In this state, the current color is assigned to the vertex primary color. The secondary color is $(0, 0, 0, 0)$.

2. Lighting On. In this state, the vertex primary and secondary colors are computed from the current lighting parameters.

Lighting is turned on or off using the generic **Enable** or **Disable** commands with the symbolic value `LIGHTING`.

APPENDIX T

### Lighting Operation

A lighting parameter is of one of five types: color, position, direction, real, or boolean. A color parameter consists of four floating-point values, one for each of R, G, B, and A, in that order. There are no restrictions on the allowable values for these parameters. A position parameter consists of four floating-point coordinates ($x$, $y$, $z$, and $w$) that specify a position in object coordinates ($w$ may be zero, indicating a point at infinity in the direction given by $x$, $y$, and $z$). A direction parameter consists of three floating-point coordinates ($x$, $y$, and $z$) that specify a direction in object coordinates. A real parameter is one floating-point value. The various values and their types are summarized in Table 2.7. The result of a lighting computation is undefined if a value for a parameter is specified that is outside the range given for that parameter in the table.

There are $n$ light sources, indexed by $i = 0, \ldots, n-1$. ($n$ is an implementation dependent maximum that must be at least 8.) Note that the default values for $\mathbf{d}_{cli}$ and $\mathbf{s}_{cli}$ differ for $i = 0$ and $i > 0$.

Before specifying the way that lighting computes colors, we introduce operators and notation that simplify the expressions involved. If $\mathbf{c}_1$ and $\mathbf{c}_2$ are colors without alpha where $\mathbf{c}_1 = (r_1, g_1, b_1)$ and $\mathbf{c}_2 = (r_2, g_2, b_2)$, then define $\mathbf{c}_1 * \mathbf{c}_2 = (r_1r_2, g_1g_2, b_1b_2)$. Addition of colors is accomplished by addition of the components. Multiplication of colors by a scalar means multiplying each component by that scalar. If $\mathbf{d}_1$ and $\mathbf{d}_2$ are directions, then define

$$\mathbf{d}_1 \odot \mathbf{d}_2 = \max\{\mathbf{d}_1 \cdot \mathbf{d}_2, 0\}.$$

(Directions are taken to have three coordinates.) If $\mathbf{P}_1$ and $\mathbf{P}_2$ are (homogeneous, with four coordinates) points then let $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$ be the unit vector that points from $\mathbf{P}_1$ to $\mathbf{P}_2$. Note that if $\mathbf{P}_2$ has a zero $w$ coordinate and $\mathbf{P}_1$ has non-zero $w$ coordinate, then $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$ is the unit vector corresponding to the direction specified by the $x$, $y$, and $z$ coordinates of $\mathbf{P}_2$; if $\mathbf{P}_1$ has a zero $w$ coordinate and $\mathbf{P}_2$ has a non-zero $w$ coordinate then $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$ is the unit vector that is the negative of that corresponding to the direction specified by $\mathbf{P}_1$. If both $\mathbf{P}_1$ and $\mathbf{P}_2$ have zero $w$ coordinates, then $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$ is the unit vector obtained by normalizing the direction corresponding to $\mathbf{P}_2 - \mathbf{P}_1$.

If $\mathbf{d}$ is an arbitrary direction, then let $\hat{\mathbf{d}}$ be the unit vector in $\mathbf{d}$'s direction. Let $\|\mathbf{P}_1\mathbf{P}_2\|$ be the distance between $\mathbf{P}_1$ and $\mathbf{P}_2$. Finally, let $\mathbf{V}$ be the point corresponding to the vertex being lit, and $\mathbf{n}$ be the corresponding normal. Let $\mathbf{P}_e$ be the eyepoint ($(0, 0, 0, 1)$ in eye coordinates).

Lighting produces two colors at a vertex: a primary color $\mathbf{c}_{pri}$ and a secondary color $\mathbf{c}_{sec}$. The values of $\mathbf{c}_{pri}$ and $\mathbf{c}_{sec}$ depend on the light model

APPENDIX T

| Parameter | Type | Default Value | Description |
|---|---|---|---|
| Material Parameters | | | |
| $\mathbf{a}_{cm}$ | color | $(0.2, 0.2, 0.2, 1.0)$ | ambient color of material |
| $\mathbf{d}_{cm}$ | color | $(0.8, 0.8, 0.8, 1.0)$ | diffuse color of material |
| $\mathbf{s}_{cm}$ | color | $(0.0, 0.0, 0.0, 1.0)$ | specular color of material |
| $\mathbf{e}_{cm}$ | color | $(0.0, 0.0, 0.0, 1.0)$ | emissive color of material |
| $s_{rm}$ | real | $0.0$ | specular exponent (range: $[0.0, 128.0]$) |
| $a_m$ | real | $0.0$ | ambient color index |
| $d_m$ | real | $1.0$ | diffuse color index |
| $s_m$ | real | $1.0$ | specular color index |
| Light Source Parameters | | | |
| $\mathbf{a}_{cli}$ | color | $(0.0, 0.0, 0.0, 1.0)$ | ambient intensity of light $i$ |
| $\mathbf{d}_{cli}(i = 0)$<br>$\mathbf{d}_{cli}(i > 0)$ | color<br>color | $(1.0, 1.0, 1.0, 1.0)$<br>$(0.0, 0.0, 0.0, 1.0)$ | diffuse intensity of light 0<br>diffuse intensity of light $i$ |
| $\mathbf{s}_{cli}(i = 0)$<br>$\mathbf{s}_{cli}(i > 0)$ | color<br>color | $(1.0, 1.0, 1.0, 1.0)$<br>$(0.0, 0.0, 0.0, 1.0)$ | specular intensity of light 0<br>specular intensity of light $i$ |
| $\mathbf{P}_{pli}$ | position | $(0.0, 0.0, 1.0, 0.0)$ | position of light $i$ |
| $\mathbf{s}_{dli}$ | direction | $(0.0, 0.0, -1.0)$ | direction of spotlight for light $i$ |
| $s_{rli}$ | real | $0.0$ | spotlight exponent for light $i$ (range: $[0.0, 128.0]$) |
| $c_{rli}$ | real | $180.0$ | spotlight cutoff angle for light $i$ (range: $[0.0, 90.0]$, $180.0$) |
| $k_{0i}$ | real | $1.0$ | constant attenuation factor for light i (range: $[0.0, \infty)$) |
| $k_{1i}$ | real | $0.0$ | linear attenuation factor for light i (range: $[0.0, \infty)$) |
| $k_{2i}$ | real | $0.0$ | quadratic attenuation factor for light i (range: $[0.0, \infty)$) |
| Lighting Model Parameters | | | |
| $\mathbf{a}_{cs}$ | color | $(0.2, 0.2, 0.2, 1.0)$ | ambient color of scene |
| $v_{bs}$ | boolean | `FALSE` | viewer assumed to be at $(0, 0, 0)$ in eye coordinates (`TRUE`) or $(0, 0, \infty)$ (`FALSE`) |
| $c_{es}$ | enum | `SINGLE_COLOR` | controls computation of colors |
| $t_{bs}$ | boolean | `FALSE` | use two-sided lighting mode |

Table 2.7: Summary of lighting parameters. The range of individual color components is $(-\infty, +\infty)$.

APPENDIX T

color control, $c_{es}$. If $c_{es} =$ SINGLE_COLOR, then the equations to compute $\mathbf{c}_{pri}$ and $\mathbf{c}_{sec}$ are

$$\begin{aligned}
\mathbf{c}_{pri} &= \mathbf{e}_{cm} \\
&+ \mathbf{a}_{cm} * \mathbf{a}_{cs} \\
&+ \sum_{i=0}^{n-1} (att_i)(spot_i)\,[\mathbf{a}_{cm} * \mathbf{a}_{cli} \\
&\qquad\qquad + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli} \\
&\qquad\qquad + (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}\mathbf{s}_{cm} * \mathbf{s}_{cli}] \\
\mathbf{c}_{sec} &= (0, 0, 0, 0)
\end{aligned}$$

If $c_{es} =$ SEPARATE_SPECULAR_COLOR, then

$$\begin{aligned}
\mathbf{c}_{pri} &= \mathbf{e}_{cm} \\
&+ \mathbf{a}_{cm} * \mathbf{a}_{cs} \\
&+ \sum_{i=0}^{n-1} (att_i)(spot_i)\,[\mathbf{a}_{cm} * \mathbf{a}_{cli} \\
&\qquad\qquad + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli}] \\
\mathbf{c}_{sec} &= \sum_{i=0}^{n-1} (att_i)(spot_i)(f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}\mathbf{s}_{cm} * \mathbf{s}_{cli}
\end{aligned}$$

where

$$f_i = \begin{cases} 1, & \mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli} \neq 0, \\ 0, & \text{otherwise,} \end{cases} \tag{2.2}$$

$$\mathbf{h}_i = \begin{cases} \overrightarrow{\mathbf{VP}}_{pli} + \overrightarrow{\mathbf{VP}}_e, & v_{bs} = \texttt{TRUE}, \\ \overrightarrow{\mathbf{VP}}_{pli} + \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T, & v_{bs} = \texttt{FALSE}, \end{cases} \tag{2.3}$$

$$att_i = \begin{cases} \dfrac{1}{k_{0i} + k_{1i}\|\mathbf{VP}_{pli}\| + k_{2i}\|\mathbf{VP}_{pli}\|^2}, & \text{if } \mathbf{P}_{pli}\text{'s } w \neq 0, \\ 1.0, & \text{otherwise.} \end{cases} \tag{2.4}$$

APPENDIX T

$$spot_i = \begin{cases} (\overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli})^{s_{rli}}, & c_{rli} \neq 180.0, \overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} \geq \cos(c_{rli}), \\ 0.0, & c_{rli} \neq 180.0, \overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} < \cos(c_{rli}), \\ 1.0, & c_{rli} = 180.0. \end{cases} \tag{2.5}$$

$$\tag{2.6}$$

All computations are carried out in eye coordinates.

The value of A produced by lighting is the alpha value associated with $\mathbf{d}_{cm}$. A is always associated with the primary color $\mathbf{c}_{pri}$; the alpha component of $\mathbf{c}_{sec}$ is 0. Results of lighting are undefined if the $w_e$ coordinate ($w$ in eye coordinates) of $\mathbf{V}$ is zero.

Lighting may operate in *two-sided* mode ($t_{bs}$ = `TRUE`), in which a *front* color is computed with one set of material parameters (the *front material*) and a *back* color is computed with a second set of material parameters (the *back material*). This second computation replaces $\mathbf{n}$ with $-\mathbf{n}$. If $t_{bs}$ = `FALSE`, then the back color and front color are both assigned the color computed using the front material with $\mathbf{n}$.

The selection between back color and front color depends on the primitive of which the vertex being lit is a part. If the primitive is a point or a line segment, the front color is always selected. If it is a polygon, then the selection is based on the sign of the (clipped or unclipped) polygon's signed area computed in window coordinates. One way to compute this area is

$$a = \frac{1}{2} \sum_{i=0}^{n-1} x_w^i y_w^{i \oplus 1} - x_w^{i \oplus 1} y_w^i \tag{2.7}$$

where $x_w^i$ and $y_w^i$ are the $x$ and $y$ window coordinates of the $i$th vertex of the $n$-vertex polygon (vertices are numbered starting at zero for purposes of this computation) and $i \oplus 1$ is $(i+1) \bmod n$. The interpretation of the sign of this value is controlled with

```
void FrontFace( enum dir );
```

Setting *dir* to `CCW` (corresponding to counter-clockwise orientation of the projected polygon in window coordinates) indicates that if $a \leq 0$, then the color of each vertex of the polygon becomes the back color computed for that vertex while if $a > 0$, then the front color is selected. If *dir* is `CW`, then $a$ is replaced by $-a$ in the above inequalities. This requires one bit of state; initially, it indicates `CCW`.

### 2.13.2 Lighting Parameter Specification

Lighting parameters are divided into three categories: material parameters, light source parameters, and lighting model parameters (see Table 2.7). Sets of lighting parameters are specified with

```
void Material{if}( enum face, enum pname, T param );
void Material{if}v( enum face, enum pname, T params );
void Light{if}( enum light, enum pname, T param );
void Light{if}v( enum light, enum pname, T params );
void LightModel{if}( enum pname, T param );
void LightModel{if}v( enum pname, T params );
```

*pname* is a symbolic constant indicating which parameter is to be set (see Table 2.8). In the vector versions of the commands, *params* is a pointer to a group of values to which to set the indicated parameter. The number of values pointed to depends on the parameter being set. In the non-vector versions, *param* is a value to which to set a single-valued parameter. (If *param* corresponds to a multi-valued parameter, the error INVALID_ENUM results.) For the **Material** command, *face* must be one of FRONT, BACK, or FRONT_AND_BACK, indicating that the property *name* of the front or back material, or both, respectively, should be set. In the case of **Light**, *light* is a symbolic constant of the form LIGHT$i$, indicating that light $i$ is to have the specified parameter set. The constants obey LIGHT$i$ = LIGHT0 + $i$.

Table 2.8 gives, for each of the three parameter groups, the correspondence between the pre-defined constant names and their names in the lighting equations, along with the number of values that must be specified with each. Color parameters specified with **Material** and **Light** are converted to floating-point values (if specified as integers) as indicated in Table 2.6 for signed integers. The error INVALID_VALUE occurs if a specified lighting parameter lies outside the allowable range given in Table 2.7. (The symbol "$\infty$" indicates the maximum representable magnitude for the indicated type.)

The current model-view matrix is applied to the position parameter indicated with **Light** for a particular light source when that position is specified. These transformed values are the values used in the lighting equation.

The spotlight direction is transformed when it is specified using only the upper leftmost 3x3 portion of the model-view matrix. That is, if $\mathbf{M}_u$ is the upper left 3x3 matrix taken from the current model-view matrix $M$, then

APPENDIX T

| Parameter | Name | Number of values |
|---|---|---|
| Material Parameters (**Material**) | | |
| $\mathbf{a}_{cm}$ | AMBIENT | 4 |
| $\mathbf{d}_{cm}$ | DIFFUSE | 4 |
| $\mathbf{a}_{cm}, \mathbf{d}_{cm}$ | AMBIENT_AND_DIFFUSE | 4 |
| $\mathbf{s}_{cm}$ | SPECULAR | 4 |
| $\mathbf{e}_{cm}$ | EMISSION | 4 |
| $s_{rm}$ | SHININESS | 1 |
| $a_m, d_m, s_m$ | COLOR_INDEXES | 3 |
| Light Source Parameters (**Light**) | | |
| $\mathbf{a}_{cli}$ | AMBIENT | 4 |
| $\mathbf{d}_{cli}$ | DIFFUSE | 4 |
| $\mathbf{s}_{cli}$ | SPECULAR | 4 |
| $\mathbf{P}_{pli}$ | POSITION | 4 |
| $\mathbf{s}_{dli}$ | SPOT_DIRECTION | 3 |
| $s_{rli}$ | SPOT_EXPONENT | 1 |
| $c_{rli}$ | SPOT_CUTOFF | 1 |
| $k_0$ | CONSTANT_ATTENUATION | 1 |
| $k_1$ | LINEAR_ATTENUATION | 1 |
| $k_2$ | QUADRATIC_ATTENUATION | 1 |
| Lighting Model Parameters (**LightModel**) | | |
| $\mathbf{a}_{cs}$ | LIGHT_MODEL_AMBIENT | 4 |
| $v_{bs}$ | LIGHT_MODEL_LOCAL_VIEWER | 1 |
| $t_{bs}$ | LIGHT_MODEL_TWO_SIDE | 1 |
| $c_{es}$ | LIGHT_MODEL_COLOR_CONTROL | 1 |

Table 2.8: Correspondence of lighting parameter symbols to names. AMBIENT_AND_DIFFUSE is used to set $\mathbf{a}_{cm}$ and $\mathbf{d}_{cm}$ to the same value.

the spotlight direction

$$\begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}$$

is transformed to

$$\begin{pmatrix} d'_x \\ d'_y \\ d'_z \end{pmatrix} = M_u \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}.$$

An individual light is enabled or disabled by calling **Enable** or **Disable** with the symbolic value `LIGHT`$i$ ($i$ is in the range 0 to $n-1$, where $n$ is the implementation-dependent number of lights). If light $i$ is disabled, the $i$th term in the lighting equation is effectively removed from the summation.

### 2.13.3 ColorMaterial

It is possible to attach one or more material properties to the current color, so that they continuously track its component values. This behavior is enabled and disabled by calling **Enable** or **Disable** with the symbolic value `COLOR_MATERIAL`.

The command that controls which of these modes is selected is

void **ColorMaterial**( enum *face*, enum *mode* );

*face* is one of `FRONT`, `BACK`, or `FRONT_AND_BACK`, indicating whether the front material, back material, or both are affected by the current color. *mode* is one of `EMISSION`, `AMBIENT`, `DIFFUSE`, `SPECULAR`, or `AMBIENT_AND_DIFFUSE` and specifies which material property or properties track the current color. If *mode* is `EMISSION`, `AMBIENT`, `DIFFUSE`, or `SPECULAR`, then the value of $\mathbf{e}_{cm}$, $\mathbf{a}_{cm}$, $\mathbf{d}_{cm}$ or $\mathbf{s}_{cm}$, respectively, will track the current color. If *mode* is `AMBIENT_AND_DIFFUSE`, both $\mathbf{a}_{cm}$ and $\mathbf{d}_{cm}$ track the current color. The replacements made to material properties are permanent; the replaced values remain until changed by either sending a new color or by setting a new material value when **ColorMaterial** is not currently enabled to override that particular value. When `COLOR_MATERIAL` is enabled, the indicated parameter or parameters always track the current color. For instance, calling

**ColorMaterial**(`FRONT`, `AMBIENT`)

while `COLOR_MATERIAL` is enabled sets the front material $\mathbf{a}_{cm}$ to the value of the current color.

APPENDIX T

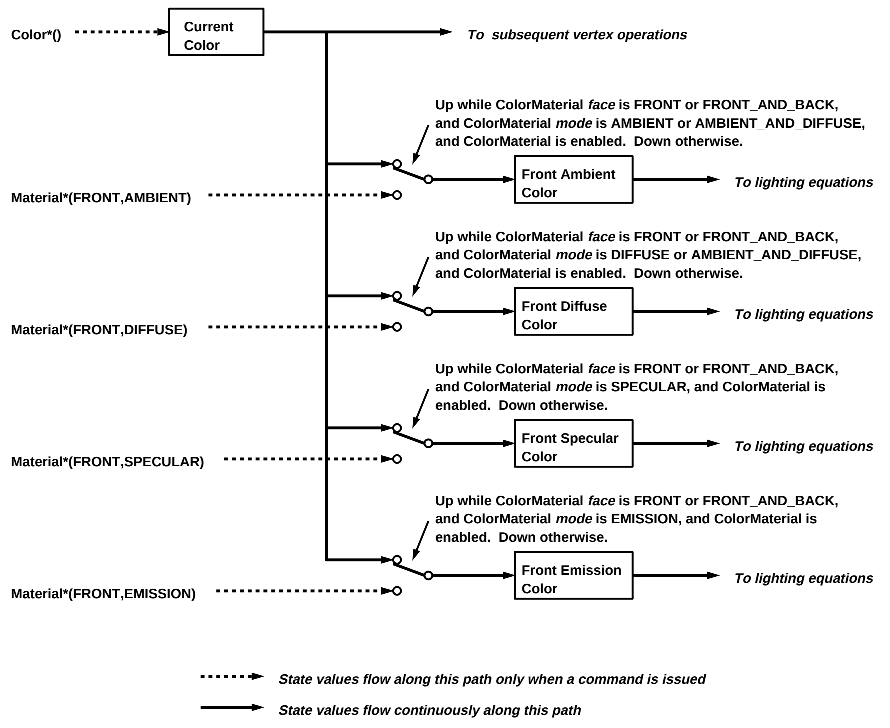

Color*()
Current Color
To subsequent vertex operations
Up while ColorMaterial *face* is FRONT or FRONT_AND_BACK, and ColorMaterial *mode* is AMBIENT or AMBIENT_AND_DIFFUSE, and ColorMaterial is enabled. Down otherwise.
Front Ambient Color
To lighting equations
Material*(FRONT,AMBIENT)
Up while ColorMaterial *face* is FRONT or FRONT_AND_BACK, and ColorMaterial *mode* is DIFFUSE or AMBIENT_AND_DIFFUSE, and ColorMaterial is enabled. Down otherwise.
Front Diffuse Color
To lighting equations
Material*(FRONT,DIFFUSE)
Up while ColorMaterial *face* is FRONT or FRONT_AND_BACK, and ColorMaterial *mode* is SPECULAR, and ColorMaterial is enabled. Down otherwise.
Front Specular Color
To lighting equations
Material*(FRONT,SPECULAR)
Up while ColorMaterial *face* is FRONT or FRONT_AND_BACK, and ColorMaterial *mode* is EMISSION, and ColorMaterial is enabled. Down otherwise.
Front Emission Color
To lighting equations
Material*(FRONT,EMISSION)
State values flow along this path only when a command is issued
State values flow continuously along this path

Figure 2.10. **ColorMaterial** operation. Material properties are continuously updated from the current color while **ColorMaterial** is enabled and has the appropriate mode. Only the front material properties are included in this figure. The back material properties are treated identically, except that *face* must be BACK or FRONT_AND_BACK.

APPENDIX T

### 2.13.4 Lighting State

The state required for lighting consists of all of the lighting parameters (front and back material parameters, lighting model parameters, and at least 8 sets of light parameters), a bit indicating whether a back color distinct from the front color should be computed, at least 8 bits to indicate which lights are enabled, a five-valued variable indicating the current **ColorMaterial** mode, a bit indicating whether or not COLOR_MATERIAL is enabled, and a single bit to indicate whether lighting is enabled or disabled. In the initial state, all lighting parameters have their default values. Back color evaluation does not take place, **ColorMaterial** is FRONT_AND_BACK and AMBIENT_AND_DIFFUSE, and both lighting and COLOR_MATERIAL are disabled.

### 2.13.5 Color Index Lighting

A simplified lighting computation applies in color index mode that uses many of the parameters controlling RGBA lighting, but none of the RGBA material parameters. First, the RGBA diffuse and specular intensities of light $i$ ($\mathbf{d}_{cli}$ and $\mathbf{s}_{cli}$, respectively) determine color index diffuse and specular light intensities, $d_{li}$ and $s_{li}$ from

$$d_{li} = (.30)R(\mathbf{d}_{cli}) + (.59)G(\mathbf{d}_{cli}) + (.11)B(\mathbf{d}_{cli})$$

and

$$s_{li} = (.30)R(\mathbf{s}_{cli}) + (.59)G(\mathbf{s}_{cli}) + (.11)B(\mathbf{s}_{cli}).$$

$R(\mathbf{x})$ indicates the R component of the color $\mathbf{x}$ and similarly for $G(\mathbf{x})$ and $B(\mathbf{x})$.

Next, let

$$s = \sum_{i=0}^{n} (att_i)(spot_i)(s_{li})(f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}$$

where $att_i$ and $spot_i$ are given by equations 2.4 and 2.5, respectively, and $f_i$ and $\hat{\mathbf{h}}_i$ are given by equations 2.2 and 2.3, respectively. Let $s' = \min\{s, 1\}$. Finally, let

$$d = \sum_{i=0}^{n} (att_i)(spot_i)(d_{li})(\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli}).$$

Then color index lighting produces a value $c$, given by

$$c = a_m + d(1 - s')(d_m - a_m) + s'(s_m - a_m).$$

The final color index is

$$c' = \min\{c, s_m\}.$$

APPENDIX T

The values $a_m$, $d_m$ and $s_m$ are material properties described in Tables 2.7 and 2.8. Any ambient light intensities are incorporated into $a_m$. As with RGBA lighting, disabled lights cause the corresponding terms from the summations to be omitted. The interpretation of $t_{bs}$ and the calculation of front and back colors is carried out as has already been described for RGBA lighting.

The values $a_m$, $d_m$, and $s_m$ are set with **Material** using a *pname* of `COLOR_INDEXES`. Their initial values are 0, 1, and 1, respectively. The additional state consists of three floating-point values. These values have no effect on RGBA lighting.

### 2.13.6 Clamping or Masking

After lighting (whether enabled or not), all components of both primary and secondary colors are clamped to the range $[0, 1]$.

For a color index, the index is first converted to fixed-point with an unspecified number of bits to the right of the binary point; the nearest fixed-point value is selected. Then, the bits to the right of the binary point are left alone while the integer portion is masked (bitwise ANDed) with $2^n - 1$, where $n$ is the number of bits in a color in the color index buffer (buffers are discussed in chapter 4).

### 2.13.7 Flatshading

A primitive may be *flatshaded*, meaning that all vertices of the primitive are assigned the same color index or the same primary and secondary colors. These colors are the colors of the vertex that spawned the primitive. For a point, these are the colors associated with the point. For a line segment, they are the colors of the second (final) vertex of the segment. For a polygon, they come from a selected vertex depending on how the polygon was generated. Table 2.9 summarizes the possibilities.

Flatshading is controlled by

```
void ShadeModel( enum mode );
```

*mode* value must be either of the symbolic constants `SMOOTH` or `FLAT`. If *mode* is `SMOOTH` (the initial state), vertex colors are treated individually. If *mode* is `FLAT`, flatshading is turned on. **ShadeModel** thus requires one bit of state.

APPENDIX T

| Primitive type of polygon $i$ | Vertex |
|---|---|
| single polygon ($i \equiv 1$) | 1 |
| triangle strip | $i + 2$ |
| triangle fan | $i + 2$ |
| independent triangle | $3i$ |
| quad strip | $2i + 2$ |
| independent quad | $4i$ |

Table 2.9: Polygon flatshading color selection. The colors used for flatshading the $i$th polygon generated by the indicated **Begin**/**End** type are derived from the current color (if lighting is disabled) in effect when the indicated vertex is specified. If lighting is enabled, the colors are produced by lighting the indicated vertex. Vertices are numbered 1 through $n$, where $n$ is the number of vertices between the **Begin**/**End** pair.

### 2.13.8 Color and Texture Coordinate Clipping

After lighting, clamping or masking and possible flatshading, colors are clipped. Those colors associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the colors assigned to vertices produced by clipping are clipped colors.

Let the colors assigned to the two vertices $\mathbf{P}_1$ and $\mathbf{P}_2$ of an unclipped edge be $\mathbf{c}_1$ and $\mathbf{c}_2$. The value of $t$ (section 2.11) for a clipped point $\mathbf{P}$ is used to obtain the color associated with $\mathbf{P}$ as

$$\mathbf{c} = t\mathbf{c}_1 + (1 - t)\mathbf{c}_2.$$

(For a color index color, multiplying a color by a scalar means multiplying the index by the scalar. For an RGBA color, it means multiplying each of R, G, B, and A by the scalar. Both primary and secondary colors are treated in the same fashion.) Polygon clipping may create a clipped vertex along an edge of the clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one plane of the clip volume's boundary at a time. Color clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

Texture coordinates must also be clipped when a primitive is clipped. The method is exactly analogous to that used for color clipping.

Microsoft et al. Exhibit 1005

APPENDIX T

### 2.13.9 Final Color Processing

For an RGBA color, each color component (which lies in $[0, 1]$) is converted (by rounding to nearest) to a fixed-point value with $m$ bits. We assume that the fixed-point representation used represents each value $k/(2^m - 1)$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as $k$ (e.g. 1.0 is represented in binary as a string of all ones). $m$ must be at least as large as the number of bits in the corresponding component of the framebuffer. $m$ must be at least 2 for A if the framebuffer does not contain an A component, or if there is only 1 bit of A in the framebuffer. A color index is converted (by rounding to nearest) to a fixed-point value with at least as many bits as there are in the color index portion of the framebuffer.

Because a number of the form $k/(2^m - 1)$ may not be represented exactly as a limited-precision floating-point quantity, we place a further requirement on the fixed-point conversion of RGBA components. Suppose that lighting is disabled, the color associated with a vertex has not been clipped, and one of **Colorub**, **Colorus**, or **Colorui** was used to specify that color. When these conditions are satisfied, an RGBA component must convert to a value that matches the component as specified in the **Color** command: if $m$ is less than the number of bits $b$ with which the component was specified, then the converted value must equal the most significant $m$ bits of the specified value; otherwise, the most significant $b$ bits of the converted value must equal the specified value.

APPENDIX T

# Chapter 3

# Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color and depth. Thus, rasterizing a primitive consists of two parts. The first is to determine which squares of an integer grid in window coordinates are occupied by the primitive. The second is assigning a color and a depth value to each such square. The results of this process are passed on to the next stage of the GL (per-fragment operations), which uses the information to update the appropriate locations in the framebuffer. Figure 3.1 diagrams the rasterization process.

A grid square along with its parameters of assigned color, $z$ (depth), and texture coordinates is called a *fragment*; the parameters are collectively dubbed the fragment's *associated data.* A fragment is located by its lower-left corner, which lies on integer grid coordinates. Rasterization operations also refer to a fragment's *center*, which is offset by $(1/2, 1/2)$ from its lower-left corner (and so lies on half-integer coordinates).

Grid squares need not actually be square in the GL. Rasterization rules are not affected by the actual aspect ratio of the grid squares. Display of non-square grids, however, will cause rasterized points and line segments to appear fatter in one direction than the other. We assume that fragments are square, since it simplifies antialiasing and texturing.

Several factors affect rasterization. Lines and polygons may be stippled. Points may be given differing diameters and line segments differing widths. A point, line segment, or polygon may be antialiased.

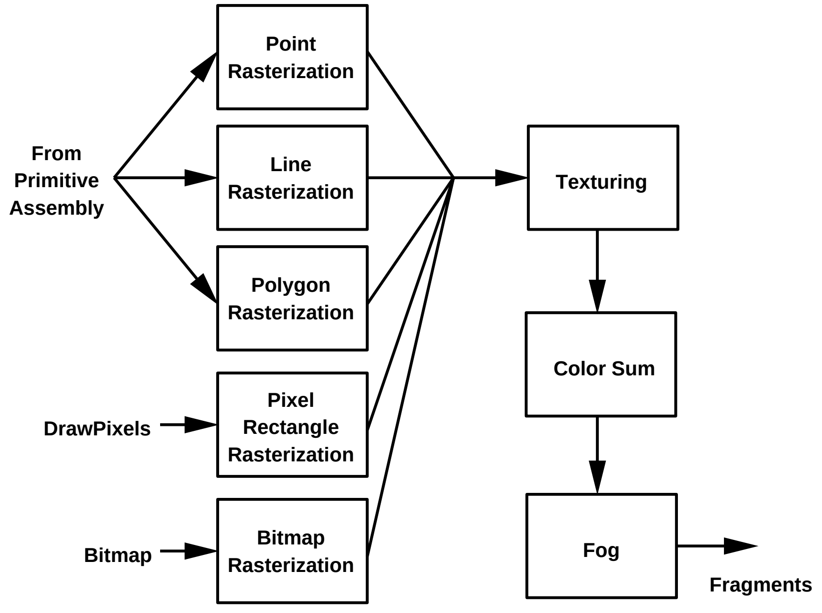


Figure 3.1. Rasterization.

APPENDIX T

## 3.1 Invariance

Consider a primitive $p'$ obtained by translating a primitive $p$ through an offset $(x, y)$ in window coordinates, where $x$ and $y$ are integers. As long as neither $p'$ nor $p$ is clipped, it must be the case that each fragment $f'$ produced from $p'$ is identical to a corresponding fragment $f$ from $p$ except that the center of $f'$ is offset by $(x, y)$ from the center of $f$.

## 3.2 Antialiasing

Antialiasing of a point, line, or polygon is effected in one of two ways depending on whether the GL is in RGBA or color index mode.

In RGBA mode, the R, G, and B values of the rasterized fragment are left unaffected, but the A value is multiplied by a floating-point value in the range $[0, 1]$ that describes a fragment's screen pixel coverage. The per-fragment stage of the GL can be set up to use the A value to blend the incoming fragment with the corresponding pixel already present in the framebuffer.

In color index mode, the least significant $b$ bits (to the left of the binary point) of the color index are used for antialiasing; $b = \min\{4, m\}$, where $m$ is the number of bits in the color index portion of the framebuffer. The antialiasing process sets these $b$ bits based on the fragment's coverage value: the bits are set to zero for no coverage and to all ones for complete coverage.

The details of how antialiased fragment coverage values are computed are difficult to specify in general. The reason is that high-quality antialiasing may take into account perceptual issues as well as characteristics of the monitor on which the contents of the framebuffer are displayed. Such details cannot be addressed within the scope of this document. Further, the coverage value computed for a fragment of some primitive may depend on the primitive's relationship to a number of grid squares neighboring the one corresponding to the fragment, and not just on the fragment's grid square. Another consideration is that accurate calculation of coverage values may be computationally expensive; consequently we allow a given GL implementation to approximate true coverage values by using a fast but not entirely accurate coverage computation.

In light of these considerations, we chose to specify the behavior of exact antialiasing in the prototypical case that each displayed pixel is a perfect square of uniform intensity. The square is called a *fragment square* and has lower left corner $(x, y)$ and upper right corner $(x+1, y+1)$. We recognize

APPENDIX T

that this simple box filter may not produce the most favorable antialiasing results, but it provides a simple, well-defined model.

A GL implementation may use other methods to perform antialiasing, subject to the following conditions:

1. If $f_1$ and $f_2$ are two fragments, and the portion of $f_1$ covered by some primitive is a subset of the corresponding portion of $f_2$ covered by the primitive, then the coverage computed for $f_1$ must be less than or equal to that computed for $f_2$.

2. The coverage computation for a fragment $f$ must be local: it may depend only on $f$'s relationship to the boundary of the primitive being rasterized. It may not depend on $f$'s $x$ and $y$ coordinates.

Another property that is desirable, but not required, is:

3. The sum of the coverage values for all fragments produced by rasterizing a particular primitive must be constant, independent of any rigid motions in window coordinates, as long as none of those fragments lies along window edges.

In some implementations, varying degrees of antialiasing quality may be obtained by providing GL hints (section 5.6), allowing a user to make an image quality versus speed tradeoff.

## 3.3 Points

The rasterization of points is controlled with

void **PointSize**( float *size* );

*size* specifies the width or diameter of a point. The default value is 1.0. A value less than or equal to zero results in the error INVALID_VALUE.

Point antialiasing is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant POINT_SMOOTH. The default state is for point antialiasing to be disabled.

In the default state, a point is rasterized by truncating its $x_w$ and $y_w$ coordinates (recall that the subscripts indicate that these are $x$ and $y$ window coordinates) to integers. This $(x, y)$ address, along with data derived from the data associated with the vertex corresponding to the point, is sent as a single fragment to the per-fragment stage of the GL.

APPENDIX T

The effect of a point width other than 1.0 depends on the state of point antialiasing. If antialiasing is disabled, the actual width is determined by rounding the supplied width to the nearest integer, then clamping it to the implementation-dependent maximum non-antialiased point width. This implementation-dependent value must be no less than the implementation-dependent maximum antialiased point width, rounded to the nearest integer value, and in any event no less than 1. If rounding the specified width results in the value 0, then it is as if the value were 1. If the resulting width is odd, then the point

$$(x, y) = (\lfloor x_w \rfloor + \frac{1}{2}, \lfloor y_w \rfloor + \frac{1}{2})$$

is computed from the vertex's $x_w$ and $y_w$, and a square grid of the odd width centered at $(x, y)$ defines the centers of the rasterized fragments (recall that fragment centers lie at half-integer window coordinate values). If the width is even, then the center point is

$$(x, y) = (\lfloor x_w + \frac{1}{2} \rfloor, \lfloor y_w + \frac{1}{2} \rfloor);$$

the rasterized fragment centers are the half-integer window coordinate values within the square of the even width centered on $(x, y)$. See figure 3.2.

All fragments produced in rasterizing a non-antialiased point are assigned the same associated data, which are those of the vertex corresponding to the point, with texture coordinates $s$, $t$, and $r$ replaced with $s/q$, $t/q$, and $r/q$, respectively. If $q$ is less than or equal to zero, the results are undefined.

If antialiasing is enabled, then point rasterization produces a fragment for each fragment square that intersects the region lying within the circle having diameter equal to the current point width and centered at the point's $(x_w, y_w)$ (figure 3.3). The coverage value for each fragment is the window coordinate area of the intersection of the circular region with the corresponding fragment square (but see section 3.2). This value is saved and used in the final step of rasterization (section 3.11). The data associated with each fragment are otherwise the data associated with the point being rasterized, with texture coordinates $s$, $t$, and $r$ replaced with $s/q$, $t/q$, and $r/q$, respectively. If $q$ is less than or equal to zero, the results are undefined.

Not all widths need be supported when point antialiasing is on, but the width 1.0 must be provided. If an unsupported width is requested, the nearest supported width is used instead. The range of supported widths and the width of evenly-spaced gradations within that range are implementation dependent. The range and gradations may be obtained using the query

APPENDIX T

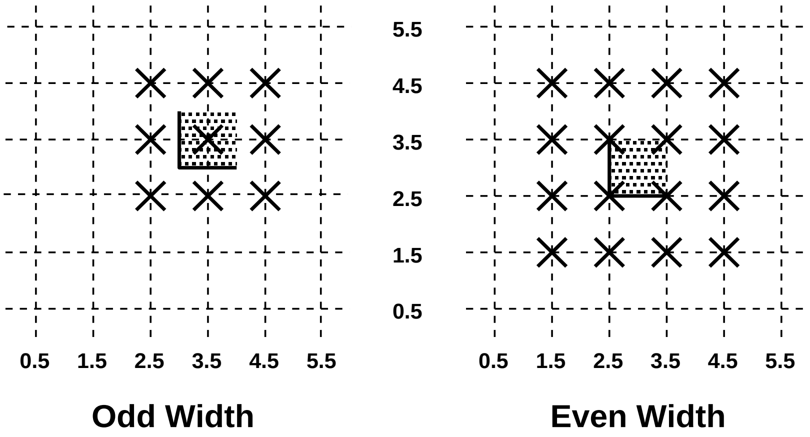


Figure 3.2. Rasterization of non-antialiased wide points. The crosses show fragment centers produced by rasterization for any point that lies within the shaded region. The dotted grid lines lie on half-integer coordinates.

mechanism described in Chapter 6. If, for instance, the width range is from 0.1 to 2.0 and the gradation width is 0.1, then the widths $0.1, 0.2, \ldots, 1.9, 2.0$ are supported.

### 3.3.1 Point Rasterization State

The state required to control point rasterization consists of the floating-point point width and a bit indicating whether or not antialiasing is enabled.

## 3.4 Line Segments

A line segment results from a line strip **Begin**/**End** object, a line loop, or a series of separate line segments. Line segment rasterization is controlled by several variables. Line width, which may be set by calling

void **LineWidth**( float *width* );

with an appropriate positive floating-point width, controls the width of rasterized line segments. The default width is 1.0. Values less than or equal

APPENDIX T

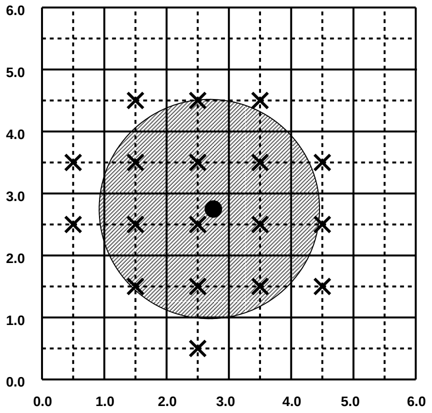


Figure 3.3. Rasterization of antialiased wide points. The black dot indicates the point to be rasterized. The shaded region has the specified width. The X marks indicate those fragment centers produced by rasterization. A fragment's computed coverage value is based on the portion of the shaded region that covers the corresponding fragment square. Solid lines lie on integer coordinates.

APPENDIX T

to 0.0 generate the error INVALID_VALUE. Antialiasing is controlled with **Enable** and **Disable** using the symbolic constant LINE_SMOOTH. Finally, line segments may be stippled. Stippling is controlled by a GL command that sets a *stipple pattern* (see below).

### 3.4.1 Basic Line Segment Rasterization

Line segment rasterization begins by characterizing the segment as either *$x$-major* or *$y$-major*. $x$-major line segments have slope in the closed interval $[-1, 1]$; all other line segments are $y$-major (slope is determined by the segment's endpoints). We shall specify rasterization only for $x$-major segments except in cases where the modifications for $y$-major segments are not self-evident.

Ideally, the GL uses a "diamond-exit" rule to determine those fragments that are produced by rasterizing a line segment. For each fragment $f$ with center at window coordinates $x_f$ and $y_f$, define a diamond-shaped region that is the intersection of four half planes:

$$R_f = \{\,(x, y) \mid |x - x_f| + |y - y_f| < 1/2.\}$$

Essentially, a line segment starting at $\mathbf{p}_a$ and ending at $\mathbf{p}_b$ produces those fragments $f$ for which the segment intersects $R_f$, except if $\mathbf{p}_b$ is contained in $R_f$. See figure 3.4.

To avoid difficulties when an endpoint lies on a boundary of $R_f$ we (in principle) perturb the supplied endpoints by a tiny amount. Let $\mathbf{p}_a$ and $\mathbf{p}_b$ have window coordinates $(x_a, y_a)$ and $(x_b, y_b)$, respectively. Obtain the perturbed endpoints $\mathbf{p}'_a$ given by $(x_a, y_a) - (\epsilon, \epsilon^2)$ and $\mathbf{p}'_b$ given by $(x_b, y_b) - (\epsilon, \epsilon^2)$. Rasterizing the line segment starting at $\mathbf{p}_a$ and ending at $\mathbf{p}_b$ produces those fragments $f$ for which the segment starting at $\mathbf{p}'_a$ and ending on $\mathbf{p}'_b$ intersects $R_f$, except if $\mathbf{p}'_b$ is contained in $R_f$. $\epsilon$ is chosen to be so small that rasterizing the line segment produces the same fragments when $\delta$ is substituted for $\epsilon$ for any $0 < \delta \leq \epsilon$.

When $\mathbf{p}_a$ and $\mathbf{p}_b$ lie on fragment centers, this characterization of fragments reduces to Bresenham's algorithm with one modification: lines produced in this description are "half-open," meaning that the final fragment (corresponding to $\mathbf{p}_b$) is not drawn. This means that when rasterizing a series of connected line segments, shared endpoints will be produced only once rather than twice (as would occur with Bresenham's algorithm).

Because the initial and final conditions of the diamond-exit rule may be difficult to implement, other line segment rasterization algorithms are allowed, subject to the following rules:

APPENDIX T

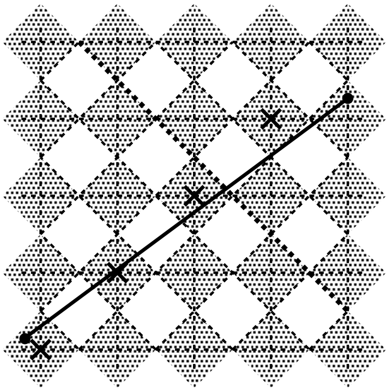


Figure 3.4. Visualization of Bresenham's algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.

1. The coordinates of a fragment produced by the algorithm may not deviate by more than one unit in either $x$ or $y$ window coordinates from a corresponding fragment produced by the diamond-exit rule.

2. The total number of fragments produced by the algorithm may differ from that produced by the diamond-exit rule by no more than one.

3. For an $x$-major line, no two fragments may be produced that lie in the same window-coordinate column (for a $y$-major line, no two fragments may appear in the same row).

4. If two line segments share a common endpoint, and both segments are either $x$-major (both left-to-right or both right-to-left) or $y$-major (both bottom-to-top or both top-to-bottom), then rasterizing both segments may not produce duplicate fragments, nor may any fragments be omitted so as to interrupt continuity of the connected segments.

Next we must specify how the data associated with each rasterized fragment are obtained. Let the window coordinates of a produced fragment center be given by $\mathbf{p}_r = (x_d, y_d)$ and let $\mathbf{p}_a = (x_a, y_a)$ and $\mathbf{p}_b = (x_b, y_b)$. Set

$$t = \frac{(\mathbf{p}_r - \mathbf{p}_a) \cdot (\mathbf{p}_b - \mathbf{p}_a)}{\|\mathbf{p}_b - \mathbf{p}_a\|^2}. \tag{3.1}$$

(Note that $t = 0$ at $\mathbf{p}_a$ and $t = 1$ at $\mathbf{p}_b$.) The value of an associated datum $f$ for the fragment, whether it be R, G, B, or A (in RGBA mode) or a color index (in color index mode), or the $s$, $t$, or $r$ texture coordinate (the depth value, window $z$, must be found using equation 3.3, below), is found as

$$f = \frac{(1-t)f_a/w_a + tf_b/w_b}{(1-t)\alpha_a/w_a + t\alpha_b/w_b} \tag{3.2}$$

where $f_a$ and $f_b$ are the data associated with the starting and ending endpoints of the segment, respectively; $w_a$ and $w_b$ are the clip $w$ coordinates of the starting and ending endpoints of the segments, respectively. $\alpha_a = \alpha_b = 1$ for all data except texture coordinates, in which case $\alpha_a = q_a$ and $\alpha_b = q_b$ ($q_a$ and $q_b$ are the homogeneous texture coordinates at the starting and ending endpoints of the segment; results are undefined if either of these is less than or equal to 0). Note that linear interpolation would use

$$f = (1-t)f_a/\alpha_a + tf_b/\alpha_b. \tag{3.3}$$

The reason that this formula is incorrect (except for the depth value) is that it interpolates a datum in window space, which may be distorted by perspective. What is actually desired is to find the corresponding value when interpolated in clip space, which equation 3.2 does. A GL implementation may choose to approximate equation 3.2 with 3.3, but this will normally lead to unacceptable distortion effects when interpolating texture coordinates.

### 3.4.2 Other Line Segment Features

We have just described the rasterization of non-antialiased line segments of width one using the default line stipple of $FFFF_{16}$. We now describe the rasterization of line segments for general values of the line segment rasterization parameters.

#### Line Stipple

The command

void **LineStipple**( int *factor*, ushort *pattern* );

defines a *line stipple*. *pattern* is an unsigned short integer. The *line stipple* is taken from the lowest order 16 bits of *pattern*. It determines those fragments that are to be drawn when the line is rasterized. *factor* is a count that is used to modify the effective line stipple by causing each bit in *line stipple* to be used *factor* times. $factor$ is clamped to the range $[1, 256]$. Line stippling may be enabled or disabled using **Enable** or **Disable** with the constant `LINE_STIPPLE`. When disabled, it is as if the line stipple has its default value.

Line stippling masks certain fragments that are produced by rasterization so that they are not sent to the per-fragment stage of the GL. The masking is achieved using three parameters: the 16-bit line stipple $p$, the line repeat count $r$, and an integer stipple counter $s$. Let

$$b = \lfloor s/r \rfloor \bmod 16,$$

Then a fragment is produced if the $b$th bit of $p$ is 1, and not produced otherwise. The bits of $p$ are numbered with 0 being the least significant and 15 being the most significant. The initial value of $s$ is zero; $s$ is incremented after production of each fragment of a line segment (fragments are produced in order, beginning at the starting point and working towards the ending point). $s$ is reset to 0 whenever a **Begin** occurs, and before every line segment in a group of independent segments (as specified when **Begin** is invoked with `LINES`).

If the line segment has been clipped, then the value of $s$ at the beginning of the line segment is indeterminate.

### Wide Lines

The actual width of non-antialiased lines is determined by rounding the supplied width to the nearest integer, then clamping it to the implementation-dependent maximum non-antialiased line width. This implementation-dependent value must be no less than the implementation-dependent maximum antialiased line width, rounded to the nearest integer value, and in any event no less than 1. If rounding the specified width results in the value 0, then it is as if the value were 1.

Non-antialiased line segments of width other than one are rasterized by offsetting them in the minor direction (for an $x$-major line, the minor direction is $y$, and for a $y$-major line, the minor direction is $x$) and replicating fragments in the minor direction (see figure 3.5). Let $w$ be the width rounded to the nearest integer (if $w = 0$, then it is as if $w = 1$). If the line segment has endpoints given by $(x_0, y_0)$ and $(x_1, y_1)$ in window coordinates, the segment with endpoints $(x_0, y_0 - (w-1)/2)$ and $(x_1, y_1 - (w-1)/2)$ is rasterized, but

**APPENDIX T**

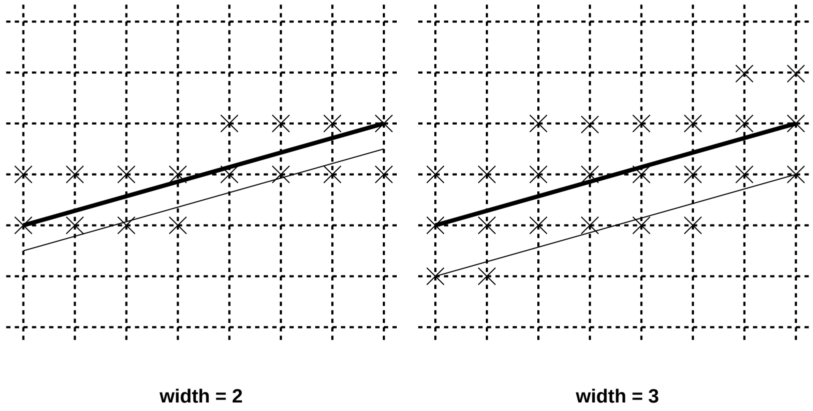


Figure 3.5. Rasterization of non-antialiased wide lines. x-major line segments are shown. The heavy line segment is the one specified to be rasterized; the light segment is the offset segment used for rasterization. x marks indicate the fragment centers produced by rasterization.

instead of a single fragment, a column of fragments of height $w$ (a row of fragments of length $w$ for a $y$-major segment) is produced at each $x$ ($y$ for $y$-major) location. The lowest fragment of this column is the fragment that would be produced by rasterizing the segment of width 1 with the modified coordinates. The whole column is not produced if the stipple bit for the column's $x$ location is zero; otherwise, the whole column is produced.

### Antialiasing

Rasterized antialiased line segments produce fragments whose fragment squares intersect a rectangle centered on the line segment. Two of the edges are parallel to the specified line segment; each is at a distance of one-half the current width from that segment: one above the segment and one below it. The other two edges pass through the line endpoints and are perpendicular to the direction of the specified line segment. Coverage values are computed for each fragment by computing the area of the intersection of the rectangle with the fragment square (see figure 3.6; see also section 3.2). Equation 3.2 is used to compute associated data values just as with non-antialiased lines; equation 3.1 is used to find the value of $t$ for each fragment whose square is intersected by the line segment's rectangle. Not all widths need be sup-

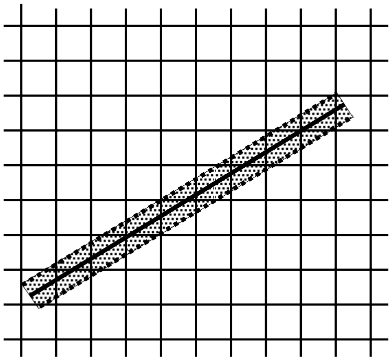


Figure 3.6. The region used in rasterizing and finding corresponding coverage values for an antialiased line segment (an x-major line segment is shown).

ported for line segment antialiasing, but width 1.0 antialiased segments must be provided. As with the point width, a GL implementation may be queried for the range and number of gradations of available antialiased line widths.

For purposes of antialiasing, a stippled line is considered to be a sequence of contiguous rectangles centered on the line segment. Each rectangle has width equal to the current line width and length equal to 1 pixel (except the last, which may be shorter). These rectangles are numbered from 0 to $n$, starting with the rectangle incident on the starting endpoint of the segment. Each of these rectangles is either eliminated or produced according to the procedure given under **Line Stipple**, above, where "fragment" is replaced with "rectangle." Each rectangle so produced is rasterized as if it were an antialiased polygon, described below (but culling, non-default settings of **PolygonMode**, and polygon stippling are not applied).

### 3.4.3 Line Rasterization State

The state required for line rasterization consists of the floating-point line width, a 16-bit line stipple, the line stipple repeat count, a bit indicating whether stippling is enabled or disabled, and a bit indicating whether line antialiasing is on or off. In addition, during rasterization, an integer stipple counter must be maintained to implement line stippling. The initial value of the line width is 1.0. The initial value of the line stipple is $FFFF_{16}$ (a stipple of all ones). The initial value of the line stipple repeat count is one.

The initial state of line stippling is disabled. The initial state of line segment antialiasing is disabled.

## 3.5 Polygons

A polygon results from a polygon **Begin**/**End** object, a triangle resulting from a triangle strip, triangle fan, or series of separate triangles, or a quadrilateral arising from a quadrilateral strip, series of separate quadrilaterals, or a **Rect** command. Like points and line segments, polygon rasterization is controlled by several variables. Polygon antialiasing is controlled with **Enable** and **Disable** with the symbolic constant `POLYGON_SMOOTH`. The analog to line segment stippling for polygons is polygon stippling, described below.

### 3.5.1 Basic Polygon Rasterization

The first step of polygon rasterization is to determine if the polygon is *back facing* or *front facing.* This determination is made by examining the sign of the area computed by equation 2.7 of section 2.13.1 (including the possible reversal of this sign as indicated by the last call to **FrontFace**). If this sign is positive, the polygon is frontfacing; otherwise, it is back facing. This determination is used in conjunction with the **CullFace** enable bit and mode value to decide whether or not a particular polygon is rasterized. The **CullFace** mode is set by calling

void **CullFace**( enum *mode* );

*mode* is a symbolic constant: one of `FRONT`, `BACK` or `FRONT_AND_BACK`. Culling is enabled or disabled with **Enable** or **Disable** using the symbolic constant `CULL_FACE`. Front facing polygons are rasterized if either culling is disabled or the **CullFace** mode is `BACK` while back facing polygons are rasterized only if either culling is disabled or the **CullFace** mode is `FRONT`. The initial setting of the **CullFace** mode is `BACK`. Initially, culling is disabled.

The rule for determining which fragments are produced by polygon rasterization is called *point sampling.* The two-dimensional projection obtained by taking the $x$ and $y$ window coordinates of the polygon's vertices is formed. Fragment centers that lie inside of this polygon are produced by rasterization. Special treatment is given to a fragment whose center lies on a polygon boundary edge. In such a case we require that if two polygons lie on either side of a common edge (with identical endpoints) on which a fragment center lies, then exactly one of the polygons results in the production of the fragment during rasterization.

As for the data associated with each fragment produced by rasterizing a polygon, we begin by specifying how these values are produced for fragments in a triangle. Define *barycentric coordinates* for a triangle. Barycentric coordinates are a set of three numbers, $a$, $b$, and $c$, each in the range $[0, 1]$, with $a + b + c = 1$. These coordinates uniquely specify any point $p$ within the triangle or on the triangle's boundary as

$$p = ap_a + bp_b + cp_c,$$

where $p_a$, $p_b$, and $p_c$ are the vertices of the triangle. $a$, $b$, and $c$ can be found as

$$a = \frac{\mathrm{A}(pp_bp_c)}{\mathrm{A}(p_ap_bp_c)}, \quad b = \frac{\mathrm{A}(pp_ap_c)}{\mathrm{A}(p_ap_bp_c)}, \quad c = \frac{\mathrm{A}(pp_ap_b)}{\mathrm{A}(p_ap_bp_c)},$$

where $\mathrm{A}(lmn)$ denotes the area in window coordinates of the triangle with vertices $l$, $m$, and $n$.

Denote a datum at $p_a$, $p_b$, or $p_c$ as $f_a$, $f_b$, or $f_c$, respectively. Then the value $f$ of a datum at a fragment produced by rasterizing a triangle is given by

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a\alpha_a/w_a + b\alpha_b/w_b + c\alpha_c/w_c} \tag{3.4}$$

where $w_a$, $w_b$ and $w_c$ are the clip $w$ coordinates of $p_a$, $p_b$, and $p_c$, respectively. $a$, $b$, and $c$ are the barycentric coordinates of the fragment for which the data are produced. $\alpha_a = \alpha_b = \alpha_c = 1$ except for texture $s$, $t$, and $r$ coordinates, for which $\alpha_a = q_a$, $\alpha_b = q_b$, and $\alpha_c = q_c$ (if any of $q_a$, $q_b$, or $q_c$ are less than or equal to zero, results are undefined). $a$, $b$, and $c$ must correspond precisely to the exact coordinates of the center of the fragment. Another way of saying this is that the data associated with a fragment must be sampled at the fragment's center.

Just as with line segment rasterization, equation 3.4 may be approximated by

$$f = af_a/\alpha_a + bf_b/\alpha_b + cf_c/\alpha_c;$$

this may yield acceptable results for color values (it *must* be used for depth values), but will normally lead to unacceptable distortion effects if used for texture coordinates.

For a polygon with more than three edges, we require only that a convex combination of the values of the datum at the polygon's vertices can be used to obtain the value assigned to each fragment produced by the rasterization

APPENDIX T

algorithm. That is, it must be the case that at every fragment

$$f = \sum_{i=1}^{n} a_i f_i$$

where $n$ is the number of vertices in the polygon, $f_i$ is the value of the $f$ at vertex $i$; for each $i$ $0 \leq a_i \leq 1$ and $\sum_{i=1}^{n} a_i = 1$. The values of the $a_i$ may differ from fragment to fragment, but at vertex $i$, $a_j = 0, j \neq i$ and $a_i = 1$.

One algorithm that achieves the required behavior is to triangulate a polygon (without adding any vertices) and then treat each triangle individually as already discussed. A scan-line rasterizer that linearly interpolates data along each edge and then linearly interpolates data across each horizontal span from edge to edge also satisfies the restrictions (in this case, the numerator and denominator of equation 3.4 should be iterated independently and a division performed for each fragment).

### 3.5.2 Stippling

Polygon stippling works much the same way as line stippling, masking out certain fragments produced by rasterization so that they are not sent to the next stage of the GL. This is the case regardless of the state of polygon antialiasing. Stippling is controlled with

void **PolygonStipple**( ubyte **pattern* );

*pattern* is a pointer to memory into which a 32 × 32 pattern is packed. The pattern is unpacked from memory according to the procedure given in section 3.6.4 for **DrawPixels**; it is as if the *height* and *width* passed to that command were both equal to 32, the *type* were BITMAP, and the *format* were COLOR_INDEX. The unpacked values (before any conversion or arithmetic would have been performed) form a stipple pattern of zeros and ones.

If $x_w$ and $y_w$ are the window coordinates of a rasterized polygon fragment, then that fragment is sent to the next stage of the GL if and only if the bit of the pattern $(x_w \bmod 32, y_w \bmod 32)$ is 1.

Polygon stippling may be enabled or disabled with **Enable** or **Disable** using the constant POLYGON_STIPPLE. When disabled, it is as if the stipple pattern were all ones.

### 3.5.3 Antialiasing

Polygon antialiasing rasterizes a polygon by producing a fragment wherever the interior of the polygon intersects that fragment's square. A coverage

APPENDIX T

value is computed at each such fragment, and this value is saved to be applied as described in section 3.11. An associated datum is assigned to a fragment by integrating the datum's value over the region of the intersection of the fragment square with the polygon's interior and dividing this integrated value by the area of the intersection. For a fragment square lying entirely within the polygon, the value of a datum at the fragment's center may be used instead of integrating the value across the fragment.

Polygon stippling operates in the same way whether polygon antialiasing is enabled or not. The polygon point sampling rule defined in section 3.5.1, however, is not enforced for antialiased polygons.

### 3.5.4 Options Controlling Polygon Rasterization

The interpretation of polygons for rasterization is controlled using

void **PolygonMode**( enum *face*, enum *mode* );

*face* is one of `FRONT`, `BACK`, or `FRONT_AND_BACK`, indicating that the rasterizing method described by *mode* replaces the rasterizing method for front facing polygons, back facing polygons, or both front and back facing polygons, respectively. *mode* is one of the symbolic constants `POINT`, `LINE`, or `FILL`. Calling **PolygonMode** with `POINT` causes certain vertices of a polygon to be treated, for rasterization purposes, just as if they were enclosed within a **Begin**(`POINT`) and **End** pair. The vertices selected for this treatment are those that have been tagged as having a polygon boundary edge beginning on them (see section 2.6.2). `LINE` causes edges that are tagged as boundary to be rasterized as line segments. (The line stipple counter is reset at the beginning of the first rasterized edge of the polygon, but not for subsequent edges.) `FILL` is the default mode of polygon rasterization, corresponding to the description in sections 3.5.1, 3.5.2, and 3.5.3. Note that these modes affect only the final rasterization of polygons: in particular, a polygon's vertices are lit, and the polygon is clipped and possibly culled before these modes are applied.

Polygon antialiasing applies only to the `FILL` state of **PolygonMode**. For `POINT` or `LINE`, point antialiasing or line segment antialiasing, respectively, apply.

### 3.5.5 Depth Offset

The depth values of all fragments generated by the rasterization of a polygon may be offset by a single value that is computed for that polygon. The

APPENDIX T

function that determines this value is specified by calling

void **PolygonOffset**( float *factor*, float *units* );

*factor* scales the maximum depth slope of the polygon, and *units* scales an implementation dependent constant that relates to the usable resolution of the depth buffer. The resulting values are summed to produce the polygon offset value. Both *factor* and *units* may be either positive or negative.

The maximum depth slope $m$ of a triangle is

$$m = \sqrt{\left(\frac{\partial z_w}{\partial x_w}\right)^2 + \left(\frac{\partial z_w}{\partial y_w}\right)^2} \tag{3.5}$$

where $(x_w, y_w, z_w)$ is a point on the triangle. $m$ may be approximated as

$$m = \max\left\{\left|\frac{\partial z_w}{\partial x_w}\right|, \left|\frac{\partial z_w}{\partial y_w}\right|\right\}. \tag{3.6}$$

If the polygon has more than three vertices, one or more values of $m$ may be used during rasterization. Each may take any value in the range $[min,max]$, where $min$ and $max$ are the smallest and largest values obtained by evaluating Equation 3.5 or Equation 3.6 for the triangles formed by all three-vertex combinations.

The minimum resolvable difference $r$ is an implementation constant. It is the smallest difference in window coordinate $z$ values that is guaranteed to remain distinct throughout polygon rasterization and in the depth buffer. All pairs of fragments generated by the rasterization of two polygons with otherwise identical vertices, but $z_w$ values that differ by $r$, will have distinct depth values.

The offset value $o$ for a polygon is

$$o = m * factor + r * units. \tag{3.7}$$

$m$ is computed as described above, as a function of depth values in the range [0,1], and $o$ is applied to depth values in the same range.

Boolean state values POLYGON_OFFSET_POINT, POLYGON_OFFSET_LINE, and POLYGON_OFFSET_FILL determine whether $o$ is applied during the rasterization of polygons in POINT, LINE, and FILL modes. These boolean state values are enabled and disabled as argument values to the commands **Enable** and **Disable**. If POLYGON_OFFSET_POINT is enabled, $o$ is added to the depth value of each fragment produced by the rasterization of a polygon in POINT mode. Likewise, if POLYGON_OFFSET_LINE or POLYGON_OFFSET_FILL is enabled, $o$

APPENDIX T

is added to the depth value of each fragment produced by the rasterization of a polygon in LINE or FILL modes, respectively.

Fragment depth values are always limited to the range [0,1], either by clamping after offset addition is performed (preferred), or by clamping the vertex values used in the rasterization of the polygon.

### 3.5.6 Polygon Rasterization State

The state required for polygon rasterization consists of a polygon stipple pattern, whether stippling is enabled or disabled, the current state of polygon antialiasing (enabled or disabled), the current values of the **PolygonMode** setting for each of front and back facing polygons, whether point, line, and fill mode polygon offsets are enabled or disabled, and the factor and bias values of the polygon offset equation. The initial stipple pattern is all ones; initially stippling is disabled. The initial setting of polygon antialiasing is disabled. The initial state for **PolygonMode** is FILL for both front and back facing polygons. The initial polygon offset factor and bias values are both 0; initially polygon offset is disabled for all modes.

## 3.6 Pixel Rectangles

Rectangles of color, depth, and certain other values may be converted to fragments using the **DrawPixels** command (described in section 3.6.4). Some of the parameters and operations governing the operation of **DrawPixels** are shared by **ReadPixels** (used to obtain pixel values from the framebuffer) and **CopyPixels** (used to copy pixels from one framebuffer location to another); the discussion of **ReadPixels** and **CopyPixels**, however, is deferred until Chapter 4 after the framebuffer has been discussed in detail. Nevertheless, we note in this section when parameters and state pertaining to **DrawPixels** also pertain to **ReadPixels** or **CopyPixels**.

A number of parameters control the encoding of pixels in client memory (for reading and writing) and how pixels are processed before being placed in or after being read from the framebuffer (for reading, writing, and copying). These parameters are set with three commands: **PixelStore**, **PixelTransfer**, and **PixelMap**.

### 3.6.1 Pixel Storage Modes

Pixel storage modes affect the operation of **DrawPixels** and **ReadPixels** (as well as other commands; see sections 3.5.2, 3.7, and 3.8) when one of

| Parameter Name | Type | Initial Value | Valid Range |
|---|---|---|---|
| UNPACK_SWAP_BYTES | boolean | FALSE | TRUE/FALSE |
| UNPACK_LSB_FIRST | boolean | FALSE | TRUE/FALSE |
| UNPACK_ROW_LENGTH | integer | 0 | $[0, \infty)$ |
| UNPACK_SKIP_ROWS | integer | 0 | $[0, \infty)$ |
| UNPACK_SKIP_PIXELS | integer | 0 | $[0, \infty)$ |
| UNPACK_ALIGNMENT | integer | 4 | 1,2,4,8 |
| UNPACK_IMAGE_HEIGHT | integer | 0 | $[0, \infty)$ |
| UNPACK_SKIP_IMAGES | integer | 0 | $[0, \infty)$ |

Table 3.1: **PixelStore** parameters pertaining to one or more of **DrawPixels**, **TexImage1D**, **TexImage2D**, and **TexImage3D**.

these commands is issued. This may differ from the time that the command is executed if the command is placed in a display list (see section 5.4). Pixel storage modes are set with

```
void PixelStore{if}( enum pname, T param );
```

*pname* is a symbolic constant indicating a parameter to be set, and *param* is the value to set it to. Table 3.1 summarizes the pixel storage parameters, their types, their initial values, and their allowable ranges. Setting a parameter to a value outside the given range results in the error INVALID_VALUE.

The version of **PixelStore** that takes a floating-point value may be used to set any type of parameter; if the parameter is boolean, then it is set to FALSE if the passed value is 0.0 and TRUE otherwise, while if the parameter is an integer, then the passed value is rounded to the nearest integer. The integer version of the command may also be used to set any type of parameter; if the parameter is boolean, then it is set to FALSE if the passed value is 0 and TRUE otherwise, while if the parameter is a floating-point value, then the passed value is converted to floating-point.

### 3.6.2 The Imaging Subset

Some pixel transfer and per-fragment operations are only made available in GL implementations which incorporate the optional *imaging subset*. The imaging subset includes both new commands, and new enumerants allowed as parameters to existing commands. If the subset is supported, *all* of these

APPENDIX T

calls and enumerants must be implemented as described later in the GL specification. If the subset is not supported, calling any of the new commands generates the error `INVALID_OPERATION`, and using any of the new enumerants generates the error `INVALID_ENUM`.

The individual operations available only in the imaging subset are described in section 3.6.3, except for blending features, which are described in chapter 4. Imaging subset operations include:

1. Color tables, including all commands and enumerants described in subsections **Color Table Specification**, **Alternate Color Table Specification Commands**, **Color Table State and Proxy State**, **Color Table Lookup**, **Post Convolution Color Table Lookup**, and **Post Color Matrix Color Table Lookup**, as well as the query commands described in section 6.1.7.

2. Convolution, including all commands and enumerants described in subsections **Convolution Filter Specification**, **Alternate Convolution Filter Specification Commands**, and **Convolution**, as well as the query commands described in section 6.1.8.

3. Color matrix, including all commands and enumerants described in subsections **Color Matrix Specification** and **Color Matrix Transformation**, as well as the simple query commands described in section 6.1.6.

4. Histogram and minmax, including all commands and enumerants described in subsections **Histogram Table Specification**, **Histogram State and Proxy State**, **Histogram**, **Minmax Table Specification**, and **Minmax**, as well as the query commands described in section 6.1.9 and section 6.1.10.

5. The subset of blending features described by **BlendEquation**, **BlendColor**, and the **BlendFunc** *modes* `CONSTANT_COLOR`, `ONE_MINUS_CONSTANT_COLOR`, `CONSTANT_ALPHA`, and `ONE_MINUS_CONSTANT_ALPHA`. These are described separately in section 4.1.6.

The imaging subset is supported only if the `EXTENSIONS` string includes the substring `"ARB_imaging"`. Querying `EXTENSIONS` is described in section 6.1.11.

If the imaging subset is not supported, the related pixel transfer operations are not performed; pixels are passed unchanged to the next operation.

| Parameter Name | Type | Initial Value | Valid Range |
|---|---|---|---|
| MAP_COLOR | boolean | FALSE | TRUE/FALSE |
| MAP_STENCIL | boolean | FALSE | TRUE/FALSE |
| INDEX_SHIFT | integer | 0 | $(-\infty, \infty)$ |
| INDEX_OFFSET | integer | 0 | $(-\infty, \infty)$ |
| $x$_SCALE | float | 1.0 | $(-\infty, \infty)$ |
| DEPTH_SCALE | float | 1.0 | $(-\infty, \infty)$ |
| $x$_BIAS | float | 0.0 | $(-\infty, \infty)$ |
| DEPTH_BIAS | float | 0.0 | $(-\infty, \infty)$ |
| POST_CONVOLUTION_$x$_SCALE | float | 1.0 | $(-\infty, \infty)$ |
| POST_CONVOLUTION_$x$_BIAS | float | 0.0 | $(-\infty, \infty)$ |
| POST_COLOR_MATRIX_$x$_SCALE | float | 1.0 | $(-\infty, \infty)$ |
| POST_COLOR_MATRIX_$x$_BIAS | float | 0.0 | $(-\infty, \infty)$ |

Table 3.2: **PixelTransfer** parameters. $x$ is RED, GREEN, BLUE, or ALPHA.

### 3.6.3 Pixel Transfer Modes

Pixel transfer modes affect the operation of **DrawPixels** (section 3.6.4), **ReadPixels** (section 4.3.2), and **CopyPixels** (section 4.3.3) at the time when one of these commands is executed (which may differ from the time the command is issued). Some pixel transfer modes are set with

void **PixelTransfer{if}**( enum *param*, T *value* );

*param* is a symbolic constant indicating a parameter to be set, and *value* is the value to set it to. Table 3.2 summarizes the pixel transfer parameters that are set with **PixelTransfer**, their types, their initial values, and their allowable ranges. Setting a parameter to a value outside the given range results in the error INVALID_VALUE. The same versions of the command exist as for **PixelStore**, and the same rules apply to accepting and converting passed values to set parameters.

The pixel map lookup tables are set with

void **PixelMap{ui us f}v**( enum *map*, sizei *size*, T *values* );

*map* is a symbolic map name, indicating the map to set, *size* indicates the size of the map, and *values* is a pointer to an array of *size* map values.

APPENDIX T

| Map Name | Address | Value | Init. Size | Init. Value |
|---|---|---|---|---|
| PIXEL_MAP_I_TO_I | color idx | color idx | 1 | 0.0 |
| PIXEL_MAP_S_TO_S | stencil idx | stencil idx | 1 | 0 |
| PIXEL_MAP_I_TO_R | color idx | R | 1 | 0.0 |
| PIXEL_MAP_I_TO_G | color idx | G | 1 | 0.0 |
| PIXEL_MAP_I_TO_B | color idx | B | 1 | 0.0 |
| PIXEL_MAP_I_TO_A | color idx | A | 1 | 0.0 |
| PIXEL_MAP_R_TO_R | R | R | 1 | 0.0 |
| PIXEL_MAP_G_TO_G | G | G | 1 | 0.0 |
| PIXEL_MAP_B_TO_B | B | B | 1 | 0.0 |
| PIXEL_MAP_A_TO_A | A | A | 1 | 0.0 |

Table 3.3: **PixelMap** parameters.

The entries of a table may be specified using one of three types: single-precision floating-point, unsigned short integer, or unsigned integer, depending on which of the three versions of **PixelMap** is called. A table entry is converted to the appropriate type when it is specified. An entry giving a color component value is converted according to table 2.6. An entry giving a color index value is converted from an unsigned short integer or unsigned integer to floating-point. An entry giving a stencil index is converted from single-precision floating-point to an integer by rounding to nearest. The various tables and their initial sizes and entries are summarized in table 3.3. A table that takes an index as an address must have $size = 2^n$ or the error INVALID_VALUE results. The maximum allowable *size* of each table is specified by the implementation dependent value MAX_PIXEL_MAP_TABLE, but must be at least 32 (a single maximum applies to all tables). The error INVALID_VALUE is generated if a *size* larger than the implemented maximum, or less than one, is given to **PixelMap**.

### Color Table Specification

Color lookup tables are specified with

void **ColorTable**( enum *target*, enum *internalformat*,
  sizei *width*, enum *format*, enum *type*, void **data* );

*target* must be one of the *regular* color table names listed in table 3.4 to define the table. A *proxy* table name is a special case discussed later in

APPENDIX T

| Table Name | Type |
|---|---|
| COLOR_TABLE | regular |
| POST_CONVOLUTION_COLOR_TABLE | |
| POST_COLOR_MATRIX_COLOR_TABLE | |
| PROXY_COLOR_TABLE | proxy |
| PROXY_POST_CONVOLUTION_COLOR_TABLE | |
| PROXY_POST_COLOR_MATRIX_COLOR_TABLE | |

Table 3.4: Color table names. Regular tables have associated image data. Proxy tables have no image data, and are used only to determine if an image can be loaded into the corresponding regular table.

this section. *width*, *format*, *type*, and *data* specify an image in memory with the same meaning and allowed values as the corresponding arguments to **DrawPixels** (see section 3.6.4), with *height* taken to be 1. The maximum allowable *width* of a table is implementation-dependent, but must be at least 32. The *format*s COLOR_INDEX, DEPTH_COMPONENT, and STENCIL_INDEX and the *type* BITMAP are not allowed.

The specified image is taken from memory and processed just as if **DrawPixels** were called, stopping after the final expansion to RGBA. The R, G, B, and A components of each pixel are then scaled by the four COLOR_TABLE_SCALE parameters, biased by the four COLOR_TABLE_BIAS parameters, and clamped to $[0, 1]$. These parameters are set by calling **ColorTableParameterfv** as described below.

Components are then selected from the resulting R, G, B, and A values to obtain a table with the *base internal format* specified by (or derived from) *internalformat*, in the same manner as for textures (section 3.8.1). *internalformat* must be one of the formats in table 3.15 or table 3.16.

The color lookup table is redefined to have *width* entries, each with the specified internal format. The table is formed with indices 0 through $width-1$. Table location $i$ is specified by the $i$th image pixel, counting from zero.

The error INVALID_VALUE is generated if *width* is not zero or a non-negative power of two. The error TABLE_TOO_LARGE is generated if the specified color lookup table is too large for the implementation.

The scale and bias parameters for a table are specified by calling

```
void ColorTableParameter{if}v( enum target,
   enum pname, T params );
```

APPENDIX T

*target* must be a regular color table name. *pname* is one of `COLOR_TABLE_SCALE` or `COLOR_TABLE_BIAS`. *params* points to an array of four values: red, green, blue, and alpha, in that order.

A GL implementation may vary its allocation of internal component resolution based on any **ColorTable** parameter, but the allocation must not be a function of any other factor, and cannot be changed once it is established. Allocations must be invariant; the same allocation must be made each time a color table is specified with the same parameter values. These allocation rules also apply to proxy color tables, which are described later in this section.

### Alternate Color Table Specification Commands

Color tables may also be specified using image data taken directly from the framebuffer, and portions of existing tables may be respecified.

The command

```
void CopyColorTable( enum target, enum internalformat,
    int x, int y, sizei width );
```

defines a color table in exactly the manner of **ColorTable**, except that table data are taken from the framebuffer, rather than from client memory. *target* must be a regular color table name. *x*, *y*, and *width* correspond precisely to the corresponding arguments of **CopyPixels** (refer to section 4.3.3); they specify the image's *width* and the lower left $(x, y)$ coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **CopyPixels** with argument *type* set to `COLOR` and *height* set to 1, stopping after the final expansion to RGBA.

Subsequent processing is identical to that described for **ColorTable**, beginning with scaling by `COLOR_TABLE_SCALE`. Parameters *target*, *internalformat* and *width* are specified using the same values, with the same meanings, as the equivalent arguments of **ColorTable**. *format* is taken to be `RGBA`.

Two additional commands,

```
void ColorSubTable( enum target, sizei start,
    sizei count, enum format, enum type, void *data );
void CopyColorSubTable( enum target, sizei start,
    int x, int y, sizei count );
```

respecify only a portion of an existing color table. No change is made to the *internalformat* or *width* parameters of the specified color table, nor is any change made to table entries outside the specified portion. *target* must be a regular color table name.

**ColorSubTable** arguments *format*, *type*, and *data* match the corresponding arguments to **ColorTable**, meaning that they are specified using the same values, and have the same meanings. Likewise, **CopyColorSubTable** arguments *x*, *y*, and *count* match the *x*, *y*, and *width* arguments of **CopyColorTable**. Both of the **ColorSubTable** commands interpret and process pixel groups in exactly the manner of their **ColorTable** counterparts, except that the assignment of R, G, B, and A pixel group values to the color table components is controlled by the *internalformat* of the table, not by an argument to the command.

Arguments *start* and *count* of **ColorSubTable** and **CopyColorSubTable** specify a subregion of the color table starting at index *start* and ending at index $start + count - 1$. Counting from zero, the $n$th pixel group is assigned to the table entry with index $count + n$. The error INVALID_VALUE is generated if $start + count > width$.

### Color Table State and Proxy State

The state necessary for color tables can be divided into two categories. For each of the three tables, there is an array of values. Each array has associated with it a width, an integer describing the internal format of the table, six integer values describing the resolutions of each of the red, green, blue, alpha, luminance, and intensity components of the table, and two groups of four floating-point numbers to store the table scale and bias. Each initial array is null (zero width, internal format RGBA, with zero-sized components). The initial value of the scale parameters is (1,1,1,1) and the initial value of the bias parameters is (0,0,0,0).

In addition to the color lookup tables, partially instantiated proxy color lookup tables are maintained. Each proxy table includes width and internal format state values, as well as state for the red, green, blue, alpha, luminance, and intensity component resolutions. Proxy tables do not include image data, nor do they include scale and bias parameters. When **ColorTable** is executed with *target* specified as one of the proxy color table names listed in table 3.4, the proxy state values of the table are recomputed and updated. If the table is too large, no error is generated, but the proxy format, width and component resolutions are set to zero. If the color table would be accommodated by **ColorTable** called with *target* set to the corresponding regular table name (COLOR_TABLE is the regular name corresponding to PROXY_COLOR_TABLE, for example), the proxy state values are set exactly as

APPENDIX T

though the regular table were being specified. Calling **ColorTable** with a proxy *target* has no effect on the image or state of any actual color table.

There is no image associated with any of the proxy targets. They cannot be used as color tables, and they must never be queried using **GetColorTable**. The error INVALID_ENUM is generated if this is attempted.

### Convolution Filter Specification

A two-dimensional convolution filter image is specified by calling

```
void ConvolutionFilter2D( enum target,
   enum internalformat, sizei width, sizei height,
   enum format, enum type, void *data );
```

*target* must be CONVOLUTION_2D. *width*, *height*, *format*, *type*, and *data* specify an image in memory with the same meaning and allowed values as the corresponding parameters to **DrawPixels**. The *format*s COLOR_INDEX, DEPTH_COMPONENT, and STENCIL_INDEX and the *type* BITMAP are not allowed.

The specified image is extracted from memory and processed just as if **DrawPixels** were called, stopping after the final expansion to RGBA. The R, G, B, and A components of each pixel are then scaled by the four two-dimensional CONVOLUTION_FILTER_SCALE parameters and biased by the four two-dimensional CONVOLUTION_FILTER_BIAS parameters. These parameters are set by calling **ConvolutionParameterfv** as described below. No clamping takes place at any time during this process.

Components are then selected from the resulting R, G, B, and A values to obtain a table with the *base internal format* specified by (or derived from) *internalformat*, in the same manner as for textures (section 3.8.1). *internalformat* must be one of the formats in table 3.15 or table 3.16.

The red, green, blue, alpha, luminance, and/or intensity components of the pixels are stored in floating point, rather than integer format. They form a two-dimensional image indexed with coordinates $i, j$ such that $i$ increases from left to right, starting at zero, and $j$ increases from bottom to top, also starting at zero. Image location $i, j$ is specified by the $N$th pixel, counting from zero, where

$$N = i + j * width$$

The error INVALID_VALUE is generated if *width* or *height* is greater than the maximum supported value. These values are queried with **GetConvolutionParameteriv**, setting *target* to CONVOLUTION_2D and *pname* to MAX_CONVOLUTION_WIDTH or MAX_CONVOLUTION_HEIGHT, respectively.

APPENDIX T

The scale and bias parameters for a two-dimensional filter are specified by calling

```
void ConvolutionParameter{if}v( enum target,
    enum pname, T params );
```

with *target* CONVOLUTION_2D. *pname* is one of CONVOLUTION_FILTER_SCALE or CONVOLUTION_FILTER_BIAS. *params* points to an array of four values: red, green, blue, and alpha, in that order.

A one-dimensional convolution filter is defined using

```
void ConvolutionFilter1D( enum target,
    enum internalformat, sizei width, enum format,
    enum type, void *data );
```

*target* must be CONVOLUTION_1D. *internalformat*, *width*, *format*, and *type* have identical semantics and accept the same values as do their two-dimensional counterparts. *data* must point to a one-dimensional image, however.

The image is extracted from memory and processed as if **ConvolutionFilter2D** were called with a *height* of 1, except that it is scaled and biased by the one-dimensional CONVOLUTION_FILTER_SCALE and CONVOLUTION_FILTER_BIAS parameters. These parameters are specified exactly as the two-dimensional parameters, except that **ConvolutionParameterfv** is called with *target* CONVOLUTION_1D.

The image is formed with coordinates $i$ such that $i$ increases from left to right, starting at zero. Image location $i$ is specified by the $i$th pixel, counting from zero.

The error INVALID_VALUE is generated if *width* is greater than the maximum supported value. This value is queried using **GetConvolutionParameteriv**, setting *target* to CONVOLUTION_1D and *pname* to MAX_CONVOLUTION_WIDTH.

Special facilities are provided for the definition of two-dimensional *separable* filters – filters whose image can be represented as the product of two one-dimensional images, rather than as full two-dimensional images. A two-dimensional separable convolution filter is specified with

```
void SeparableFilter2D( enum target, enum internalformat,
    sizei width, sizei height, enum format, enum type,
    void *row, void *column );
```

*target* must be SEPARABLE_2D. *internalformat* specifies the formats of the table entries of the two one-dimensional images that will be retained. *row* points to a *width* pixel wide image of the specified *format* and *type*. *column* points to a *height* pixel high image, also of the specified *format* and *type*.

The two images are extracted from memory and processed as if **ConvolutionFilter1D** were called separately for each, except that each image is scaled and biased by the two-dimensional separable CONVOLUTION_FILTER_SCALE and CONVOLUTION_FILTER_BIAS parameters. These parameters are specified exactly as the one-dimensional and two-dimensional parameters, except that **ConvolutionParameteriv** is called with *target* SEPARABLE_2D.

### Alternate Convolution Filter Specification Commands

One and two-dimensional filters may also be specified using image data taken directly from the framebuffer.

The command

```
void CopyConvolutionFilter2D( enum target,
   enum internalformat, int x, int y, sizei width,
   sizei height );
```

defines a two-dimensional filter in exactly the manner of **ConvolutionFilter2D**, except that image data are taken from the framebuffer, rather than from client memory. *target* must be CONVOLUTION_2D. *x*, *y*, *width*, and *height* correspond precisely to the corresponding arguments of **CopyPixels** (refer to section 4.3.3); they specify the image's *width* and *height*, and the lower left $(x, y)$ coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **CopyPixels** with argument *type* set to COLOR, stopping after the final expansion to RGBA.

Subsequent processing is identical to that described for **ConvolutionFilter2D**, beginning with scaling by CONVOLUTION_FILTER_SCALE. Parameters *target*, *internalformat*, *width*, and *height* are specified using the same values, with the same meanings, as the equivalent arguments of **ConvolutionFilter2D**. *format* is taken to be RGBA.

The command

```
void CopyConvolutionFilter1D( enum target,
   enum internalformat, int x, int y, sizei width );
```

defines a one-dimensional filter in exactly the manner of **ConvolutionFilter1D**, except that image data are taken from the framebuffer, rather than from client memory. *target* must be CONVOLUTION_1D. $x$, $y$, and *width* correspond precisely to the corresponding arguments of **CopyPixels** (refer to section 4.3.3); they specify the image's *width* and the lower left $(x, y)$ coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **CopyPixels** with argument *type* set to COLOR and *height* set to 1, stopping after the final expansion to RGBA.

Subsequent processing is identical to that described for **ConvolutionFilter1D**, beginning with scaling by CONVOLUTION_FILTER_SCALE. Parameters *target*, *internalformat*, and *width* are specified using the same values, with the same meanings, as the equivalent arguments of **ConvolutionFilter2D**. *format* is taken to be RGBA.

### Convolution Filter State

The required state for convolution filters includes a one-dimensional image array, two one-dimensional image arrays for the separable filter, and a two-dimensional image array. The two-dimensional array has associated with it a height. Each array has associated with it a width, an integer describing the internal format of the table, and six integer values describing the resolutions of each of the red, green, blue, alpha, luminance, and intensity components of the table. Each filter (one-dimensional, two-dimensional, and two-dimensional separable) also has associated with it two groups of four floating-point numbers to store the filter scale and bias.

Each initial convolution filter is null (zero width and height, internal format RGBA, with zero-sized components). The initial value of all scale parameters is (1,1,1,1) and the initial value of all bias parameters is (0,0,0,0).

### Color Matrix Specification

Setting the matrix mode to COLOR causes the matrix operations described in section 2.10.2 to apply to the top matrix on the color matrix stack. All matrix operations have the same effect on the color matrix as they do on the other matrices.

### Histogram Table Specification

The histogram table is specified with

void **Histogram**( enum *target*, sizei *width*,
enum *internalformat*, boolean *sink* );

*target* must be HISTOGRAM if a histogram table is to be specified. *target* value PROXY_HISTOGRAM is a special case discussed later in this section. *width* specifies the number of entries in the histogram table, and *internalformat* specifies the format of each table entry. The maximum allowable *width* of the histogram table is implementation-dependent, but must be at least 32. *sink* specifies whether pixel groups will be consumed by the histogram operation (TRUE) or passed on to the minmax operation (FALSE).

If no error results from the execution of **Histogram**, the specified histogram table is redefined to have *width* entries, each with the specified internal format. The entries are indexed 0 through $width - 1$. Each component in each entry is set to zero. The values in the previous histogram table, if any, are lost.

The error INVALID_VALUE is generated if *width* is not zero or a non-negative power of 2. The error TABLE_TOO_LARGE is generated if the specified histogram table is too large for the implementation. The error INVALID_ENUM is generated if *internalformat* is not one of the values accepted by the corresponding parameter of **TexImage2D**, or is 1, 2, 3, 4, INTENSITY, INTENSITY4, INTENSITY8, INTENSITY12, or INTENSITY16.

A GL implementation may vary its allocation of internal component resolution based on any **Histogram** parameter, but the allocation must not be a function of any other factor, and cannot be changed once it is established. In particular, allocations must be invariant; the same allocation must be made each time a histogram is specified with the same parameter values. These allocation rules also apply to the proxy histogram, which is described later in this section.

### Histogram State and Proxy State

The state necessary for histogram operation is an array of values, with which is associated a width, an integer describing the internal format of the histogram, five integer values describing the resolutions of each of the red, green, blue, alpha, and luminance components of the table, and a flag indicating whether or not pixel groups are consumed by the operation. The initial array is null (zero width, internal format RGBA, with zero-sized components). The initial value of the flag is false.

In addition to the histogram table, a partially instantiated proxy histogram table is maintained. It includes width, internal format, and red,

APPENDIX T

green, blue, alpha, and luminance component resolutions. The proxy table does not include image data or the flag. When **Histogram** is executed with *target* set to PROXY_HISTOGRAM, the proxy state values are recomputed and updated. If the histogram array is too large, no error is generated, but the proxy format, width, and component resolutions are set to zero. If the histogram table would be accomodated by **Histogram** called with *target* set to HISTOGRAM, the proxy state values are set exactly as though the actual histogram table were being specified. Calling **Histogram** with *target* PROXY_HISTOGRAM has no effect on the actual histogram table.

There is no image associated with PROXY_HISTOGRAM. It cannot be used as a histogram, and its image must never queried using **GetHistogram**. The error INVALID_ENUM results if this is attempted.

### Minmax Table Specification

The minmax table is specified with

```
void Minmax( enum target, enum internalformat,
   boolean sink );
```

*target* must be MINMAX. *internalformat* specifies the format of the table entries. *sink* specifies whether pixel groups will be consumed by the minmax operation (TRUE) or passed on to final conversion (FALSE).

The error INVALID_ENUM is generated if *internalformat* is not one of the values accepted by the corresponding parameter of **TexImage2D**, or is 1, 2, 3, 4, INTENSITY, INTENSITY4, INTENSITY8, INTENSITY12, or INTENSITY16. The resulting table always has 2 entries, each with values corresponding only to the components of the internal format.

The state necessary for minmax operation is a table containing two elements (the first element stores the minimum values, the second stores the maximum values), an integer describing the internal format of the table, and a flag indicating whether or not pixel groups are consumed by the operation. The initial state is a minimum table entry set to the maximum representable value and a maximum table entry set to the minimum representable value. Internal format is set to RGBA and the initial value of the flag is false.

## 3.6.4 Rasterization of Pixel Rectangles

The process of drawing pixels encoded in host memory is diagrammed in figure 3.7. We describe the stages of this process in the order in which they occur.

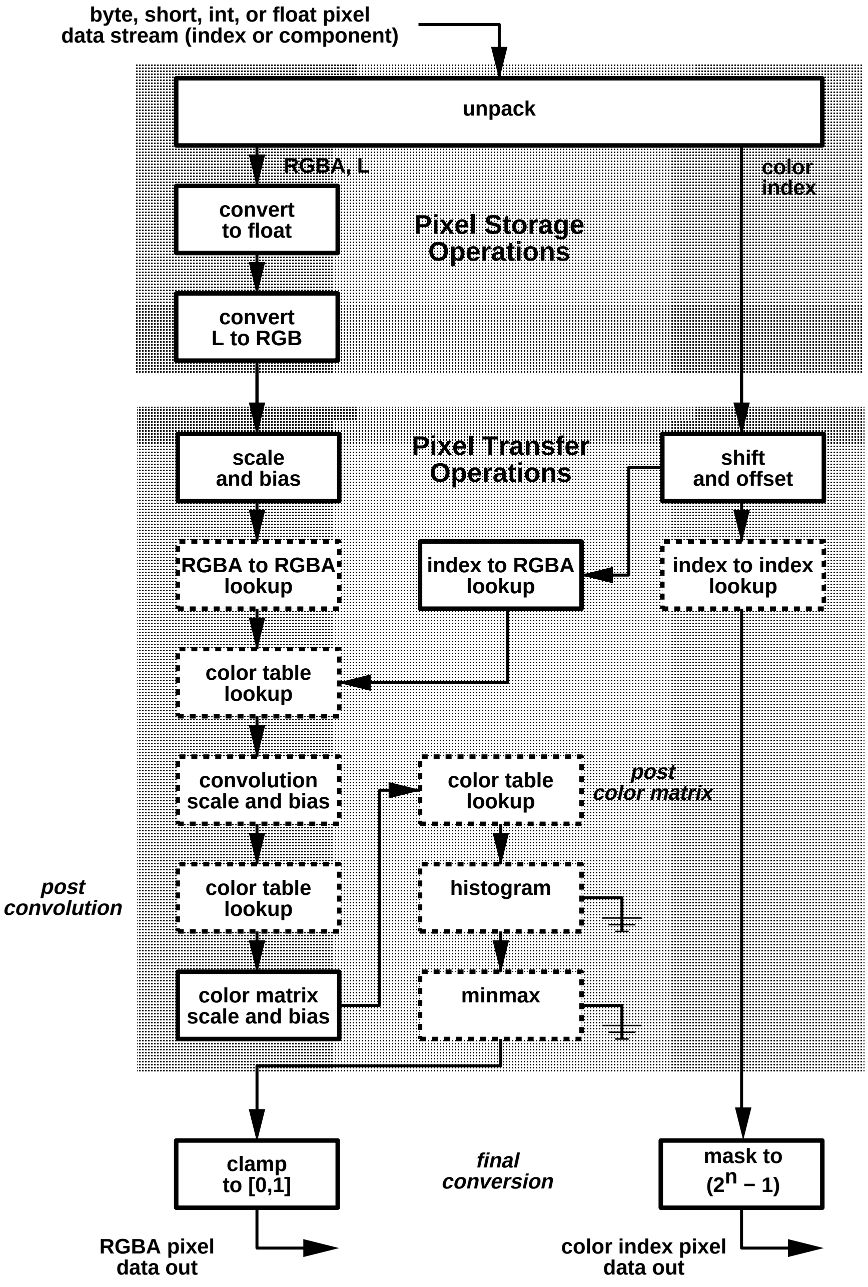


Figure 3.7. Operation of **DrawPixels**. Output is RGBA pixels if the GL is in RGBA mode, color index pixels otherwise. Operations in dashed boxes may be enabled or disabled. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

Pixels are drawn using

void **DrawPixels**( sizei *width*, sizei *height*, enum *format*, enum *type*, void **data* );

*format* is a symbolic constant indicating what the values in memory represent. *width* and *height* are the width and height, respectively, of the pixel rectangle to be drawn. *data* is a pointer to the data to be drawn. These data are represented with one of seven GL data types, specified by *type*. The correspondence between the twenty *type* token values and the GL data types they indicate is given in table 3.5. If the GL is in color index mode and *format* is not one of COLOR_INDEX, STENCIL_INDEX, or DEPTH_COMPONENT, then the error INVALID_OPERATION occurs. If *type* is BITMAP and *format* is not COLOR_INDEX or STENCIL_INDEX then the error INVALID_ENUM occurs. Some additional constraints on the combinations of *format* and *type* values that are accepted is discussed below.

### Unpacking

Data are taken from host memory as a sequence of signed or unsigned bytes (GL data types byte and ubyte), signed or unsigned short integers (GL data types short and ushort), signed or unsigned integers (GL data types int and uint), or floating point values (GL data type float). These elements are grouped into sets of one, two, three, or four values, depending on the *format*, to form a group. Table 3.6 summarizes the format of groups obtained from memory; it also indicates those formats that yield indices and those that yield components.

By default the values of each GL data type are interpreted as they would be specified in the language of the client's GL binding. If UNPACK_SWAP_BYTES is enabled, however, then the values are interpreted with the bit orderings modified as per table 3.7. The modified bit orderings are defined only if the GL data type ubyte has eight bits, and then for each specific GL data type only if that type is represented with 8, 16, or 32 bits.

The groups in memory are treated as being arranged in a rectangle. This rectangle consists of a series of *rows*, with the first element of the first group of the first row pointed to by the pointer passed to **DrawPixels**. If the value of UNPACK_ROW_LENGTH is not positive, then the number of groups in a row is *width*; otherwise the number of groups is UNPACK_ROW_LENGTH. If $p$ indicates the location in memory of the first element of the first row, then the first element of the $N$th row is indicated by

APPENDIX T

| *type* Parameter Token Name | Corresponding GL Data Type | Special Interpretation |
|---|---|---|
| `UNSIGNED_BYTE` | `ubyte` | No |
| `BITMAP` | `ubyte` | Yes |
| `BYTE` | `byte` | No |
| `UNSIGNED_SHORT` | `ushort` | No |
| `SHORT` | `short` | No |
| `UNSIGNED_INT` | `uint` | No |
| `INT` | `int` | No |
| `FLOAT` | `float` | No |
| `UNSIGNED_BYTE_3_3_2` | `ubyte` | Yes |
| `UNSIGNED_BYTE_2_3_3_REV` | `ubyte` | Yes |
| `UNSIGNED_SHORT_5_6_5` | `ushort` | Yes |
| `UNSIGNED_SHORT_5_6_5_REV` | `ushort` | Yes |
| `UNSIGNED_SHORT_4_4_4_4` | `ushort` | Yes |
| `UNSIGNED_SHORT_4_4_4_4_REV` | `ushort` | Yes |
| `UNSIGNED_SHORT_5_5_5_1` | `ushort` | Yes |
| `UNSIGNED_SHORT_1_5_5_5_REV` | `ushort` | Yes |
| `UNSIGNED_INT_8_8_8_8` | `uint` | Yes |
| `UNSIGNED_INT_8_8_8_8_REV` | `uint` | Yes |
| `UNSIGNED_INT_10_10_10_2` | `uint` | Yes |
| `UNSIGNED_INT_2_10_10_10_REV` | `uint` | Yes |

Table 3.5: **DrawPixels** and **ReadPixels** *type* parameter values and the corresponding GL data types. Refer to table 2.2 for definitions of GL data types. Special interpretations are described near the end of section 3.6.4.

| Format Name | Element Meaning and Order | Target Buffer |
|---|---|---|
| COLOR_INDEX | Color Index | Color |
| STENCIL_INDEX | Stencil Index | Stencil |
| DEPTH_COMPONENT | Depth | Depth |
| RED | R | Color |
| GREEN | G | Color |
| BLUE | B | Color |
| ALPHA | A | Color |
| RGB | R, G, B | Color |
| RGBA | R, G, B, A | Color |
| BGR | B, G, R | Color |
| BGRA | B, G, R, A | Color |
| LUMINANCE | Luminance | Color |
| LUMINANCE_ALPHA | Luminance, A | Color |

Table 3.6: **DrawPixels** and **ReadPixels** formats. The second column gives a description of and the number and order of elements in a group. Unless specified as an index, formats yield components.

| Element Size | Default Bit Ordering | Modified Bit Ordering |
|---|---|---|
| 8 bit | [7..0] | [7..0] |
| 16 bit | [15..0] | [7..0][15..8] |
| 32 bit | [31..0] | [7..0][15..8][23..16][31..24] |

Table 3.7: Bit ordering modification of elements when UNPACK_SWAP_BYTES is enabled. These reorderings are defined only when GL data type ubyte has 8 bits, and then only for GL data types with 8, 16, or 32 bits. Bit 0 is the least significant.

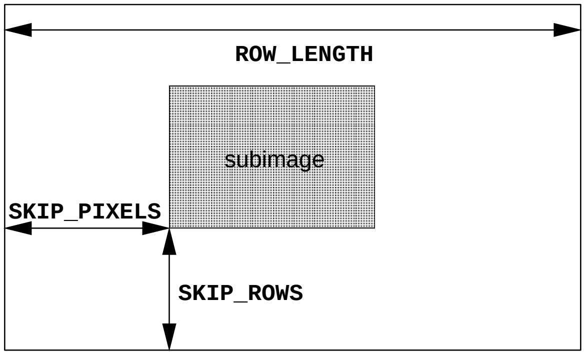


Figure 3.8. Selecting a subimage from an image. The indicated parameter names are prefixed by UNPACK_ for **DrawPixels** and by PACK_ for **ReadPixels**.

$$p + Nk \tag{3.8}$$

where $N$ is the row number (counting from zero) and k is defined as

$$k = \begin{cases} nl & s \geq a, \\ a/s \lceil snl/a \rceil & s < a \end{cases} \tag{3.9}$$

where $n$ is the number of elements in a group, $l$ is the number of groups in the row, $a$ is the value of UNPACK_ALIGNMENT, and $s$ is the size, in units of GL ubytes, of an element. If the number of bits per element is not 1, 2, 4, or 8 times the number of bits in a GL ubyte, then $k = nl$ for all values of $a$.

There is a mechanism for selecting a sub-rectangle of groups from a larger containing rectangle. This mechanism relies on three integer parameters: UNPACK_ROW_LENGTH, UNPACK_SKIP_ROWS, and UNPACK_SKIP_PIXELS. Before obtaining the first group from memory, the pointer supplied to **DrawPixels** is effectively advanced by (UNPACK_SKIP_PIXELS)$n$ + (UNPACK_SKIP_ROWS)$k$ elements. Then *width* groups are obtained from contiguous elements in memory (without advancing the pointer), after which the pointer is advanced by $k$ elements. *height* sets of *width* groups of values are obtained this way. See figure 3.8.

Calling **DrawPixels** with a *type* of UNSIGNED_BYTE_3_3_2, UNSIGNED_BYTE_2_3_3_REV, UNSIGNED_SHORT_5_6_5, UNSIGNED_SHORT_5_6_5_REV,

APPENDIX T

| *type* Parameter Token Name | GL Data Type | Number of Components | Matching Pixel Formats |
|---|---|---|---|
| UNSIGNED_BYTE_3_3_2 | ubyte | 3 | RGB |
| UNSIGNED_BYTE_2_3_3_REV | ubyte | 3 | RGB |
| UNSIGNED_SHORT_5_6_5 | ushort | 3 | RGB |
| UNSIGNED_SHORT_5_6_5_REV | ushort | 3 | RGB |
| UNSIGNED_SHORT_4_4_4_4 | ushort | 4 | RGBA,BGRA |
| UNSIGNED_SHORT_4_4_4_4_REV | ushort | 4 | RGBA,BGRA |
| UNSIGNED_SHORT_5_5_5_1 | ushort | 4 | RGBA,BGRA |
| UNSIGNED_SHORT_1_5_5_5_REV | ushort | 4 | RGBA,BGRA |
| UNSIGNED_INT_8_8_8_8 | uint | 4 | RGBA,BGRA |
| UNSIGNED_INT_8_8_8_8_REV | uint | 4 | RGBA,BGRA |
| UNSIGNED_INT_10_10_10_2 | uint | 4 | RGBA,BGRA |
| UNSIGNED_INT_2_10_10_10_REV | uint | 4 | RGBA,BGRA |

Table 3.8: Packed pixel formats.

UNSIGNED_SHORT_4_4_4_4, UNSIGNED_SHORT_4_4_4_4_REV, UNSIGNED_SHORT_5_5_5_1, UNSIGNED_SHORT_1_5_5_5_REV, UNSIGNED_INT_8_8_8_8, UNSIGNED_INT_8_8_8_8_REV, UNSIGNED_INT_10_10_10_2, or UNSIGNED_INT_2_10_10_10_REV is a special case in which all the components of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type. The number of components per packed pixel is fixed by the type, and must match the number of components per group indicated by the *format* parameter, as listed in table 3.8. The error INVALID_OPERATION is generated if a mismatch occurs. This constraint also holds for all other functions that accept or return pixel data using *type* and *format* parameters to define the type and format of that data.

Bitfield locations of the first, second, third, and fourth components of each packed pixel type are illustrated in tables 3.9, 3.10, and 3.11. Each bitfield is interpreted as an unsigned integer value. If the base GL type is supported with more than the minimum precision (e.g. a 9-bit byte) the packed components are right-justified in the pixel.

Components are normally packed with the first component in the most significant bits of the bitfield, and successive component occupying progressively less significant locations. Types whose token names end with _REV reverse the component packing order from least to most significant locations. In all cases, the most significant bit of each component is packed in

APPENDIX T

the most significant bit location of its location in the bitfield.

UNSIGNED_BYTE_3_3_2:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="3">1st Component</td><td colspan="3">2nd</td><td colspan="2">3rd</td></tr>
</table>

UNSIGNED_BYTE_2_3_3_REV:

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="2">3rd</td><td colspan="3">2nd</td><td colspan="3">1st Component</td></tr>
</table>

Table 3.9: UNSIGNED_BYTE formats. Bit numbers are indicated for each component.

UNSIGNED_SHORT_5_6_5:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="5">1st Component</td><td colspan="6">2nd</td><td colspan="5">3rd</td></tr>
</table>

UNSIGNED_SHORT_5_6_5_REV:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="5">3rd</td><td colspan="6">2nd</td><td colspan="5">1st Component</td></tr>
</table>

UNSIGNED_SHORT_4_4_4_4:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="4">1st Component</td><td colspan="4">2nd</td><td colspan="4">3rd</td><td colspan="4">4th</td></tr>
</table>

UNSIGNED_SHORT_4_4_4_4_REV:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="4">4th</td><td colspan="4">3rd</td><td colspan="4">2nd</td><td colspan="4">1st Component</td></tr>
</table>

UNSIGNED_SHORT_5_5_5_1:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="5">1st Component</td><td colspan="5">2nd</td><td colspan="5">3rd</td><td>4th</td></tr>
</table>

UNSIGNED_SHORT_1_5_5_5_REV:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>4th</td><td colspan="5">3rd</td><td colspan="5">2nd</td><td colspan="5">1st Component</td></tr>
</table>

Table 3.10: UNSIGNED_SHORT formats

APPENDIX T

UNSIGNED_INT_8_8_8_8:

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="8">1st Component</td><td colspan="8">2nd</td><td colspan="8">3rd</td><td colspan="8">4th</td></tr>
</table>

UNSIGNED_INT_8_8_8_8_REV:

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="8">4th</td><td colspan="8">3rd</td><td colspan="8">2nd</td><td colspan="8">1st Component</td></tr>
</table>

UNSIGNED_INT_10_10_10_2:

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="10">1st Component</td><td colspan="10">2nd</td><td colspan="10">3rd</td><td colspan="2">4th</td></tr>
</table>

UNSIGNED_INT_2_10_10_10_REV:

<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="2">4th</td><td colspan="10">3rd</td><td colspan="10">2nd</td><td colspan="10">1st Component</td></tr>
</table>

Table 3.11: UNSIGNED_INT formats

| Format | First Component | Second Component | Third Component | Fourth Component |
|---|---|---|---|---|
| RGB | red | green | blue | |
| RGBA | red | green | blue | alpha |
| BGRA | blue | green | red | alpha |

Table 3.12: Packed pixel field assignments

The assignment of component to fields in the packed pixel is as described in table 3.12

Byte swapping, if enabled, is performed before the component are extracted from each pixel. The above discussions of row length and image extraction are valid for packed pixels, if "group" is substituted for "component" and the number of components per group is understood to be one.

Calling **DrawPixels** with a *type* of BITMAP is a special case in which the data are a series of GL ubyte values. Each ubyte value specifies 8 1-bit elements with its 8 least-significant bits. The 8 single-bit elements are ordered from most significant to least significant if the value of UNPACK_LSB_FIRST is FALSE; otherwise, the ordering is from least significant to most significant. The values of bits other than the 8 least significant in each ubyte are not significant.

The first element of the first row is the first bit (as defined above) of the ubyte pointed to by the pointer passed to **DrawPixels**. The first element of the second row is the first bit (again as defined above) of the ubyte at location $p + k$, where $k$ is computed as

$$k = a \left\lceil \frac{l}{8a} \right\rceil \tag{3.10}$$

There is a mechanism for selecting a sub-rectangle of elements from a BITMAP image as well. Before obtaining the first element from memory, the pointer supplied to **DrawPixels** is effectively advanced by UNPACK_SKIP_ROWS $*\ k$ ubytes. Then UNPACK_SKIP_PIXELS 1-bit elements are ignored, and the subsequent *width* 1-bit elements are obtained, without advancing the ubyte pointer, after which the pointer is advanced by $k$ ubytes. *height* sets of *width* elements are obtained this way.

APPENDIX T

### Conversion to floating-point

This step applies only to groups of components. It is not performed on indices. Each element in a group is converted to a floating-point value according to the appropriate formula in table 2.6 (section 2.13). For packed pixel types, each element in the group is converted by computing $c\ /\ (2^N - 1)$, where $c$ is the unsigned integer value of the bitfield containing the element and $N$ is the number of bits in the bitfield.

### Conversion to RGB

This step is applied only if the *format* is LUMINANCE or LUMINANCE_ALPHA. If the *format* is LUMINANCE, then each group of one element is converted to a group of R, G, and B (three) elements by copying the original single element into each of the three new elements. If the *format* is LUMINANCE_ALPHA, then each group of two elements is converted to a group of R, G, B, and A (four) elements by copying the first original element into each of the first three new elements and copying the second original element to the A (fourth) new element.

### Final Expansion to RGBA

This step is performed only for non-depth component groups. Each group is converted to a group of 4 elements as follows: if a group does not contain an A element, then A is added and set to 1.0. If any of R, G, or B is missing from the group, each missing element is added and assigned a value of 0.0.

### Pixel Transfer Operations

This step is actually a sequence of steps. Because the pixel transfer operations are performed equivalently during the drawing, copying, and reading of pixels, and during the specification of texture images (either from memory or from the framebuffer), they are described separately in section 3.6.5. After the processing described in that section is completed, groups are processed as described in the following sections.

### Final Conversion

For a color index, final conversion consists of masking the bits of the index to the left of the binary point by $2^n - 1$, where $n$ is the number of bits in an index buffer. For RGBA components, each element is clamped to $[0, 1]$. The

APPENDIX T

resulting values are converted to fixed-point according to the rules given in section 2.13.9 (Final Color Processing).

For a depth component, an element is first clamped to $[0, 1]$ and then converted to fixed-point as if it were a window $z$ value (see section 2.10.1, Controlling the Viewport).

Stencil indices are masked by $2^n - 1$, where $n$ is the number of bits in the stencil buffer.

### Conversion to Fragments

The conversion of a group to fragments is controlled with

`void` **PixelZoom**( `float` $z_x$, `float` $z_y$ );

Let $(x_{rp}, y_{rp})$ be the current raster position (section 2.12). (If the current raster position is invalid, then **DrawPixels** is ignored; pixel transfer operations do not update the histogram or minmax tables, and no fragments are generated. However, the histogram and minmax tables are updated even if the corresponding fragments are later rejected by the pixel ownership (section 4.1.1) or scissor (section 4.1.2) tests.) If a particular group (index or components) is the $n$th in a row and belongs to the $m$th row, consider the region in window coordinates bounded by the rectangle with corners

$$(x_{rp} + z_x n, y_{rp} + z_y m) \qquad \text{and} \qquad (x_{rp} + z_x(n+1), y_{rp} + z_y(m+1))$$

(either $z_x$ or $z_y$ may be negative). Any fragments whose centers lie inside of this rectangle (or on its bottom or left boundaries) are produced in correspondence with this particular group of elements.

A fragment arising from a group consisting of color data takes on the color index or color components of the group; the depth and texture coordinates are taken from the current raster position's associated data. A fragment arising from a depth component takes the component's depth value; the color and texture coordinates are given by those associated with the current raster position. In both cases texture coordinates $s$, $t$, and $r$ are replaced with $s/q$, $t/q$, and $r/q$, respectively. If $q$ is less than or equal to zero, the results are undefined. Groups arising from **DrawPixels** with a *format* of `STENCIL_INDEX` are treated specially and are described in section 4.3.1.

## 3.6.5 Pixel Transfer Operations

The GL defines four kinds of pixel groups:

Microsoft et al. Exhibit 1005

1. *RGBA component:* Each group comprises four color components: red, green, blue, and alpha.
2. *Depth component:* Each group comprises a single depth component.
3. *Color index:* Each group comprises a single color index.
4. *Stencil index:* Each group comprises a single stencil index.

Each operation described in this section is applied sequentially to each pixel group in an image. Many operations are applied only to pixel groups of certain kinds; if an operation is not applicable to a given group, it is skipped.

### Arithmetic on Components

This step applies only to RGBA component and depth component groups. Each component is multiplied by an appropriate signed scale factor: RED_SCALE for an R component, GREEN_SCALE for a G component, BLUE_SCALE for a B component, and ALPHA_SCALE for an A component, or DEPTH_SCALE for a depth component. Then the result is added to the appropriate signed bias: RED_BIAS, GREEN_BIAS, BLUE_BIAS, ALPHA_BIAS, or DEPTH_BIAS.

### Arithmetic on Indices

This step applies only to color index and stencil index groups. If the index is a floating-point value, it is converted to fixed-point, with an unspecified number of bits to the right of the binary point and at least $\lceil \log_2(\texttt{MAX\_PIXEL\_MAP\_TABLE}) \rceil$ bits to the left of the binary point. Indices that are already integers remain so; any fraction bits in the resulting fixed-point value are zero.

The fixed-point index is then shifted by $|\texttt{INDEX\_SHIFT}|$ bits, left if INDEX_SHIFT $> 0$ and right otherwise. In either case the shift is zero-filled. Then, the signed integer offset INDEX_OFFSET is added to the index.

### RGBA to RGBA Lookup

This step applies only to RGBA component groups, and is skipped if MAP_COLOR is FALSE. First, each component is clamped to the range $[0, 1]$. There is a table associated with each of the R, G, B, and A component elements: PIXEL_MAP_R_TO_R for R, PIXEL_MAP_G_TO_G for G, PIXEL_MAP_B_TO_B for B, and PIXEL_MAP_A_TO_A for A. Each element is multiplied by an integer one less than the size of the corresponding table, and, for each element, an address is found by rounding this value to the nearest integer. For each element, the addressed value in the corresponding table replaces the element.

### Color Index Lookup

This step applies only to color index groups. If the GL command that invokes the pixel transfer operation requires that RGBA component pixel groups be generated, then a conversion is performed at this step. RGBA component pixel groups are required if

1. The groups will be rasterized, and the GL is in RGBA mode, or
2. The groups will be loaded as an image into texture memory, or
3. The groups will be returned to client memory with a format other than `COLOR_INDEX`.

If RGBA component groups are required, then the integer part of the index is used to reference 4 tables of color components: `PIXEL_MAP_I_TO_R`, `PIXEL_MAP_I_TO_G`, `PIXEL_MAP_I_TO_B`, and `PIXEL_MAP_I_TO_A`. Each of these tables must have $2^n$ entries for some integer value of $n$ ($n$ may be different for each table). For each table, the index is first rounded to the nearest integer; the result is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The indexed value becomes an R, G, B, or A value, as appropriate. The group of four elements so obtained replaces the index, changing the group's type to RGBA component.

If RGBA component groups are not required, and if `MAP_COLOR` is enabled, then the index is looked up in the `PIXEL_MAP_I_TO_I` table (otherwise, the index is not looked up). Again, the table must have $2^n$ entries for some integer $n$. The index is first rounded to the nearest integer; the result is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The value in the table replaces the index. The floating-point table value is first rounded to a fixed-point value with unspecified precision. The group's type remains color index.

### Stencil Index Lookup

This step applies only to stencil index groups. If `MAP_STENCIL` is enabled, then the index is looked up in the `PIXEL_MAP_S_TO_S` table (otherwise, the index is not looked up). The table must have $2^n$ entries for some integer $n$. The integer index is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The integer value in the table replaces the index.

APPENDIX T

| Base Internal Format | R | G | B | A |
|---|---|---|---|---|
| ALPHA | | | | $A_t$ |
| LUMINANCE | $L_t$ | $L_t$ | $L_t$ | |
| LUMINANCE_ALPHA | $L_t$ | $L_t$ | $L_t$ | $A_t$ |
| INTENSITY | $I_t$ | $I_t$ | $I_t$ | $I_t$ |
| RGB | $R_t$ | $G_t$ | $B_t$ | |
| RGBA | $R_t$ | $G_t$ | $B_t$ | $A_t$ |

Table 3.13: Color table lookup. $R_t$, $G_t$, $B_t$, $A_t$, $L_t$, and $I_t$ are color table values that are assigned to pixel components $R$, $G$, $B$, and $A$ depending on the table format. When there is no assignment, the component value is left unchanged by lookup.

### Color Table Lookup

This step applies only to RGBA component groups. Color table lookup is only done if COLOR_TABLE is enabled. If a zero-width table is enabled, no lookup is performed.

The internal format of the table determines which components of the group will be replaced (see table 3.13). The components to be replaced are converted to indices by clamping to $[0, 1]$, multiplying by an integer one less than the width of the table, and rounding to the nearest integer. Components are replaced by the table entry at the index.

The required state is one bit indicating whether color table lookup is enabled or disabled. In the initial state, lookup is disabled.

### Convolution

This step applies only to RGBA component groups. If CONVOLUTION_1D is enabled, the one-dimensional convolution filter is applied only to the one-dimensional texture images passed to **TexImage1D**, **TexSubImage1D**, **CopyTexImage1D**, and **CopyTexSubImage1D**, and returned by **GetTexImage** (see section 6.1.4) with target TEXTURE_1D. If CONVOLUTION_2D is enabled, the two-dimensional convolution filter is applied only to the two-dimensional images passed to **DrawPixels**, **CopyPixels**, **ReadPixels**, **TexImage2D**, **TexSubImage2D**, **CopyTexImage2D**, **CopyTexSubImage2D**, and **CopyTexSubImage3D**, and returned by **GetTexImage** with target TEXTURE_2D. If SEPARABLE_2D is enabled, and CONVOLUTION_2D is disabled, the separable two-dimensional convolution filter is instead ap-

| Base Filter Format | R | G | B | A |
|---|---|---|---|---|
| ALPHA | $R_s$ | $G_s$ | $B_s$ | $A_s * A_f$ |
| LUMINANCE | $R_s * L_f$ | $G_s * L_f$ | $B_s * L_f$ | $A_s$ |
| LUMINANCE_ALPHA | $R_s * L_f$ | $G_s * L_f$ | $B_s * L_f$ | $A_s * A_f$ |
| INTENSITY | $R_s * I_f$ | $G_s * I_f$ | $B_s * I_f$ | $A_s * I_f$ |
| RGB | $R_s * R_f$ | $G_s * G_f$ | $B_s * B_f$ | $A_s$ |
| RGBA | $R_s * R_f$ | $G_s * G_f$ | $B_s * B_f$ | $A_s * A_f$ |

Table 3.14: Computation of filtered color components depending on filter image format. $C * F$ indicates the convolution of image component $C$ with filter $F$.

plied these images.

The convolution operation is a sum of products of source image pixels and convolution filter pixels. Source image pixels always have four components: red, green, blue, and alpha, denoted in the equations below as $R_s$, $G_s$, $B_s$, and $A_s$. Filter pixels may be stored in one of five formats, with 1, 2, 3, or 4 components. These components are denoted as $R_f$, $G_f$, $B_f$, $A_f$, $L_f$, and $I_f$ in the equations below. The result of the convolution operation is the 4-tuple R,G,B,A. Depending on the internal format of the filter, individual color components of each source image pixel are convolved with one filter component, or are passed unmodified. The rules for this are defined in table 3.14.

The convolution operation is defined differently for each of the three convolution filters. The variables $W_f$ and $H_f$ refer to the dimensions of the convolution filter. The variables $W_s$ and $H_s$ refer to the dimensions of the source pixel image.

The convolution equations are defined as follows, where $C$ refers to the filtered result, $C_f$ refers to the one- or two-dimensional convolution filter, and $C_{row}$ and $C_{column}$ refer to the two one-dimensional filters comprising the two-dimensional separable filter. $C'_s$ depends on the source image color $C_s$ and the convolution border mode as described below. $C_r$, the filtered output image, depends on all of these variables and is described separately for each border mode. The pixel indexing nomenclature is decribed in the **Convolution Filter Specification** subsection of section 3.6.3.

**One-dimensional filter:**

$$C[i'] = \sum_{n=0}^{W_f-1} C'_s[i'+n] * C_f[n]$$

APPENDIX T

**Two-dimensional filter:**

$$C[i',j'] = \sum_{n=0}^{W_f-1} \sum_{m=0}^{H_f-1} C'_s[i'+n, j'+m] * C_f[n,m]$$

**Two-dimensional separable filter:**

$$C[i',j'] = \sum_{n=0}^{W_f-1} \sum_{m=0}^{H_f-1} C'_s[i'+n, j'+m] * C_{row}[n] * C_{column}[m]$$

If $W_f$ of a one-dimensional filter is zero, then $C[i]$ is always set to zero. Likewise, if either $W_f$ or $H_f$ of a two-dimensional filter is zero, then $C[i,j]$ is always set to zero.

The convolution border mode for a specific convolution filter is specified by calling

void **ConvolutionParameter{if}**( enum *target*,
  enum *pname*, T *param* );

where *target* is the name of the filter, *pname* is CONVOLUTION_BORDER_MODE, and *param* is one of REDUCE, CONSTANT_BORDER or REPLICATE_BORDER.

### Border Mode REDUCE

The width and height of source images convolved with border mode REDUCE are reduced by $W_f - 1$ and $H_f - 1$, respectively. If this reduction would generate a resulting image with zero or negative width and/or height, the output is simply null, with no error generated. The coordinates of the image that results from a convolution with border mode REDUCE are zero through $W_s - W_f$ in width, and zero through $H_s - H_f$ in height. In cases where errors can result from the specification of invalid image dimensions, it is these resulting dimensions that are tested, not the dimensions of the source image. (A specific example is **TexImage1D** and **TexImage2D**, which specify constraints for image dimensions. Even if **TexImage1D** or **TexImage2D** is called with a null pixel pointer, the dimensions of the resulting texture image are those that would result from the convolution of the specified image).

When the border mode is REDUCE, $C'_s$ equals the source image color $C_s$ and $C_r$ equals the filtered result $C$.

For the remaining border modes, define $C_w = \lfloor W_f/2 \rfloor$ and $C_h = \lfloor H_f/2 \rfloor$. The coordinates $(C_w, C_h)$ define the center of the convolution filter.

APPENDIX T

### Border Mode CONSTANT_BORDER

If the convolution border mode is CONSTANT_BORDER, the output image has the same dimensions as the source image. The result of the convolution is the same as if the source image were surrounded by pixels with the same color as the current convolution border color. Whenever the convolution filter extends beyond one of the edges of the source image, the constant-color border pixels are used as input to the filter. The current convolution border color is set by calling **ConvolutionParameterfv** or **ConvolutionParameteriv** with *pname* set to CONVOLUTION_BORDER_COLOR and *params* containing four values that comprise the RGBA color to be used as the image border. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to -1.0. Floating point color components are not clamped when they are specified.

For a one-dimensional filter, the result color is defined by

$$C_r[i] = C[i - C_w]$$

where $C[i']$ is computed using the following equation for $C'_s[i']$:

$$C'_s[i'] = \begin{cases} C_s[i'], & 0 \le i' < W_s \\ C_c, & otherwise \end{cases}$$

and $C_c$ is the convolution border color.

For a two-dimensional or two-dimensional separable filter, the result color is defined by

$$C_r[i,j] = C[i - C_w, j - C_h]$$

where $C[i',j']$ is computed using the following equation for $C'_s[i',j']$:

$$C'_s[i',j'] = \begin{cases} C_s[i',j'], & 0 \le i' < W_s, 0 \le j' < H_s \\ C_c, & otherwise \end{cases}$$

### Border Mode REPLICATE_BORDER

The convolution border mode REPLICATE_BORDER also produces an output image with the same dimensions as the source image. The behavior of this mode is identical to that of the CONSTANT_BORDER mode except for the treatment of pixel locations where the convolution filter extends beyond the edge of the source image. For these locations, it is as if the outermost one-pixel border of the source image was replicated. Conceptually, each pixel in

APPENDIX T

the leftmost one-pixel column of the source image is replicated $C_w$ times to provide additional image data along the left edge, each pixel in the rightmost one-pixel column is replicated $C_w$ times to provide additional image data along the right edge, and each pixel value in the top and bottom one-pixel rows is replicated to create $C_h$ rows of image data along the top and bottom edges. The pixel value at each corner is also replicated in order to provide data for the convolution operation at each corner of the source image.

For a one-dimensional filter, the result color is defined by

$$C_r[i] = C[i - C_w]$$

where $C[i']$ is computed using the following equation for $C'_s[i']$:

$$C'_s[i'] = C_s[\mathrm{clamp}(i', W_s)]$$

and the clamping function $\mathrm{clamp}(val, max)$ is defined as

$$\mathrm{clamp}(val, max) = \begin{cases} 0, & val < 0 \\ val, & 0 \le val < max \\ max - 1, & val >= max \end{cases}$$

For a two-dimensional or two-dimensional separable filter, the result color is defined by

$$C_r[i, j] = C[i - C_w, j - C_h]$$

where $C[i', j']$ is computed using the following equation for $C'_s[i', j']$:

$$C'_s[i', j'] = C_s[\mathrm{clamp}(i', W_s), \mathrm{clamp}(j', H_s)]$$

After convolution, each component of the resulting image is scaled by the corresponding **PixelTransfer** parameters: `POST_CONVOLUTION_RED_SCALE` for an R component, `POST_CONVOLUTION_GREEN_SCALE` for a G component, `POST_CONVOLUTION_BLUE_SCALE` for a B component, and `POST_CONVOLUTION_ALPHA_SCALE` for an A component. The result is added to the corresponding bias: `POST_CONVOLUTION_RED_BIAS`, `POST_CONVOLUTION_GREEN_BIAS`, `POST_CONVOLUTION_BLUE_BIAS`, or `POST_CONVOLUTION_ALPHA_BIAS`.

The required state is three bits indicating whether each of one-dimensional, two-dimensional, or separable two-dimensional convolution is enabled or disabled, an integer describing the current convolution border mode, and four floating-point values specifying the convolution border color. In the initial state, all convolution operations are disabled, the border mode is `REDUCE`, and the border color is $(0, 0, 0, 0)$.

### Post Convolution Color Table Lookup

This step applies only to RGBA component groups. Post convolution color table lookup is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant `POST_CONVOLUTION_COLOR_TABLE`. The post convolution table is defined by calling **ColorTable** with a *target* argument of `POST_CONVOLUTION_COLOR_TABLE`. In all other respects, operation is identical to color table lookup, as defined earlier in section 3.6.5.

The required state is one bit indicating whether post convolution table lookup is enabled or disabled. In the initial state, lookup is disabled.

### Color Matrix Transformation

This step applies only to RGBA component groups. The components are transformed by the color matrix. Each transformed component is multiplied by an appropriate signed scale factor: `POST_COLOR_MATRIX_RED_SCALE` for an R component, `POST_COLOR_MATRIX_GREEN_SCALE` for a G component, `POST_COLOR_MATRIX_BLUE_SCALE` for a B component, and `POST_COLOR_MATRIX_ALPHA_SCALE` for an A component. The result is added to a signed bias: `POST_COLOR_MATRIX_RED_BIAS`, `POST_COLOR_MATRIX_GREEN_BIAS`, `POST_COLOR_MATRIX_BLUE_BIAS`, or `POST_COLOR_MATRIX_ALPHA_BIAS`. The resulting components replace each component of the original group.

That is, if $M_c$ is the color matrix, a subscript of $s$ represents the scale term for a component, and a subscript of $b$ represents the bias term, then the components

$$\begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}$$

are transformed to

$$\begin{pmatrix} R' \\ G' \\ B' \\ A' \end{pmatrix} = \begin{pmatrix} R_s & 0 & 0 & 0 \\ 0 & G_s & 0 & 0 \\ 0 & 0 & B_s & 0 \\ 0 & 0 & 0 & A_s \end{pmatrix} M_c \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} + \begin{pmatrix} R_b \\ G_b \\ B_b \\ A_b \end{pmatrix}.$$

### Post Color Matrix Color Table Lookup

This step applies only to RGBA component groups. Post color matrix color table lookup is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant POST_COLOR_MATRIX_COLOR_TABLE. The post color matrix table is defined by calling **ColorTable** with a *target* argument of POST_COLOR_MATRIX_COLOR_TABLE. In all other respects, operation is identical to color table lookup, as defined in section 3.6.5.

The required state is one bit indicating whether post color matrix lookup is enabled or disabled. In the initial state, lookup is disabled.

### Histogram

This step applies only to RGBA component groups. Histogram operation is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant HISTOGRAM.

If the width of the table is non-zero, then indices $R_i$, $G_i$, $B_i$, and $A_i$ are derived from the red, green, blue, and alpha components of each pixel group (without modifying these components) by clamping each component to $[0, 1]$ , multiplying by one less than the width of the histogram table, and rounding to the nearest integer. If the format of the HISTOGRAM table includes red or luminance, the red or luminance component of histogram entry $R_i$ is incremented by one. If the format of the HISTOGRAM table includes green, the green component of histogram entry $G_i$ is incremented by one. The blue and alpha components of histogram entries $B_i$ and $A_i$ are incremented in the same way. If a histogram entry component is incremented beyond its maximum value, its value becomes undefined; this is not an error.

If the **Histogram** *sink* parameter is FALSE, histogram operation has no effect on the stream of pixel groups being processed. Otherwise, all RGBA pixel groups are discarded immediately after the histogram operation is completed. Because histogram precedes minmax, no minmax operation is performed. No pixel fragments are generated, no change is made to texture memory contents, and no pixel values are returned. However, texture object state is modified whether or not pixel groups are discarded.

### Minmax

This step applies only to RGBA component groups. Minmax operation is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant MINMAX.

If the format of the minmax table includes red or luminance, the red component value replaces the red or luminance value in the minimum table element if and only if it is less than that component. Likewise, if the format includes red or luminance and the red component of the group is greater

APPENDIX T

than the red or luminance value in the maximum element, the red group component replaces the red or luminance maximum component. If the format of the table includes green, the green group component conditionally replaces the green minimum and/or maximum if it is smaller or larger, respectively. The blue and alpha group components are similarly tested and replaced, if the table format includes blue and/or alpha. The internal type of the minimum and maximum component values is floating point, with at least the same representable range as a floating point number used to represent colors (section 2.1.1). There are no semantics defined for the treatment of group component values that are outside the representable range.

If the **Minmax** *sink* parameter is `FALSE`, minmax operation has no effect on the stream of pixel groups being processed. Otherwise, all RGBA pixel groups are discarded immediately after the minmax operation is completed. No pixel fragments are generated, no change is made to texture memory contents, and no pixel values are returned. However, texture object state is modified whether or not pixel groups are discarded.

## 3.7 Bitmaps

Bitmaps are rectangles of zeros and ones specifying a particular pattern of fragments to be produced. Each of these fragments has the same associated data. These data are those associated with the *current raster position.*

Bitmaps are sent using

> `void` **Bitmap**( `sizei` $w$, `sizei` $h$, `float` $x_{bo}$, `float` $y_{bo}$,
> `float` $x_{bi}$, `float` $y_{bi}$, `ubyte` **data* );

$w$ and $h$ comprise the integer width and height of the rectangular bitmap, respectively. $(x_{bo}, y_{bo})$ gives the floating-point $x$ and $y$ values of the bitmap's origin. $(x_{bi}, y_{bi})$ gives the floating-point $x$ and $y$ increments that are added to the raster position after the bitmap is rasterized. *data* is a pointer to a bitmap.

Like a polygon pattern, a bitmap is unpacked from memory according to the procedure given in section 3.6.4 for **DrawPixels**; it is as if the *width* and *height* passed to that command were equal to $w$ and $h$, respectively, the *type* were `BITMAP`, and the *format* were `COLOR_INDEX`. The unpacked values (before any conversion or arithmetic would have been performed) form a stipple pattern of zeros and ones. See figure 3.9.

A bitmap sent using **Bitmap** is rasterized as follows. First, if the current raster position is invalid (the valid bit is reset), the bitmap is ignored.

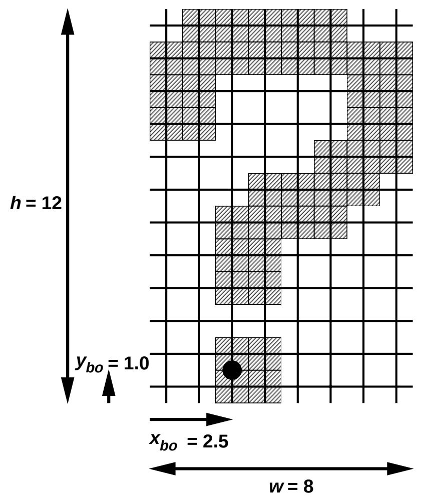


Figure 3.9. A bitmap and its associated parameters. $x_{bi}$ and $y_{bi}$ are not shown.

Otherwise, a rectangular array of fragments is constructed, with lower left corner at

$$(x_{ll}, y_{ll}) = (\lfloor x_{rp} - x_{bo} \rfloor, \lfloor y_{rp} - y_{bo} \rfloor)$$

and upper right corner at $(x_{ll} + w, y_{ll} + h)$ where $w$ and $h$ are the width and height of the bitmap, respectively. Fragments in the array are produced if the corresponding bit in the bitmap is 1 and not produced otherwise. The associated data for each fragment are those associated with the current raster position, with texture coordinates $s$, $t$, and $r$ replaced with $s/q$, $t/q$, and $r/q$, respectively. If $q$ is less than or equal to zero, the results are undefined. Once the fragments have been produced, the current raster position is updated:

$$(x_{rp}, y_{rp}) \leftarrow (x_{rp} + x_{bi}, y_{rp} + y_{bi}).$$

The $z$ and $w$ values of the current raster position remain unchanged.

## 3.8 Texturing

Texturing maps a portion of a specified image onto each primitive for which texturing is enabled. This mapping is accomplished by using the color of

APPENDIX T

an image at the location indicated by a fragment's $(s, t, r)$ coordinates to modify the fragment's primary RGBA color. Texturing does not affect the secondary color.

Texturing is specified only for RGBA mode; its use in color index mode is undefined.

The GL provides a means to specify the details of how texturing of a primitive is effected. These details include specification of the image to be texture mapped, the means by which the image is filtered when applied to the primitive, and the function that determines what RGBA value is produced given a fragment color and an image value.

### 3.8.1 Texture Image Specification

The command

```
void TexImage3D( enum target, int level,
    int internalformat, sizei width, sizei height,
    sizei depth, int border, enum format, enum type,
    void *data );
```

is used to specify a three-dimensional texture image. *target* must be either TEXTURE_3D, or PROXY_TEXTURE_3D in the special case discussed in section 3.8.7. *format*, *type*, and *data* match the corresponding arguments to **DrawPixels** (refer to section 3.6.4); they specify the format of the image data, the type of those data, and a pointer to the image data in host memory. The *format*s STENCIL_INDEX and DEPTH_COMPONENT are not allowed.

The groups in memory are treated as being arranged in a sequence of adjacent rectangles. Each rectangle is a two-dimensional image, whose size and organization are specified by the *width* and *height* parameters to **TexImage3D**. The values of UNPACK_ROW_LENGTH and UNPACK_ALIGNMENT control the row-to-row spacing in these images in the same manner as **DrawPixels**. If the value of the integer parameter UNPACK_IMAGE_HEIGHT is not positive, then the number of rows in each two-dimensional image is *height*; otherwise the number of rows is UNPACK_IMAGE_HEIGHT. Each two-dimensional image comprises an integral number of rows, and is exactly adjacent to its neighbor images.

The mechanism for selecting a sub-volume of a three-dimensional image relies on the integer parameter UNPACK_SKIP_IMAGES. If UNPACK_SKIP_IMAGES is positive, the pointer is advanced by UNPACK_SKIP_IMAGES times the number of elements in one two-dimensional image before obtaining the first group from memory. Then *depth* two-dimensional images are processed, each having a subimage extracted in the same manner as **DrawPixels**.

The selected groups are processed exactly as for **DrawPixels**, stopping just before final conversion. Each R, G, B, and A value so generated is clamped to $[0, 1]$.

Components are then selected from the resulting R, G, B, and A values to obtain a texture with the *base internal format* specified by (or derived from) *internalformat.* Table 3.15 summarizes the mapping of R, G, B, and A values to texture components, as a function of the base internal format of the texture image. *internalformat* may be specified as one of the six base internal format symbolic constants listed in table 3.15, or as one of the *sized internal format* symbolic constants listed in table 3.16. *internalformat* may (for backwards compatibility with the 1.0 version of the GL) also take on the integer values `1`, `2`, `3`, and `4`, which are equivalent to symbolic constants `LUMINANCE`, `LUMINANCE_ALPHA`, `RGB`, and `RGBA` respectively. Specifying a value for *internalformat* that is not one of the above values generates the error `INVALID_VALUE`.

The *internal component resolution* is the number of bits allocated to each value in a texture image. If *internalformat* is specified as a base internal format, the GL stores the resulting texture with internal component resolutions of its own choosing. If a sized internal format is specified, the mapping of the R, G, B, and A values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 3.15, and the memory allocation per texture component is assigned by the GL to match the allocations listed in table 3.16 as closely as possible. (The definition of closely is left up to the implementation. Implementations are not required to support more than one resolution for each base internal format.)

A GL implementation may vary its allocation of internal component resolution based on any **TexImage3D**, **TexImage2D** (see below), or **TexImage1D** (see below) parameter (except *target*), but the allocation must not be a function of any other state, and cannot be changed once it is established. Allocations must be invariant; the same allocation must be made each time a texture image is specified with the same parameter values. These allocation rules also apply to proxy textures, which are described in section 3.8.7.

The image itself (pointed to by *data*) is a sequence of groups of values. The first group is the lower left back corner of the texture image. Subsequent groups fill out rows of width *width* from left to right; *height* rows are stacked from bottom to top forming a single two-dimensional image slice; and *depth* slices are stacked from back to front. When the final R, G, B,

| Base Internal Format | RGBA Values | Internal Components |
|---|---|---|
| ALPHA | A | $A$ |
| LUMINANCE | R | $L$ |
| LUMINANCE_ALPHA | R,A | $L$,$A$ |
| INTENSITY | R | $I$ |
| RGB | R,G,B | $R$,$G$,$B$ |
| RGBA | R,G,B,A | $R$,$G$,$B$,$A$ |

Table 3.15: Conversion from RGBA pixel components to internal texture, table, or filter components. See section 3.8.9 for a description of the texture components $R$, $G$, $B$, $A$, $L$, and $I$.

and A components have been computed for a group, they are assigned to components of a *texel* as described by table 3.15. Counting from zero, each resulting $N$th texel is assigned internal integer coordinates $(i, j, k)$, where

$$i = (N \bmod width) - b_s$$

$$j = (\lfloor \frac{N}{width} \rfloor \bmod height) - b_s$$

$$k = (\lfloor \frac{N}{width \times height} \rfloor \bmod depth) - b_s$$

and $b_s$ is the specified *border* width. Thus the last two-dimensional image slice of the three-dimensional image is indexed with the highest value of $k$.

Each color component is converted (by rounding to nearest) to a fixed-point value with $n$ bits, where $n$ is the number of bits of storage allocated to that component in the image array. We assume that the fixed-point representation used represents each value $k/(2^n - 1)$, where $k \in \{0, 1, \ldots, 2^n - 1\}$, as $k$ (e.g. 1.0 is represented in binary as a string of all ones).

The *level* argument to **TexImage3D** is an integer *level-of-detail* number. Levels of detail are discussed below, under **Mipmapping**. The main texture image has a level of detail number of 0. If a level-of-detail less than zero is specified, the error INVALID_VALUE is generated.

The *border* argument to **TexImage3D** is a border width. The significance of borders is described below. The border width affects the required dimensions of the texture image: it must be the case that

$$w_s = 2^n + 2b_s \tag{3.11}$$

APPENDIX T

| Sized<br>Internal Format | Base<br>Internal Format | *R*<br>bits | *G*<br>bits | *B*<br>bits | *A*<br>bits | *L*<br>bits | *I*<br>bits |
|---|---|---|---|---|---|---|---|
| ALPHA4 | ALPHA | | | | 4 | | |
| ALPHA8 | ALPHA | | | | 8 | | |
| ALPHA12 | ALPHA | | | | 12 | | |
| ALPHA16 | ALPHA | | | | 16 | | |
| LUMINANCE4 | LUMINANCE | | | | | 4 | |
| LUMINANCE8 | LUMINANCE | | | | | 8 | |
| LUMINANCE12 | LUMINANCE | | | | | 12 | |
| LUMINANCE16 | LUMINANCE | | | | | 16 | |
| LUMINANCE4_ALPHA4 | LUMINANCE_ALPHA | | | | 4 | 4 | |
| LUMINANCE6_ALPHA2 | LUMINANCE_ALPHA | | | | 2 | 6 | |
| LUMINANCE8_ALPHA8 | LUMINANCE_ALPHA | | | | 8 | 8 | |
| LUMINANCE12_ALPHA4 | LUMINANCE_ALPHA | | | | 4 | 12 | |
| LUMINANCE12_ALPHA12 | LUMINANCE_ALPHA | | | | 12 | 12 | |
| LUMINANCE16_ALPHA16 | LUMINANCE_ALPHA | | | | 16 | 16 | |
| INTENSITY4 | INTENSITY | | | | | | 4 |
| INTENSITY8 | INTENSITY | | | | | | 8 |
| INTENSITY12 | INTENSITY | | | | | | 12 |
| INTENSITY16 | INTENSITY | | | | | | 16 |
| R3_G3_B2 | RGB | 3 | 3 | 2 | | | |
| RGB4 | RGB | 4 | 4 | 4 | | | |
| RGB5 | RGB | 5 | 5 | 5 | | | |
| RGB8 | RGB | 8 | 8 | 8 | | | |
| RGB10 | RGB | 10 | 10 | 10 | | | |
| RGB12 | RGB | 12 | 12 | 12 | | | |
| RGB16 | RGB | 16 | 16 | 16 | | | |
| RGBA2 | RGBA | 2 | 2 | 2 | 2 | | |
| RGBA4 | RGBA | 4 | 4 | 4 | 4 | | |
| RGB5_A1 | RGBA | 5 | 5 | 5 | 1 | | |
| RGBA8 | RGBA | 8 | 8 | 8 | 8 | | |
| RGB10_A2 | RGBA | 10 | 10 | 10 | 2 | | |
| RGBA12 | RGBA | 12 | 12 | 12 | 12 | | |
| RGBA16 | RGBA | 16 | 16 | 16 | 16 | | |

Table 3.16: Correspondence of sized internal formats to base internal formats, and *desired* component resolutions for each sized internal format.

APPENDIX T

$$h_s = 2^m + 2b_s \tag{3.12}$$

$$d_s = 2^l + 2b_s \tag{3.13}$$

for some integers $n$, $m$, and $l$, where $w_s$, $h_s$, and $d_s$ are the specified image *width*, *height*, and *depth*. If any one of these relationships cannot be satisfied, then the error INVALID_VALUE is generated.

Currently, the maximum border width $b_t$ is 1. If $b_s$ is less than zero, or greater than $b_t$, then the error INVALID_VALUE is generated.

The maximum allowable width, height, or depth of a three-dimensional texture image is an implementation dependent function of the level-of-detail and internal format of the resulting image array. It must be at least $2^{k-lod} + 2b_t$ for image arrays of level-of-detail 0 through $k$, where $k$ is the log base 2 of MAX_3D_TEXTURE_SIZE, *lod* is the level-of-detail of the image array, and $b_t$ is the maximum border width. It may be zero for image arrays of any level-of-detail greater than $k$. The error INVALID_VALUE is generated if the specified image is too large to be stored under any conditions.

In a similar fashion, the maximum allowable width of a one- or two-dimensional texture image, and the maximum allowable height of a two-dimensional texture image, must be at least $2^{k-lod} + 2b_t$ for image arrays of level 0 through $k$, where $k$ is the log base 2 of MAX_TEXTURE_SIZE.

Furthermore, an implementation may allow a one-, two-, or three-dimensional image array of level 1 or greater to be created only if a complete[1] set of image arrays consistent with the requested array can be supported. Likewise, an implementation may allow an image array of level 0 to be created only if that single image array can be supported.

The command

```
void TexImage2D( enum target, int level,
    int internalformat, sizei width, sizei height,
    int border, enum format, enum type, void *data );
```

is used to specify a two-dimensional texture image. *target* must be either TEXTURE_2D, or PROXY_TEXTURE_2D in the special case discussed in section 3.8.7. The other parameters match the corresponding parameters of **TexImage3D**.

[1] For this purpose the definition of "complete", as provided under **Mipmapping**, is augmented as follows: 1) it is as though TEXTURE_BASE_LEVEL is 0 and TEXTURE_MAX_LEVEL is 1000. 2) Excluding borders, the dimensions of the next lower numbered array are all understood to be twice the corresponding dimensions of the specified array.

Microsoft et al. Exhibit 1005

APPENDIX T

For the purposes of decoding the texture image, **TexImage2D** is equivalent to calling **TexImage3D** with corresponding arguments and *depth* of 1, except that

- The *depth* of the image is always 1 regardless of the value of *border*.
- Convolution will be performed on the image (possibly changing its *width* and *height*) if SEPARABLE_2D or CONVOLUTION_2D is enabled.
- UNPACK_SKIP_IMAGES is ignored.

Finally, the command

```
void TexImage1D( enum target, int level,
    int internalformat, sizei width, int border,
    enum format, enum type, void *data );
```

is used to specify a one-dimensional texture image. *target* must be either TEXTURE_1D, or PROXY_TEXTURE_1D in the special case discussed in section 3.8.7.)

For the purposes of decoding the texture image, **TexImage1D** is equivalent to calling **TexImage2D** with corresponding arguments and *height* of 1, except that

- The *height* of the image is always 1 regardless of the value of *border*.
- Convolution will be performed on the image (possibly changing its *width*) only if CONVOLUTION_1D is enabled.

An image with zero width, height (**TexImage2D** and **TexImage3D** only), or depth (**TexImage3D** only) indicates the null texture. If the null texture is specified for the level-of-detail specified by TEXTURE_BASE_LEVEL, it is as if texturing were disabled.

The image indicated to the GL by the image pointer is decoded and copied into the GL's internal memory. This copying effectively places the decoded image inside a border of the maximum allowable width $b_t$ whether or not a border has been specified (see figure 3.10) [2]. If no border or a border smaller than the maximum allowable width has been specified, then the image is still stored as if it were surrounded by a border of the maximum possible width. Any excess border (which surrounds the specified image,

[2] Figure 3.10 needs to show a three-dimensional texture image.

including any border) is assigned unspecified values. A two-dimensional texture has a border only at its left, right, top, and bottom ends, and a one-dimensional texture has a border only at its left and right ends.

We shall refer to the (possibly border augmented) decoded image as the *texture array.* A three-dimensional texture array has width, height, and depth

$$w_t = 2^n + 2b_t$$

$$h_t = 2^m + 2b_t$$

$$d_t = 2^l + 2b_t$$

where $b_t$ is the maximum allowable border width and $n$, $m$, and $l$ are defined in equations 3.11, 3.12, and 3.13. A two-dimensional texture array has depth $d_t = 1$, with height $h_t$ and width $w_t$ as above, and a one-dimensional texture array has depth $d_t = 1$, height $h_t = 1$, and width $w_t$ as above.

An element $(i, j, k)$ of the texture array is called a *texel* (for a two-dimensional texture, $k$ is irrelevant; for a one-dimensional texture, $j$ and $k$ are both irrelevant). The *texture value* used in texturing a fragment is determined by that fragment's associated $(s, t, r)$ coordinates, but may not correspond to any actual texel. See figure 3.10.

If the *data* argument of **TexImage1D**, **TexImage2D**, or **TexImage3D** is a null pointer (a zero-valued pointer in the C implementation), a one-, two-, or three-dimensional texture array is created with the specified *target*, *level*, *internalformat*, *width*, *height*, and *depth*, but with unspecified image contents. In this case no pixel values are accessed in client memory, and no pixel processing is performed. Errors are generated, however, exactly as though the *data* pointer were valid.

### 3.8.2 Alternate Texture Image Specification Commands

Two-dimensional and one-dimensional texture images may also be specified using image data taken directly from the framebuffer, and rectangular subregions of existing texture images may be respecified.

The command

```
void CopyTexImage2D( enum target, int level,
    enum internalformat, int x, int y, sizei width,
    sizei height, int border );
```

APPENDIX T

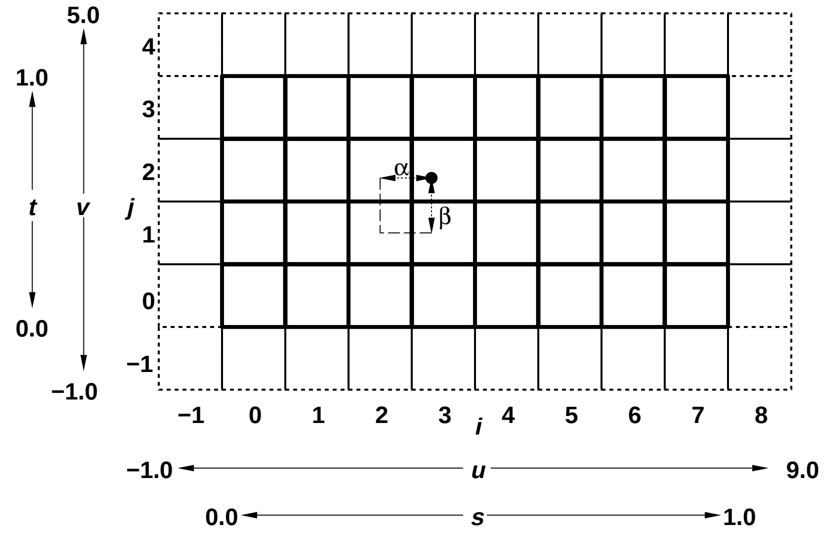


Figure 3.10. A texture image and the coordinates used to access it. This is a two-dimensional texture with $n = 3$ and $m = 2$. A one-dimensional texture would consist of a single horizontal strip. $\alpha$ and $\beta$, values used in blending adjacent texels to obtain a texture value, are also shown.

APPENDIX T

defines a two-dimensional texture array in exactly the manner of **TexImage2D**, except that the image data are taken from the framebuffer rather than from client memory. Currently, *target* must be TEXTURE_2D. $x$, $y$, *width*, and *height* correspond precisely to the corresponding arguments to **CopyPixels** (refer to section 4.3.3); they specify the image's *width* and *height*, and the lower left $(x, y)$ coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **CopyPixels**, with argument *type* set to COLOR, stopping after pixel transfer processing is complete. Subsequent processing is identical to that described for **TexImage2D**, beginning with clamping of the R, G, B, and A values from the resulting pixel groups. Parameters *level*, *internalformat*, and *border* are specified using the same values, with the same meanings, as the equivalent arguments of **TexImage2D**, except that *internalformat* may not be specified as 1, 2, 3, or 4. An invalid value specified for *internalformat* generates the error INVALID_ENUM. The constraints on *width*, *height*, and *border* are exactly those for the equivalent arguments of **TexImage2D**.

The command

```
void CopyTexImage1D( enum target, int level,
    enum internalformat, int x, int y, sizei width,
    int border );
```

defines a one-dimensional texture array in exactly the manner of **TexImage1D**, except that the image data are taken from the framebuffer, rather than from client memory. Currently, *target* must be TEXTURE_1D. For the purposes of decoding the texture image, **CopyTexImage1D** is equivalent to calling **CopyTexImage2D** with corresponding arguments and *height* of 1, except that the *height* of the image is always 1, regardless of the value of *border*. *level*, *internalformat*, and *border* are specified using the same values, with the same meanings, as the equivalent arguments of **TexImage1D**, except that *internalformat* may not be specified as 1, 2, 3, or 4. The constraints on *width* and *border* are exactly those of the equivalent arguments of **TexImage1D**.

Six additional commands,

```
void TexSubImage3D( enum target, int level, int xoffset,
    int yoffset, int zoffset, sizei width, sizei height,
    sizei depth, enum format, enum type, void *data );
void TexSubImage2D( enum target, int level, int xoffset,
    int yoffset, sizei width, sizei height, enum format,
    enum type, void *data );
```

```
void TexSubImage1D( enum target, int level, int xoffset,
   sizei width, enum format, enum type, void *data );
void CopyTexSubImage3D( enum target, int level,
   int xoffset, int yoffset, int zoffset, int x, int y,
   sizei width, sizei height );
void CopyTexSubImage2D( enum target, int level,
   int xoffset, int yoffset, int x, int y, sizei width,
   sizei height );
void CopyTexSubImage1D( enum target, int level,
   int xoffset, int x, int y, sizei width );
```

respecify only a rectangular subregion of an existing texture array. No change is made to the *internalformat*, *width*, *height*, *depth*, or *border* parameters of the specified texture array, nor is any change made to texel values outside the specified subregion. Currently the *target* arguments of **TexSubImage1D** and **CopyTexSubImage1D** must be TEXTURE_1D, the *target* arguments of **TexSubImage2D** and **CopyTexSubImage2D** must be TEXTURE_2D, and the *target* arguments of **TexSubImage3D** and **CopyTexSubImage3D** must be TEXTURE_3D. The *level* parameter of each command specifies the level of the texture array that is modified. If *level* is less than zero or greater than the base 2 logarithm of the maximum texture width or height, the error INVALID_VALUE is generated.

**TexSubImage3D** arguments *width*, *height*, *depth*, *format*, *type*, and *data* match the corresponding arguments to **TexImage3D**, meaning that they are specified using the same values, and have the same meanings. Likewise, **TexSubImage2D** arguments *width*, *height*, *format*, *type*, and *data* match the corresponding arguments to **TexImage2D**, and **TexSubImage1D** arguments *width*, *format*, *type*, and *data* match the corresponding arguments to **TexImage1D**.

**CopyTexSubImage3D** and **CopyTexSubImage2D** arguments $x$, $y$, *width*, and *height* match the corresponding arguments to **CopyTexImage2D**[3]. **CopyTexSubImage1D** arguments $x$, $y$, and *width* match the corresponding arguments to **CopyTexImage1D**. Each of the **TexSubImage** commands interprets and processes pixel groups in exactly the manner of its **TexImage** counterpart, except that the assignment of R, G, B, and A pixel group values to the texture components is controlled by the *internalformat* of the texture array, not by an argument to the command.

[3] Because the framebuffer is inherently two-dimensional, there is no **CopyTexImage3D** command.

Arguments *xoffset*, *yoffset*, and *zoffset* of **TexSubImage3D** and **CopyTexSubImage3D** specify the lower left texel coordinates of a *width*-wide by *height*-high by *depth*-deep rectangular subregion of the texture array. The *depth* argument associated with **CopyTexSubImage3D** is always 1, because framebuffer memory is two-dimensional - only a portion of a single $s, t$ slice of a three-dimensional texture is replaced by **CopyTexSubImage3D**.

Negative values of *xoffset*, *yoffset*, and *zoffset* correspond to the coordinates of border texels, addressed as in figure 3.10. Taking $w_s$, $h_s$, $d_s$, and $b_s$ to be the specified width, height, depth, and border width of the texture array, (not the actual array dimensions $w_t$, $h_t$, $d_t$, and $b_t$), and taking $x$, $y$, $z$, $w$, $h$, and $d$ to be the *xoffset*, *yoffset*, *zoffset*, *width*, *height*, and *depth* argument values, any of the following relationships generates the error `INVALID_VALUE`:

$$x < -b_s$$

$$x + w > w_s - b_s$$

$$y < -b_s$$

$$y + h > h_s - b_s$$

$$z < -b_s$$

$$z + d > d_s - b_s$$

(Recall that $d_s$, $w_s$, and $h_s$ include twice the specified border width $b_s$.) Counting from zero, the $n$th pixel group is assigned to the texel with internal integer coordinates $[i, j, k]$, where

$$i = x + (n \bmod w)$$

$$j = y + (\lfloor \frac{n}{w} \rfloor \bmod h)$$

$$k = z + (\lfloor \frac{n}{width * height} \rfloor \bmod d$$

Arguments *xoffset* and *yoffset* of **TexSubImage2D** and **CopyTexSubImage2D** specify the lower left texel coordinates of a *width*-wide by *height*-high rectangular subregion of the texture array. Negative values of *xoffset* and *yoffset* correspond to the coordinates of border texels, addressed as in figure 3.10. Taking $w_s$, $h_s$, and $b_s$ to be the specified width, height, and border width of the texture array, (not the actual array dimensions $w_t$, $h_t$, and $b_t$), and taking $x$, $y$, $w$, and $h$ to be the *xoffset*, *yoffset*, *width*, and *height* argument values, any of the following relationships generates the error `INVALID_VALUE`:

$$x < -b_s$$
$$x + w > w_s - b_s$$
$$y < -b_s$$
$$y + h > h_s - b_s$$

(Recall that $w_s$ and $h_s$ include twice the specified border width $b_s$.) Counting from zero, the $n$th pixel group is assigned to the texel with internal integer coordinates $[i, j]$, where

$$i = x + (n \bmod w)$$
$$j = y + (\lfloor \frac{n}{w} \rfloor \bmod h)$$

The *xoffset* argument of **TexSubImage1D** and **CopyTexSubImage1D** specifies the left texel coordinate of a *width*-wide subregion of the texture array. Negative values of *xoffset* correspond to the coordinates of border texels. Taking $w_s$ and $b_s$ to be the specified width and border width of the texture array, and $x$ and $w$ to be the *xoffset* and *width* argument values, either of the following relationships generates the error `INVALID_VALUE`:

$$x < -b_s$$
$$x + w > w_s - b_s$$

Counting from zero, the $n$th pixel group is assigned to the texel with internal integer coordinates $[i]$, where

$$i = x + (n \bmod w)$$

### 3.8.3 Texture Parameters

Various parameters control how the texture array is treated when applied to a fragment. Each parameter is set by calling

```
void TexParameter{if}( enum target, enum pname,
   T param );
void TexParameter{if}v( enum target, enum pname,
   T params );
```

| Name | Type | Legal Values |
|---|---|---|
| TEXTURE_WRAP_S | integer | CLAMP, CLAMP_TO_EDGE, REPEAT |
| TEXTURE_WRAP_T | integer | CLAMP, CLAMP_TO_EDGE, REPEAT |
| TEXTURE_WRAP_R | integer | CLAMP, CLAMP_TO_EDGE, REPEAT |
| TEXTURE_MIN_FILTER | integer | NEAREST,<br>LINEAR,<br>NEAREST_MIPMAP_NEAREST,<br>NEAREST_MIPMAP_LINEAR,<br>LINEAR_MIPMAP_NEAREST,<br>LINEAR_MIPMAP_LINEAR, |
| TEXTURE_MAG_FILTER | integer | NEAREST,<br>LINEAR |
| TEXTURE_BORDER_COLOR | 4 floats | any 4 values in $[0,1]$ |
| TEXTURE_PRIORITY | float | any value in $[0,1]$ |
| TEXTURE_MIN_LOD | float | any value |
| TEXTURE_MAX_LOD | float | any value |
| TEXTURE_BASE_LEVEL | integer | any non-negative integer |
| TEXTURE_MAX_LEVEL | integer | any non-negative integer |

Table 3.17: Texture parameters and their values.

*target* is the target, either TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D. *pname* is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 3.17. In the first form of the command, *param* is a value to which to set a single-valued parameter; in the second form of the command, *params* is an array of parameters whose type depends on the parameter being set. If the values for TEXTURE_BORDER_COLOR are specified as integers, the conversion for signed integers from table 2.6 is applied to convert the values to floating-point. Each of the four values set by TEXTURE_BORDER_COLOR is clamped to lie in $[0,1]$.

### 3.8.4 Texture Wrap Modes

If TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R is set to REPEAT, then the GL ignores the integer part of $s$, $t$, or $r$ coordinates, respectively, using only the fractional part. (For a number $f$, the fractional part is $f - \lfloor f \rfloor$, regardless of the sign of $f$; recall that the *floor* function truncates towards $-\infty$.) CLAMP causes $s$, $t$, or $r$ coordinates to be clamped to the range $[0,1]$.

The initial state is for all of $s$, $t$, and $r$ behavior to be that given by `REPEAT`.

`CLAMP_TO_EDGE` clamps texture coordinates at all mipmap levels such that the texture filter never samples a border texel. The color returned when clamping is derived only from texels at the edge of the texture image.

Texture coordinates are clamped to the range $[min, max]$. The minimum value is defined as

$$min = \frac{1}{2N}$$

where $N$ is the size of the one-, two-, or three-dimensional texture image in the direction of clamping. The maximum value is defined as

$$max = 1 - min$$

so that clamping is always symmetric about the $[0, 1]$ mapped range of a texture coordinate.

### 3.8.5 Texture Minification

Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment. In the GL this mapping is approximated by one of two simple filtering schemes. One of these schemes is selected based on whether the mapping from texture space to framebuffer space is deemed to *magnify* or *minify* the texture image.

#### Scale Factor and Level of Detail

The choice is governed by a scale factor $\rho(x, y)$ and the *level of detail* parameter $\lambda(x, y)$, defined as

$$\lambda'(x, y) = \log_2[\rho(x, y)]$$

$$\lambda = \begin{cases} \texttt{TEXTURE\_MAX\_LOD}, & \lambda' > \texttt{TEXTURE\_MAX\_LOD} \\ \lambda', & \texttt{TEXTURE\_MIN\_LOD} \leq \lambda' \leq \texttt{TEXTURE\_MAX\_LOD} \\ \texttt{TEXTURE\_MIN\_LOD}, & \lambda' < \texttt{TEXTURE\_MIN\_LOD} \\ undefined, & \texttt{TEXTURE\_MIN\_LOD} > \texttt{TEXTURE\_MAX\_LOD} \end{cases} \quad (3.14)$$

APPENDIX T

If $\lambda(x, y)$ is less than or equal to the constant $c$ (described below in section 3.8.6) the texture is said to be magnified; if it is greater, the texture is minified.

The initial values of `TEXTURE_MIN_LOD` and `TEXTURE_MAX_LOD` are chosen so as to never clamp the normal range of $\lambda$. They may be respecified for a specific texture by calling **TexParameter[if]**.

Let $s(x, y)$ be the function that associates an $s$ texture coordinate with each set of window coordinates $(x, y)$ that lie within a primitive; define $t(x, y)$ and $r(x, y)$ analogously. Let $u(x, y) = 2^n s(x, y)$, $v(x, y) = 2^m t(x, y)$, and $w(x, y) = 2^l r(x, y)$, where $n$, $m$, and $l$ are as defined by equations 3.11, 3.12, and 3.13 with $w_s$, $h_s$, and $d_s$ equal to the width, height, and depth of the image array whose level is `TEXTURE_BASE_LEVEL`. For a one-dimensional texture, define $v(x, y) \equiv 0$ and $w(x, y) \equiv 0$; for a two-dimensional texture, define $w(x, y) \equiv 0$. For a polygon, $\rho$ is given at a fragment with window coordinates $(x, y)$ by

$$\rho = \max\left\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y}\right)^2}\right\} \tag{3.15}$$

where $\partial u/\partial x$ indicates the derivative of $u$ with respect to window $x$, and similarly for the other derivatives.

For a line, the formula is

$$\rho = \sqrt{\left(\frac{\partial u}{\partial x}\Delta x + \frac{\partial u}{\partial y}\Delta y\right)^2 + \left(\frac{\partial v}{\partial x}\Delta x + \frac{\partial v}{\partial y}\Delta y\right)^2 + \left(\frac{\partial w}{\partial x}\Delta x + \frac{\partial w}{\partial y}\Delta y\right)^2} \Big/ l, \tag{3.16}$$

where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$ with $(x_1, y_1)$ and $(x_2, y_2)$ being the segment's window coordinate endpoints and $l = \sqrt{\Delta x^2 + \Delta y^2}$. For a point, pixel rectangle, or bitmap, $\rho \equiv 1$.

While it is generally agreed that equations 3.15 and 3.16 give the best results when texturing, they are often impractical to implement. Therefore, an implementation may approximate the ideal $\rho$ with a function $f(x, y)$ subject to these conditions:

1. $f(x, y)$ is continuous and monotonically increasing in each of $|\partial u/\partial x|$, $|\partial u/\partial y|$, $|\partial v/\partial x|$, $|\partial v/\partial y|$, $|\partial w/\partial x|$, and $|\partial w/\partial y|$
2. Let
$$m_u = \max\left\{\left|\frac{\partial u}{\partial x}\right|, \left|\frac{\partial u}{\partial y}\right|\right\}$$

$$m_v = \max\left\{\left|\frac{\partial v}{\partial x}\right|, \left|\frac{\partial v}{\partial y}\right|\right\}$$

$$m_w = \max\left\{\left|\frac{\partial w}{\partial x}\right|, \left|\frac{\partial w}{\partial y}\right|\right\}.$$

Then $\max\{m_u, m_v, m_w\} \leq f(x,y) \leq m_u + m_v + m_w$.

When $\lambda$ indicates minification, the value assigned to `TEXTURE_MIN_FILTER` is used to determine how the texture value for a fragment is selected. When `TEXTURE_MIN_FILTER` is `NEAREST`, the texel in the image array of level `TEXTURE_BASE_LEVEL` that is nearest (in Manhattan distance) to that specified by $(s, t, r)$ is obtained. This means the texel at location $(i, j, k)$ becomes the texture value, with $i$ given by

$$i = \begin{cases} \lfloor u \rfloor, & s < 1 \\ 2^n - 1, & s = 1 \end{cases} \tag{3.17}$$

(Recall that if `TEXTURE_WRAP_S` is `REPEAT`, then $0 \leq s < 1$.) Similarly, $j$ is found as

$$j = \begin{cases} \lfloor v \rfloor, & t < 1 \\ 2^m - 1, & t = 1 \end{cases} \tag{3.18}$$

and $k$ is found as

$$k = \begin{cases} \lfloor w \rfloor, & r < 1 \\ 2^l - 1, & r = 1 \end{cases} \tag{3.19}$$

For a one-dimensional texture, $j$ and $k$ are irrelevant; the texel at location $i$ becomes the texture value. For a two-dimensional texture, $k$ is irrelevant; the texel at location $(i, j)$ becomes the texture value.

When `TEXTURE_MIN_FILTER` is `LINEAR`, a $2 \times 2 \times 2$ cube of texels in the image array of level `TEXTURE_BASE_LEVEL` is selected. This cube is obtained by first clamping texture coordinates as described above under **Texture Wrap Modes** (if the wrap mode for a coordinate is `CLAMP` or `CLAMP_TO_EDGE`) and computing

$$i_0 = \begin{cases} \lfloor u - 1/2 \rfloor \bmod 2^n, & \texttt{TEXTURE\_WRAP\_S}\ \text{is}\ \texttt{REPEAT} \\ \lfloor u - 1/2 \rfloor, & \textit{otherwise} \end{cases}$$

$$j_0 = \begin{cases} \lfloor v - 1/2 \rfloor \bmod 2^m, & \texttt{TEXTURE\_WRAP\_T} \text{ is } \texttt{REPEAT} \\ \lfloor v - 1/2 \rfloor, & \textit{otherwise} \end{cases}$$

and

$$k_0 = \begin{cases} \lfloor w - 1/2 \rfloor \bmod 2^l, & \texttt{TEXTURE\_WRAP\_R} \text{ is } \texttt{REPEAT} \\ \lfloor w - 1/2 \rfloor, & \textit{otherwise} \end{cases}$$

Then

$$i_1 = \begin{cases} (i_0 + 1) \bmod 2^n, & \texttt{TEXTURE\_WRAP\_S} \text{ is } \texttt{REPEAT} \\ i_0 + 1, & \textit{otherwise} \end{cases}$$

$$j_1 = \begin{cases} (j_0 + 1) \bmod 2^m, & \texttt{TEXTURE\_WRAP\_T} \text{ is } \texttt{REPEAT} \\ j_0 + 1, & \textit{otherwise} \end{cases}$$

and

$$k_1 = \begin{cases} (k_0 + 1) \bmod 2^l, & \texttt{TEXTURE\_WRAP\_R} \text{ is } \texttt{REPEAT} \\ k_0 + 1, & \textit{otherwise} \end{cases}$$

Let

$$\alpha = \operatorname{frac}(u - 1/2)$$
$$\beta = \operatorname{frac}(v - 1/2)$$
$$\gamma = \operatorname{frac}(w - 1/2)$$

where $\operatorname{frac}(x)$ denotes the fractional part of $x$.

For a three-dimensional texture, the texture value $\tau$ is found as

$$\begin{aligned} \tau \;=\; & (1-\alpha)(1-\beta)(1-\gamma)\tau_{i_0j_0k_0} + \alpha(1-\beta)(1-\gamma)\tau_{i_1j_0k_0} \\ & + (1-\alpha)\beta(1-\gamma)\tau_{i_0j_1k_0} + \alpha\beta(1-\gamma)\tau_{i_1j_1k_0} \\ & + (1-\alpha)(1-\beta)\gamma\tau_{i_0j_0k_1} + \alpha(1-\beta)\gamma\tau_{i_1j_0k_1} \\ & + (1-\alpha)\beta\gamma\tau_{i_0j_1k_1} + \alpha\beta\gamma\tau_{i_1j_1k_1} \end{aligned}$$

where $\tau_{ijk}$ is the texel at location $(i, j, k)$ in the three-dimensional texture image.

For a two-dimensional texture,

APPENDIX T

$$\tau = (1-\alpha)(1-\beta)\tau_{i_0 j_0} + \alpha(1-\beta)\tau_{i_1 j_0} + (1-\alpha)\beta\tau_{i_0 j_1} + \alpha\beta\tau_{i_1 j_1} \qquad (3.20)$$

where $\tau_{ij}$ is the texel at location $(i, j)$ in the two-dimensional texture image.

And for a one-dimensional texture,

$$\tau = (1-\alpha)\tau_{i_0} + \alpha\tau_{i_1}$$

where $\tau_i$ is the texel at location $i$ in the one-dimensional texture.

If any of the selected $\tau_{ijk}$, $\tau_{ij}$, or $\tau_i$ in the above equations refer to a border texel with $i < -b_s$, $j < -b_s$, $k < -b_s$, $i \geq w_s - b_s$, $j \geq h_s - b_s$, or $j \geq d_s - b_s$, then the border color given by the current setting of TEXTURE_BORDER_COLOR is used instead of the unspecified value or values. The RGBA values of the TEXTURE_BORDER_COLOR are interpreted to match the texture's internal format in a manner consistent with table 3.15.

### Mipmapping

TEXTURE_MIN_FILTER values NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, and LINEAR_MIPMAP_LINEAR each require the use of a *mipmap*. A mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one. If the image array of level TEXTURE_BASE_LEVEL, excluding its border, has dimensions $2^n \times 2^m \times 2^l$, then there are $\max\{n, m, l\} + 1$ image arrays in the mipmap. Each array subsequent to the array of level TEXTURE_BASE_LEVEL has dimensions

$$\sigma(i-1) \times \sigma(j-1) \times \sigma(k-1)$$

where the dimensions of the previous array are

$$\sigma(i) \times \sigma(j) \times \sigma(k)$$

and

$$\sigma(x) = \begin{cases} 2^x & x > 0 \\ 1 & x \leq 0 \end{cases}$$

until the last array is reached with dimension $1 \times 1 \times 1$.

Each array in a mipmap is defined using **TexImage3D**, **TexImage2D**, **CopyTexImage2D**, **TexImage1D**, or **CopyTexImage1D**; the array being set is indicated with the level-of-detail argument *level*. Level-of-detail numbers proceed from TEXTURE_BASE_LEVEL for the original texture array

APPENDIX T

through $p = \max\{n, m, l\} + \mathtt{TEXTURE\_BASE\_LEVEL}$ with each unit increase indicating an array of half the dimensions of the previous one as already described. If texturing is enabled (and TEXTURE_MIN_FILTER is one that requires a mipmap) at the time a primitive is rasterized and if the set of arrays TEXTURE_BASE_LEVEL through $q = \min\{p, \mathtt{TEXTURE\_MAX\_LEVEL}\}$ is incomplete, then it is as if texture mapping were disabled. The set of arrays TEXTURE_BASE_LEVEL through $q$ is incomplete if the internal formats of all the mipmap arrays were not specified with the same symbolic constant, if the border widths of the mipmap arrays are not the same, if the dimensions of the mipmap arrays do not follow the sequence described above, if $\mathtt{TEXTURE\_MAX\_LEVEL} < \mathtt{TEXTURE\_BASE\_LEVEL}$, or if $\mathtt{TEXTURE\_BASE\_LEVEL} > p$. Array levels $k$ where $k < \mathtt{TEXTURE\_BASE\_LEVEL}$ or $k > q$ are insignificant.

The values of TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL may be respecified for a specific texture by calling **TexParameter[if]**. The error INVALID_VALUE is generated if either value is negative.

The mipmap is used in conjunction with the level of detail to approximate the application of an appropriately filtered texture to a fragment. Let $c$ be the value of $\lambda$ at which the transition from minification to magnification occurs (since this discussion pertains to minification, we are concerned only with values of $\lambda$ where $\lambda > c$). In the following equations, let

$b = \mathtt{TEXTURE\_BASE\_LEVEL}$

For mipmap filters NEAREST_MIPMAP_NEAREST and LINEAR_MIPMAP_NEAREST, the $d$th mipmap array is selected, where

$$d = \begin{cases} b, & \lambda \le \frac{1}{2} \\ \lceil b + \lambda + \frac{1}{2} \rceil - 1, & \lambda > \frac{1}{2}, b + \lambda \le q + \frac{1}{2} \\ q, & \lambda > \frac{1}{2}, b + \lambda > q + \frac{1}{2} \end{cases} \tag{3.21}$$

The rules for NEAREST or LINEAR filtering are then applied to the selected array.

For mipmap filters NEAREST_MIPMAP_LINEAR and LINEAR_MIPMAP_LINEAR, the level $d_1$ and $d_2$ mipmap arrays are selected, where

$$d_1 = \begin{cases} q, & b + \lambda \ge q \\ \lfloor b + \lambda \rfloor, & otherwise \end{cases} \tag{3.22}$$

$$d_2 = \begin{cases} q, & b + \lambda \ge q \\ d_1 + 1, & otherwise \end{cases} \tag{3.23}$$

The rules for NEAREST or LINEAR filtering are then applied to each of the selected arrays, yielding two corresponding texture values $\tau_1$ and $\tau_2$. The final texture value is then found as

Microsoft et al. Exhibit 1005

APPENDIX T

$$\tau = [1 - \mathrm{frac}(\lambda)]\tau_1 + \mathrm{frac}(\lambda)\tau_2.$$

### 3.8.6 Texture Magnification

When $\lambda$ indicates magnification, the value assigned to TEXTURE_MAG_FILTER determines how the texture value is obtained. There are two possible values for TEXTURE_MAG_FILTER: NEAREST and LINEAR. NEAREST behaves exactly as NEAREST for TEXTURE_MIN_FILTER (equations 3.17, 3.18, and 3.19 are used); LINEAR behaves exactly as LINEAR for TEXTURE_MIN_FILTER (equation 3.20 is used). The level-of-detail TEXTURE_BASE_LEVEL texture array is always used for magnification.

Finally, there is the choice of $c$, the minification vs. magnification switch-over point. If the magnification filter is given by LINEAR and the minification filter is given by NEAREST_MIPMAP_NEAREST or NEAREST_MIPMAP_LINEAR, then $c = 0.5$. This is done to ensure that a minified texture does not appear "sharper" than a magnified texture. Otherwise $c = 0$.

### 3.8.7 Texture State and Proxy State

The state necessary for texture can be divided into two categories. First, there are the three sets of mipmap arrays (one-, two-, and three-dimensional) and their number. Each array has associated with it a width, height (two- or three-dimensional only), and depth (three-dimensional only), a border width, an integer describing the internal format of the image, and six integer values describing the resolutions of each of the red, green, blue, alpha, luminance, and intensity components of the image. Each initial texture array is null (zero width, height, and depth, zero border width, internal format 1, with zero-sized components). Next, there are the two sets of texture properties; each consists of the selected minification and magnification filters, the wrap modes for $s$, $t$ (two- and three-dimensional only), and $r$ (three-dimensional only), the TEXTURE_BORDER_COLOR, two integers describing the minimum and maximum level of detail, two integers describing the base and maximum mipmap array, a boolean flag indicating whether the texture is resident and the priority associated with each set of properties. The value of the resident flag is determined by the GL and may change as a result of other GL operations. The flag may only be queried, not set, by applications. See section 3.8.8). In the initial state, the value assigned to TEXTURE_MIN_FILTER is NEAREST_MIPMAP_LINEAR, and the value for TEXTURE_MAG_FILTER is LINEAR. $s$, $t$, and $r$ wrap modes are all set to REPEAT.

The values of TEXTURE_MIN_LOD and TEXTURE_MAX_LOD are -1000 and 1000 respectively. The values of TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL are 0 and 1000 respectively. TEXTURE_PRIORITY is 1.0, and TEXTURE_BORDER_COLOR is (0,0,0,0). The initial value of TEXTURE_RESIDENT is determined by the GL.

In addition to the one-, two-, and three-dimensional sets of image arrays, partially instantiated one-, two-, and three-dimensional sets of proxy image arrays are maintained. Each proxy array includes width, height (two- and three-dimensional arrays only), depth (three-dimensional arrays only), border width, and internal format state values, as well as state for the red, green, blue, alpha, luminance, and intensity component resolutions. Proxy arrays do not include image data, nor do they include texture properties. When **TexImage3D** is executed with *target* specified as PROXY_TEXTURE_3D, the three-dimensional proxy state values of the specified level-of-detail are recomputed and updated. If the image array would not be supported by **TexImage3D** called with *target* set to TEXTURE_3D, no error is generated, but the proxy width, height, depth, border width, and component resolutions are set to zero. If the image array would be supported by such a call to **TexImage3D**, the proxy state values are set exactly as though the actual image array were being specified. No pixel data are transferred or processed in either case.

One- and two-dimensional proxy arrays are operated on in the same way when **TexImage1D** is executed with *target* specified as PROXY_TEXTURE_1D, or **TexImage2D** is executed with *target* specified as PROXY_TEXTURE_2D.

There is no image associated with any of the proxy textures. Therefore PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, and PROXY_TEXTURE_3D cannot be used as textures, and their images must never be queried using **GetTexImage**. The error INVALID_ENUM is generated if this is attempted. Likewise, there is no nonlevel-related state associated with a proxy texture, and **GetTexParameteriv** or **GetTexParameterfv** may not be called with a proxy texture *target*. The error INVALID_ENUM is generated if this is attempted.

### 3.8.8 Texture Objects

In addition to the default textures TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D named one-, two-, and three-dimensional texture objects can be created and operated upon. The name space for texture objects is the unsigned integers, with zero reserved by the GL.

A texture object is created by *binding* an unused name to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D. The binding is effected by calling

APPENDIX T

void **BindTexture**( enum *target*, uint *texture* );

with *target* set to the desired texture target and *texture* set to the unused name. The resulting texture object is a new state vector, comprising all the state values listed in section 3.8.7, set to the same initial values. If the new texture object is bound to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D respectively, it is and remains a one-, two-, or three-dimensional texture until it is deleted.

**BindTexture** may also be used to bind an existing texture object to either TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D. The error INVALID_OPERATION is generated if an attempt is made to bind a texture object of different dimensionality than the specified *target*. If the bind is successful no change is made to the state of the bound texture object, and any previous binding to *target* is broken.

While a texture object is bound, GL operations on the target to which it is bound affect the bound object, and queries of the target to which it is bound return state from the bound object. If texture mapping of the dimensionality of the target to which a texture object is bound is enabled, the state of the bound texture object directs the texturing operation.

In the initial state, TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D have one-, two-, and three-dimensional texture state vectors associated with them. In order that access to these initial textures not be lost, they are treated as texture objects all of whose names are 0. The initial one-, two-, or three-dimensional texture is therefore operated upon, queried, and applied as TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D respectively while 0 is bound to the corresponding targets.

Texture objects are deleted by calling

void **DeleteTextures**( sizei *n*, uint **textures* );

*textures* contains *n* names of texture objects to be deleted. After a texture object is deleted, it has no contents or dimensionality, and its name is again unused. If a texture that is currently bound to one of the targets TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D is deleted, it is as though **BindTexture** had been executed with the same *target* and *texture* zero. Unused names in *textures* are silently ignored, as is the value zero.

The command

void **GenTextures**( sizei *n*, uint **textures* );

APPENDIX T

returns *n* previously unused texture object names in *textures*. These names are marked as used, for the purposes of **GenTextures** only, but they acquire texture state and a dimensionality only when they are first bound, just as if they were unused.

An implementation may choose to establish a working set of texture objects on which binding operations are performed with higher performance. A texture object that is currently part of the working set is said to be *resident*. The command

```
boolean AreTexturesResident( sizei n, uint *textures,
   boolean *residences );
```

returns `TRUE` if all of the *n* texture objects named in *textures* are resident, or if the implementation does not distinguish a working set. If at least one of the texture objects named in *textures* is not resident, then `FALSE` is returned, and the residence of each texture object is returned in *residences*. Otherwise the contents of *residences* are not changed. If any of the names in *textures* are unused or are zero, `FALSE` is returned, the error `INVALID_VALUE` is generated, and the contents of *residences* are indeterminate. The residence status of a single bound texture object can also be queried by calling **GetTexParameteriv** or **GetTexParameterfv** with *target* set to the target to which the texture object is bound, and *pname* set to `TEXTURE_RESIDENT`.

**AreTexturesResident** indicates only whether a texture object is currently resident, not whether it could not be made resident. An implementation may choose to make a texture object resident only on first use, for example. The client may guide the GL implementation in determining which texture objects should be resident by specifying a priority for each texture object. The command

```
void PrioritizeTextures( sizei n, uint *textures,
   clampf *priorities );
```

sets the priorities of the *n* texture objects named in *textures* to the values in *priorities*. Each priority value is clamped to the range [0,1] before it is assigned. Zero indicates the lowest priority, with the least likelihood of being resident. One indicates the highest priority, with the greatest likelihood of being resident. The priority of a single bound texture object may also be changed by calling **TexParameteri**, **TexParameterf**, **TexParameteriv**, or **TexParameterfv** with *target* set to the target to which the texture object is bound, *pname* set to `TEXTURE_PRIORITY`, and *param* or *params* specifying the new priority value (which is clamped to the range [0,1] before being assigned). **PrioritizeTextures** silently ignores attempts to prioritize unused texture object names or zero (default textures).

### 3.8.9 Texture Environments and Texture Functions

The command

```
void TexEnv{if}( enum target, enum pname, T param );
void TexEnv{if}v( enum target, enum pname, T params );
```

sets parameters of the *texture environment* that specifies how texture values are interpreted when texturing a fragment. *target* must currently be the symbolic constant TEXTURE_ENV. *pname* is a symbolic constant indicating the parameter to be set. In the first form of the command, *param* is a value to which to set a single-valued parameter; in the second form, *params* is a pointer to an array of parameters: either a single symbolic constant or a value or group of values to which the parameter should be set. The possible environment parameters are TEXTURE_ENV_MODE and TEXTURE_ENV_COLOR. TEXTURE_ENV_MODE may be set to one of REPLACE, MODULATE, DECAL, or BLEND; TEXTURE_ENV_COLOR is set to an RGBA color by providing four single-precision floating-point values in the range $[0,1]$ (values outside this range are clamped to it). If integers are provided for TEXTURE_ENV_COLOR, then they are converted to floating-point as specified in table 2.6 for signed integers.

The value of TEXTURE_ENV_MODE specifies a *texture function.* The result of this function depends on the fragment and the texture array value. The precise form of the function depends on the base internal formats of the texture arrays that were last specified. In the following two tables, $R_f$, $G_f$, $B_f$, and $A_f$ are the primary color components of the incoming fragment; $R_t$, $G_t$, $B_t$, $A_t$, $L_t$, and $I_t$ are the filtered texture values; $R_c$, $G_c$, $B_c$, and $A_c$ are the texture environment color values; and $R_v$, $G_v$, $B_v$, and $A_v$ are the primary color components computed by the texture function. All of these color values are in the range $[0,1]$. The REPLACE and MODULATE texture functions are specified in table 3.18, and the DECAL and BLEND texture functions are specified in table 3.19.

The state required for the current texture environment consists of the four-valued integer indicating the texture function and four floating-point TEXTURE_ENV_COLOR values. In the initial state, the texture function is given by MODULATE and TEXTURE_ENV_COLOR is $(0,0,0,0)$.

APPENDIX T

| Base<br>Internal Format | REPLACE<br>Texture Function | MODULATE<br>Texture Function |
|---|---|---|
| ALPHA | $R_v = R_f$<br>$G_v = G_f$<br>$B_v = B_f$<br>$A_v = A_t$ | $R_v = R_f$<br>$G_v = G_f$<br>$B_v = B_f$<br>$A_v = A_f A_t$ |
| LUMINANCE<br>(or 1) | $R_v = L_t$<br>$G_v = L_t$<br>$B_v = L_t$<br>$A_v = A_f$ | $R_v = R_f L_t$<br>$G_v = G_f L_t$<br>$B_v = B_f L_t$<br>$A_v = A_f$ |
| LUMINANCE_ALPHA<br>(or 2) | $R_v = L_t$<br>$G_v = L_t$<br>$B_v = L_t$<br>$A_v = A_t$ | $R_v = R_f L_t$<br>$G_v = G_f L_t$<br>$B_v = B_f L_t$<br>$A_v = A_f A_t$ |
| INTENSITY | $R_v = I_t$<br>$G_v = I_t$<br>$B_v = I_t$<br>$A_v = I_t$ | $R_v = R_f I_t$<br>$G_v = G_f I_t$<br>$B_v = B_f I_t$<br>$A_v = A_f I_t$ |
| RGB<br>(or 3) | $R_v = R_t$<br>$G_v = G_t$<br>$B_v = B_t$<br>$A_v = A_f$ | $R_v = R_f R_t$<br>$G_v = G_f G_t$<br>$B_v = B_f B_t$<br>$A_v = A_f$ |
| RGBA<br>(or 4) | $R_v = R_t$<br>$G_v = G_t$<br>$B_v = B_t$<br>$A_v = A_t$ | $R_v = R_f R_t$<br>$G_v = G_f G_t$<br>$B_v = B_f B_t$<br>$A_v = A_f A_t$ |

Table 3.18: Replace and modulate texture functions.

APPENDIX T

| Base<br>Internal Format | DECAL<br>Texture Function | BLEND<br>Texture Function |
|---|---|---|
| ALPHA | undefined | $R_v = R_f$<br>$G_v = G_f$<br>$B_v = B_f$<br>$A_v = A_f A_t$ |
| LUMINANCE<br>(or 1) | undefined | $R_v = R_f(1 - L_t) + R_c L_t$<br>$G_v = G_f(1 - L_t) + G_c L_t$<br>$B_v = B_f(1 - L_t) + B_c L_t$<br>$A_v = A_f$ |
| LUMINANCE_ALPHA<br>(or 2) | undefined | $R_v = R_f(1 - L_t) + R_c L_t$<br>$G_v = G_f(1 - L_t) + G_c L_t$<br>$B_v = B_f(1 - L_t) + B_c L_t$<br>$A_v = A_f A_t$ |
| INTENSITY | undefined | $R_v = R_f(1 - I_t) + R_c I_t$<br>$G_v = G_f(1 - I_t) + G_c I_t$<br>$B_v = B_f(1 - I_t) + B_c I_t$<br>$A_v = A_f(1 - I_t) + A_c I_t$ |
| RGB<br>(or 3) | $R_v = R_t$<br>$G_v = G_t$<br>$B_v = B_t$<br>$A_v = A_f$ | $R_v = R_f(1 - R_t) + R_c R_t$<br>$G_v = G_f(1 - G_t) + G_c G_t$<br>$B_v = B_f(1 - B_t) + B_c B_t$<br>$A_v = A_f$ |
| RGBA<br>(or 4) | $R_v = R_f(1 - A_t) + R_t A_t$<br>$G_v = G_f(1 - A_t) + G_t A_t$<br>$B_v = B_f(1 - A_t) + B_t A_t$<br>$A_v = A_f$ | $R_v = R_f(1 - R_t) + R_c R_t$<br>$G_v = G_f(1 - G_t) + G_c G_t$<br>$B_v = B_f(1 - B_t) + B_c B_t$<br>$A_v = A_f A_t$ |

Table 3.19: Decal and blend texture functions.

Microsoft et al. Exhibit 1005

APPENDIX T

### 3.8.10 Texture Application

Texturing is enabled or disabled using the generic **Enable** and **Disable** commands, respectively, with the symbolic constants TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D to enable the one-, two-, or three-dimensional texture, respectively. If both two- and one-dimensional textures are enabled, the two-dimensional texture is used. If the three-dimensional and either of the two- or one-dimensional textures is enabled, the three-dimensional texture is used. If all texturing is disabled, a rasterized fragment is passed on unaltered to the next stage of the GL (although its texture coordinates may be discarded). Otherwise, a texture value is found according to the parameter values of the currently bound texture image of the appropriate dimensionality using the rules given in sections 3.8.5 and 3.8.6. This texture value is used along with the incoming fragment in computing the texture function indicated by the currently bound texture environment. The result of this function replaces the incoming fragment's primary R, G, B, and A values. These are the color values passed to subsequent operations. Other data associated with the incoming fragment remain unchanged, except that the texture coordinates may be discarded.

The required state is three bits indicating whether each of one-, two-, or three-dimensional texturing is enabled or disabled. In the initial state, all texturing is disabled.

## 3.9 Color Sum

At the beginning of color sum, a fragment has two RGBA colors: a primary color $\mathbf{c}_{pri}$ (which texturing, if enabled, may have modified) and a secondary color $\mathbf{c}_{sec}$. The components of these two colors are summed to produce a single post-texturing RGBA color $\mathbf{c}$. The components of $\mathbf{c}$ are then clamped to the range $[0, 1]$.

Color sum has no effect in color index mode.

## 3.10 Fog

If enabled, fog blends a fog color with a rasterized fragment's post-texturing color using a blending factor $f$. Fog is enabled and disabled with the **Enable** and **Disable** commands using the symbolic constant FOG.

This factor $f$ is computed according to one of three equations:

$$f = \exp(-d \cdot z), \qquad (3.24)$$

$$f = \exp(-(d \cdot z)^2), \text{or} \tag{3.25}$$

$$f = \frac{e - z}{e - s} \tag{3.26}$$

($z$ is the eye-coordinate distance from the eye, $(0, 0, 0, 1)$ in eye coordinates, to the fragment center). The equation, along with either $d$ or $e$ and $s$, is specified with

```
void Fog{if}( enum pname, T param );
void Fog{if}v( enum pname, T params );
```

If *pname* is `FOG_MODE`, then *param* must be, or *params* must point to an integer that is one of the symbolic constants `EXP`, `EXP2`, or `LINEAR`, in which case equation 3.24, 3.25, or 3.26, respectively, is selected for the fog calculation (if, when 3.26 is selected, $e = s$, results are undefined). If *pname* is `FOG_DENSITY`, `FOG_START`, or `FOG_END`, then *param* is or *params* points to a value that is $d$, $s$, or $e$, respectively. If $d$ is specified less than zero, the error `INVALID_VALUE` results.

An implementation may choose to approximate the eye-coordinate distance from the eye to each fragment center by $|z_e|$. Further, $f$ need not be computed at each fragment, but may be computed at each vertex and interpolated as other data are.

No matter which equation and approximation is used to compute $f$, the result is clamped to $[0, 1]$ to obtain the final $f$.

$f$ is used differently depending on whether the GL is in RGBA or color index mode. In RGBA mode, if $C_r$ represents a rasterized fragment's R, G, or B value, then the corresponding value produced by fog is

$$C = fC_r + (1 - f)C_f.$$

(The rasterized fragment's A value is not changed by fog blending.) The R, G, B, and A values of $C_f$ are specified by calling **Fog** with *pname* equal to `FOG_COLOR`; in this case *params* points to four values comprising $C_f$. If these are not floating-point values, then they are converted to floating-point using the conversion given in table 2.6 for signed integers. Each component of $C_f$ is clamped to $[0, 1]$ when specified.

In color index mode, the formula for fog blending is

$$I = i_r + (1 - f)i_f$$

where $i_r$ is the rasterized fragment's color index and $i_f$ is a single-precision floating-point value. $(1 - f)i_f$ is rounded to the nearest fixed-point value with the same number of bits to the right of the binary point as $i_r$, and the integer portion of $I$ is masked (bitwise ANDed) with $2^n - 1$, where $n$ is the number of bits in a color in the color index buffer (buffers are discussed in chapter 4). The value of $i_f$ is set by calling **Fog** with *pname* set to `FOG_INDEX` and *param* being or *params* pointing to a single value for the fog index. The integer part of $i_f$ is masked with $2^n - 1$.

The state required for fog consists of a three valued integer to select the fog equation, three floating-point values $d$, $e$, and $s$, an RGBA fog color and a fog color index, and a single bit to indicate whether or not fog is enabled. In the initial state, fog is disabled, `FOG_MODE` is `EXP`, $d = 1.0$, $e = 1.0$, and $s = 0.0$; $C_f = (0, 0, 0, 0)$ and $i_f = 0$.

## 3.11 Antialiasing Application

Finally, if antialiasing is enabled for the primitive from which a rasterized fragment was produced, then the computed coverage value is applied to the fragment. In RGBA mode, the value is multiplied by the fragment's alpha (A) value to yield a final alpha value. In color index mode, the value is used to set the low order bits of the color index value as described in section 3.2.

APPENDIX T

# Chapter 4

# Per-Fragment Operations and the Framebuffer

The framebuffer consists of a set of pixels arranged as a two-dimensional array. The height and width of this array may vary from one GL implementation to another. For purposes of this discussion, each pixel in the framebuffer is simply a set of some number of bits. The number of bits per pixel may also vary depending on the particular GL implementation or context.

Corresponding bits from each pixel in the framebuffer are grouped together into a *bitplane*; each bitplane contains a single bit from each pixel. These bitplanes are grouped into several *logical buffers*. These are the *color*, *depth*, *stencil*, and *accumulation* buffers. The color buffer actually consists of a number of buffers: the *front left* buffer, the *front right* buffer, the *back left* buffer, the *back right* buffer, and some number of *auxiliary* buffers. Typically the contents of the front buffers are displayed on a color monitor while the contents of the back buffers are invisible. (Monoscopic contexts display only the front left buffer; stereoscopic contexts display both the front left and the front right buffers.) The contents of the auxiliary buffers are never visible. All color buffers must have the same number of bitplanes, although an implementation or context may choose not to provide right buffers, back buffers, or auxiliary buffers at all. Further, an implementation or context may not provide depth, stencil, or accumulation buffers.

Color buffers consist of either unsigned integer color indices or R, G, B, and, optionally, A unsigned integer values. The number of bitplanes in each of the color buffers, the depth buffer, the stencil buffer, and the accumulation buffer is fixed and window dependent. If an accumulation buffer is provided,

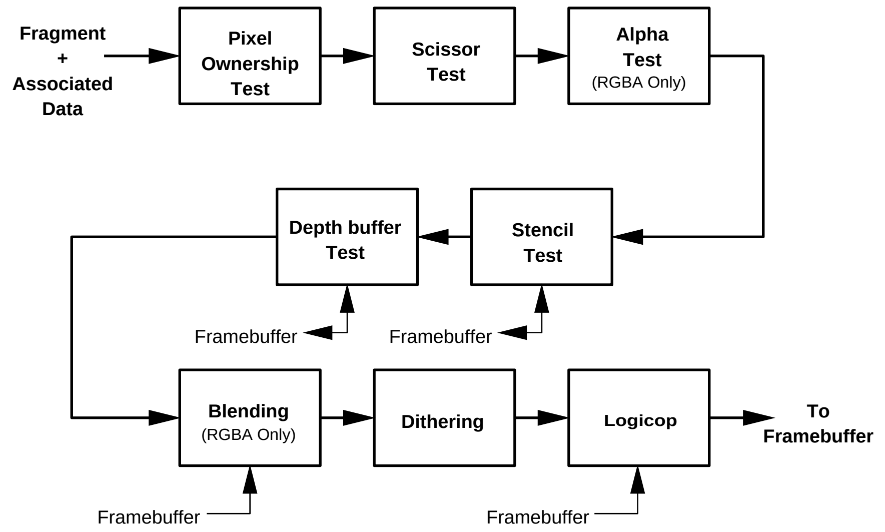


Figure 4.1. Per-fragment operations.

it must have at least as many bitplanes per R, G, and B color component as do the color buffers.

The initial state of all provided bitplanes is undefined.

## 4.1 Per-Fragment Operations

A fragment produced by rasterization with window coordinates of $(x_w, y_w)$ modifies the pixel in the framebuffer at that location based on a number of parameters and conditions. We describe these modifications and tests, diagrammed in Figure 4.1, in the order in which they are performed. Figure 4.1 diagrams these modifications and tests.

### 4.1.1 Pixel Ownership Test

The first test is to determine if the pixel at location $(x_w, y_w)$ in the framebuffer is currently owned by the GL (more precisely, by this GL context). If it is not, the window system decides the fate the incoming fragment. Possible results are that the fragment is discarded or that some subset of the subsequent per-fragment operations are applied to the fragment. This test allows the window system to control the GL's behavior, for instance, when a GL window is obscured.

### 4.1.2 Scissor test

The scissor test determines if $(x_w, y_w)$ lies within the scissor rectangle defined by four values. These values are set with

void **Scissor**( int *left*, int *bottom*, sizei *width*,
  sizei *height* );

If $left \leq x_w < left + width$ and $bottom \leq y_w < bottom + height$, then the scissor test passes. Otherwise, the test fails and the fragment is discarded. The test is enabled or disabled using **Enable** or **Disable** using the constant SCISSOR_TEST. When disabled, it is as if the scissor test always passes. If either *width* or *height* is less than zero, then the error INVALID_VALUE is generated. The state required consists of four integer values and a bit indicating whether the test is enabled or disabled. In the initial state $left = bottom = 0$; *width* and *height* are determined by the size of the GL window. Initially, the scissor test is disabled.

### 4.1.3 Alpha test

This step applies only in RGBA mode. In color index mode, proceed to the next step. The alpha test discards a fragment conditional on the outcome of a comparison between the incoming fragment's alpha value and a constant value. The comparison is enabled or disabled with the generic **Enable** and **Disable** commands using the symbolic constant ALPHA_TEST. When disabled, it is as if the comparison always passes. The test is controlled with

void **AlphaFunc**( enum *func*, clampf *ref* );

*func* is a symbolic constant indicating the alpha test function; *ref* is a reference value. *ref* is clamped to lie in $[0, 1]$, and then converted to a fixed-point value according to the rules given for an A component in section 2.13.9. For purposes of the alpha test, the fragment's alpha value is also rounded to the nearest integer. The possible constants specifying the test function are NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GEQUAL, GREATER, or NOTEQUAL, meaning pass the fragment never, always, if the fragment's alpha value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the reference value, respectively.

The required state consists of the floating-point reference value, an eight-valued integer indicating the comparison function, and a bit indicating if the comparison is enabled or disabled. The initial state is for the reference value to be 0 and the function to be ALWAYS. Initially, the alpha test is disabled.

### 4.1.4 Stencil test

The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer at location $(x_w, y_w)$ and a reference value. The test is controlled with

void **StencilFunc**( enum *func*, int *ref*, uint *mask* );
void **StencilOp**( enum *sfail*, enum *dpfail*, enum *dppass* );

The test is enabled or disabled with the **Enable** and **Disable** commands, using the symbolic constant STENCIL_TEST. When disabled, the stencil test and associated modifications are not made, and the fragment is always passed.

*ref* is an integer reference value that is used in the unsigned stencil comparison. It is clamped to the range $[0, 2^s - 1]$, where $s$ is the number of bits in the stencil buffer. *func* is a symbolic constant that determines the stencil comparison function; the eight symbolic constants are NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GEQUAL, GREATER, or NOTEQUAL. Accordingly, the stencil test passes never, always, if the reference value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the masked stored value in the stencil buffer. The $s$ least significant bits of *mask* are bitwise ANDed with both the reference and the stored stencil value. The ANDed values are those that participate in the comparison.

**StencilOp** takes three arguments that indicate what happens to the stored stencil value if this or certain subsequent tests fail or pass. *sfail* indicates what action is taken if the stencil test fails. The symbolic constants are KEEP, ZERO, REPLACE, INCR, DECR, and INVERT. These correspond to keeping the current value, setting it to zero, replacing it with the reference value, incrementing it, decrementing it, or bitwise inverting it. For purposes of increment and decrement, the stencil bits are considered as an unsigned integer; values clamp at 0 and the maximum representable value. The same symbolic values are given to indicate the stencil action if the depth buffer test (below) fails (*dpfail*), or if it passes (*dppass*).

If the stencil test fails, the incoming fragment is discarded. The state required consists of the most recent values passed to **StencilFunc** and **StencilOp**, and a bit indicating whether stencil testing is enabled or disabled.

APPENDIX T

In the initial state, stenciling is disabled, the stencil reference value is zero, the stencil comparison function is ALWAYS, and the stencil *mask* is all ones. Initially, all three stencil operations are KEEP. If there is no stencil buffer, no stencil modification can occur, and it is as if the stencil tests always pass, regardless of any calls to **StencilOp**.

### 4.1.5 Depth buffer test

The depth buffer test discards the incoming fragment if a depth comparison fails. The comparison is enabled or disabled with the generic **Enable** and **Disable** commands using the symbolic constant DEPTH_TEST. When disabled, the depth comparison and subsequent possible updates to the depth buffer value are bypassed and the fragment is passed to the next operation. The stencil value, however, is modified as indicated below as if the depth buffer test passed. If enabled, the comparison takes place and the depth buffer and stencil value may subsequently be modified.

The comparison is specified with

void **DepthFunc**( enum *func* );

This command takes a single symbolic constant: one of NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GREATER, GEQUAL, NOTEQUAL. Accordingly, the depth buffer test passes never, always, if the incoming fragment's $z_w$ value is less than, less than or equal to, equal to, greater than, greater than or equal to, or not equal to the depth value stored at the location given by the incoming fragment's $(x_w, y_w)$ coordinates.

If the depth buffer test fails, the incoming fragment is discarded. The stencil value at the fragment's $(x_w, y_w)$ coordinates is updated according to the function currently in effect for depth buffer test failure. Otherwise, the fragment continues to the next operation and the value of the depth buffer at the fragment's $(x_w, y_w)$ location is set to the fragment's $z_w$ value. In this case the stencil value is updated according to the function currently in effect for depth buffer test success.

The necessary state is an eight-valued integer and a single bit indicating whether depth buffering is enabled or disabled. In the initial state the function is LESS and the test is disabled.

If there is no depth buffer, it is as if the depth buffer test always passes.

APPENDIX T

### 4.1.6 Blending

Blending combines the incoming fragment's R, G, B, and A values with the R, G, B, and A values stored in the framebuffer at the incoming fragment's $(x_w, y_w)$ location.

This blending is dependent on the incoming fragment's alpha value and that of the corresponding currently stored pixel. Blending applies only in RGBA mode; in color index mode it is bypassed. Blending is enabled or disabled using **Enable** or **Disable** with the symbolic constant `BLEND`. If it is disabled, or if logical operation on color values is enabled (section 4.1.8), proceed to the next stage.

In the following discussion, $C_s$ refers to the source color for an incoming fragment, $C_d$ refers to the destination color at the corresponding framebuffer location, and $C_c$ refers to a constant color in the GL state. Individual RGBA components of these colors are denoted by subscripts of $s$, $d$, and $c$ respectively.

Destination (framebuffer) components are taken to be fixed-point values represented according to the scheme given in section 2.13.9 (Final Color Processing), as are source (fragment) components. Constant color components are taken to be floating point values.

Prior to blending, each fixed-point color component undergoes an implied conversion to floating point. This conversion must leave the values 0 and 1 invariant. Blending computations are treated as if carried out in floating point.

The commands that control blending are

void **BlendColor**( clampf *red*, clampf *green*, clampf *blue*,
clampf *alpha* );
void **BlendEquation**( enum *mode* );

void **BlendFunc**( enum *src*, enum *dst* );

#### Using BlendColor

The constant color $C_c$ to be used in blending is specified with **BlendColor**. The four parameters are clamped to the range $[0, 1]$ before being stored. The constant color can be used in both the source and destination blending factors.

**BlendColor** is an imaging subset feature (see section 3.6.2), and is only allowed when the imaging subset is supported.

APPENDIX T

### Using BlendEquation

Blending capability is defined by the *blend equation*. **BlendEquation** *mode* `FUNC_ADD` defines the blending equation as

$$C = C_s S + C_d D$$

where $C_s$ and $C_d$ are the source and destination colors, and $S$ and $D$ are quadruplets of weighting factors as specified by **BlendFunc**.

If *mode* is `FUNC_SUBTRACT`, the blending equation is defined as

$$C = C_s S - C_d D$$

If *mode* is `FUNC_REVERSE_SUBTRACT`, the blending equation is defined as

$$C = C_d D - C_s S$$

If *mode* is `MIN`, the blending equation is defined as

$$C = min(C_s, C_d)$$

Finally, if *mode* is `MAX`, the blending equation is defined as

$$C = max(C_s, C_d)$$

The blending equation is evaluated separately for each color component and the corresponding weighting factors.

**BlendEquation** is an imaging subset feature (see section 3.6.2). If the imaging subset is not available, then blending always uses the blending equation `FUNC_ADD`.

### Using BlendFunc

**BlendFunc** *src* indicates how to compute a source blending factor, while *dst* indicates how to compute a destination factor. The possible arguments and their corresponding computed source and destination factors are summarized in Tables 4.1 and 4.2. Addition or subtraction of quadruplets means adding or subtracting them component-wise.

The computed source and destination blending quadruplets are applied to the source and destination R, G, B, and A values to obtain a new set of values that are sent to the next operation. Let the source and destination blending quadruplets be $S$ and $D$, respectively. Then a quadruplet of values is computed using the blend equation specified by **BlendEquation**. Each

APPENDIX T

| Value | Blend Factors |
|---|---|
| ZERO | $(0, 0, 0, 0)$ |
| ONE | $(1, 1, 1, 1)$ |
| DST_COLOR | $(R_d, G_d, B_d, A_d)$ |
| ONE_MINUS_DST_COLOR | $(1, 1, 1, 1) - (R_d, G_d, B_d, A_d)$ |
| SRC_ALPHA | $(A_s, A_s, A_s, A_s)$ |
| ONE_MINUS_SRC_ALPHA | $(1, 1, 1, 1) - (A_s, A_s, A_s, A_s)$ |
| DST_ALPHA | $(A_d, A_d, A_d, A_d)$ |
| ONE_MINUS_DST_ALPHA | $(1, 1, 1, 1) - (A_d, A_d, A_d, A_d)$ |
| CONSTANT_COLOR | $(R_c, G_c, B_c, A_c)$ |
| ONE_MINUS_CONSTANT_COLOR | $(1, 1, 1, 1) - (R_c, G_c, B_c, A_c)$ |
| CONSTANT_ALPHA | $(A_c, A_c, A_c, A_c)$ |
| ONE_MINUS_CONSTANT_ALPHA | $(1, 1, 1, 1) - (A_c, A_c, A_c, A_c)$ |
| SRC_ALPHA_SATURATE | $(f, f, f, 1)$ |

Table 4.1: Values controlling the source blending function and the source blending values they compute. $f = \min(A_s, 1 - A_d)$.

| Value | Blend factors |
|---|---|
| ZERO | $(0, 0, 0, 0)$ |
| ONE | $(1, 1, 1, 1)$ |
| SRC_COLOR | $(R_s, G_s, B_s, A_s)$ |
| ONE_MINUS_SRC_COLOR | $(1, 1, 1, 1) - (R_s, G_s, B_s, A_s)$ |
| SRC_ALPHA | $(A_s, A_s, A_s, A_s)$ |
| ONE_MINUS_SRC_ALPHA | $(1, 1, 1, 1) - (A_s, A_s, A_s, A_s)$ |
| DST_ALPHA | $(A_d, A_d, A_d, A_d)$ |
| ONE_MINUS_DST_ALPHA | $(1, 1, 1, 1) - (A_d, A_d, A_d, A_d)$ |
| CONSTANT_COLOR | $(R_c, G_c, B_c, A_c)$ |
| ONE_MINUS_CONSTANT_COLOR | $(1, 1, 1, 1) - (R_c, G_c, B_c, A_c)$ |
| CONSTANT_ALPHA | $(A_c, A_c, A_c, A_c)$ |
| ONE_MINUS_CONSTANT_ALPHA | $(1, 1, 1, 1) - (A_c, A_c, A_c, A_c)$ |

Table 4.2: Values controlling the destination blending function and the destination blending values they compute.

floating-point value in this quadruplet is clamped to $[0, 1]$ and converted back to a fixed-point value in the manner described in section 2.13.9. The resulting four values are sent to the next operation.

**BlendFunc** arguments CONSTANT_COLOR, ONE_MINUS_CONSTANT_COLOR, CONSTANT_ALPHA, and ONE_MINUS_CONSTANT_ALPHA are imaging subset features (see section 3.6.2), and are only allowed when the imaging subset is provided.

#### Blending State

The state required for blending is an integer indicating the blending equation, two integers indicating the source and destination blending functions, four floating-point values to store the RGBA constant blend color, and a bit indicating whether blending is enabled or disabled. The initial blending equation is FUNC_ADD. The initial blending functions are ONE for the source function and ZERO for the destination function. The initial constant blend color is $(\mathrm{R}, \mathrm{G}, \mathrm{B}, \mathrm{A}) = (0, 0, 0, 0)$. Initially, blending is disabled.

Blending occurs once for each color buffer currently enabled for writing (section 4.2.1) using each buffer's color for $C_d$. If a color buffer has no A value, then $A_d$ is taken to be 1.

### 4.1.7 Dithering

Dithering selects between two color values or indices. In RGBA mode, consider the value of any of the color components as a fixed-point value with $m$ bits to the left of the binary point, where $m$ is the number of bits allocated to that component in the framebuffer; call each such value $c$. For each $c$, dithering selects a value $c_1$ such that $c_1 \in \{\max\{0, \lceil c \rceil - 1\}, \lceil c \rceil\}$ (after this selection, treat $c_1$ as a fixed point value in [0,1] with $m$ bits). This selection may depend on the $x_w$ and $y_w$ coordinates of the pixel. In color index mode, the same rule applies with $c$ being a single color index. $c$ must not be larger than the maximum value representable in the framebuffer for either the component or the index, as appropriate.

Many dithering algorithms are possible, but a dithered value produced by any algorithm must depend only the incoming value and the fragment's $x$ and $y$ window coordinates. If dithering is disabled, then each color component is truncated to a fixed-point value with as many bits as there are in the corresponding component in the framebuffer; a color index is rounded to the nearest integer representable in the color index portion of the framebuffer.

Dithering is enabled with **Enable** and disabled with **Disable** using the symbolic constant DITHER. The state required is thus a single bit. Initially, dithering is enabled.

### 4.1.8 Logical Operation

Finally, a logical operation is applied between the incoming fragment's color or index values and the color or index values stored at the corresponding location in the framebuffer. The result replaces the values in the framebuffer at the fragment's $(x, y)$ coordinates. The logical operation on color indices is enabled or disabled with **Enable** or **Disable** using the symbolic constant INDEX_LOGIC_OP. (For compatibility with GL version 1.0, the symbolic constant LOGIC_OP may also be used.) The logical operation on color values is enabled or disabled with **Enable** or **Disable** using the symbolic constant COLOR_LOGIC_OP. If the logical operation is enabled for color values, it is as if blending were disabled, regardless of the value of BLEND.

The logical operation is selected by

void **LogicOp**( enum *op* );

*op* is a symbolic constant; the possible constants and corresponding operations are enumerated in Table 4.3. In this table, *s* is the value of the incoming fragment and *d* is the value stored in the framebuffer. The numeric values assigned to the symbolic constants are the same as those assigned to the corresponding symbolic values in the X window system.

Logical operations are performed independently for each color index buffer that is selected for writing, or for each red, green, blue, and alpha value of each color buffer that is selected for writing. The required state is an integer indicating the logical operation, and two bits indicating whether the logical operation is enabled or disabled. The initial state is for the logic operation to be given by COPY, and to be disabled.

## 4.2 Whole Framebuffer Operations

The preceding sections described the operations that occur as individual fragments are sent to the framebuffer. This section describes operations that control or affect the whole framebuffer.

### 4.2.1 Selecting a Buffer for Writing

The first such operation is controlling the buffer into which color values are written. This is accomplished with

APPENDIX T

| Argument value | Operation |
|---|---|
| CLEAR | 0 |
| AND | $s \wedge d$ |
| AND_REVERSE | $s \wedge \neg d$ |
| COPY | $s$ |
| AND_INVERTED | $\neg s \wedge d$ |
| NOOP | $d$ |
| XOR | $s$ xor $d$ |
| OR | $s \vee d$ |
| NOR | $\neg(s \vee d)$ |
| EQUIV | $\neg(s \text{ xor } d)$ |
| INVERT | $\neg d$ |
| OR_REVERSE | $s \vee \neg d$ |
| COPY_INVERTED | $\neg s$ |
| OR_INVERTED | $\neg s \vee d$ |
| NAND | $\neg(s \wedge d)$ |
| SET | all 1's |

Table 4.3: Arguments to **LogicOp** and their corresponding operations.

```
void DrawBuffer( enum buf );
```

*buf* is a symbolic constant specifying zero, one, two, or four buffers for writing. The constants are NONE, FRONT_LEFT, FRONT_RIGHT, BACK_LEFT, BACK_RIGHT, FRONT, BACK, LEFT, RIGHT, FRONT_AND_BACK, and AUX0 through AUX$n$, where $n+1$ is the number of available auxiliary buffers.

The constants refer to the four potentially visible buffers *front_left*, *front_right*, *back_left*, and *back_right*, and to the *auxiliary* buffers. Arguments other than AUX$i$ that omit reference to LEFT or RIGHT refer to both left and right buffers. Arguments other than AUX$i$ that omit reference to FRONT or BACK refer to both front and back buffers. AUX$i$ enables drawing only to *auxiliary* buffer $i$. Each AUX$i$ adheres to AUX$i$ = AUX0 + $i$. The constants and the buffers they indicate are summarized in Table 4.4. If **DrawBuffer** is is supplied with a constant (other than NONE) that does not indicate any of the color buffers allocated to the GL context, the error INVALID_OPERATION results.

Indicating a buffer or buffers using **DrawBuffer** causes subsequent pixel color value writes to affect the indicated buffers. If more than one color buffer is selected for drawing, blending and logical operations are computed

| symbolic<br>constant | front<br>left | front<br>right | back<br>left | back<br>right | aux<br>*i* |
|---|---|---|---|---|---|
| NONE | | | | | |
| FRONT_LEFT | ● | | | | |
| FRONT_RIGHT | | ● | | | |
| BACK_LEFT | | | ● | | |
| BACK_RIGHT | | | | ● | |
| FRONT | ● | ● | | | |
| BACK | | | ● | ● | |
| LEFT | ● | | ● | | |
| RIGHT | | ● | | ● | |
| FRONT_AND_BACK | ● | ● | ● | ● | |
| AUX*i* | | | | | ● |

Table 4.4: Arguments to **DrawBuffer** and the buffers that they indicate.

and applied independently for each buffer. Calling **DrawBuffer** with a value of NONE inhibits the writing of color values to any buffer.

Monoscopic contexts include only left buffers, while stereoscopic contexts include both left and right buffers. Likewise, single buffered contexts include only front buffers, while double buffered contexts include both front and back buffers. The type of context is selected at GL initialization.

The state required to handle buffer selection is a set of up to $4 + n$ bits. 4 bits indicate if the front left buffer, the front right buffer, the back left buffer, or the back right buffer, are enabled for color writing. The other $n$ bits indicate which of the auxiliary buffers is enabled for color writing. In the initial state, the front buffer or buffers are enabled if there are no back buffers; otherwise, only the back buffer or buffers are enabled.

### 4.2.2 Fine Control of Buffer Updates

Four commands are used to mask the writing of bits to each of the logical framebuffers after all per-fragment operations have been performed. The commands

```
void IndexMask( uint mask );
void ColorMask( boolean r, boolean g, boolean b,
   boolean a );
```

control the color buffer or buffers (depending on which buffers are currently indicated for writing). The least significant $n$ bits of *mask*, where $n$ is the number of bits in a color index buffer, specify a mask. Where a 1 appears in this mask, the corresponding bit in the color index buffer (or buffers) is written; where a 0 appears, the bit is not written. This mask applies only in color index mode. In RGBA mode, **ColorMask** is used to mask the writing of R, G, B and A values to the color buffer or buffers. $r$, $g$, $b$, and $a$ indicate whether R, G, B, or A values, respectively, are written or not (a value of `TRUE` means that the corresponding value is written). In the initial state, all bits (in color index mode) and all color values (in RGBA mode) are enabled for writing.

The depth buffer can be enabled or disabled for writing $z_w$ values using

`void` **DepthMask**( `boolean` *mask* );

If *mask* is non-zero, the depth buffer is enabled for writing; otherwise, it is disabled. In the initial state, the depth buffer is enabled for writing.

The command

`void` **StencilMask**( `uint` *mask* );

controls the writing of particular bits into the stencil planes. The least significant $s$ bits of *mask* comprise an integer mask ($s$ is the number of bits in the stencil buffer), just as for **IndexMask**. The initial state is for the stencil plane mask to be all ones.

The state required for the various masking operations is two integers and a bit: an integer for color indices, an integer for stencil values, and a bit for depth values. A set of four bits is also required indicating which color components of an RGBA value should be written. In the initial state, the integer masks are all ones as are the bits controlling depth value and RGBA component writing.

### 4.2.3 Clearing the Buffers

The GL provides a means for setting portions of every pixel in a particular buffer to the same value. The argument to

`void` **Clear**( `bitfield` *buf* );

is the bitwise OR of a number of values indicating which buffers are to be cleared. The values are `COLOR_BUFFER_BIT`, `DEPTH_BUFFER_BIT`,

APPENDIX T

`STENCIL_BUFFER_BIT`, and `ACCUM_BUFFER_BIT`, indicating the buffers currently enabled for color writing, the depth buffer, the stencil buffer, and the accumulation buffer (see below), respectively. The value to which each buffer is cleared depends on the setting of the clear value for that buffer. If the mask is not a bitwise OR of the specified values, then the error `INVALID_VALUE` is generated.

```
void ClearColor( clampf r, clampf g, clampf b,
   clampf a );
```

sets the clear value for the color buffers in RGBA mode. Each of the specified components is clamped to $[0, 1]$ and converted to fixed-point according to the rules of section 2.13.9.

```
void ClearIndex( float index );
```

sets the clear color index. *index* is converted to a fixed-point value with unspecified precision to the left of the binary point; the integer part of this value is then masked with $2^m - 1$, where $m$ is the number of bits in a color index value stored in the framebuffer.

```
void ClearDepth( clampd d );
```

takes a floating-point value that is clamped to the range $[0, 1]$ and converted to fixed-point according to the rules for a window $z$ value given in section 2.10.1. Similarly,

```
void ClearStencil( int s );
```

takes a single integer argument that is the value to which to clear the stencil buffer. $s$ is masked to the number of bitplanes in the stencil buffer.

```
void ClearAccum( float r, float g, float b, float a );
```

takes four floating-point arguments that are the values, in order, to which to set the R, G, B, and A values of the accumulation buffer (see the next section). These values are clamped to the range $[-1, 1]$ when they are specified.

When **Clear** is called, the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test, and dithering. The masking operations described in the last section (4.2.2) are also effective. If a buffer is not present, then a **Clear** directed at that buffer has no effect.

APPENDIX T

The state required for clearing is a clear value for each of the color buffer, the depth buffer, the stencil buffer, and the accumulation buffer. Initially, the RGBA color clear value is (0,0,0,0), the clear color index is 0, and the stencil buffer and accumulation buffer clear values are all 0. The depth buffer clear value is initially 1.0.

### 4.2.4 The Accumulation Buffer

Each portion of a pixel in the accumulation buffer consists of four values: one for each of R, G, B, and A. The accumulation buffer is controlled exclusively through the use of

void **Accum**( enum *op*, float *value* );

(except for clearing it). *op* is a symbolic constant indicating an accumulation buffer operation, and *value* is a floating-point value to be used in that operation. The possible operations are ACCUM, LOAD, RETURN, MULT, and ADD.

When the scissor test is enabled (section 4.1.2), then only those pixels within the current scissor box are updated by any **Accum** operation; otherwise, all pixels in the window are updated. The accumulation buffer operations apply identically to every affected pixel, so we describe the effect of each operation on an individual pixel. Accumulation buffer values are taken to be signed values in the range $[-1, 1]$. Using ACCUM obtains R, G, B, and A components from the buffer currently selected for reading (section 4.3.2). Each component, considered as a fixed-point value in $[0, 1]$. (see section 2.13.9), is converted to floating-point. Each result is then multiplied by *value*. The results of this multiplication are then added to the corresponding color component currently in the accumulation buffer, and the resulting color value replaces the current accumulation buffer color value.

The LOAD operation has the same effect as ACCUM, but the computed values replace the corresponding accumulation buffer components rather than being added to them.

The RETURN operation takes each color value from the accumulation buffer, multiplies each of the R, G, B, and A components by *value*, and clamps the results to the range $[0, 1]$ The resulting color value is placed in the buffers currently enabled for color writing as if it were a fragment produced from rasterization, except that the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test (section 4.1.2), and dithering (section 4.1.7). Color masking (section 4.2.2) is also applied.

APPENDIX T

The MULT operation multiplies each R, G, B, and A in the accumulation buffer by *value* and then returns the scaled color components to their corresponding accumulation buffer locations. ADD is the same as MULT except that *value* is added to each of the color components.

The color components operated on by **Accum** must be clamped only if the operation is RETURN. In this case, a value sent to the enabled color buffers is first clamped to $[0, 1]$. Otherwise, results are undefined if the result of an operation on a color component is out of the range $[-1, 1]$. If there is no accumulation buffer, or if the GL is in color index mode, **Accum** generates the error INVALID_OPERATION.

No state (beyond the accumulation buffer itself) is required for accumulation buffering.

## 4.3 Drawing, Reading, and Copying Pixels

Pixels may be written to and read from the framebuffer using the **DrawPixels** and **ReadPixels** commands. **CopyPixels** can be used to copy a block of pixels from one portion of the framebuffer to another.

### 4.3.1 Writing to the Stencil Buffer

The operation of **DrawPixels** was described in section 3.6.4, except if the *format* argument was STENCIL_INDEX. In this case, all operations described for **DrawPixels** take place, but window $(x, y)$ coordinates, each with the corresponding stencil index, are produced in lieu of fragments. Each coordinate-stencil index pair is sent directly to the per-fragment operations, bypassing the texture, fog, and antialiasing application stages of rasterization. Each pair is then treated as a fragment for purposes of the pixel ownership and scissor tests; all other per-fragment operations are bypassed. Finally, each stencil index is written to its indicated location in the framebuffer, subject to the current setting of **StencilMask**.

The error INVALID_OPERATION results if there is no stencil buffer.

### 4.3.2 Reading Pixels

The method for reading pixels from the framebuffer and placing them in client memory is diagrammed in Figure 4.2. We describe the stages of the pixel reading process in the order in which they occur.

Pixels are read using

APPENDIX T

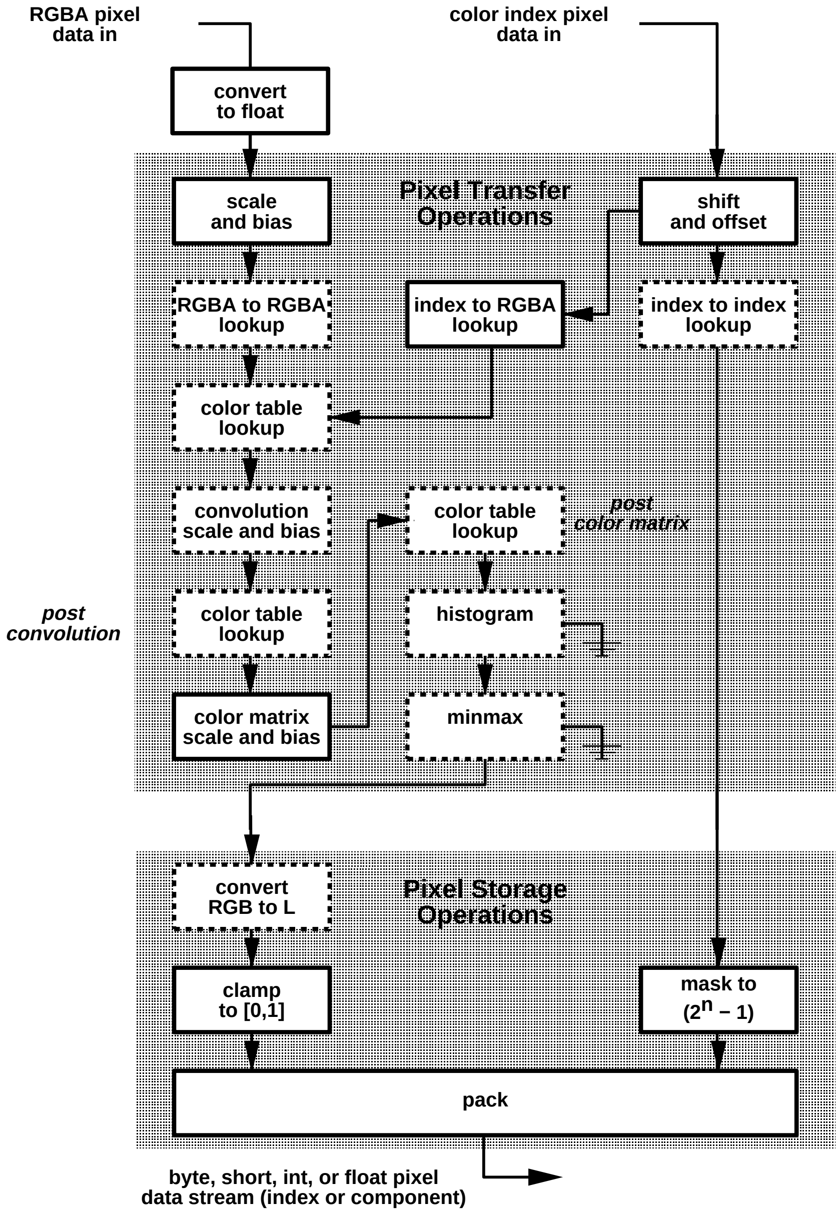


Figure 4.2. Operation of **ReadPixels**. Operations in dashed boxes may be enabled or disabled. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

| Parameter Name | Type | Initial Value | Valid Range |
|---|---|---|---|
| PACK_SWAP_BYTES | boolean | FALSE | TRUE/FALSE |
| PACK_LSB_FIRST | boolean | FALSE | TRUE/FALSE |
| PACK_ROW_LENGTH | integer | 0 | $[0, \infty)$ |
| PACK_SKIP_ROWS | integer | 0 | $[0, \infty)$ |
| PACK_SKIP_PIXELS | integer | 0 | $[0, \infty)$ |
| PACK_ALIGNMENT | integer | 4 | 1,2,4,8 |
| PACK_IMAGE_HEIGHT | integer | 0 | $[0, \infty)$ |
| PACK_SKIP_IMAGES | integer | 0 | $[0, \infty)$ |

Table 4.5: **PixelStore** parameters pertaining to **ReadPixels**, **GetTexImage1D**, **GetTexImage2D**, **GetTexImage3D**, **GetColorTable**, **GetConvolutionFilter**, **GetSeparableFilter**, **GetHistogram**, and **GetMinmax**.

void **ReadPixels**( int *x*, int *y*, sizei *width*, sizei *height*,
   enum *format*, enum *type*, void **data* );

The arguments after *x* and *y* to **ReadPixels** correspond to those of **DrawPixels**. The pixel storage modes that apply to **ReadPixels** and other commands that query images (see section 6.1) are summarized in Table 4.5.

### Obtaining Pixels from the Framebuffer

If the *format* is DEPTH_COMPONENT, then values are obtained from the depth buffer. If there is no depth buffer, the error INVALID_OPERATION occurs.

If the *format* is STENCIL_INDEX, then values are taken from the stencil buffer; again, if there is no stencil buffer, the error INVALID_OPERATION occurs.

For all other formats, the buffer from which values are obtained is one of the color buffers; the selection of color buffer is controlled with **ReadBuffer**.

The command

void **ReadBuffer**( enum *src* );

takes a symbolic constant as argument. The possible values are FRONT_LEFT, FRONT_RIGHT, BACK_LEFT, BACK_RIGHT, FRONT, BACK, LEFT, RIGHT, and AUX0 through AUX*n*. FRONT and LEFT refer to the front left buffer, BACK refers to the back left buffer, and RIGHT refers to the front right buffer. The other constants correspond directly to the buffers that they name. If the requested

APPENDIX T

buffer is missing, then the error INVALID_OPERATION is generated. The initial setting for **ReadBuffer** is FRONT if there is no back buffer and BACK otherwise.

**ReadPixels** obtains values from the selected buffer from each pixel with lower left hand corner at $(x + i, y + j)$ for $0 \leq i < width$ and $0 \leq j < height$; this pixel is said to be the $i$th pixel in the $j$th row. If any of these pixels lies outside of the window allocated to the current GL context, the values obtained for those pixels are undefined. Results are also undefined for individual pixels that are not owned by the current context. Otherwise, **ReadPixels** obtains values from the selected buffer, regardless of how those values were placed there.

If the GL is in RGBA mode, and *format* is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, BGR, BGRA, LUMINANCE, or LUMINANCE_ALPHA, then red, green, blue, and alpha values are obtained from the selected buffer at each pixel location. If the framebuffer does not support alpha values then the A that is obtained is 1.0. If *format* is COLOR_INDEX and the GL is in RGBA mode then the error INVALID_OPERATION occurs. If the GL is in color index mode, and *format* is not DEPTH_COMPONENT or STENCIL_INDEX, then the color index is obtained at each pixel location.

### Conversion of RGBA values

This step applies only if the GL is in RGBA mode, and then only if *format* is neither STENCIL_INDEX nor DEPTH_COMPONENT. The R, G, B, and A values form a group of elements. Each element is taken to be a fixed-point value in $[0, 1]$ with $m$ bits, where $m$ is the number of bits in the corresponding color component of the selected buffer (see section 2.13.9).

### Conversion of Depth values

This step applies only if *format* is DEPTH_COMPONENT. An element is taken to be a fixed-point value in [0,1] with $m$ bits, where $m$ is the number of bits in the depth buffer (see section 2.10.1).

### Pixel Transfer Operations

This step is actually the sequence of steps that was described separately in section 3.6.5. After the processing described in that section is completed, groups are processed as described in the following sections.

| *type* Parameter | Index Mask |
|---|---|
| UNSIGNED_BYTE | $2^8 - 1$ |
| BITMAP | $1$ |
| BYTE | $2^7 - 1$ |
| UNSIGNED_SHORT | $2^{16} - 1$ |
| SHORT | $2^{15} - 1$ |
| UNSIGNED_INT | $2^{32} - 1$ |
| INT | $2^{31} - 1$ |

Table 4.6: Index masks used by **ReadPixels**. Floating point data are not masked.

### Conversion to L

This step applies only to RGBA component groups, and only if the *format* is either LUMINANCE or LUMINANCE_ALPHA. A value L is computed as

$$L = R + G + B$$

where $R$, $G$, and $B$ are the values of the R, G, and B components. The single computed L component replaces the R, G, and B components in the group.

### Final Conversion

For an index, if the *type* is not FLOAT, final conversion consists of masking the index with the value given in Table 4.6; if the *type* is FLOAT, then the integer index is converted to a GL float data value.

For an RGBA color, each component is first clamped to $[0, 1]$. Then the appropriate conversion formula from table 4.7 is applied to the component.

### Placement in Client Memory

Groups of elements are placed in memory just as they are taken from memory for **DrawPixels**. That is, the $i$th group of the $j$th row (corresponding to the $i$th pixel in the $j$th row) is placed in memory just where the $i$th group of the $j$th row would be taken from for **DrawPixels**. See **Unpacking** under section 3.6.4. The only difference is that the storage mode parameters whose names begin with PACK_ are used instead of those whose names begin with UNPACK_. If the *format* is RED, GREEN, BLUE, ALPHA, or LUMINANCE,

APPENDIX T

| *type* Parameter | GL Data Type | Component Conversion Formula |
|---|---|---|
| UNSIGNED_BYTE | ubyte | $c = (2^8 - 1)f$ |
| BYTE | byte | $c = [(2^8 - 1)f - 1]/2$ |
| UNSIGNED_SHORT | ushort | $c = (2^{16} - 1)f$ |
| SHORT | short | $c = [(2^{16} - 1)f - 1]/2$ |
| UNSIGNED_INT | uint | $c = (2^{32} - 1)f$ |
| INT | int | $c = [(2^{32} - 1)f - 1]/2$ |
| FLOAT | float | $c = f$ |
| UNSIGNED_BYTE_3_3_2 | ubyte | $c = (2^N - 1)f$ |
| UNSIGNED_BYTE_2_3_3_REV | ubyte | $c = (2^N - 1)f$ |
| UNSIGNED_SHORT_5_6_5 | ushort | $c = (2^N - 1)f$ |
| UNSIGNED_SHORT_5_6_5_REV | ushort | $c = (2^N - 1)f$ |
| UNSIGNED_SHORT_4_4_4_4 | ushort | $c = (2^N - 1)f$ |
| UNSIGNED_SHORT_4_4_4_4_REV | ushort | $c = (2^N - 1)f$ |
| UNSIGNED_SHORT_5_5_5_1 | ushort | $c = (2^N - 1)f$ |
| UNSIGNED_SHORT_1_5_5_5_REV | ushort | $c = (2^N - 1)f$ |
| UNSIGNED_INT_8_8_8_8 | uint | $c = (2^N - 1)f$ |
| UNSIGNED_INT_8_8_8_8_REV | uint | $c = (2^N - 1)f$ |
| UNSIGNED_INT_10_10_10_2 | uint | $c = (2^N - 1)f$ |
| UNSIGNED_INT_2_10_10_10_REV | uint | $c = (2^N - 1)f$ |

Table 4.7: Reversed component conversions - used when component data are being returned to client memory. Color, normal, and depth components are converted from the internal floating-point representation ($f$) to a datum of the specified GL data type ($c$) using the equations in this table. All arithmetic is done in the internal floating point format. These conversions apply to component data returned by GL query commands and to components of pixel data returned to client memory. The equations remain the same even if the implemented ranges of the GL data types are greater than the minimum required ranges. (See Table 2.2.) Equations with $N$ as the exponent are performed for each bitfield of the packed data type, with $N$ set to the number of bits in the bitfield.

only the corresponding single element is written. Likewise if the *format* is LUMINANCE_ALPHA, RGB, or BGR, only the corresponding two or three elements are written. Otherwise all the elements of each group are written.

### 4.3.3 Copying Pixels

**CopyPixels** transfers a rectangle of pixel values from one region of the framebuffer to another. Pixel copying is diagrammed in Figure 4.3.

void **CopyPixels**( int *x*, int *y*, sizei *width*, sizei *height*,
  enum *type* );

*type* is a symbolic constant that must be one of COLOR, STENCIL, or DEPTH, indicating that the values to be transferred are colors, stencil values, or depth values, respectively. The first four arguments have the same interpretation as the corresponding arguments to **ReadPixels**.

Values are obtained from the framebuffer, converted (if appropriate), then subjected to the pixel transfer operations described in section 3.6.5, just as if **ReadPixels** were called with the corresponding arguments. If the *type* is STENCIL or DEPTH, then it is as if the *format* for **ReadPixels** were STENCIL_INDEX or DEPTH_COMPONENT, respectively. If the *type* is COLOR, then if the GL is in RGBA mode, it is as if the *format* were RGBA, while if the GL is in color index mode, it is as if the *format* were COLOR_INDEX.

The groups of elements so obtained are then written to the framebuffer just as if **DrawPixels** had been given *width* and *height*, beginning with final conversion of elements. The effective *format* is the same as that already described.

### 4.3.4 Pixel Draw/Read state

The state required for pixel operations consists of the parameters that are set with **PixelStore**, **PixelTransfer**, and **PixelMap**. This state has been summarized in Tables 3.1, 3.2, and 3.3. The current setting of **ReadBuffer**, an integer, is also required, along with the current raster position (section 2.12). State set with **PixelStore** is GL client state.

APPENDIX T

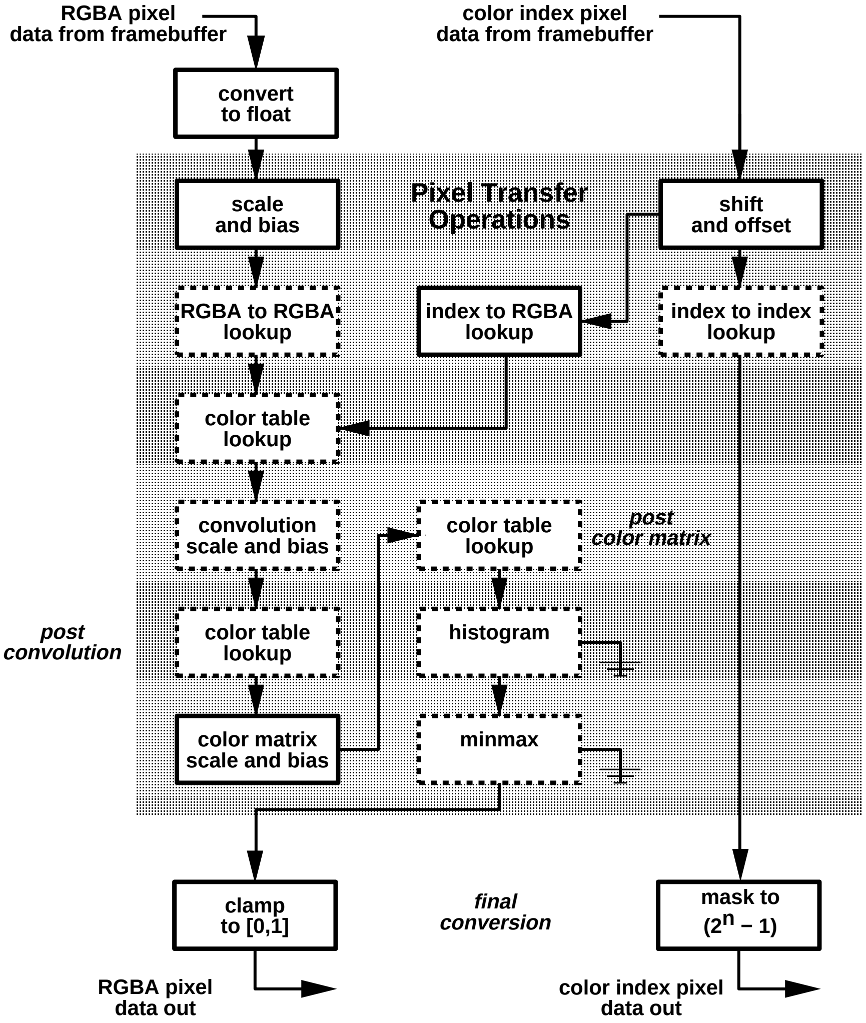


Figure 4.3. Operation of **CopyPixels**. Operations in dashed boxes may be enabled or disabled. Index-to-RGBA lookup is currently never performed. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

# Chapter 5

# Special Functions

This chapter describes additional GL functionality that does not fit easily into any of the preceding chapters. This functionality consists of evaluators (used to model curves and surfaces), selection (used to locate rendered primitives on the screen), feedback (which returns GL results before rasterization), display lists (used to designate a group of GL commands for later execution by the GL), flushing and finishing (used to synchronize the GL command stream), and hints.

## 5.1 Evaluators

Evaluators provide a means to use a polynomial or rational polynomial mapping to produce vertex, normal, and texture coordinates, and colors. The values so produced are sent on to further stages of the GL as if they had been provided directly by the client. Transformations, lighting, primitive assembly, rasterization, and per-pixel operations are not affected by the use of evaluators.

Consider the $R^k$-valued polynomial $\mathbf{p}(u)$ defined by

$$\mathbf{p}(u) = \sum_{i=0}^{n} B_i^n(u)\mathbf{R}_i \qquad (5.1)$$

with $\mathbf{R}_i \in R^k$ and

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i},$$

the $i$th Bernstein polynomial of degree $n$ (recall that $0^0 \equiv 1$ and $\binom{n}{0} \equiv 1$). Each $\mathbf{R}_i$ is a *control point*. The relevant command is

| *target* | $k$ | Values |
|---|---|---|
| MAP1_VERTEX_3 | 3 | $x$, $y$, $z$ vertex coordinates |
| MAP1_VERTEX_4 | 4 | $x$, $y$, $z$, $w$ vertex coordinates |
| MAP1_INDEX | 1 | color index |
| MAP1_COLOR_4 | 4 | R, G, B, A |
| MAP1_NORMAL | 3 | $x$, $y$, $z$ normal coordinates |
| MAP1_TEXTURE_COORD_1 | 1 | $s$ texture coordinate |
| MAP1_TEXTURE_COORD_2 | 2 | $s$, $t$ texture coordinates |
| MAP1_TEXTURE_COORD_3 | 3 | $s$, $t$, $r$ texture coordinates |
| MAP1_TEXTURE_COORD_4 | 4 | $s$, $t$, $r$, $q$ texture coordinates |

Table 5.1: Values specified by the *target* to **Map1**. Values are given in the order in which they are taken.

```
void Map1{fd}( enum type, T u1, T u2, int stride,
   int order, T points );
```

*type* is a symbolic constant indicating the range of the defined polynomial. Its possible values, along with the evaluations that each indicates, are given in Table 5.1. *order* is equal to $n + 1$; The error INVALID_VALUE is generated if *order* is less than one or greater than MAX_EVAL_ORDER. *points* is a pointer to a set of $n+1$ blocks of storage. Each block begins with $k$ single-precision floating-point or double-precision floating-point values, respectively. The rest of the block may be filled with arbitrary data. Table 5.1 indicates how $k$ depends on *type* and what the $k$ values represent in each case.

*stride* is the number of single- or double-precision values (as appropriate) in each block of storage. The error INVALID_VALUE results if *stride* is less than $k$. The order of the polynomial, *order*, is also the number of blocks of storage containing control points.

$u_1$ and $u_2$ give two floating-point values that define the endpoints of the pre-image of the map. When a value $u'$ is presented for evaluation, the formula used is

$$\mathbf{p}'(u') = \mathbf{p}(\frac{u' - u_1}{u_2 - u_1}).$$

The error INVALID_VALUE results if $u_1 = u_2$.

**Map2** is analogous to **Map1**, except that it describes bivariate polyno-

**APPENDIX T**

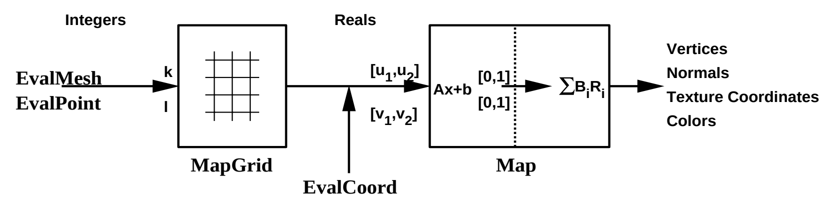


Figure 5.1. Map Evaluation.

mials of the form

$$\mathbf{p}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)B_j^m(v)\mathbf{R}_{ij}.$$

The form of the **Map2** command is

> `void` **Map2**{**fd**}( `enum` *target*, `T` $u_1$, `T` $u_2$, `int` *ustride*,
> `int` *uorder*, `T` $v_1$, `T` $v_2$, `int` *vstride*, `int` *vorder*, `T` *points* );

*target* is a range type selected from the same group as is used for **Map1**, except that the string `MAP1` is replaced with `MAP2`. *points* is a pointer to $(n+1)(m+1)$ blocks of storage ($uorder = n+1$ and $vorder = m+1$; the error `INVALID_VALUE` is generated if either *uorder* or *vorder* is less than one or greater than `MAX_EVAL_ORDER`). The values comprising $\mathbf{R}_{ij}$ are located

$$(ustride)i + (vstride)j$$

values (either single- or double-precision floating-point, as appropriate) past the first value pointed to by *points*. $u_1$, $u_2$, $v_1$, and $v_2$ define the pre-image rectangle of the map; a domain point $(u', v')$ is evaluated as

$$\mathbf{p}'(u',v') = \mathbf{p}(\frac{u'-u_1}{u_2-u_1}, \frac{v'-v_1}{v_2-v_1}).$$

The evaluation of a defined map is enabled or disabled with **Enable** and **Disable** using the constant corresponding to the map as described above. The error `INVALID_VALUE` results if either *ustride* or *vstride* is less than $k$, or if $u_1$ is equal to $u_2$, or if $v_1$ is equal to $v_2$.

Figure 5.1 describes map evaluation schematically; an evaluation of enabled maps is effected in one of two ways. The first way is to use

APPENDIX T

`void` **EvalCoord{12}{fd}**( `T` *arg* );
`void` **EvalCoord{12}{fd}v**( `T` *arg* );

**EvalCoord1** causes evaluation of the enabled one-dimensional maps. The argument is the value (or a pointer to the value) that is the domain coordinate, $u'$. **EvalCoord2** causes evaluation of the enabled two-dimensional maps. The two values specify the two domain coordinates, $u'$ and $v'$, in that order.

When one of the **EvalCoord** commands is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if a corresponding GL command were issued with the resulting coordinates, with one important difference. The difference is that when an evaluation is performed, the GL uses evaluated values instead of current values for those evaluations that are enabled (otherwise, the current values are used). The order of the effective commands is immaterial, except that **Vertex** (for vertex coordinate evaluation) must be issued last. Use of evaluators has no effect on the current color, normal, or texture coordinates. If **ColorMaterial** is enabled, evaluated color values affect the result of the lighting equation as if the current color was being modified, but no change is made to the tracking lighting parameters or to the current color.

No command is effectively issued if the corresponding map (of the indicated dimension) is not enabled. If more than one evaluation is enabled for a particular dimension (e.g. `MAP1_TEXTURE_COORD_1` and `MAP1_TEXTURE_COORD_2`), then only the result of the evaluation of the map with the highest number of coordinates is used.

Finally, if either `MAP2_VERTEX_3` or `MAP2_VERTEX_4` is enabled, then the normal to the surface is computed. Analytic computation, which sometimes yields normals of length zero, is one method which may be used. If automatic normal generation is enabled, then this computed normal is used as the normal associated with a generated vertex. Automatic normal generation is controlled with **Enable** and **Disable** with symbolic the constant `AUTO_NORMAL`. If automatic normal generation is disabled, then a corresponding normal map, if enabled, is used to produce a normal. If neither automatic normal generation nor a normal map are enabled, then no normal is sent with a vertex resulting from an evaluation (the effect is that the current normal is used).

For `MAP_VERTEX_3`, let $\mathbf{q} = \mathbf{p}$. For `MAP_VERTEX_4`, let $\mathbf{q} = (x/w, y/w, z/w)$, where $(x, y, z, w) = \mathbf{p}$. Then let

$$\mathbf{m} = \frac{\partial \mathbf{q}}{\partial u} \times \frac{\partial \mathbf{q}}{\partial v}.$$

Microsoft et al. Exhibit 1005

APPENDIX T

Then the generated analytic normal, $\mathbf{n}$, is given by $\mathbf{n} = \mathbf{m}/\|\mathbf{m}\|$.

The second way to carry out evaluations is to use a set of commands that provide for efficient specification of a series of evenly spaced values to be mapped. This method proceeds in two steps. The first step is to define a grid in the domain. This is done using

void **MapGrid1{fd}**( int $n$, T $u'_1$, T $u'_2$ );

for a one-dimensional map or

void **MapGrid2{fd}**( int $n_u$, T $u'_1$, T $u'_2$, int $n_v$, T $v'_1$,
T $v'_2$ );

for a two-dimensional map. In the case of **MapGrid1** $u'_1$ and $u'_2$ describe an interval, while $n$ describes the number of partitions of the interval. The error INVALID_VALUE results if $n \leq 0$. For **MapGrid2**, $(u'_1, v'_1)$ specifies one two-dimensional point and $(u'_2, v'_2)$ specifies another. $n_u$ gives the number of partitions between $u'_1$ and $u'_2$, and $n_v$ gives the number of partitions between $v'_1$ and $v'_2$. If either $n_u \leq 0$ or $n_v \leq 0$, then the error INVALID_VALUE occurs.

Once a grid is defined, an evaluation on a rectangular subset of that grid may be carried out by calling

void **EvalMesh1**( enum *mode*, int $p_1$, int $p_2$ );

*mode* is either POINT or LINE. The effect is the same as performing the following code fragment, with $\Delta u' = (u'_2 - u'_1)/n$:

```
Begin(type);
  for $i = p_1$ to $p_2$ step 1.0
    EvalCoord1($i * \Delta u' + u'_1$);
End();
```

where **EvalCoord1f** or **EvalCoord1d** is substituted for **EvalCoord1** as appropriate. If *mode* is POINT, then *type* is POINTS; if *mode* is LINE, then *type* is LINE_STRIP. The one requirement is that if either $i = 0$ or $i = n$, then the value computed from $i * \Delta u' + u'_1$ is precisely $u'_1$ or $u'_2$, respectively.

The corresponding commands for two-dimensional maps are

void **EvalMesh2**( enum *mode*, int $p_1$, int $p_2$, int $q_1$,
int $q_2$ );

Microsoft et al. Exhibit 1005

*mode* must be `FILL`, `LINE`, or `POINT`. When *mode* is `FILL`, then these commands are equivalent to the following, with $\Delta u' = (u'_2 - u'_1)/n$ and $\Delta v' = (v'_2 - v'_1)/m$:

```
for $i = q_1$ to $q_2 - 1$ step 1.0
    Begin(QUAD_STRIP);
        for $j = p_1$ to $p_2$ step 1.0
            EvalCoord2($j * \Delta u' + u'_1$ , $i * \Delta v' + v'_1$);
            EvalCoord2($j * \Delta u' + u'_1$ , $(i+1) * \Delta v' + v'_1$);
    End();
```

If *mode* is `LINE`, then a call to **EvalMesh2** is equivalent to

```
for $i = q_1$ to $q_2$ step 1.0
    Begin(LINE_STRIP);
    for $j = p_1$ to $p_2$ step 1.0
        EvalCoord2($j * \Delta u' + u'_1$ , $i * \Delta v' + v'_1$);
    End();;
for $i = p_1$ to $p_2$ step 1.0
    Begin(LINE_STRIP);
    for $j = q_1$ to $q_2$ step 1.0
        EvalCoord2($i * \Delta u' + u'_1$ , $j * \Delta v' + v'_1$);
    End();
```

If *mode* is `POINT`, then a call to **EvalMesh2** is equivalent to

```
Begin(POINTS);
    for $i = q_1$ to $q_2$ step 1.0
        for $j = p_1$ to $p_2$ step 1.0
            EvalCoord2($j * \Delta u' + u'_1$ , $i * \Delta v' + v'_1$);
End();
```

Again, in all three cases, there is the requirement that $0 * \Delta u' + u'_1 = u'_1$, $n * \Delta u' + u'_1 = u'_2$, $0 * \Delta v' + v'_1 = v'_1$, and $m * \Delta v' + v'_1 = v'_2$.

An evaluation of a single point on the grid may also be carried out:

```
void EvalPoint1( int p );
```

Calling it is equivalent to the command

```
EvalCoord1($p * \Delta u' + u'_1$);
```

with $\Delta u'$ and $u'_1$ defined as above.

**APPENDIX T**

```
void EvalPoint2( int p, int q );
```

is equivalent to the command

$$\textbf{EvalCoord2}(p * \Delta u' + u_1' \;,\; q * \Delta v' + v_1');$$

The state required for evaluators potentially consists of 9 one-dimensional map specifications and 9 two-dimensional map specifications, as well as corresponding flags for each specification indicating which are enabled. Each map specification consists of one or two orders, an appropriately sized array of control points, and a set of two values (for a one-dimensional map) or four values (for a two-dimensional map) to describe the domain. The maximum possible order, for either $u$ or $v$, is implementation dependent (one maximum applies to both $u$ and $v$), but must be at least 8. Each control point consists of between one and four floating-point values (depending on the type of the map). Initially, all maps have order 1 (making them constant maps). All vertex coordinate maps produce the coordinates $(0, 0, 0, 1)$ (or the appropriate subset); all normal coordinate maps produce $(0, 0, 1)$; RGBA maps produce $(1, 1, 1, 1)$; color index maps produce 1.0; texture coordinate maps produce $(0, 0, 0, 1)$; In the initial state, all maps are disabled. A flag indicates whether or not automatic normal generation is enabled for two-dimensional maps. In the initial state, automatic normal generation is disabled. Also required are two floating-point values and an integer number of grid divisions for the one-dimensional grid specification and four floating-point values and two integer grid divisions for the two-dimensional grid specification. In the initial state, the bounds of the domain interval for 1-D is 0 and 1.0, respectively; for 2-D, they are $(0, 0)$ and $(1.0, 1.0)$, respectively. The number of grid divisions is 1 for 1-D and 1 in both directions for 2-D. If any evaluation command is issued when no vertex map is enabled, nothing happens.

## 5.2 Selection

Selection is used by a programmer to determine which primitives are drawn into some region of a window. The region is defined by the current model-view and perspective matrices.

Selection works by returning an array of integer-valued *names*. This array represents the current contents of the *name stack*. This stack is controlled with the commands

APPENDIX T

```
void InitNames( void );
void PopName( void );
void PushName( uint name );
void LoadName( uint name );
```

**InitNames** empties (clears) the name stack. **PopName** pops one name off the top of the name stack. **PushName** causes *name* to be pushed onto the name stack. **LoadName** replaces the value on the top of the stack with *name*. Loading a name onto an empty stack generates the error `INVALID_OPERATION`. Popping a name off of an empty stack generates `STACK_UNDERFLOW`; pushing a name onto a full stack generates `STACK_OVERFLOW`. The maximum allowable depth of the name stack is implementation dependent but must be at least 64.

In selection mode, no fragments are rendered into the framebuffer. The GL is placed in selection mode with

```
int RenderMode( enum mode );
```

*mode* is a symbolic constant: one of `RENDER`, `SELECT`, or `FEEDBACK`. `RENDER` is the default, corresponding to rendering as described until now. `SELECT` specifies selection mode, and `FEEDBACK` specifies feedback mode (described below). Use of any of the name stack manipulation commands while the GL is not in selection mode has no effect.

Selection is controlled using

```
void SelectBuffer( sizei n, uint *buffer );
```

*buffer* is a pointer to an array of unsigned integers (called the selection array) to be potentially filled with names, and *n* is an integer indicating the maximum number of values that can be stored in that array. Placing the GL in selection mode before **SelectBuffer** has been called results in an error of `INVALID_OPERATION` as does calling **SelectBuffer** while in selection mode.

In selection mode, if a point, line, polygon, or the valid coordinates produced by a **RasterPos** command intersects the clip volume (section 2.11) then this primitive (or **RasterPos** command) causes a selection *hit*. In the case of polygons, no hit occurs if the polygon would have been culled, but selection is based on the polygon itself, regardless of the setting of **PolygonMode**. When in selection mode, whenever a name stack manipulation command is executed or **RenderMode** is called and there has been a hit since the last time the stack was manipulated or **RenderMode** was called, then a *hit record* is written into the selection array.

APPENDIX T

A hit record consists of the following items in order: a non-negative integer giving the number of elements on the name stack at the time of the hit, a minimum depth value, a maximum depth value, and the name stack with the bottommost element first. The minimum and maximum depth values are the minimum and maximum taken over all the window coordinate $z$ values of each (post-clipping) vertex of each primitive that intersects the clipping volume since the last hit record was written. The minimum and maximum (each of which lies in the range $[0, 1]$) are each multiplied by $2^{32} - 1$ and rounded to the nearest unsigned integer to obtain the values that are placed in the hit record. No depth offset arithmetic (section 3.5.5) is performed on these values.

Hit records are placed in the selection array by maintaining a pointer into that array. When selection mode is entered, the pointer is initialized to the beginning of the array. Each time a hit record is copied, the pointer is updated to point at the array element after the one into which the topmost element of the name stack was stored. If copying the hit record into the selection array would cause the total number of values to exceed $n$, then as much of the record as fits in the array is written and an overflow flag is set.

Selection mode is exited by calling **RenderMode** with an argument value other than SELECT. Whenever **RenderMode** is called in selection mode, it returns the number of hit records copied into the selection array and resets the **SelectBuffer** pointer to its last specified value. Values are not guaranteed to be written into the selection array until **RenderMode** is called. If the selection array overflow flag was set, then **RenderMode** returns $-1$ and clears the overflow flag. The name stack is cleared and the stack pointer reset whenever **RenderMode** is called.

The state required for selection consists of the address of the selection array and its maximum size, the name stack and its associated pointer, a minimum and maximum depth value, and several flags. One flag indicates the current **RenderMode** value. In the initial state, the GL is in the RENDER mode. Another flag is used to indicate whether or not a hit has occurred since the last name stack manipulation. This flag is reset upon entering selection mode and whenever a name stack manipulation takes place. One final flag is required to indicate whether the maximum number of copied names would have been exceeded. This flag is reset upon entering selection mode. This flag, the address of the selection array, and its maximum size are GL client state.

APPENDIX T

## 5.3 Feedback

Feedback, like selection, is a GL mode. The mode is selected by calling **RenderMode** with `FEEDBACK`. When the GL is in feedback mode, no fragments are written to the framebuffer. Instead, information about primitives that would have been rasterized is fed back to the application using the GL.

Feedback is controlled using

void **FeedbackBuffer**( sizei *n*, enum *type*, float **buffer* );

*buffer* is a pointer to an array of floating-point values into which feedback information will be placed, and *n* is a number indicating the maximum number of values that can be written to that array. *type* is a symbolic constant describing the information to be fed back for each vertex (see Figure 5.2). The error `INVALID_OPERATION` results if the GL is placed in feedback mode before a call to **FeedbackBuffer** has been made, or if a call to **FeedbackBuffer** is made while in feedback mode.

While in feedback mode, each primitive that would be rasterized (or bitmap or call to **DrawPixels** or **CopyPixels**, if the raster position is valid) generates a block of values that get copied into the feedback array. If doing so would cause the number of entries to exceed the maximum, the block is partially written so as to fill the array (if there is any room left at all). The first block of values generated after the GL enters feedback mode is placed at the beginning of the feedback array, with subsequent blocks following. Each block begins with a code indicating the primitive type, followed by values that describe the primitive's vertices and associated data. Entries are also written for bitmaps and pixel rectangles. Feedback occurs after polygon culling (section 3.5.1) and **PolygonMode** interpretation of polygons (section 3.5.4) has taken place. It may also occur after polygons with more than three edges are broken up into triangles (if the GL implementation renders polygons by performing this decomposition). $x$, $y$, and $z$ coordinates returned by feedback are window coordinates; if $w$ is returned, it is in clip coordinates. No depth offset arithmetic (section 3.5.5) is performed on the $z$ values. In the case of bitmaps and pixel rectangles, the coordinates returned are those of the current raster position.

The texture coordinates and colors returned are these resulting from the clipping operations described in Section 2.13.8. The colors returned are the primary colors.

The ordering rules for GL command interpretation also apply in feedback mode. Each command must be fully interpreted and its effects on both GL

| Type | coordinates | color | texture | total values |
|---|---|---|---|---|
| 2D | $x$, $y$ | – | – | 2 |
| 3D | $x$, $y$, $z$ | – | – | 3 |
| 3D_COLOR | $x$, $y$, $z$ | $k$ | – | $3+k$ |
| 3D_COLOR_TEXTURE | $x$, $y$, $z$ | $k$ | 4 | $7+k$ |
| 4D_COLOR_TEXTURE | $x$, $y$, $z$, $w$ | $k$ | 4 | $8+k$ |

Table 5.2: Correspondence of feedback type to number of values per vertex. $k$ is 1 in color index mode and 4 in RGBA mode.

state and the values to be written to the feedback buffer completed before a subsequent command may be executed.

The GL is taken out of feedback mode by calling **RenderMode** with an argument value other than FEEDBACK. When called while in feedback mode, **RenderMode** returns the number of values placed in the feedback array and resets the feedback array pointer to be *buffer*. The return value never exceeds the maximum number of values passed to **FeedbackBuffer**.

If writing a value to the feedback buffer would cause more values to be written than the specified maximum number of values, then the value is not written and an overflow flag is set. In this case, **RenderMode** returns $-1$ when it is called, after which the overflow flag is reset. While in feedback mode, values are not guaranteed to be written into the feedback buffer before **RenderMode** is called.

Figure 5.2 gives a grammar for the array produced by feedback. Each primitive is indicated with a unique identifying value followed by some number of vertices. A vertex is fed back as some number of floating-point values determined by the feedback *type*. Table 5.2 gives the correspondence between feedback *buffer* and the number of values returned for each vertex.

The command

void **PassThrough**( float *token* );

may be used as a marker in feedback mode. *token* is returned as if it were a primitive; it is indicated with its own unique identifying value. The ordering of any **PassThrough** commands with respect to primitive specification is maintained by feedback. **PassThrough** may not occur between **Begin** and **End**. It has no effect when the GL is not in feedback mode.

The state required for feedback is the pointer to the feedback array, the maximum number of values that may be placed there, and the feedback *type*.

An overflow flag is required to indicate whether the maximum allowable number of feedback values has been written; initially this flag is cleared. These state variables are GL client state. Feedback also relies on the same mode flag as selection to indicate whether the GL is in feedback, selection, or normal rendering mode.

## 5.4 Display Lists

A display list is simply a group of GL commands and arguments that has been stored for subsequent execution. The GL may be instructed to process a particular display list (possibly repeatedly) by providing a number that uniquely specifies it. Doing so causes the commands within the list to be executed just as if they were given normally. The only exception pertains to commands that rely upon client state. When such a command is accumulated into the display list (that is, when issued, not when executed), the client state in effect at that time applies to the command. Only server state is affected when the command is executed. As always, pointers which are passed as arguments to commands are dereferenced when the command is issued. (Vertex array pointers are dereferenced when the commands **ArrayElement**, **DrawArrays**, or **DrawElements** are accumulated into a display list.)

A display list is begun by calling

void **NewList**( uint *n*, enum *mode* );

*n* is a positive integer to which the display list that follows is assigned, and *mode* is a symbolic constant that controls the behavior of the GL during display list creation. If *mode* is `COMPILE`, then commands are not executed as they are placed in the display list. If *mode* is `COMPILE_AND_EXECUTE` then commands are executed as they are encountered, then placed in the display list. If $n = 0$, then the error `INVALID_VALUE` is generated.

After calling **NewList** all subsequent GL commands are placed in the display list (in the order the commands are issued) until a call to

void **EndList**( void );

occurs, after which the GL returns to its normal command execution state. It is only when **EndList** occurs that the specified display list is actually associated with the index indicated with **NewList**. The error `INVALID_OPERATION` is generated if **EndList** is called without a previous matching **NewList**,

APPENDIX T

```
feedback-list:
        feedback-item feedback-list
        feedback-item

feedback-item:
        point
        line-segment
        polygon
        bitmap
        pixel-rectangle
        passthrough

point:
        POINT_TOKEN vertex
line-segment:
        LINE_TOKEN vertex vertex
        LINE_RESET_TOKEN vertex vertex
polygon:
        POLYGON_TOKEN n polygon-spec
polygon-spec:
        polygon-spec vertex
        vertex vertex vertex
bitmap:
        BITMAP_TOKEN vertex

pixel-rectangle:
        DRAW_PIXEL_TOKEN vertex
        COPY_PIXEL_TOKEN vertex
passthrough:
        PASS_THROUGH_TOKEN f

vertex:
2D:
        f f
3D:
        f f f
3D_COLOR:
        f f f color
3D_COLOR_TEXTURE:
        f f f color tex
4D_COLOR_TEXTURE:
        f f f f color tex

color:
        f f f f
        f

tex:
        f f f f
```

Figure 5.2: Feedback syntax. $f$ is a floating-point number. $n$ is a floating-point integer giving the number of vertices in a polygon. The symbols ending with _TOKEN are symbolic floating-point constants. The labels under the "vertex" rule show the different data returned for vertices depending on the feedback *type*. LINE_TOKEN and LINE_RESET_TOKEN are identical except that the latter is returned only when the line stipple is reset for that line segment.

Microsoft et al. Exhibit 1005

**APPENDIX T**

or if **NewList** is called a second time before calling **EndList**. The error `OUT_OF_MEMORY` is generated if **EndList** is called and the specified display list cannot be stored because insufficient memory is available. In this case GL implementations of revision 1.1 or greater insure that no change is made to the previous contents of the display list, if any, and that no other change is made to the GL state, except for the state changed by execution of GL commands when the display list mode is `COMPILE_AND_EXECUTE`.

Once defined, a display list is executed by calling

void **CallList**( uint *n* );

*n* gives the index of the display list to be called. This causes the commands saved in the display list to be executed, in order, just as if they were issued without using a display list. If $n = 0$, then the error `INVALID_VALUE` is generated.

The command

void **CallLists**( sizei *n*, enum *type*, void **lists* );

provides an efficient means for executing a number of display lists. *n* is an integer indicating the number of display lists to be called, and *lists* is a pointer that points to an array of offsets. Each offset is constructed as determined by *lists* as follows. First, *type* may be one of the constants `BYTE`, `UNSIGNED_BYTE`, `SHORT`, `UNSIGNED_SHORT`, `INT`, `UNSIGNED_INT`, or `FLOAT` indicating that the array pointed to by *lists* is an array of bytes, unsigned bytes, shorts, unsigned shorts, integers, unsigned integers, or floats, respectively. In this case each offset is found by simply converting each array element to an integer (floating point values are truncated). Further, *type* may be one of `2_BYTES`, `3_BYTES`, or `4_BYTES`, indicating that the array contains sequences of 2, 3, or 4 unsigned bytes, in which case each integer offset is constructed according to the following algorithm:

```
$offset \leftarrow 0$
for $i = 1$ to $b$
    $offset \leftarrow offset$ shifted left 8 bits
    $offset \leftarrow offset + byte$
    advance to next $byte$ in the array
```

*b* is 2, 3, or 4, as indicated by *type*. If $n = 0$, **CallLists** does nothing.

Each of the *n* constructed offsets is taken in order and added to a display list base to obtain a display list number. For each number, the indicated display list is executed. The base is set by calling

APPENDIX T

```
void ListBase( uint base );
```

to specify the offset.

Indicating a display list index that does not correspond to any display list has no effect. **CallList** or **CallLists** may appear inside a display list. (If the *mode* supplied to **NewList** is COMPILE_AND_EXECUTE, then the appropriate lists are executed, but the **CallList** or **CallLists**, rather than those lists' constituent commands, is placed in the list under construction.) To avoid the possibility of infinite recursion resulting from display lists calling one another, an implementation dependent limit is placed on the nesting level of display lists during display list execution. This limit must be at least 64.

Two commands are provided to manage display list indices.

```
uint GenLists( sizei s );
```

returns an integer $n$ such that the indices $n, \ldots, n+s-1$ are previously unused (i.e. there are $s$ previously unused display list indices starting at $n$). **GenLists** also has the effect of creating an empty display list for each of the indices $n, \ldots, n+s-1$, so that these indices all become used. **GenLists** returns 0 if there is no group of $s$ contiguous previously unused display list indices, or if $s = 0$.

```
boolean IsList( uint list );
```

returns TRUE if *list* is the index of some display list.

A contiguous group of display lists may be deleted by calling

```
void DeleteLists( uint list, sizei range );
```

where *list* is the index of the first display list to be deleted and *range* is the number of display lists to be deleted. All information about the display lists is lost, and the indices become unused. Indices to which no display list corresponds are ignored. If $range = 0$, nothing happens.

Certain commands, when called while compiling a display list, are not compiled into the display list but are executed immediately. These are: **IsList**, **GenLists**, **DeleteLists**, **FeedbackBuffer**, **SelectBuffer**, **RenderMode**, **VertexPointer**, **NormalPointer**, **ColorPointer**, **IndexPointer**, **TexCoordPointer**, **EdgeFlagPointer**, **InterleavedArrays**, **EnableClientState**, **DisableClientState**, **PushClientAttrib**, **PopClientAttrib**, **ReadPixels**, **PixelStore**, **GenTextures**, **DeleteTextures**, **AreTexturesResident**, **IsTexture**, **Flush**, **Finish**, as well as **IsEnabled** and all of the **Get** commands (see Chapter 6).

**TexImage3D**, **TexImage2D**, **TexImage1D**, **Histogram**, and **ColorTable** are executed immediately when called with the corresponding proxy arguments PROXY_TEXTURE_3D, PROXY_TEXTURE_2D, PROXY_TEXTURE_1D, PROXY_HISTOGRAM, and PROXY_COLOR_TABLE, PROXY_POST_CONVOLUTION_COLOR_TABLE, or PROXY_POST_COLOR_MATRIX_COLOR_TABLE.

Display lists require one bit of state to indicate whether a GL command should be executed immediately or placed in a display list. In the initial state, commands are executed immediately. If the bit indicates display list creation, an index is required to indicate the current display list being defined. Another bit indicates, during display list creation, whether or not commands should be executed as they are compiled into the display list. One integer is required for the current **ListBase** setting; its initial value is zero. Finally, state must be maintained to indicate which integers are currently in use as display list indices. In the initial state, no indices are in use.

## 5.5 Flush and Finish

The command

```
void Flush( void );
```

indicates that all commands that have previously been sent to the GL must complete in finite time.

The command

```
void Finish( void );
```

forces all previous GL commands to complete. **Finish** does not return until all effects from previously issued commands on GL client and server state and the framebuffer are fully realized.

## 5.6 Hints

Certain aspects of GL behavior, when there is room for variation, may be controlled with hints. A hint is specified using

```
void Hint( enum target, enum hint );
```

APPENDIX T

*target* is a symbolic constant indicating the behavior to be controlled, and *hint* is a symbolic constant indicating what type of behavior is desired. *target* may be one of `PERSPECTIVE_CORRECTION_HINT`, indicating the desired quality of parameter interpolation; `POINT_SMOOTH_HINT`, indicating the desired sampling quality of points; `LINE_SMOOTH_HINT`, indicating the desired sampling quality of lines; `POLYGON_SMOOTH_HINT`, indicating the desired sampling quality of polygons; and `FOG_HINT`, indicating whether fog calculations are done per pixel or per vertex. *hint* must be one of `FASTEST`, indicating that the most efficient option should be chosen; `NICEST`, indicating that the highest quality option should be chosen; and `DONT_CARE`, indicating no preference in the matter.

The interpretation of hints is implementation dependent. An implementation may ignore them entirely.

The initial value of all hints is `DONT_CARE`.

# Chapter 6

# State and State Requests

The state required to describe the GL machine is enumerated in section 6.2. Most state is set through the calls described in previous chapters, and can be queried using the calls described in section 6.1.

## 6.1 Querying GL State

### 6.1.1 Simple Queries

Much of the GL state is completely identified by symbolic constants. The values of these state variables can be obtained using a set of **Get** commands. There are four commands for obtaining simple state variables:

```
void GetBooleanv( enum value, boolean *data );
void GetIntegerv( enum value, int *data );
void GetFloatv( enum value, float *data );
void GetDoublev( enum value, double *data );
```

The commands obtain boolean, integer, floating-point, or double-precision state variables. *value* is a symbolic constant indicating the state variable to return. *data* is a pointer to a scalar or array of the indicated type in which to place the returned data. In addition

```
boolean IsEnabled( enum value );
```

can be used to determine if *value* is currently enabled (as with **Enable**) or disabled.

### 6.1.2 Data Conversions

If a **Get** command is issued that returns value types different from the type of the value being obtained, a type conversion is performed. If **GetBooleanv** is called, a floating-point or integer value converts to `FALSE` if and only if it is zero (otherwise it converts to `TRUE`). If **GetIntegerv** (or any of the **Get** commands below) is called, a boolean value is interpreted as either 1 or 0, and a floating-point value is rounded to the nearest integer, unless the value is an RGBA color component, a **DepthRange** value, a depth buffer clear value, or a normal coordinate. In these cases, the **Get** command converts the floating-point value to an integer according the `INT` entry of Table 4.7; a value not in $[-1, 1]$ converts to an undefined value. If **GetFloatv** is called, a boolean value is interpreted as either 1.0 or 0.0, an integer is coerced to floating-point, and a double-precision floating-point value is converted to single-precision. Analogous conversions are carried out in the case of **GetDoublev**. If a value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned.

Unless otherwise indicated, multi-valued state variables return their multiple values in the same order as they are given as arguments to the commands that set them. For instance, the two **DepthRange** parameters are returned in the order $n$ followed by $f$. Similarly, points for evaluator maps are returned in the order that they appeared when passed to **Map1**. **Map2** returns $\mathbf{R}_{ij}$ in the $[(uorder)i + j]$th block of values (see page 166 for $i$, $j$, $uorder$, and $\mathbf{R}_{ij}$).

### 6.1.3 Enumerated Queries

Other commands exist to obtain state variables that are identified by a category (clip plane, light, material, etc.) as well as a symbolic constant. These are

```
void GetClipPlane( enum plane, double eqn[4] );
void GetLight{if}v( enum light, enum value, T data );
void GetMaterial{if}v( enum face, enum value, T data );
void GetTexEnv{if}v( enum env, enum value, T data );
void GetTexGen{if}v( enum coord, enum value, T data );
void GetTexParameter{if}v( enum target, enum value,
    T data );
void GetTexLevelParameter{if}v( enum target, int lod,
    enum value, T data );
```

void **GetPixelMap{ui us f}v**( enum *map*, T *data* );
void **GetMap{ifd}v**( enum *map*, enum *value*, T *data* );

**GetClipPlane** always returns four double-precision values in *eqn*; these are the coefficients of the plane equation of *plane* in eye coordinates (these coordinates are those that were computed when the plane was specified).

**GetLight** places information about *value* (a symbolic constant) for *light* (also a symbolic constant) in *data*. POSITION or SPOT_DIRECTION returns values in eye coordinates (again, these are the coordinates that were computed when the position or direction was specified).

**GetMaterial**, **GetTexGen**, **GetTexEnv**, and **GetTexParameter** are similar to **GetLight**, placing information about *value* for the target indicated by their first argument into *data*. The *face* argument to **GetMaterial** must be either FRONT or BACK, indicating the front or back material, respectively. The *env* argument to **GetTexEnv** must currently be TEXTURE_ENV. The *coord* argument to **GetTexGen** must be one of S, T, R, or Q. For **GetTexGen**, EYE_LINEAR coefficients are returned in the eye coordinates that were computed when the plane was specified; OBJECT_LINEAR coefficients are returned in object coordinates.

**GetTexParameter** and **GetTexLevelParameter** parameter *target* may be one of TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D, indicating the currently bound one-, two-, or three-dimensional texture object. For **GetTexLevelParameter**, *target* may also be one of PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, or PROXY_TEXTURE_3D, indicating the one-, two-, or three-dimensional proxy state vector. *value* is a symbolic value indicating which texture parameter is to be obtained. The *lod* argument to **GetTexLevelParameter** determines which level-of-detail's state is returned. If the *lod* argument is less than zero or if it is larger than the maximum allowable level-of-detail then the error INVALID_VALUE occurs. Queries of TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE, TEXTURE_BLUE_SIZE, TEXTURE_ALPHA_SIZE, TEXTURE_LUMINANCE_SIZE, and TEXTURE_INTENSITY_SIZE return the actual resolutions of the stored image array components, not the resolutions specified when the image array was defined. Queries of TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH, and TEXTURE_BORDER return the width, height, depth, and border as specified when the image array was created. The internal format of the image array is queried as TEXTURE_INTERNAL_FORMAT, or as TEXTURE_COMPONENTS for compatibility with GL version 1.0.

For **GetPixelMap**, the *map* must be a map name from Table 3.3. For **GetMap**, *map* must be one of the map types described in section 5.1, and

**APPENDIX T**

*value* must be one of ORDER, COEFF, or DOMAIN.

### 6.1.4 Texture Queries

The command

```
void GetTexImage( enum tex, int lod, enum format,
   enum type, void *img );
```

is used to obtain texture images. It is somewhat different from the other get commands; *tex* is a symbolic value indicating which texture is to be obtained. TEXTURE_1D indicates a one-dimensional texture, TEXTURE_2D indicates a two-dimensional texture, and TEXTURE_3D indicates a three-dimensional texture. *lod* is a level-of-detail number, *format* is a pixel format from Table 3.6, *type* is a pixel type from Table 3.5, and *img* is a pointer to a block of memory.

**GetTexImage** obtains component groups from a texture image with the indicated level-of-detail. The components are assigned among R, G, B, and A according to Table 6.1, starting with the first group in the first row, and continuing by obtaining groups in order from each row and proceeding from the first row to the last, and from the first image to the last for three-dimensional textures. These groups are then packed and placed in client memory. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to **ReadPixels** are applied.

For three-dimensional textures, pixel storage operations are applied as if the image were two-dimensional, except that the additional pixel storage state values PACK_IMAGE_HEIGHT and PACK_SKIP_IMAGES are applied. The correspondence of texels to memory locations is as defined for **TexImage3D** in section 3.8.1.

The row length, number of rows, image depth, and number of images are determined by the size of the texture image (including any borders). Calling **GetTexImage** with *lod* less than zero or larger than the maximum allowable causes the error INVALID_VALUE . Calling **GetTexImage** with *format* of COLOR_INDEX, STENCIL_INDEX, or DEPTH_COMPONENT causes the error INVALID_ENUM.

The command

```
boolean IsTexture( uint texture );
```

returns TRUE if *texture* is the name of a texture object. If *texture* is zero, or is a non-zero value that is not the name of a texture object, or if an error condition occurs, **IsTexture** returns FALSE. A name returned by **GenTextures**, but not yet bound, is not the name of a texture object.

APPENDIX T

| Base Internal Format | R | G | B | A |
|---|---|---|---|---|
| ALPHA | 0 | 0 | 0 | $A_i$ |
| LUMINANCE (or 1) | $L_i$ | 0 | 0 | 1 |
| LUMINANCE_ALPHA (or 2) | $L_i$ | 0 | 0 | $A_i$ |
| INTENSITY | $I_i$ | 0 | 0 | 1 |
| RGB (or 3) | $R_i$ | $G_i$ | $B_i$ | 1 |
| RGBA (or 4) | $R_i$ | $G_i$ | $B_i$ | $A_i$ |

Table 6.1: Texture, table, and filter return values. $R_i$, $G_i$, $B_i$, $A_i$, $L_i$, and $I_i$ are components of the internal format that are assigned to pixel values R, G, B, and A. If a requested pixel value is not present in the internal format, the specified constant value is used.

### 6.1.5 Stipple Query

The command

```
void GetPolygonStipple( void *pattern );
```

obtains the polygon stipple. The pattern is packed into memory according to the procedure given in section 4.3.2 for **ReadPixels**; it is as if the *height* and *width* passed to that command were both equal to 32, the *type* were BITMAP, and the *format* were COLOR_INDEX.

### 6.1.6 Color Matrix Query

The scale and bias variables are queried using **GetFloatv** with *pname* set to the appropriate variable name. The top matrix on the color matrix stack is returned by **GetFloatv** called with *pname* set to COLOR_MATRIX. The depth of the color matrix stack, and the maximum depth of the color matrix stack, are queried with **GetIntegerv**, setting *pname* to COLOR_MATRIX_STACK_DEPTH and MAX_COLOR_MATRIX_STACK_DEPTH respectively.

### 6.1.7 Color Table Query

The current contents of a color table are queried using

```
void GetColorTable( enum target, enum format, enum type,
   void *table );
```

APPENDIX T

*target* must be one of the *regular* color table names listed in table 3.4. *format* and *type* accept the same values as do the corresponding parameters of **GetTexImage**. The one-dimensional color table image is returned to client memory starting at *table*. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to **ReadPixels** are performed. Color components that are requested in the specified *format*, but which are not included in the internal format of the color lookup table, are returned as zero. The assignments of internal color components to the components requested by *format* are described in Table 6.1.

The functions

```
void GetColorTableParameter{if}v( enum target,
    enum pname, T params );
```

are used for integer and floating point query.

*target* must be one of the regular or proxy color table names listed in table 3.4. *pname* is one of COLOR_TABLE_SCALE, COLOR_TABLE_BIAS, COLOR_TABLE_FORMAT, COLOR_TABLE_WIDTH, COLOR_TABLE_RED_SIZE, COLOR_TABLE_GREEN_SIZE, COLOR_TABLE_BLUE_SIZE, COLOR_TABLE_ALPHA_SIZE, COLOR_TABLE_LUMINANCE_SIZE, or COLOR_TABLE_INTENSITY_SIZE. The value of the specified parameter is returned in *params*.

### 6.1.8 Convolution Query

The current contents of a convolution filter image are queried with the command

```
void GetConvolutionFilter( enum target, enum format,
    enum type, void *image );
```

*target* must be CONVOLUTION_1D or CONVOLUTION_2D. *format* and *type* accept the same values as do the corresponding parameters of **GetTexImage**. The one-dimensional or two-dimensional images is returned to client memory starting at *image*. Pixel processing and component mapping are identical to those of **GetTexImage**.

The current contents of a separable filter image are queried using

```
void GetSeparableFilter( enum target, enum format,
    enum type, void *row, void *column, void *span );
```

**APPENDIX T**

*target* must be SEPARABLE_2D. *format* and *type* accept the same values as do the corresponding parameters of **GetTexImage**. The row and column images are returned to client memory starting at *row* and *column* respectively. *span* is currently unused. Pixel processing and component mapping are identical to those of **GetTexImage**.

The functions

```
void GetConvolutionParameter{if}v( enum target,
    enum pname, T params );
```

are used for integer and floating point query. *target* must be CONVOLUTION_1D, CONVOLUTION_2D, or SEPARABLE_2D. *pname* is one of CONVOLUTION_BORDER_COLOR, CONVOLUTION_BORDER_MODE, CONVOLUTION_FILTER_SCALE, CONVOLUTION_FILTER_BIAS, CONVOLUTION_FORMAT, CONVOLUTION_WIDTH, CONVOLUTION_HEIGHT, MAX_CONVOLUTION_WIDTH, or MAX_CONVOLUTION_HEIGHT. The value of the specified parameter is returned in *params*.

### 6.1.9 Histogram Query

The current contents of the histogram table are queried using

```
void GetHistogram( enum target, boolean reset,
    enum format, enum type, void* values );
```

*target* must be HISTOGRAM. *type* and *format* accept the same values as do the corresponding parameters of **GetTexImage**. The one-dimensional histogram table image is returned to *values*. Pixel processing and component mapping are identical to those of **GetTexImage**.

If *reset* is TRUE, then all counters of all elements of the histogram are reset to zero. Counters are reset whether returned or not.

No counters are modified if *reset* is FALSE.

Calling

```
void ResetHistogram( enum target );
```

resets all counters of all elements of the histogram table to zero. *target* must be HISTOGRAM.

It is not an error to reset or query the contents of a histogram table with zero entries.

The functions

**APPENDIX T**

```
void GetHistogramParameter{if}v( enum target,
    enum pname, T params );
```

are used for integer and floating point query. *target* must be `HISTOGRAM` or `PROXY_HISTOGRAM`. *pname* is one of `HISTOGRAM_FORMAT`, `HISTOGRAM_WIDTH`, `HISTOGRAM_RED_SIZE`, `HISTOGRAM_GREEN_SIZE`, `HISTOGRAM_BLUE_SIZE`, `HISTOGRAM_ALPHA_SIZE`, or `HISTOGRAM_LUMINANCE_SIZE`. *pname* may be `HISTOGRAM_SINK` only for *target* `HISTOGRAM`. The value of the specified parameter is returned in *params*.

### 6.1.10 Minmax Query

The current contents of the minmax table are queried using

```
void GetMinmax( enum target, boolean reset,
    enum format, enum type, void* values );
```

*target* must be `MINMAX`. *type* and *format* accept the same values as do the corresponding parameters of **GetTexImage**. A one-dimensional image of width 2 is returned to *values*. Pixel processing and component mapping are identical to those of **GetTexImage**.

If *reset* is `TRUE`, then each minimum value is reset to the maximum representable value, and each maximum value is reset to the minimum representable value. All values are reset, whether returned or not.

No values are modified if *reset* is `FALSE`.

Calling

```
void ResetMinmax( enum target );
```

resets all minimum and maximum values of *target* to to their maximum and minimum representable values, respectively, *target* must be `MINMAX`.

The functions

```
void GetMinmaxParameter{if}v( enum target,
    enum pname, T params );
```

are used for integer and floating point query. *target* must be `MINMAX`. *pname* is `MINMAX_FORMAT` or `MINMAX_SINK`. The value of the specified parameter is returned in *params*.

### 6.1.11 Pointer and String Queries

The command

void **GetPointerv**( enum *pname*, void ***params* );

obtains the pointer or pointers named *pname* in the array *params*. The possible values for *pname* are SELECTION_BUFFER_POINTER, FEEDBACK_BUFFER_POINTER, VERTEX_ARRAY_POINTER, NORMAL_ARRAY_POINTER, COLOR_ARRAY_POINTER, INDEX_ARRAY_POINTER, TEXTURE_COORD_ARRAY_POINTER, and EDGE_FLAG_ARRAY_POINTER. Each returns a single pointer value.

Finally,

ubyte ***GetString**( enum *name* );

returns a pointer to a static string describing some aspect of the current GL connection. The possible values for *name* are VENDOR, RENDERER, VERSION, and EXTENSIONS. The format of the RENDERER and VERSION strings is implementation dependent. The EXTENSIONS string contains a space separated list of extension names (The extension names themselves do not contain any spaces); the VERSION string is laid out as follows:

<version number><space><vendor-specific information>

The version number is either of the form *major_number.minor_number* or *major_number.minor_number.release_number*, where the numbers all have one or more digits. The vendor specific information is optional. However, if it is present then it pertains to the server and the format and contents are implementation dependent.

**GetString** returns the version number (returned in the VERSION string) and the extension names (returned in the EXTENSIONS string) that can be supported on the connection. Thus, if the client and server support different versions and/or extensions, a compatible version and list of extensions is returned.

### 6.1.12 Saving and Restoring State

Besides providing a means to obtain the values of state variables, the GL also provides a means to save and restore groups of state variables. The **PushAttrib**, **PushClientAttrib**, **PopAttrib** and **PopClientAttrib** commands are used for this purpose. The commands

APPENDIX T

```
void PushAttrib( bitfield mask );
void PushClientAttrib( bitfield mask );
```

take a bitwise OR of symbolic constants indicating which groups of state variables to push onto an attribute stack. **PushAttrib** uses a server attribute stack while **PushClientAttrib** uses a client attribute stack. Each constant refers to a group of state variables. The classification of each variable into a group is indicated in the following tables of state variables. The error STACK_OVERFLOW is generated if **PushAttrib** or **PushClientAttrib** is executed while the corresponding stack depth is MAX_ATTRIB_STACK_DEPTH or MAX_CLIENT_ATTRIB_STACK_DEPTH respectively. The commands

```
void PopAttrib( void );
void PopClientAttrib( void );
```

reset the values of those state variables that were saved with the last corresponding **PushAttrib** or **PopClientAttrib**. Those not saved remain unchanged. The error STACK_UNDERFLOW is generated if **PopAttrib** or **PopClientAttrib** is executed while the respective stack is empty.

Table 6.2 shows the attribute groups with their corresponding symbolic constant names and stacks.

When **PushAttrib** is called with TEXTURE_BIT set, the priorities, border colors, filter modes, and wrap modes of the currently bound texture objects, as well as the current texture bindings and enables, are pushed onto the attribute stack. (Unbound texture objects are not pushed or restored.) When an attribute set that includes texture information is popped, the bindings and enables are first restored to their pushed values, then the bound texture objects' priorities, border colors, filter modes, and wrap modes are restored to their pushed values.

The depth of each attribute stack is implementation dependent but must be at least 16. The state required for each attribute stack is potentially 16 copies of each state variable, 16 masks indicating which groups of variables are stored in each stack entry, and an attribute stack pointer. In the initial state, both attribute stacks are empty.

In the tables that follow, a type is indicated for each variable. Table 6.3 explains these types. The type actually identifies all state associated with the indicated description; in certain cases only a portion of this state is returned. This is the case with all matrices, where only the top entry on the stack is returned; with clip planes, where only the selected clip plane is returned, with parameters describing lights, where only the value pertaining

APPENDIX T

| Stack | Attribute | Constant |
|---|---|---|
| server | accum-buffer | ACCUM_BUFFER_BIT |
| server | color-buffer | COLOR_BUFFER_BIT |
| server | current | CURRENT_BIT |
| server | depth-buffer | DEPTH_BUFFER_BIT |
| server | enable | ENABLE_BIT |
| server | eval | EVAL_BIT |
| server | fog | FOG_BIT |
| server | hint | HINT_BIT |
| server | lighting | LIGHTING_BIT |
| server | line | LINE_BIT |
| server | list | LIST_BIT |
| server | pixel | PIXEL_MODE_BIT |
| server | point | POINT_BIT |
| server | polygon | POLYGON_BIT |
| server | polygon-stipple | POLYGON_STIPPLE_BIT |
| server | scissor | SCISSOR_BIT |
| server | stencil-buffer | STENCIL_BUFFER_BIT |
| server | texture | TEXTURE_BIT |
| server | transform | TRANSFORM_BIT |
| server | viewport | VIEWPORT_BIT |
| server |  | ALL_ATTRIB_BITS |
| client | vertex-array | CLIENT_VERTEX_ARRAY_BIT |
| client | pixel-store | CLIENT_PIXEL_STORE_BIT |
| client | select | can't be pushed or pop'd |
| client | feedback | can't be pushed or pop'd |
| client |  | ALL_CLIENT_ATTRIB_BITS |

Table 6.2: Attribute groups

APPENDIX T

| Type code | Explanation |
|---|---|
| $B$ | Boolean |
| $C$ | Color (floating-point R, G, B, and A values) |
| $CI$ | Color index (floating-point index value) |
| $T$ | Texture coordinates (floating-point $s$, $t$, $r$, $q$ values) |
| $N$ | Normal coordinates (floating-point $x$, $y$, $z$ values) |
| $V$ | Vertex, including associated data |
| $Z$ | Integer |
| $Z^+$ | Non-negative integer |
| $Z_k$, $Z_{k*}$ | $k$-valued integer ($k*$ indicates $k$ is minimum) |
| $R$ | Floating-point number |
| $R^+$ | Non-negative floating-point number |
| $R^{[a,b]}$ | Floating-point number in the range $[a, b]$ |
| $R^k$ | $k$-tuple of floating-point numbers |
| $P$ | Position ($x$, $y$, $z$, $w$ floating-point coordinates) |
| $D$ | Direction ($x$, $y$, $z$ floating-point coordinates) |
| $M^4$ | $4 \times 4$ floating-point matrix |
| $I$ | Image |
| $A$ | Attribute stack entry, including mask |
| $Y$ | Pointer (data type unspecified) |
| $n \times type$ | $n$ copies of type $type$ ($n*$ indicates $n$ is minimum) |

Table 6.3: State variable types

to the selected light is returned; with textures, where only the selected texture or texture parameter is returned; and with evaluator maps, where only the selected map is returned. Finally, a “–” in the attribute column indicates that the indicated value is not included in any attribute group (and thus can not be pushed or popped with **PushAttrib**, **PushClientAttrib**, **PopAttrib**, or **PopClientAttrib**).

The $M$ and $m$ entries for initial minmax table values represent the maximum and minimum possible representable values, respectively.

Microsoft et al. Exhibit 1005

APPENDIX T

## 6.2 State Tables

The tables on the following pages indicate which state variables are obtained with what commands. State variables that can be obtained using any of **GetBooleanv**, **GetIntegerv**, **GetFloatv**, or **GetDoublev** are listed with just one of these commands – the one that is most appropriate given the type of the data to be returned. These state variables cannot be obtained using **IsEnabled**. However, state variables for which **IsEnabled** is listed as the query command can also be obtained using **GetBooleanv**, **GetIntegerv**, **GetFloatv**, and **GetDoublev**. State variables for which any other command is listed as the query command can be obtained only by using that command.

State table entries which are required only by the imaging subset (see section 3.6.2) are typeset against a gray background .

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| – | $Z_{11}$ | – | 0 | When $\neq 0$, indicates **begin/end** object | 2.6.1 | – |
| – | $V$ | – | – | Previous vertex in **Begin/End line** | 2.6.1 | – |
| – | $B$ | – | – | Indicates if *line-vertex* is the first | 2.6.1 | – |
| – | $V$ | – | – | First vertex of a **Begin/End line loop** | 2.6.1 | – |
| – | $Z^+$ | – | – | Line stipple counter | 3.4 | – |
| – | $n \times V$ | – | – | Vertices inside of **Begin/End polygon** | 2.6.1 | – |
| – | $Z^+$ | – | – | Number of *polygon-vertices* | 2.6.1 | – |
| – | $2 \times V$ | – | – | Previous two vertices in a **Begin/End triangle strip** | 2.6.1 | – |
| – | $Z_3$ | – | – | Number of vertices so far in triangle strip: 0, 1, or more | 2.6.1 | – |
| – | $Z_2$ | – | – | Triangle strip A/B vertex pointer | 2.6.1 | – |
| – | $3 \times V$ | – | – | Vertices of the quad under construction | 2.6.1 | – |
| – | $Z_4$ | – | – | Number of vertices so far in quad strip: 0, 1, 2, or more | 2.6.1 | – |

Table 6.4. GL Internal begin-end state variables (inaccessible)

APPENDIX T

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| CURRENT_COLOR | $C$ | **GetIntegerv, GetFloatv** | 1,1,1,1 | Current color | 2.7 | current |
| CURRENT_INDEX | $CI$ | **GetIntegerv, GetFloatv** | 1 | Current color index | 2.7 | current |
| CURRENT_TEXTURE_COORDS | $T$ | **GetFloatv** | 0,0,0,1 | Current texture coordinates | 2.7 | current |
| CURRENT_NORMAL | $N$ | **GetFloatv** | 0,0,1 | Current normal | 2.7 | current |
| – | $C$ | – | - | Color associated with last vertex | 2.6 | – |
| – | $CI$ | – | - | Color index associated with last vertex | 2.6 | – |
| – | $T$ | – | - | Texture coordinates associated with last vertex | 2.6 | – |
| CURRENT_RASTER_POSITION | $R^4$ | **GetFloatv** | 0,0,0,1 | Current raster position | 2.12 | current |
| CURRENT_RASTER_DISTANCE | $R^+$ | **GetFloatv** | 0 | Current raster distance | 2.12 | current |
| CURRENT_RASTER_COLOR | $C$ | **GetIntegerv, GetFloatv** | 1,1,1,1 | Color associated with raster position | 2.12 | current |
| CURRENT_RASTER_INDEX | $CI$ | **GetIntegerv, GetFloatv** | 1 | Color index associated with raster position | 2.12 | current |
| CURRENT_RASTER_TEXTURE_COORDS | $T$ | **GetFloatv** | 0,0,0,1 | Texture coordinates associated with raster position | 2.12 | current |
| CURRENT_RASTER_POSITION_VALID | $B$ | **GetBooleanv** | *True* | Raster position valid bit | 2.12 | current |
| EDGE_FLAG | $B$ | **GetBooleanv** | *True* | Edge flag | 2.6.2 | current |

Table 6.5. Current Values and Associated Data

APPENDIX T

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| VERTEX_ARRAY | $B$ | **IsEnabled** | *False* | Vertex array enable | 2.8 | vertex-array |
| VERTEX_ARRAY_SIZE | $Z^+$ | **GetIntegerv** | 4 | Coordinates per vertex | 2.8 | vertex-array |
| VERTEX_ARRAY_TYPE | $Z_4$ | **GetIntegerv** | FLOAT | Type of vertex coordinates | 2.8 | vertex-array |
| VERTEX_ARRAY_STRIDE | $Z^+$ | **GetIntegerv** | 0 | Stride between vertices | 2.8 | vertex-array |
| VERTEX_ARRAY_POINTER | $Y$ | **GetPointerv** | 0 | Pointer to the vertex array | 2.8 | vertex-array |
| NORMAL_ARRAY | $B$ | **IsEnabled** | *False* | Normal array enable | 2.8 | vertex-array |
| NORMAL_ARRAY_TYPE | $Z_5$ | **GetIntegerv** | FLOAT | Type of normal coordinates | 2.8 | vertex-array |
| NORMAL_ARRAY_STRIDE | $Z^+$ | **GetIntegerv** | 0 | Stride between normals | 2.8 | vertex-array |
| NORMAL_ARRAY_POINTER | $Y$ | **GetPointerv** | 0 | Pointer to the normal array | 2.8 | vertex-array |
| COLOR_ARRAY | $B$ | **IsEnabled** | *False* | Color array enable | 2.8 | vertex-array |
| COLOR_ARRAY_SIZE | $Z^+$ | **GetIntegerv** | 4 | Colors per vertex | 2.8 | vertex-array |
| COLOR_ARRAY_TYPE | $Z_8$ | **GetIntegerv** | FLOAT | Type of color components | 2.8 | vertex-array |
| COLOR_ARRAY_STRIDE | $Z^+$ | **GetIntegerv** | 0 | Stride between colors | 2.8 | vertex-array |
| COLOR_ARRAY_POINTER | $Y$ | **GetPointerv** | 0 | Pointer to the color array | 2.8 | vertex-array |
| INDEX_ARRAY | $B$ | **IsEnabled** | *False* | Index array enable | 2.8 | vertex-array |
| INDEX_ARRAY_TYPE | $Z_4$ | **GetIntegerv** | FLOAT | Type of indices | 2.8 | vertex-array |
| INDEX_ARRAY_STRIDE | $Z^+$ | **GetIntegerv** | 0 | Stride between indices | 2.8 | vertex-array |
| INDEX_ARRAY_POINTER | $Y$ | **GetPointerv** | 0 | Pointer to the index array | 2.8 | vertex-array |
| TEXTURE_COORD_ARRAY | $B$ | **IsEnabled** | *False* | Texture coordinate array enable | 2.8 | vertex-array |
| TEXTURE_COORD_ARRAY_SIZE | $Z^+$ | **GetIntegerv** | 4 | Coordinates per element | 2.8 | vertex-array |
| TEXTURE_COORD_ARRAY_TYPE | $Z_4$ | **GetIntegerv** | FLOAT | Type of texture coordinates | 2.8 | vertex-array |
| TEXTURE_COORD_ARRAY_STRIDE | $Z^+$ | **GetIntegerv** | 0 | Stride between texture coordinates | 2.8 | vertex-array |
| TEXTURE_COORD_ARRAY_POINTER | $Y$ | **GetPointerv** | 0 | Pointer to the texture coordinate array | 2.8 | vertex-array |
| EDGE_FLAG_ARRAY | $B$ | **IsEnabled** | *False* | Edge flag array enable | 2.8 | vertex-array |
| EDGE_FLAG_ARRAY_STRIDE | $Z^+$ | **GetIntegerv** | 0 | Stride between edge flags | 2.8 | vertex-array |
| EDGE_FLAG_ARRAY_POINTER | $Y$ | **GetPointerv** | 0 | Pointer to the edge flag array | 2.8 | vertex-array |

Table 6.6. Vertex Array Data

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| COLOR_MATRIX | $2 * \times M^4$ | **GetFloatv** | Identity | Color matrix stack | 3.6.3 | – |
| MODELVIEW_MATRIX | $32 * \times M^4$ | **GetFloatv** | Identity | Model-view matrix stack | 2.10.2 | – |
| PROJECTION_MATRIX | $2 * \times M^4$ | **GetFloatv** | Identity | Projection matrix stack | 2.10.2 | – |
| TEXTURE_MATRIX | $2 * \times M^4$ | **GetFloatv** | Identity | Texture matrix stack | 2.10.2 | – |
| VIEWPORT | $4 \times Z$ | **GetIntegerv** | see 2.10.1 | Viewport origin & extent | 2.10.1 | viewport |
| DEPTH_RANGE | $2 \times R^+$ | **GetFloatv** | 0,1 | Depth range near & far | 2.10.1 | viewport |
| COLOR_MATRIX_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 1 | Color matrix stack pointer | 3.6.3 | – |
| MODELVIEW_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 1 | Model-view matrix stack pointer | 2.10.2 | – |
| PROJECTION_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 1 | Projection matrix stack pointer | 2.10.2 | – |
| TEXTURE_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 1 | Texture matrix stack pointer | 2.10.2 | – |
| MATRIX_MODE | $Z_4$ | **GetIntegerv** | MODELVIEW | Current matrix mode | 2.10.2 | transform |
| NORMALIZE | $B$ | **IsEnabled** | *False* | Current normal normalization on/off | 2.10.3 | transform/enable |
| RESCALE_NORMAL | $B$ | **IsEnabled** | *False* | Current normal rescaling on/off | 2.10.3 | transform/enable |
| CLIP_PLANE*i* | $6 * \times R^4$ | **GetClipPlane** | 0,0,0,0 | User clipping plane coefficients | 2.11 | transform |
| CLIP_PLANE*i* | $6 * \times B$ | **IsEnabled** | *False* | *i*th user clipping plane enabled | 2.11 | transform/enable |

Table 6.7. Transformation state

APPENDIX T

| Get value | Type | Get<br>Cmnd | Initial<br>Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| FOG_COLOR | $C$ | **GetFloatv** | 0,0,0,0 | Fog color | 3.10 | fog |
| FOG_INDEX | $CI$ | **GetFloatv** | 0 | Fog index | 3.10 | fog |
| FOG_DENSITY | $R$ | **GetFloatv** | 1.0 | Exponential fog density | 3.10 | fog |
| FOG_START | $R$ | **GetFloatv** | 0.0 | Linear fog start | 3.10 | fog |
| FOG_END | $R$ | **GetFloatv** | 1.0 | Linear fog end | 3.10 | fog |
| FOG_MODE | $Z_3$ | **GetIntegerv** | EXP | Fog mode | 3.10 | fog |
| FOG | $B$ | **IsEnabled** | *False* | True if fog enabled | 3.10 | fog/enable |
| SHADE_MODEL | $Z^+$ | **GetIntegerv** | SMOOTH | **ShadeModel** setting | 2.13.7 | lighting |

Table 6.8. Coloring

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| LIGHTING | $B$ | **IsEnabled** | *False* | True if lighting is enabled | 2.13.1 | lighting/enable |
| COLOR_MATERIAL | $B$ | **IsEnabled** | *False* | True if color tracking is enabled | 2.13.3 | lighting/enable |
| COLOR_MATERIAL_PARAMETER | $Z_5$ | **GetIntegerv** | `AMBIENT_AND_DIFFUSE` | Material properties tracking current color | 2.13.3 | lighting |
| COLOR_MATERIAL_FACE | $Z_3$ | **GetIntegerv** | `FRONT_AND_BACK` | Face(s) affected by color tracking | 2.13.3 | lighting |
| AMBIENT | $2 \times C$ | **GetMaterialfv** | (0.2,0.2,0.2,1.0) | Ambient material color | 2.13.1 | lighting |
| DIFFUSE | $2 \times C$ | **GetMaterialfv** | (0.8,0.8,0.8,1.0) | Diffuse material color | 2.13.1 | lighting |
| SPECULAR | $2 \times C$ | **GetMaterialfv** | (0.0,0.0,0.0,1.0) | Specular material color | 2.13.1 | lighting |
| EMISSION | $2 \times C$ | **GetMaterialfv** | (0.0,0.0,0.0,1.0) | Emissive mat. color | 2.13.1 | lighting |
| SHININESS | $2 \times R$ | **GetMaterialfv** | 0.0 | Specular exponent of material | 2.13.1 | lighting |
| LIGHT_MODEL_AMBIENT | $C$ | **GetFloatv** | (0.2,0.2,0.2,1.0) | Ambient scene color | 2.13.1 | lighting |
| LIGHT_MODEL_LOCAL_VIEWER | $B$ | **GetBooleanv** | *False* | Viewer is local | 2.13.1 | lighting |
| LIGHT_MODEL_TWO_SIDE | $B$ | **GetBooleanv** | *False* | Use two-sided lighting | 2.13.1 | lighting |
| LIGHT_MODEL_COLOR_CONTROL | $Z_2$ | **GetIntegerv** | `SINGLE_COLOR` | Color control | 2.13.1 | lighting |

Table 6.9. Lighting (see also Table 2.7 for defaults)

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| AMBIENT | $8 * \times C$ | **GetLightfv** | (0.0,0.0,0.0,1.0) | Ambient intensity of light $i$ | 2.13.1 | lighting |
| DIFFUSE | $8 * \times C$ | **GetLightfv** | see 2.5 | Diffuse intensity of light $i$ | 2.13.1 | lighting |
| SPECULAR | $8 * \times C$ | **GetLightfv** | see 2.5 | Specular intensity of light $i$ | 2.13.1 | lighting |
| POSITION | $8 * \times P$ | **GetLightfv** | (0.0,0.0,1.0,0.0) | Position of light $i$ | 2.13.1 | lighting |
| CONSTANT_ATTENUATION | $8 * \times R^+$ | **GetLightfv** | 1.0 | Constant atten. factor | 2.13.1 | lighting |
| LINEAR_ATTENUATION | $8 * \times R^+$ | **GetLightfv** | 0.0 | Linear atten. factor | 2.13.1 | lighting |
| QUADRATIC_ATTENUATION | $8 * \times R^+$ | **GetLightfv** | 0.0 | Quadratic atten. factor | 2.13.1 | lighting |
| SPOT_DIRECTION | $8 * \times D$ | **GetLightfv** | (0.0,0.0,-1.0) | Spotlight direction of light $i$ | 2.13.1 | lighting |
| SPOT_EXPONENT | $8 * \times R^+$ | **GetLightfv** | 0.0 | Spotlight exponent of light $i$ | 2.13.1 | lighting |
| SPOT_CUTOFF | $8 * \times R^+$ | **GetLightfv** | 180.0 | Spot. angle of light $i$ | 2.13.1 | lighting |
| LIGHT$i$ | $8 * \times B$ | **IsEnabled** | *False* | True if light $i$ enabled | 2.13.1 | lighting/enable |
| COLOR_INDEXES | $2 \times 3 \times R$ | **GetMaterialfv** | 0,1,1 | $a_m$, $d_m$, and $s_m$ for color index lighting | 2.13.1 | lighting |

Table 6.10. Lighting (cont.)

APPENDIX T

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| POINT_SIZE | $R^+$ | **GetFloatv** | 1.0 | Point size | 3.3 | point |
| POINT_SMOOTH | $B$ | **IsEnabled** | *False* | Point antialiasing on | 3.3 | point/enable |
| LINE_WIDTH | $R^+$ | **GetFloatv** | 1.0 | Line width | 3.4 | line |
| LINE_SMOOTH | $B$ | **IsEnabled** | *False* | Line antialiasing on | 3.4 | line/enable |
| LINE_STIPPLE_PATTERN | $Z^+$ | **GetIntegerv** | 1's | Line stipple | 3.4.2 | line |
| LINE_STIPPLE_REPEAT | $Z^+$ | **GetIntegerv** | 1 | Line stipple repeat | 3.4.2 | line |
| LINE_STIPPLE | $B$ | **IsEnabled** | *False* | Line stipple enable | 3.4.2 | line/enable |
| CULL_FACE | $B$ | **IsEnabled** | *False* | Polygon culling enabled | 3.5.1 | polygon/enable |
| CULL_FACE_MODE | $Z_3$ | **GetIntegerv** | `BACK` | Cull front/back facing polygons | 3.5.1 | polygon |
| FRONT_FACE | $Z_2$ | **GetIntegerv** | `CCW` | Polygon frontface CW/CCW indicator | 3.5.1 | polygon |
| POLYGON_SMOOTH | $B$ | **IsEnabled** | *False* | Polygon antialiasing on | 3.5 | polygon/enable |
| POLYGON_MODE | $2 \times Z_3$ | **GetIntegerv** | `FILL` | Polygon rasterization mode (front & back) | 3.5.4 | polygon |
| POLYGON_OFFSET_FACTOR | $R$ | **GetFloatv** | 0 | Polygon offset factor | 3.5.5 | polygon |
| POLYGON_OFFSET_UNITS | $R$ | **GetFloatv** | 0 | Polygon offset bias | 3.5.5 | polygon |
| POLYGON_OFFSET_POINT | $B$ | **IsEnabled** | *False* | Polygon offset enable for `POINT` mode rasterization | 3.5.5 | polygon/enable |
| POLYGON_OFFSET_LINE | $B$ | **IsEnabled** | *False* | Polygon offset enable for `LINE` mode rasterization | 3.5.5 | polygon/enable |
| POLYGON_OFFSET_FILL | $B$ | **IsEnabled** | *False* | Polygon offset enable for `FILL` mode rasterization | 3.5.5 | polygon/enable |
| – | $I$ | **GetPolygonStipple** | 1's | Polygon stipple | 3.5 | polygon-stipple |
| POLYGON_STIPPLE | $B$ | **IsEnabled** | *False* | Polygon stipple enable | 3.5.2 | polygon/enable |

Table 6.11. Rasterization

APPENDIX T

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| TEXTURE_$x$D | $3 \times B$ | **IsEnabled** | *False* | True if $x$D texturing is enabled; $x$ is 1, 2, or 3 | 3.8.10 | texture/enable |
| TEXTURE_BINDING_$x$D | $3 \times Z^+$ | **GetIntegerv** | 0 | Texture object bound to TEXTURE_$x$D | 3.8.8 | texture |
| TEXTURE_$x$D | $n \times I$ | **GetTexImage** | see 3.8 | $x$D texture image at l.o.d. $i$ | 3.8 | – |
| TEXTURE_WIDTH | $n \times Z^+$ | **GetTexLevelParameter** | 0 | $x$D texture image $i$'s specified width | 3.8 | – |
| TEXTURE_HEIGHT | $n \times Z^+$ | **GetTexLevelParameter** | 0 | 2D texture image $i$'s specified height | 3.8 | – |
| TEXTURE_DEPTH | $n \times Z^+$ | **GetTexLevelParameter** | 0 | 3D texture image $i$'s specified depth | 3.8 | – |
| TEXTURE_BORDER | $n \times Z^+$ | **GetTexLevelParameter** | 0 | $x$D texture image $i$'s specified border width | 3.8 | – |
| TEXTURE_INTERNAL_FORMAT (TEXTURE_COMPONENTS) | $n \times Z_{42}$ | **GetTexLevelParameter** | 1 | $x$D texture image $i$'s internal image format | 3.8 | – |
| TEXTURE_RED_SIZE | $n \times Z^+$ | **GetTexLevelParameter** | 0 | $x$D texture image $i$'s red resolution | 3.8 | – |
| TEXTURE_GREEN_SIZE | $n \times Z^+$ | **GetTexLevelParameter** | 0 | $x$D texture image $i$'s green resolution | 3.8 | – |
| TEXTURE_BLUE_SIZE | $n \times Z^+$ | **GetTexLevelParameter** | 0 | $x$D texture image $i$'s blue resolution | 3.8 | – |
| TEXTURE_ALPHA_SIZE | $n \times Z^+$ | **GetTexLevelParameter** | 0 | $x$D texture image $i$'s alpha resolution | 3.8 | – |
| TEXTURE_LUMINANCE_SIZE | $n \times Z^+$ | **GetTexLevelParameter** | 0 | $x$D texture image $i$'s luminance resolution | 3.8 | – |
| TEXTURE_INTENSITY_SIZE | $n \times Z^+$ | **GetTexLevelParameter** | 0 | $x$D texture image $i$'s intensity resolution | 3.8 | – |

Table 6.12. Texture Objects

APPENDIX T

| Get value | Type | Get<br>Cmnd | Initial<br>Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| TEXTURE_BORDER_COLOR | $2^+ \times C$ | **GetTexParameter** | 0,0,0,0 | Texture border color | 3.8 | texture |
| TEXTURE_MIN_FILTER | $2^+ \times Z_6$ | **GetTexParameter** | see 3.8 | Texture minification function | 3.8.5 | texture |
| TEXTURE_MAG_FILTER | $2^+ \times Z_2$ | **GetTexParameter** | see 3.8 | Texture magnification function | 3.8.6 | texture |
| TEXTURE_WRAP_S | $3^+ \times Z_3$ | **GetTexParameter** | REPEAT | Texture wrap mode S | 3.8 | texture |
| TEXTURE_WRAP_T | $2^+ \times Z_3$ | **GetTexParameter** | REPEAT | Texture wrap mode T | 3.8 | texture |
| TEXTURE_WRAP_R | $1^+ \times Z_3$ | **GetTexParameter** | REPEAT | Texture wrap mode R | 3.8 | texture |
| TEXTURE_PRIORITY | $2^+ \times R^{[0,1]}$ | **GetTexParameterfv** | 1 | Texture object priority | 3.8.8 | texture |
| TEXTURE_RESIDENT | $2^+ \times B$ | **GetTexParameteriv** | see 3.8.8 | Texture residency | 3.8.8 | texture |
| TEXTURE_MIN_LOD | $n \times R$ | **GetTexParameterfv** | -1000 | Minimum level of detail | 3.8 | texture |
| TEXTURE_MAX_LOD | $n \times R$ | **GetTexParameterfv** | 1000 | Maximum level of detail | 3.8 | texture |
| TEXTURE_BASE_LEVEL | $n \times R$ | **GetTexParameterfv** | 0 | Base texture array | 3.8 | texture |
| TEXTURE_MAX_LEVEL | $n \times R$ | **GetTexParameterfv** | 1000 | Maximum texture array level | 3.8 | texture |

Table 6.13. Texture Objects (cont.)

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| TEXTURE_ENV_MODE | $Z_4$ | **GetTexEnviv** | MODULATE | Texture application function | 3.8.9 | texture |
| TEXTURE_ENV_COLOR | $C$ | **GetTexEnvfv** | 0,0,0,0 | Texture environment color | 3.8.9 | texture |
| TEXTURE_GEN_$x$ | $4 \times B$ | **IsEnabled** | *False* | Texgen enabled ($x$ is S, T, R, or Q) | 2.10.4 | texture/enable |
| EYE_PLANE | $4 \times R^4$ | **GetTexGenfv** | see 2.10.4 | Texgen plane equation coefficients (for S, T, R, and Q) | 2.10.4 | texture |
| OBJECT_PLANE | $4 \times R^4$ | **GetTexGenfv** | see 2.10.4 | Texgen object linear coefficients (for S, T, R, and Q) | 2.10.4 | texture |
| TEXTURE_GEN_MODE | $4 \times Z_3$ | **GetTexGeniv** | EYE_LINEAR | Function used for texgen (for S, T, R, and Q | 2.10.4 | texture |

Table 6.14. Texture Environment and Generation

APPENDIX T

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| SCISSOR_TEST | $B$ | **IsEnabled** | *False* | Scissoring enabled | 4.1.2 | scissor/enable |
| SCISSOR_BOX | $4 \times Z$ | **GetIntegerv** | see 4.1.2 | Scissor box | 4.1.2 | scissor |
| ALPHA_TEST | $B$ | **IsEnabled** | *False* | Alpha test enabled | 4.1.3 | color-buffer/enable |
| ALPHA_TEST_FUNC | $Z_8$ | **GetIntegerv** | ALWAYS | Alpha test function | 4.1.3 | color-buffer |
| ALPHA_TEST_REF | $R^+$ | **GetIntegerv** | 0 | Alpha test reference value | 4.1.3 | color-buffer |
| STENCIL_TEST | $B$ | **IsEnabled** | *False* | Stenciling enabled | 4.1.4 | stencil-buffer/enable |
| STENCIL_FUNC | $Z_8$ | **GetIntegerv** | ALWAYS | Stencil function | 4.1.4 | stencil-buffer |
| STENCIL_VALUE_MASK | $Z^+$ | **GetIntegerv** | 1's | Stencil mask | 4.1.4 | stencil-buffer |
| STENCIL_REF | $Z^+$ | **GetIntegerv** | 0 | Stencil reference value | 4.1.4 | stencil-buffer |
| STENCIL_FAIL | $Z_6$ | **GetIntegerv** | KEEP | Stencil fail action | 4.1.4 | stencil-buffer |
| STENCIL_PASS_DEPTH_FAIL | $Z_6$ | **GetIntegerv** | KEEP | Stencil depth buffer fail action | 4.1.4 | stencil-buffer |
| STENCIL_PASS_DEPTH_PASS | $Z_6$ | **GetIntegerv** | KEEP | Stencil depth buffer pass action | 4.1.4 | stencil-buffer |
| DEPTH_TEST | $B$ | **IsEnabled** | *False* | Depth buffer enabled | 4.1.5 | depth-buffer/enable |
| DEPTH_FUNC | $Z_8$ | **GetIntegerv** | LESS | Depth buffer test function | 4.1.5 | depth-buffer |
| BLEND | $B$ | **IsEnabled** | *False* | Blending enabled | 4.1.6 | color-buffer/enable |
| BLEND_SRC | $Z_{13}$ | **GetIntegerv** | ONE | Blending source function | 4.1.6 | color-buffer |
| BLEND_DST | $Z_{12}$ | **GetIntegerv** | ZERO | Blending destination function | 4.1.6 | color-buffer |
| BLEND_EQUATION | $Z_5$ | **GetIntegerv** | FUNC_ADD | Blending equation | 4.1.6 | color-buffer |
| BLEND_COLOR | $C$ | **GetFloatv** | 0,0,0,0 | Constant blend color | 4.1.6 | color-buffer |
| DITHER | $B$ | **IsEnabled** | *True* | Dithering enabled | 4.1.7 | color-buffer/enable |
| INDEX_LOGIC_OP (v1.0: GL_LOGIC_OP) | $B$ | **IsEnabled** | *False* | Index logic op enabled | 4.1.8 | color-buffer/enable |
| COLOR_LOGIC_OP | $B$ | **IsEnabled** | *False* | Color logic op enabled | 4.1.8 | color-buffer/enable |
| LOGIC_OP_MODE | $Z_{16}$ | **GetIntegerv** | COPY | Logic op function | 4.1.8 | color-buffer |

Table 6.15. Pixel Operations

APPENDIX T

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| DRAW_BUFFER | $Z_{10*}$ | **GetIntegerv** | see 4.2.1 | Buffers selected for drawing | 4.2.1 | color-buffer |
| INDEX_WRITEMASK | $Z^+$ | **GetIntegerv** | 1's | Color index writemask | 4.2.2 | color-buffer |
| COLOR_WRITEMASK | $4 \times B$ | **GetBooleanv** | *True* | Color write enables; R, G, B, or A | 4.2.2 | color-buffer |
| DEPTH_WRITEMASK | $B$ | **GetBooleanv** | *True* | Depth buffer enabled for writing | 4.2.2 | depth-buffer |
| STENCIL_WRITEMASK | $Z^+$ | **GetIntegerv** | 1's | Stencil buffer writemask | 4.2.2 | stencil-buffer |
| COLOR_CLEAR_VALUE | $C$ | **GetFloatv** | 0,0,0,0 | Color buffer clear value (RGBA mode) | 4.2.3 | color-buffer |
| INDEX_CLEAR_VALUE | $CI$ | **GetFloatv** | 0 | Color buffer clear value (color index mode) | 4.2.3 | color-buffer |
| DEPTH_CLEAR_VALUE | $R^+$ | **GetIntegerv** | 1 | Depth buffer clear value | 4.2.3 | depth-buffer |
| STENCIL_CLEAR_VALUE | $Z^+$ | **GetIntegerv** | 0 | Stencil clear value | 4.2.3 | stencil-buffer |
| ACCUM_CLEAR_VALUE | $4 \times R^+$ | **GetFloatv** | 0 | Accumulation buffer clear value | 4.2.3 | accum-buffer |

Table 6.16. Framebuffer Control

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| UNPACK_SWAP_BYTES | $B$ | **GetBooleanv** | *False* | Value of `UNPACK_SWAP_BYTES` | 4.3 | pixel-store |
| UNPACK_LSB_FIRST | $B$ | **GetBooleanv** | *False* | Value of `UNPACK_LSB_FIRST` | 4.3 | pixel-store |
| UNPACK_IMAGE_HEIGHT | $Z^+$ | **GetIntegerv** | 0 | Value of `UNPACK_IMAGE_HEIGHT` | 4.3 | pixel-store |
| UNPACK_SKIP_IMAGES | $Z^+$ | **GetIntegerv** | 0 | Value of `UNPACK_SKIP_IMAGES` | 4.3 | pixel-store |
| UNPACK_ROW_LENGTH | $Z^+$ | **GetIntegerv** | 0 | Value of `UNPACK_ROW_LENGTH` | 4.3 | pixel-store |
| UNPACK_SKIP_ROWS | $Z^+$ | **GetIntegerv** | 0 | Value of `UNPACK_SKIP_ROWS` | 4.3 | pixel-store |
| UNPACK_SKIP_PIXELS | $Z^+$ | **GetIntegerv** | 0 | Value of `UNPACK_SKIP_PIXELS` | 4.3 | pixel-store |
| UNPACK_ALIGNMENT | $Z^+$ | **GetIntegerv** | 4 | Value of `UNPACK_ALIGNMENT` | 4.3 | pixel-store |
| PACK_SWAP_BYTES | $B$ | **GetBooleanv** | *False* | Value of `PACK_SWAP_BYTES` | 4.3 | pixel-store |
| PACK_LSB_FIRST | $B$ | **GetBooleanv** | *False* | Value of `PACK_LSB_FIRST` | 4.3 | pixel-store |
| PACK_IMAGE_HEIGHT | $Z^+$ | **GetIntegerv** | 0 | Value of `PACK_IMAGE_HEIGHT` | 4.3 | pixel-store |
| PACK_SKIP_IMAGES | $Z^+$ | **GetIntegerv** | 0 | Value of `PACK_SKIP_IMAGES` | 4.3 | pixel-store |
| PACK_ROW_LENGTH | $Z^+$ | **GetIntegerv** | 0 | Value of `PACK_ROW_LENGTH` | 4.3 | pixel-store |
| PACK_SKIP_ROWS | $Z^+$ | **GetIntegerv** | 0 | Value of `PACK_SKIP_ROWS` | 4.3 | pixel-store |
| PACK_SKIP_PIXELS | $Z^+$ | **GetIntegerv** | 0 | Value of `PACK_SKIP_PIXELS` | 4.3 | pixel-store |
| PACK_ALIGNMENT | $Z^+$ | **GetIntegerv** | 4 | Value of `PACK_ALIGNMENT` | 4.3 | pixel-store |

Table 6.17. Pixels

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| MAP_COLOR | $B$ | **GetBooleanv** | *False* | True if colors are mapped | 4.3 | pixel |
| MAP_STENCIL | $B$ | **GetBooleanv** | *False* | True if stencil values are mapped | 4.3 | pixel |
| INDEX_SHIFT | $Z$ | **GetIntegerv** | 0 | Value of `INDEX_SHIFT` | 4.3 | pixel |
| INDEX_OFFSET | $Z$ | **GetIntegerv** | 0 | Value of `INDEX_OFFSET` | 4.3 | pixel |
| $x$_SCALE | $R$ | **GetFloatv** | 1 | Value of $x$`_SCALE`; $x$ is `RED`, `GREEN`, `BLUE`, `ALPHA`, or `DEPTH` | 4.3 | pixel |
| $x$_BIAS | $R$ | **GetFloatv** | 0 | Value of $x$`_BIAS`; $x$ is one of `RED`, `GREEN`, `BLUE`, `ALPHA`, or `DEPTH` | 4.3 | pixel |
| COLOR_TABLE | $B$ | **IsEnabled** | *False* | True if color table lookup is done | 3.6.3 | pixel/enable |
| POST_CONVOLUTION_COLOR_TABLE | $B$ | **IsEnabled** | *False* | True if post convolution color table lookup is done | 3.6.3 | pixel/enable |
| POST_COLOR_MATRIX_COLOR_TABLE | $B$ | **IsEnabled** | *False* | True if post color matrix color table lookup is done | 3.6.3 | pixel/enable |
| COLOR_TABLE | $3 \times I$ | **GetColorTable** | *empty* | Color tables | 3.6.3 | – |
| COLOR_TABLE_FORMAT | $2 \times 3 \times Z_{42}$ | **GetColorTable-Parameteriv** | `RGBA` | Color tables' internal image format | 3.6.3 | – |
| COLOR_TABLE_WIDTH | $2 \times 3 \times Z^{+}$ | **GetColorTable-Parameteriv** | 0 | Color tables' specified width | 3.6.3 | – |
| COLOR_TABLE_$x$_SIZE | $6 \times 2 \times 3 \times Z^{+}$ | **GetColorTable-Parameteriv** | 0 | Color table component resolution; $x$ is `RED`, `GREEN`, `BLUE`, `ALPHA`, `LUMINANCE`, or `INTENSITY` | 3.6.3 | – |
| COLOR_TABLE_SCALE | $3 \times R^4$ | **GetColorTable-Parameterfv** | 1,1,1,1 | Scale factors applied to color table entries | 3.6.3 | pixel |
| COLOR_TABLE_BIAS | $3 \times R^4$ | **GetColorTable-Parameterfv** | 0,0,0,0 | Bias factors applied to color table entries | 3.6.3 | pixel |

Table 6.18. Pixels (cont.)

APPENDIX T

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| CONVOLUTION_1D | $B$ | **IsEnabled** | *False* | True if 1D convolution is done | 3.6.3 | pixel/enable |
| CONVOLUTION_2D | $B$ | **IsEnabled** | *False* | True if 2D convolution is done | 3.6.3 | pixel/enable |
| SEPARABLE_2D | $B$ | **IsEnabled** | *False* | True if separable 2D convolution is done | 3.6.3 | pixel/enable |
| CONVOLUTION | $2 \times I$ | **GetConvolution-Filter** | *empty* | Convolution filters | 3.6.3 | – |
| CONVOLUTION | $2 \times I$ | **GetSeparable-Filter** | *empty* | Separable convolution filter | 3.6.3 | – |
| CONVOLUTION_BORDER_COLOR | $3 \times C$ | **GetConvolution-Parameterfv** | 0,0,0,0 | Convolution border color | 4.3 | pixel |
| CONVOLUTION_BORDER_MODE | $3 \times Z_4$ | **GetConvolution-Parameteriv** | REDUCE | Convolution border mode | 4.3 | pixel |
| CONVOLUTION_FILTER_SCALE | $3 \times R^4$ | **GetConvolution-Parameterfv** | 1,1,1,1 | Scale factors applied to convolution filter entries | 3.6.3 | pixel |
| CONVOLUTION_FILTER_BIAS | $3 \times R^4$ | **GetConvolution-Parameterfv** | 0,0,0,0 | Bias factors applied to convolution filter entries | 3.6.3 | pixel |
| CONVOLUTION_FORMAT | $3 \times Z_{42}$ | **GetConvolution-Parameteriv** | RGBA | Convolution filter internal format | 4.3 | – |
| CONVOLUTION_WIDTH | $3 \times Z^+$ | **GetConvolution-Parameteriv** | 0 | Convolution filter width | 4.3 | – |
| CONVOLUTION_HEIGHT | $2 \times Z^+$ | **GetConvolution-Parameteriv** | 0 | Convolution filter height | 4.3 | – |

Table 6.19. Pixels (cont.)

APPENDIX T

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| POST_CONVOLUTION_$x$_SCALE | $R$ | **GetFloatv** | 1 | Component scale factors after convolution: $x$ is `RED`, `GREEN`, `BLUE`, or `ALPHA` | 3.6.3 | pixel |
| POST_CONVOLUTION_$x$_BIAS | $R$ | **GetFloatv** | 0 | Component bias factors after convolution: $x$ is `RED`, `GREEN`, `BLUE`, or `ALPHA` | 3.6.3 | pixel |
| POST_COLOR_MATRIX_$x$_SCALE | $R$ | **GetFloatv** | 1 | Component scale factors after color matrix; $x$ is `RED`, `GREEN`, `BLUE`, or `ALPHA` | 3.6.3 | pixel |
| POST_COLOR_MATRIX_$x$_BIAS | $R$ | **GetFloatv** | 0 | Component bias factors after color matrix; $x$ is `RED`, `GREEN`, `BLUE`, or `ALPHA` | 3.6.3 | pixel |
| HISTOGRAM | $B$ | **IsEnabled** | False | True if histogramming is enabled | 3.6.3 | pixel/enable |
| HISTOGRAM | $I$ | **GetHistogram** | *empty* | Histogram table | 3.6.3 | – |
| HISTOGRAM_WIDTH | $2 \times Z^+$ | **GetHistogram-Parameteriv** | 0 | Histogram table width | 3.6.3 | – |
| HISTOGRAM_FORMAT | $2 \times Z_{42}$ | **GetHistogram-Parameteriv** | `RGBA` | Histogram table internal format | 3.6.3 | – |
| HISTOGRAM_$x$_SIZE | $5 \times 2 \times Z^+$ | **GetHistogram-Parameteriv** | 0 | Histogram table component resolution; $x$ is `RED`, `GREEN`, `BLUE`, `ALPHA`, or `LUMINANCE` | 3.6.3 | – |
| HISTOGRAM_SINK | $B$ | **GetHistogram-Parameteriv** | False | True if histogramming consumes pixel groups | 3.6.3 | – |

Table 6.20. Pixels (cont.)

Microsoft et al. Exhibit 1005

| Get value | Type | Get<br>Cmnd | Initial<br>Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| MINMAX | $B$ | **IsEnabled** | False | True if minmax is enabled | 3.6.3 | pixel/enable |
| MINMAX | $R^n$ | **GetMinmax** | (M,M,M,M),(m,m,m,m) | Minmax table | 3.6.3 | – |
| MINMAX_FORMAT | $Z_{42}$ | **GetMinmax-Parameteriv** | RGBA | Minmax table internal format | 3.6.3 | – |
| MINMAX_SINK | $B$ | **GetMinmax-Parameteriv** | False | True if minmax consumes pixel groups | 3.6.3 | – |
| ZOOM_X | $R$ | **GetFloatv** | 1.0 | $x$ zoom factor | 4.3 | pixel |
| ZOOM_Y | $R$ | **GetFloatv** | 1.0 | $y$ zoom factor | 4.3 | pixel |
| $x$ | $8 \times 32 * \times R$ | **GetPixelMap** | 0's | RGBA **PixelMap** translation tables; $x$ is a map name from Table 3.3 | 4.3 | – |
| $x$ | $2 \times 32 * \times Z$ | **GetPixelMap** | 0's | Index **PixelMap** translation tables; $x$ is a map name from Table 3.3 | 4.3 | – |
| $x$_SIZE | $Z^+$ | **GetIntegerv** | 1 | Size of table $x$ | 4.3 | – |
| READ_BUFFER | $Z_3$ | **GetIntegerv** | see 4.3.2 | Read source buffer | 4.3 | pixel |

Table 6.21. Pixels (cont.)

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| ORDER | $9 \times Z_{8*}$ | **GetMapiv** | 1 | 1d map order | 5.1 | – |
| ORDER | $9 \times 2 \times Z_{8*}$ | **GetMapiv** | 1,1 | 2d map orders | 5.1 | – |
| COEFF | $9 \times 8 * \times R^n$ | **GetMapfv** | see 5.1 | 1d control points | 5.1 | – |
| COEFF | $9 \times 8 * \times 8 * \times R^n$ | **GetMapfv** | see 5.1 | 2d control points | 5.1 | – |
| DOMAIN | $9 \times 2 \times R$ | **GetMapfv** | see 5.1 | 1d domain endpoints | 5.1 | – |
| DOMAIN | $9 \times 4 \times R$ | **GetMapfv** | see 5.1 | 2d domain endpoints | 5.1 | – |
| MAP1_$x$ | $9 \times B$ | **IsEnabled** | *False* | 1d map enables: $x$ is map type | 5.1 | eval/enable |
| MAP2_$x$ | $9 \times B$ | **IsEnabled** | *False* | 2d map enables: $x$ is map type | 5.1 | eval/enable |
| MAP1_GRID_DOMAIN | $2 \times R$ | **GetFloatv** | 0,1 | 1d grid endpoints | 5.1 | eval |
| MAP2_GRID_DOMAIN | $4 \times R$ | **GetFloatv** | 0,1;0,1 | 2d grid endpoints | 5.1 | eval |
| MAP1_GRID_SEGMENTS | $Z^+$ | **GetFloatv** | 1 | 1d grid divisions | 5.1 | eval |
| MAP2_GRID_SEGMENTS | $2 \times Z^+$ | **GetFloatv** | 1,1 | 2d grid divisions | 5.1 | eval |
| AUTO_NORMAL | $B$ | **IsEnabled** | *False* | True if automatic normal generation enabled | 5.1 | eval/enable |

Table 6.22. Evaluators (**GetMap** takes a map name)

APPENDIX T

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| PERSPECTIVE_CORRECTION_HINT | $Z_3$ | **GetIntegerv** | `DONT_CARE` | Perspective correction hint | 5.6 | hint |
| POINT_SMOOTH_HINT | $Z_3$ | **GetIntegerv** | `DONT_CARE` | Point smooth hint | 5.6 | hint |
| LINE_SMOOTH_HINT | $Z_3$ | **GetIntegerv** | `DONT_CARE` | Line smooth hint | 5.6 | hint |
| POLYGON_SMOOTH_HINT | $Z_3$ | **GetIntegerv** | `DONT_CARE` | Polygon smooth hint | 5.6 | hint |
| FOG_HINT | $Z_3$ | **GetIntegerv** | `DONT_CARE` | Fog hint | 5.6 | hint |

Table 6.23. Hints

APPENDIX T

| Get value | Type | Get Cmnd | Minimum Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| MAX_LIGHTS | $Z^+$ | **GetIntegerv** | 8 | Maximum number of lights | 2.13.1 | – |
| MAX_CLIP_PLANES | $Z^+$ | **GetIntegerv** | 6 | Maximum number of user clipping planes | 2.11 | – |
| MAX_COLOR_MATRIX_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 2 | Maximum color matrix stack depth | 3.6.3 | – |
| MAX_MODELVIEW_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 32 | Maximum model-view stack depth | 2.10.2 | – |
| MAX_PROJECTION_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 2 | Maximum projection matrix stack depth | 2.10.2 | – |
| MAX_TEXTURE_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 2 | Maximum number depth of texture matrix stack | 2.10.2 | – |
| SUBPIXEL_BITS | $Z^+$ | **GetIntegerv** | 4 | Number of bits of subpixel precision in screen $x_w$ and $y_w$ | 3 | – |
| MAX_3D_TEXTURE_SIZE | $Z^+$ | **GetIntegerv** | 16 | See the discussion in Section 3.8. | 3.8 | – |
| MAX_TEXTURE_SIZE | $Z^+$ | **GetIntegerv** | 64 | See the discussion in Section 3.8. | 3.8 | – |
| MAX_PIXEL_MAP_TABLE | $Z^+$ | **GetIntegerv** | 32 | Maximum size of a **PixelMap** translation table | 3.6.3 | – |
| MAX_NAME_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 64 | Maximum selection name stack depth | 5.2 | – |
| MAX_LIST_NESTING | $Z^+$ | **GetIntegerv** | 64 | Maximum display list call nesting | 5.4 | – |
| MAX_EVAL_ORDER | $Z^+$ | **GetIntegerv** | 8 | Maximum evaluator polynomial order | 5.1 | – |
| MAX_VIEWPORT_DIMS | $2 \times Z^+$ | **GetIntegerv** | see 2.10.1 | Maximum viewport dimensions | 2.10.1 | – |
| MAX_ATTRIB_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 16 | Maximum depth of the server attribute stack | 6 | – |
| MAX_CLIENT_ATTRIB_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 16 | Maximum depth of the client attribute stack | 6 | – |
| – | $3 \times Z^+$ | **-** | 32 | Maximum size of a color table | 3.6.3 | – |
| – | $Z^+$ | **-** | 32 | Maximum size of the histogram table | 3.6.3 | – |

Table 6.24. Implementation Dependent Values

Microsoft et al. Exhibit 1005

| Get value | Type | Get Cmnd | Minimum Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| AUX_BUFFERS | $Z^+$ | **GetIntegerv** | 0 | Number of auxiliary buffers | 4.2.1 | – |
| RGBA_MODE | $B$ | **GetBooleanv** | – | True if color buffers store rgba | 2.7 | – |
| INDEX_MODE | $B$ | **GetBooleanv** | – | True if color buffers store indexes | 2.7 | – |
| DOUBLEBUFFER | $B$ | **GetBooleanv** | – | True if front & back buffers exist | 4.2.1 | – |
| STEREO | $B$ | **GetBooleanv** | – | True if left & right buffers exist | 6 | – |
| ALIASED_POINT_SIZE_RANGE | $2 \times R^+$ | **GetFloatv** | 1,1 | Range (lo to hi) of aliased point sizes | 3.3 | – |
| SMOOTH_POINT_SIZE_RANGE<br>(v1.1: POINT_SIZE_RANGE) | $2 \times R^+$ | **GetFloatv** | 1,1 | Range (lo to hi) of antialiased point sizes | 3.3 | – |
| SMOOTH_POINT_SIZE_GRANULARITY<br>(v1.1: POINT_SIZE_GRANULARITY) | $R^+$ | **GetFloatv** | – | Antialiased point size granularity | 3.3 | – |
| ALIASED_LINE_WIDTH_RANGE | $2 \times R^+$ | **GetFloatv** | 1,1 | Range (lo to hi) of aliased line widths | 3.4 | – |
| SMOOTH_LINE_WIDTH_RANGE<br>(v1.1: LINE_WIDTH_RANGE) | $2 \times R^+$ | **GetFloatv** | 1,1 | Range (lo to hi) of antialiased line widths | 3.4 | – |
| SMOOTH_LINE_WIDTH_GRANULARITY<br>(v1.1: LINE_WIDTH_GRANULARITY) | $R^+$ | **GetFloatv** | – | Antialiased line width granularity | 3.4 | – |
| MAX_CONVOLUTION_WIDTH | $3 \times Z^+$ | **GetConvolutionParameteriv** | 3 | Maximum width of convolution filter | 4.3 | – |
| MAX_CONVOLUTION_HEIGHT | $2 \times Z^+$ | **GetConvolutionParameteriv** | 3 | Maximum height of convolution filter | 4.3 | – |
| MAX_ELEMENTS_INDICES | $Z^+$ | **GetIntegerv** | – | Recommended maximum number of **DrawRangeElements** indices | 2.8 | – |
| MAX_ELEMENTS_VERTICES | $Z^+$ | **GetIntegerv** | – | Recommended maximum number of **DrawRangeElements** vertices | 2.8 | – |

Table 6.25. More Implementation Dependent Values

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| $x$_BITS | $Z^+$ | **GetIntegerv** | - | Number of bits in $x$ color buffer component; $x$ is one of RED, GREEN, BLUE, ALPHA, or INDEX | 4 | – |
| DEPTH_BITS | $Z^+$ | **GetIntegerv** | - | Number of depth buffer planes | 4 | – |
| STENCIL_BITS | $Z^+$ | **GetIntegerv** | - | Number of stencil planes | 4 | – |
| ACCUM_$x$_BITS | $Z^+$ | **GetIntegerv** | - | Number of bits in $x$ accumulation buffer component ($x$ is RED, GREEN, BLUE, or ALPHA | 4 | – |

Table 6.26. Implementation Dependent Pixel Depths

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| LIST_BASE | $Z^+$ | **GetIntegerv** | 0 | Setting of **ListBase** | 5.4 | list |
| LIST_INDEX | $Z^+$ | **GetIntegerv** | 0 | number of display list under construction; 0 if none | 5.4 | – |
| LIST_MODE | $Z^+$ | **GetIntegerv** | 0 | Mode of display list under construction; undefined if none | 5.4 | – |
| – | $16 * \times A$ | – | empty | Server attribute stack | 6 | – |
| ATTRIB_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 0 | Server attribute stack pointer | 6 | – |
| – | $16 * \times A$ | – | empty | Client attribute stack | 6 | – |
| CLIENT_ATTRIB_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 0 | Client attribute stack pointer | 6 | – |
| NAME_STACK_DEPTH | $Z^+$ | **GetIntegerv** | 0 | Name stack depth | 5.2 | – |
| RENDER_MODE | $Z_3$ | **GetIntegerv** | RENDER | **RenderMode** setting | 5.2 | – |
| SELECTION_BUFFER_POINTER | $Y$ | **GetPointerv** | 0 | Selection buffer pointer | 5.2 | select |
| SELECTION_BUFFER_SIZE | $Z^+$ | **GetIntegerv** | 0 | Selection buffer size | 5.2 | select |
| FEEDBACK_BUFFER_POINTER | $Y$ | **GetPointerv** | 0 | Feedback buffer pointer | 5.3 | feedback |
| FEEDBACK_BUFFER_SIZE | $Z^+$ | **GetIntegerv** | 0 | Feedback buffer size | 5.3 | feedback |
| FEEDBACK_BUFFER_TYPE | $Z_5$ | **GetIntegerv** | 2D | Feedback type | 5.3 | feedback |
| – | $n \times Z_8$ | **GetError** | 0 | Current error code(s) | 2.5 | – |
| – | $n \times B$ | – | *False* | True if there is a corresponding error | 2.5 | – |

Table 6.27. Miscellaneous

APPENDIX T

# Appendix A

# Invariance

The OpenGL specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different GL implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

## A.1 Repeatability

The obvious and most fundamental case is repeated issuance of a series of GL commands. For any given GL and framebuffer state *vector*, and for any GL command, the resulting GL and framebuffer state must be identical whenever the command is executed on that initial GL and framebuffer state.

One purpose of repeatability is avoidance of visual artifacts when a double-buffered scene is redrawn. If rendering is not repeatable, swapping between two buffers rendered with the same command sequence may result in visible changes in the image. Such false motion is distracting to the viewer. Another reason for repeatability is testability.

Repeatability, while important, is a weak requirement. Given only repeatability as a requirement, two scenes rendered with one (small) polygon changed in position might differ at every pixel. Such a difference, while within the law of repeatability, is certainly not within its spirit. Additional invariance rules are desirable to ensure useful operation.

## A.2 Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different GL mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

- "Erasing" a primitive from the framebuffer by redrawing it, either in a different color or using the XOR logical operation.
- Using stencil operations to compute capping planes.

On the other hand, invariance rules can greatly increase the complexity of high-performance implementations of the GL. Even the weak repeatability requirement significantly constrains a parallel implementation of the GL. Because GL implementations are required to implement ALL GL capabilities, not just a convenient subset, those that utilize hardware acceleration are expected to alternate between hardware and software modules based on the current GL mode vector. A strong invariance requirement forces the behavior of the hardware and software modules to be identical, something that may be very difficult to achieve (for example, if the hardware does floating-point operations with different precision than the software).

What is desired is a compromise that results in many compliant, high-performance implementations, and in many software vendors choosing to port to OpenGL.

## A.3 Invariance Rules

For a given instantiation of an OpenGL rendering context:

**Rule 1** *For any given GL and framebuffer state vector, and for any given GL command, the resulting GL and framebuffer state must be identical each time the command is executed on that initial GL and framebuffer state.*

**Rule 2** *Changes to the following state values have no side effects (the use of any other state value is not affected by the change):*

**Required:**

- *Framebuffer contents (all bitplanes)*
- *The color buffers enabled for writing*

APPENDIX T

- *The values of matrices other than the top-of-stack matrices*
- *Scissor parameters (other than enable)*
- *Writemasks (color, index, depth, stencil)*
- *Clear values (color, index, depth, stencil, accumulation)*
- ○ *Current values (color, index, normal, texture coords, edgeflag)*
- ○ *Current raster color, index and texture coordinates.*
- ○ *Material properties (ambient, diffuse, specular, emission, shininess)*

**Strongly suggested:**

- *Matrix mode*
- *Matrix stack depths*
- *Alpha test parameters (other than enable)*
- *Stencil parameters (other than enable)*
- *Depth test parameters (other than enable)*
- *Blend parameters (other than enable)*
- *Logical operation parameters (other than enable)*
- *Pixel storage and transfer state*
- *Evaluator state (except as it affects the vertex data generated by the evaluators)*
- *Polygon offset parameters (other than enables, and except as they affect the depth values of fragments)*

**Corollary 1** *Fragment generation is invariant with respect to the state values marked with • in Rule 2.*

**Corollary 2** *The window coordinates (x, y, and z) of generated fragments are also invariant with respect to*

**Required:**

- *Current values (color, color index, normal, texture coords, edgeflag)*
- *Current raster color, color index, and texture coordinates*
- *Material properties (ambient, diffuse, specular, emission, shininess)*

APPENDIX T

**Rule 3** *The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it (the parameters that control the alpha test, for instance, are the alpha test enable, the alpha test function, and the alpha test reference value).*

**Corollary 3** *Images rendered into different color buffers sharing the same framebuffer, either simultaneously or separately using the same command sequence, are pixel identical.*

## A.4 What All This Means

Hardware accelerated GL implementations are expected to default to software operation when some GL state vectors are encountered. Even the weak repeatability requirement means, for example, that OpenGL implementations cannot apply hysteresis to this swap, but must instead guarantee that a given mode vector implies that a subsequent command *always* is executed in either the hardware or the software machine.

The stronger invariance rules constrain when the switch from hardware to software rendering can occur, given that the software and hardware renderers are not pixel identical. For example, the switch can be made when blending is enabled or disabled, but it should not be made when a change is made to the blending parameters.

Because floating point values may be represented using different formats in different renderers (hardware and software), many OpenGL state values may change subtly when renderers are swapped. This is the type of state value change that Rule 1 seeks to avoid.

Microsoft et al. Exhibit 1005

APPENDIX T

# Appendix B

# Corollaries

The following observations are derived from the body and the other appendixes of the specification. Absence of an observation from this list in no way impugns its veracity.

1. The CURRENT_RASTER_TEXTURE_COORDS must be maintained correctly at all times, including periods while texture mapping is not enabled, and when the GL is in color index mode.
2. When requested, texture coordinates returned in feedback mode are always valid, including periods while texture mapping is not enabled, and when the GL is in color index mode.
3. The error semantics of upward compatible OpenGL revisions may change. Otherwise, only additions can be made to upward compatible revisions.
4. GL query commands are not required to satisfy the semantics of the **Flush** or the **Finish** commands. All that is required is that the queried state be consistent with complete execution of all previously executed GL commands.
5. Application specified point size and line width must be returned as specified when queried. Implementation dependent clamping affects the values only while they are in use.
6. Bitmaps and pixel transfers do not cause selection hits.
7. The mask specified as the third argument to **StencilFunc** affects the operands of the stencil comparison function, but has no direct effect on

the update of the stencil buffer. The mask specified by **StencilMask** has no effect on the stencil comparison function; it limits the effect of the update of the stencil buffer.

8. Polygon shading is completed before the polygon mode is interpreted. If the shade model is `FLAT`, all of the points or lines generated by a single polygon will have the same color.

9. A display list is just a group of commands and arguments, so errors generated by commands in a display list must be generated when the list is executed. If the list is created in `COMPILE` mode, errors should not be generated while the list is being created.

10. **RasterPos** does not change the current raster index from its default value in an RGBA mode GL context. Likewise, **RasterPos** does not change the current raster color from its default value in a color index GL context. Both the current raster index and the current raster color can be queried, however, regardless of the color mode of the GL context.

11. A material property that is attached to the current color via **ColorMaterial** always takes the value of the current color. Attempts to change that material property via **Material** calls have no effect.

12. **Material** and **ColorMaterial** can be used to modify the RGBA material properties, even in a color index context. Likewise, **Material** can be used to modify the color index material properties, even in an RGBA context.

13. There is no atomicity requirement for OpenGL rendering commands, even at the fragment level.

14. Because rasterization of non-antialiased polygons is point sampled, polygons that have no area generate no fragments when they are rasterized in `FILL` mode, and the fragments generated by the rasterization of "narrow" polygons may not form a continuous array.

15. OpenGL does not force left- or right-handedness on any of its coordinates systems. Consider, however, the following conditions: (1) the object coordinate system is right-handed; (2) the only commands used to manipulate the model-view matrix are **Scale** (with positive scaling values only), **Rotate**, and **Translate**; (3) exactly one of either **Frustum** or **Ortho** is used to set the projection matrix; (4) the near value

APPENDIX T

is less than the far value for **DepthRange**. If these conditions are all satisfied, then the eye coordinate system is right-handed and the clip, normalized device, and window coordinate systems are left-handed.

16. ColorMaterial has no effect on color index lighting.

17. (No pixel dropouts or duplicates.) Let two polygons share an identical edge (that is, there exist vertices A and B of an edge of one polygon, and vertices C and D of an edge of the other polygon, and the coordinates of vertex A (resp. B) are identical to those of vertex C (resp. D), and the state of the the coordinate transfomations is identical when A, B, C, and D are specified). Then, when the fragments produced by rasterization of both polygons are taken together, each fragment intersecting the interior of the shared edge is produced exactly once.

18. OpenGL state continues to be modified in `FEEDBACK` mode and in `SELECT` mode. The contents of the framebuffer are not modified.

19. The current raster position, the user defined clip planes, the spot directions and the light positions for `LIGHT`$i$, and the eye planes for texgen are transformed when they are specified. They are not transformed during a **PopAttrib**, or when copying a context.

20. Dithering algorithms may be different for different components. In particular, alpha may be dithered differently from red, green, or blue, and an implementation may choose to not dither alpha at all.

21. For any GL and framebuffer state, and for any group of GL commands and arguments, the resulting GL and framebuffer state is identical whether the GL commands and arguments are executed normally or from a display list.

APPENDIX T

# Appendix C

# Version 1.1

OpenGL version 1.1 is the first revision since the original version 1.0 was released on 1 July 1992. Version 1.1 is upward compatible with version 1.0, meaning that any program that runs with a 1.0 GL implementation will also run unchanged with a 1.1 GL implementation. Several additions were made to the GL, especially to the texture mapping capabilities, but also to the geometry and fragment operations. Following are brief descriptions of each addition.

## C.1 Vertex Array

Arrays of vertex data may be transferred to the GL with many fewer commands than were previously necessary. Six arrays are defined, one each storing vertex positions, normal coordinates, colors, color indices, texture coordinates, and edge flags. The arrays may be specified and enabled independently, or one of the pre-defined configurations may be selected with a single command.

The primary goal was to decrease the number of subroutine calls required to transfer non-display listed geometry data to the GL. A secondary goal was to improve the efficiency of the transfer; especially to allow direct memory access (DMA) hardware to be used to effect the transfer. The additions match those of the `EXT_vertex_array` extension, except that static array data are not supported (because they complicated the interface, and were not being used), and the pre-defined configurations are added (both to reduce subroutine count even further, and to allow for efficient transfer of array data).

Microsoft et al. Exhibit 1005

**APPENDIX T**

## C.2 Polygon Offset

Depth values of fragments generated by the rasterization of a polygon may be shifted toward or away from the origin, as an affine function of the window coordinate depth slope of the polygon. Shifted depth values allow coplanar geometry, especially facet outlines, to be rendered without depth buffer artifacts. They may also be used by future shadow generation algorithms.

The additions match those of the `EXT_polygon_offset` extension, with two exceptions. First, the offset is enabled separately for `POINT`, `LINE`, and `FILL` rasterization modes, all sharing a single affine function definition. (Shifting the depth values of the outline fragments, instead of the fill fragments, allows the contents of the depth buffer to be maintained correctly.) Second, the offset bias is specified in units of depth buffer resolution, rather than in the [0,1] depth range.

## C.3 Logical Operation

Fragments generated by RGBA rendering may be merged into the framebuffer using a logical operation, just as color index fragments are in GL version 1.0. Blending is disabled during such operation because it is rarely desired, because many systems could not support it, and to match the semantics of the `EXT_blend_logic_op` extension, on which this addition is loosely based.

## C.4 Texture Image Formats

Stored texture arrays have a format, known as the *internal format*, rather than a simple count of components. The internal format is represented as a single enumerated value, indicating both the organization of the image data (`LUMINANCE`, `RGB`, etc.) and the number of bits of storage for each image component. Clients can use the internal format specification to suggest the desired storage precision of texture images. New *base formats*, `ALPHA` and `INTENSITY`, provide new texture environment operations. These additions match those of a subset of the `EXT_texture` extension.

## C.5 Texture Replace Environment

A common use of texture mapping is to replace the color values of generated fragments with texture color data. This could be specified only indirectly

APPENDIX T

in GL version 1.0, which required that client specified "white" geometry be modulated by a texture. GL version 1.1 allows such replacement to be specified explicitly, possibly improving performance. These additions match those of a subset of the EXT_texture extension.

## C.6 Texture Proxies

Texture proxies allow a GL implementation to advertise different maximum texture image sizes as a function of some other texture parameters, especially of the internal image format. Clients may use the proxy query mechanism to tailor their use of texture resources at run time. The proxy interface is designed to allow such queries without adding new routines to the GL interface. These additions match those of a subset of the EXT_texture extension, except that implementations return allocation information consistent with support for complete mipmap arrays.

## C.7 Copy Texture and Subtexture

Texture array data can be specified from framebuffer memory, as well as from client memory, and rectangular subregions of texture arrays can be redefined either from client or framebuffer memory. These additions match those defined by the EXT_copy_texture and EXT_subtexture extensions.

## C.8 Texture Objects

A set of texture arrays and their related texture state can be treated as a single object. Such treatment allows for greater implementation efficiency when multiple arrays are used. In conjunction with the subtexture capability, it also allows clients to make gradual changes to existing texture arrays, rather than completely redefining them. These additions match those of the EXT_texture_object extension, with slight additions to the texture residency semantics.

## C.9 Other Changes

1. Color indices may now be specified as unsigned bytes.

APPENDIX T

2. Texture coordinates $s$, $t$, and $r$ are divided by $q$ during the rasterization of points, pixel rectangles, and bitmaps. This division was documented only for lines and polygons in the 1.0 version.

3. The line rasterization algorithm was changed so that vertical lines on pixel borders rasterize correctly.

4. Separate pixel transfer discussions in chapter 3 and chapter 4 were combined into a single discussion in chapter 3.

5. Texture alpha values are returned as 1.0 if there is no alpha channel in the texture array. This behavior was unspecified in the 1.0 version, and was incorrectly documented in the reference manual.

6. Fog start and end values may now be negative.

7. Evaluated color values direct the evaluation of the lighting equation if **ColorMaterial** is enabled.

## C.10 Acknowledgements

OpenGL 1.1 is the result of the contributions of many people, representing a cross section of the computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

Kurt Akeley, Silicon Graphics
Bill Armstrong, Evans & Sutherland
Andy Bigos, 3Dlabs
Pat Brown, IBM
Jim Cobb, Evans & Sutherland
Dick Coulter, Digital Equipment
Bruce D'Amora, GE Medical Systems
John Dennis, Digital Equipment
Fred Fisher, Accel Graphics
Chris Frazier, Silicon Graphics
Todd Frazier, Evans & Sutherland
Tim Freese, NCD
Ken Garnett, NCD
Mike Heck, Template Graphics Software
Dave Higgins, IBM
Phil Huxley, 3Dlabs

APPENDIX T

Dale Kirkland, Intergraph
Hock San Lee, Microsoft
Kevin LeFebvre, Hewlett Packard
Jim Miller, IBM
Tim Misner, SunSoft
Jeremy Morris, 3Dlabs
Israel Pinkas, Intel
Bimal Poddar, IBM
Lyle Ramshaw, Digital Equipment
Randi Rost, Hewlett Packard
John Schimpf, Silicon Graphics
Mark Segal, Silicon Graphics
Igor Sinyak, Intel
Jeff Stevenson, Hewlett Packard
Bill Sweeney, SunSoft
Kelvin Thompson, Portable Graphics
Neil Trevett, 3Dlabs
Linas Vepstas, IBM
Andy Vesper, Digital Equipment
Henri Warren, Megatek
Paula Womack, Silicon Graphics
Mason Woo, Silicon Graphics
Steve Wright, Microsoft

# Appendix D

# Version 1.2

OpenGL version 1.2, released on March 16, 1998, is the second revision since the original version 1.0. Version 1.2 is upward compatible with version 1.1, meaning that any program that runs with a 1.1 GL implementation will also run unchanged with a 1.2 GL implementation.

Several additions were made to the GL, especially to texture mapping capabilities and the pixel processing pipeline. Following are brief descriptions of each addition.

## D.1 Three-Dimensional Texturing

Three-dimensional textures can be defined and used. In-memory formats for three-dimensional images, and pixel storage modes to support them, are also defined. The additions match those of the EXT_texture3D extension.

One important application of three-dimensional textures is rendering volumes of image data.

## D.2 BGRA Pixel Formats

BGRA extends the list of host-memory color formats. Specifically, it provides a component order matching file and framebuffer formats common on Windows platforms. The additions match those of the EXT_bgra extension.

## D.3 Packed Pixel Formats

Packed pixels in host memory are represented entirely by one unsigned byte, one unsigned short, or one unsigned integer. The fields with the packed pixel

APPENDIX T

are not proper machine types, but the pixel as a whole is. Thus the pixel storage modes and their unpacking counterparts all work correctly with packed pixels.

The additions match those of the EXT_packed_pixels extension, with the further addition of reversed component order packed formats.

## D.4 Normal Rescaling

Normals may be rescaled by a constant factor derived from the modelview matrix. Rescaling can operate faster than renormalization in many cases, while resulting in the same unit normals.

The additions are based on the EXT_rescale_normal extension.

## D.5 Separate Specular Color

Lighting calculations are modified to produce a primary color consisting of emissive, ambient and diffuse terms of the usual GL lighting equation, and a secondary color consisting of the specular term. Only the primary color is modified by the texture environment; the secondary color is added to the result of texturing to produce a single post-texturing color. This allows highlights whose color is based on the light source creating them, rather than surface properties.

The additions match those of the EXT_separate_specular_color extension.

## D.6 Texture Coordinate Edge Clamping

GL normally clamps such that the texture coordinates are limited to exactly the range $[0, 1]$. When a texture coordinate is clamped using this algorithm, the texture sampling filter straddles the edge of the texture image, taking half its sample values from within the texture image, and the other half from the texture border. It is sometimes desirable to clamp a texture without requiring a border, and without using the constant border color.

A new texture clamping algorithm, CLAMP_TO_EDGE, clamps texture coordinates at all mipmap levels such that the texture filter never samples a border texel. The color returned when clamping is derived only from texels at the edge of the texture image.

The additions match those of the SGIS_texture_edge_clamp extension.

APPENDIX T

## D.7 Texture Level of Detail Control

Two constraints related to the texture level of detail parameter $\lambda$ are added. One constraint clamps $\lambda$ to a specified floating point range. The other limits the selection of mipmap image arrays to a subset of the arrays that would otherwise be considered.

Together these constraints allow a large texture to be loaded and used initially at low resolution, and to have its resolution raised gradually as more resolution is desired or available. Image array specification is necessarily integral, rather than continuous. By providing separate, continuous clamping of the $\lambda$ parameter, it is possible to avoid "popping" artifacts when higher resolution images are provided.

The additions match those of the `SGIS_texture_lod` extension.

## D.8 Vertex Array Draw Element Range

A new form of **DrawElements** that provides explicit information on the range of vertices referred to by the index set is added. Implementations can take advantage of this additional information to process vertex data without having to scan the index data to determine which vertices are referenced.

The additions match those of the `EXT_draw_range_elements` extension.

## D.9 Imaging Subset

The remaining new features are primarily intended for advanced image processing applications, and may not be present in all GL implementations. The are collectively referred to as the *imaging subset.*

### D.9.1 Color Tables

A new RGBA-format color lookup mechanism is defined in the pixel transfer process, providing additional lookup capabilities beyond the existing lookup. The key difference is that the new lookup tables are treated as one-dimensional images with internal formats, like texture images and convolution filter images. Thus the new tables can operate on a subset of the components of passing pixel groups. For example, a table with internal format `ALPHA` modifies only the A component of each pixel group, leaving the R, G, and B components unmodified.

APPENDIX T

Three independent lookups may be performed: prior to convolution; after convolution and prior to color matrix transformation; after color matrix transformation and prior to gathering pipeline statistics.

Methods to initialize the color lookup tables from the framebuffer, in addition to the standard memory source mechanisms, are provided.

Portions of a color lookup table may be redefined without reinitializing the entire table. The affected portions may be specified either from host memory or from the framebuffer.

The additions match those of the `EXT_color_table` and `EXT_color_subtable` extensions.

### D.9.2 Convolution

One- or two-dimensional convolution operations are executed following the first color table lookup in the pixel transfer process. The convolution kernels are themselves treated as one- and two-dimensional images, which can be loaded from application memory or from the framebuffer.

The convolution framework is designed to accommodate three-dimensional convolution, but that API is left for a future extension.

The additions match those of the `EXT_convolution` and `HP_convolution_border_modes` extensions.

### D.9.3 Color Matrix

A 4x4 matrix transformation and associated matrix stack are added to the pixel transfer path. The matrix operates on RGBA pixel groups, using the equation

$$C' = MC,$$

where

$$C = \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}$$

and $M$ is the $4 \times 4$ matrix on the top of the color matrix stack. After the matrix multiplication, each resulting color component is scaled and biased by a programmed amount. Color matrix multiplication follows convolution.

The color matrix can be used to reassign and duplicate color components. It can also be used to implement simple color space conversions.

The additions match those of the `SGI_color_matrix` extension.

APPENDIX T

### D.9.4 Pixel Pipeline Statistics

Pixel operations that count occurences of specific color component values (histogram) and that track the minimum and maximum color component values (minmax) are performed at the end of the pixel transfer pipeline. An optional mode allows pixel data to be discarded after the histogram and/or minmax operations are completed. Otherwise the pixel data continues on to the next operation unaffected.

The additions match those of the `EXT_histogram` extension.

### D.9.5 Constant Blend Color

A constant color that can be used to define blend weighting factors may be defined. A typical usage is blending two RGB images. Without the constant blend factor, one image must have an alpha channel with each pixel set to the desired blend factor.

The additions match those of the `EXT_blend_color` extension.

### D.9.6 New Blending Equations

Blending equations other than the normal weighted sum of source and destination components may be used.

Two of the new equations produce the minimum (or maximum) color components of the source and destination colors. Taking the maximum is useful for applications such as maximum projection in medical imaging.

The other two equations are similar to the default blending equation, but produce the difference of its left and right hand sides, rather than the sum. Image differences are useful in many image processing applications.

The additions match those of the `EXT_blend_minmax` and `EXT_blend_subtract` extensions.

## D.10 Acknowledgements

OpenGL 1.2 is the result of the contributions of many people, representing a cross section of the computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

Kurt Akeley, Silicon Graphics
Bill Armstrong, Evans & Sutherland
Otto Berkes, Microsoft

APPENDIX T

Pierre-Luc Bisaillon, Matrox Graphics
Drew Bliss, Microsoft
David Blythe, Silicon Graphics
Jon Brewster, Hewlett Packard
Dan Brokenshire, IBM
Pat Brown, IBM
Newton Cheung, S3
Bill Clifford, Digital
Jim Cobb, Parametric Technology
Bruce D'Amora, IBM
Kevin Dallas, Microsoft
Mahesh Dandapani, Rendition
Daniel Daum, AccelGraphics
Suzy Deffeyes, IBM
Peter Doyle, Intel
Jay Duluk, Raycer
Craig Dunwoody, Silicon Graphics
Dave Erb, IBM
Fred Fisher, AccelGraphics / Dynamic Pictures
Celeste Fowler, Silicon Graphics
Allen Gallotta, ATI
Ken Garnett, NCD
Michael Gold, Nvidia / Silicon Graphics
Craig Groeschel, Metro Link
Jan Hardenbergh, Mitsubishi Electric
Mike Heck, Template Graphics Software
Dick Hessel, Raycer Graphics
Paul Ho, Silicon Graphics
Shawn Hopwood, Silicon Graphics
Jim Hurley, Intel
Phil Huxley, 3Dlabs
Dick Jay, Template Graphics Software
Paul Jensen, 3Dfx
Brett Johnson, Hewlett Packard
Michael Jones, Silicon Graphics
Tim Kelley, Real3D
Jon Khazam, Intel
Louis Khouw, Sun
Dale Kirkland, Intergraph
Chris Kitrick, Raycer

APPENDIX T

Don Kuo, S3
Herb Kuta, Quantum 3D
Phil Lacroute, Silicon Graphics
Prakash Ladia, S3
Jon Leech, Silicon Graphics
Kevin Lefebvre, Hewlett Packard
David Ligon, Raycer Graphics
Kent Lin, S3
Dan McCabe, S3
Jack Middleton, Sun
Tim Misner, Intel
Bill Mitchell, National Institute of Standards
Jeremy Morris, 3Dlabs
Gene Munce, Intel
William Newhall, Real3D
Matthew Papakipos, Nvidia / Raycer
Garry Paxinos, Metro Link
Hanspeter Pfister, Mitsubishi Electric
Richard Pimentel, Parametric Technology
Bimal Poddar, IBM / Intel
Rob Putney, IBM
Mike Quinlan, Real3D
Nate Robins, University of Utah
Detlef Roettger, Elsa
Randi Rost, Hewlett Packard
Kevin Rushforth, Sun
Richard S. Wright, Real3D
Hock San Lee, Microsoft
John Schimpf, Silicon Graphics
Stefan Seeboth, ELSA
Mark Segal, Silicon Graphics
Bob Seitsinger, S3
Min-Zhi Shao, S3
Colin Sharp, Rendition
Igor Sinyak, Intel
Bill Sweeney, Sun
William Sweeney, Sun
Nathan Tuck, Raycer
Doug Twillenger, Sun
John Tynefeld, 3dfx

APPENDIX T

Kartik Venkataraman, Intel
Andy Vesper, Digital Equipment
Henri Warren, Digital Equipment / Megatek
Paula Womack, Silicon Graphics
Steve Wright, Microsoft
David Yu, Silicon Graphics
Randy Zhao, S3

APPENDIX T

# Appendix E

# Version 1.2.1

OpenGL version 1.2.1, released on October 14, 1998, introduced ARB extensions (see Appendix F). The only ARB extension defined in this version is multitexture, allowing application of multiple textures to a fragment in one rendering pass. Multitexture is based on the `SGIS_multitexture` extension, simplified by removing the ability to route texture coordinate sets to arbitrary texture units.

A new corollary discussing display list and immediate mode invariance was added to Appendix B on April 1, 1999.

Microsoft et al. Exhibit 1005

APPENDIX T

# Appendix F

# ARB Extensions

OpenGL extensions that have been approved by the OpenGL Architectural Review Board (ARB) are described in this chapter. These extensions are not required to be supported by a conformant OpenGL implementation, but are expected to be widely available; they define functionality that is likely to move into the required feature set in a future revision of the specification.

In order not to compromise the readability of the core specification, ARB extensions are not integrated into the core language; instead, they are presented in this chapter, as changes to the core.

## F.1 Naming Conventions

To distinguish ARB extensions from core OpenGL features and from vendor-specific extensions, the following naming conventions are used:

- A unique *name string* of the form "GL_ARB_*name*" is associated with each extension. If the extension is supported by an implementation, this string will be present in the EXTENSIONS string described in section 6.1.11.

- All functions defined by the extension will have names of the form ***Function*ARB**

- All enumerants defined by the extension will have names of the form *NAME*_ARB.

Microsoft et al. Exhibit 1005

APPENDIX T

## F.2 Multitexture

Multitexture adds support for multiple texture units. The capabilities of the multiple texture units are identical, except that evaluation and feedback are supported only for texture unit 0. Each texture unit has its own state vector which includes texture vertex array specification, texture image and filtering parameters, and texture environment application.

The texture environments of the texture units are applied in a pipelined fashion whereby the output of one texture environment is used as the input fragment color for the next texture environment. Changes to texture client state and texture server state are each routed through one of two selectors which control which instance of texture state is affected.

The specification is written using four texture units though the actual number supported is implementation dependent and can be larger or smaller than four.

The name string for multitexture is `GL_ARB_multitexture`.

### F.2.1 Dependencies

Multitexture requires features of OpenGL 1.1.

### F.2.2 Issues

The extension currently requires a separate texture coordinate input for each texture unit. Modification to allow routing and/or broadcasting texcoords and **TexGen** output would be useful, possibly as a future extension layered on multitexture.

### F.2.3 Changes to Section 2.6 (Begin/End Paradigm)

*Amend paragraphs 2 and 3*

Each vertex is specified with two, three, or four coordinates. In addition, a *current normal*, multiple *current texture coordinate sets*, and *current color* may be used in processing each vertex. Normals are used by the GL in lighting calculations; the current normal is a three-dimensional vector that may be set by sending three coordinates that specify it. Texture coordinates determine how a texture image is mapped onto a primitive. Multiple sets of texture coordinates may be used to specify how multiple texture images are mapped onto a primitive. The number of texture units supported is implementation dependent but must be at least one. The number of active textures supported can be queried with the state `MAX_TEXTURE_UNITS_ARB`.

Microsoft et al. Exhibit 1005

Primary and secondary colors are associated with each vertex (see section 3.9). These *associated* colors are either based on the current color or produced by lighting, depending on whether or not lighting is enabled. Texture coordinates are similarly associated with each vertex. Multiple sets of texture coordinates may be associated with a vertex. Figure F.1 summarizes the association of auxiliary data with a transformed vertex to produce a *processed vertex.*

*Amend paragraph 6*

Before colors have been assigned to a vertex, the state required by a vertex is the vertex's coordinates, the current normal, the current edge flag (see section 2.6.2), the current material properties (see section 2.13.2), and the multiple current texture coordinate sets. Because color assignment is done vertex-by-vertex, a processed vertex comprises the vertex's coordinates, its edge flag, its assigned colors, and its multiple texture coordinate sets.

### F.2.4 Changes to Section 2.7 (Vertex Specification)

*Amend paragraph 2*

Current values are used in associating auxiliary data with a vertex as described in section 2.6. A current value may be changed at any time by issuing an appropriate command. The commands

```
void TexCoord{1234}{sifd}( T coords );
void TexCoord{1234}{sifd}v( T coords );
```

specify the current homogeneous texture coordinates, named $s$, $t$, $r$, and $q$. The **TexCoord1** family of commands set the $s$ coordinate to the provided single argument while setting $t$ and $r$ to 0 and $q$ to 1. Similarly, **TexCoord2** sets $s$ and $t$ to the specified values, $r$ to 0 and $q$ to 1; **TexCoord3** sets $s$, $t$, and $r$, with $q$ set to 1, and **TexCoord4** sets all four texture coordinates.

Implementations may support more than one texture unit, and thus more than one set of texture coordinates. The commands

```
void MultiTexCoord{1234}{sifd}ARB(enum texture,T
   coords)
void MultiTexCoord{1234}{sifd}vARB(enum texture,T
   coords)
```

take the coordinate set to be modified as the *texture* parameter. *texture* is a symbolic constant of the form TEXTURE$i$_ARB, indicating that texture coordinate set $i$ is to be modified. The constants obey TEXTURE$i$_ARB =

APPENDIX T

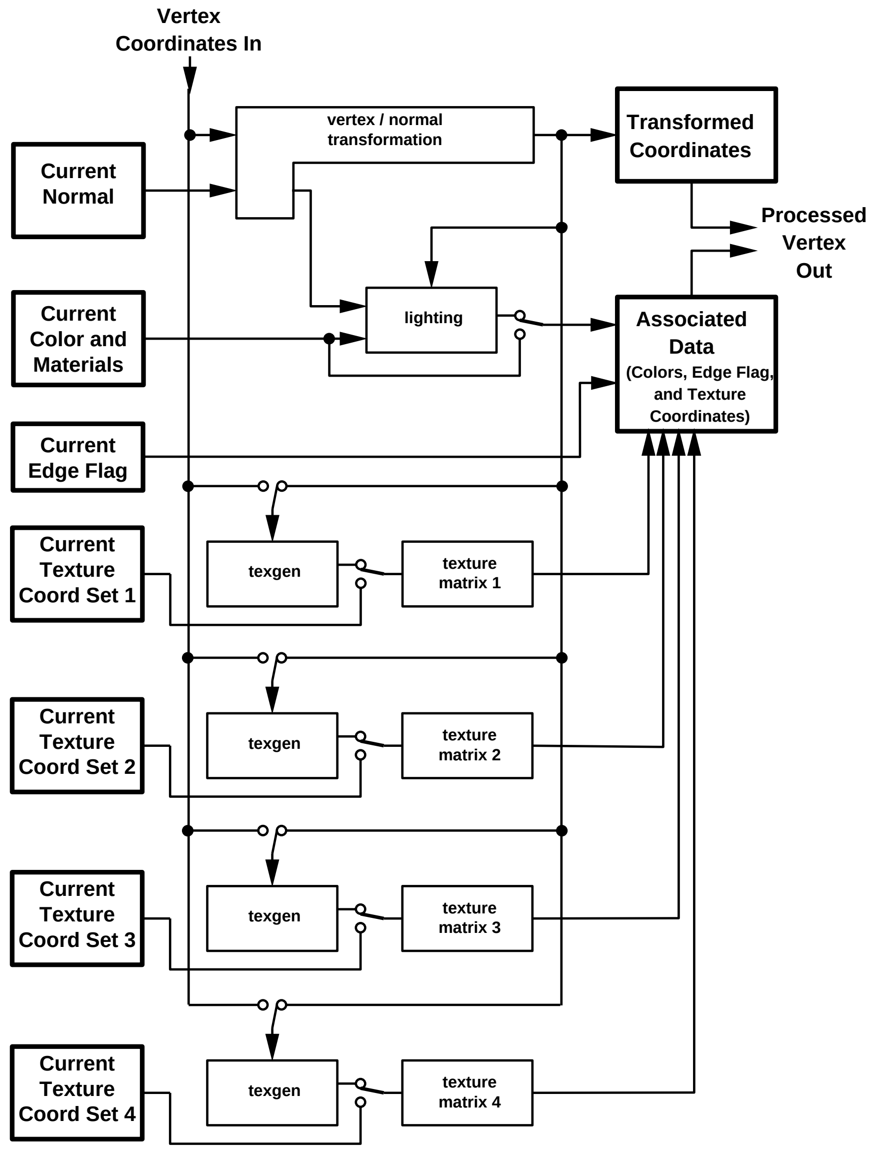


Figure F.1. Association of current values with a vertex. The heavy lined boxes represent GL state. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation.

Microsoft et al. Exhibit 1005

TEXTURE0_ARB $+ i$ ($i$ is in the range 0 to $k-1$, where $k$ is the implementation-dependent number of texture units defined by MAX_TEXTURE_UNITS_ARB).

The **TexCoord** commands are exactly equivalent to the corresponding **MultiTexCoordARB** commands with *texture* set to TEXTURE0_ARB.

**Get**s of CURRENT_TEXTURE_COORDS return the texture coordinate set defined by the value of ACTIVE_TEXTURE_ARB.

Specifying an invalid texture coordinate set for the *texture* argument of **MultiTexCoordARB** results in undefined behavior.

### F.2.5 Changes to Section 2.8 (Vertex Arrays)

*Amend paragraph 1*

The vertex specification commands described in section 2.7 accept data in almost any format, but their use requires many command executions to specify even simple geometry. Vertex data may also be placed into arrays that are stored in the client's address space. Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command. The client may specify up to 5 plus the value of MAX_TEXTURE_UNITS_ARB arrays: one each to store vertex coordinates, edge flags, colors, color indices, normals, and one or more texture coordinate sets. The commands ...

*Insert between paragraph 2 and 3*

In implementations which support more than one texture unit, the command

void **ClientActiveTextureARB**( enum *texture* );

is used to select the vertex array client state parameters to be modified by the **TexCoordPointer** command and the array affected by **EnableClientState** and **DisableClientState** with parameter TEXTURE_COORD_ARRAY. This command sets the client state variable CLIENT_ACTIVE_TEXTURE_ARB. Each texture unit has a client state vector which is selected when this command is invoked. This state vector includes the vertex array state. This call also selects which texture units' client state vector is used for queries of client state.

Specifying an invalid *texture* generates the error INVALID_ENUM. Valid values of *texture* are the same as for the **MultiTexCoordARB** commands described in section 2.7.

*Amend final paragraph*

If the number of supported texture units (the value of MAX_TEXTURE_UNITS_ARB) is $k$, then the client state required to implement vertex arrays consists of $5+k$ boolean values, $5+k$ memory pointers, $5+k$ integer stride values, $4+k$ symbolic constants representing array types, and $3+k$ integers representing values per element. In the initial state, the boolean values are each disabled, the memory pointers are each null, the strides are each zero, the array types are each FLOAT, and the integers representing values per element are each four.

### F.2.6 Changes to Section 2.10.2 (Matrices)

*Amend paragraph 8*

For each texture unit, a $4 \times 4$ matrix is applied to the corresponding texture coordinates. This matrix is applied as

$$\begin{pmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{pmatrix} \begin{pmatrix} s \\ t \\ r \\ q \end{pmatrix},$$

where the left matrix is the current texture matrix. The matrix is applied to the coordinates resulting from texture coordinate generation (which may simply be the current texture coordinates), and the resulting transformed coordinates become the texture coordinates associated with a vertex. Setting the matrix mode to TEXTURE causes the already described matrix operations to apply to the texture matrix.

There is also a corresponding texture matrix stack for each texture unit. To change the stack affected by matrix operations, set the *active texture unit selector* by calling

void **ActiveTextureARB**( enum *texture* );

The selector also affects calls modifying texture environment state, texture coordinate generation state, texture binding state, and queries of all these state values as well as current texture coordinates and current raster texture coordinates.

Specifying an invalid *texture* generates the error INVALID_ENUM. Valid values of *texture* are the same as for the **MultiTexCoordARB** commands described in section 2.7.

The active texture unit selector may be queried by calling **GetIntegerv** with *pname* set to ACTIVE_TEXTURE_ARB.

There is a stack of matrices for each of matrix modes `MODELVIEW`, `PROJECTION`, and `COLOR`, and for each texture unit. For `MODELVIEW` mode, the stack depth is at least 32 (that is, there is a stack of at least 32 model-view matrices). For the other modes, the depth is at least 2. Texture matrix stacks for all texture units have the same depth. The current matrix in any mode is the matrix on the top of the stack for that mode.

`void` **PushMatrix**( `void` );

pushes the stack down by one, duplicating the current matrix in both the top of the stack and the entry below it.

`void` **PopMatrix**( `void` );

pops the top entry off of the stack, replacing the current matrix with the matrix that was the second entry in the stack. The pushing or popping takes place on the stack corresponding to the current matrix mode. Popping a matrix off a stack with only one entry generates the error `STACK_UNDERFLOW`; pushing a matrix onto a full stack generates `STACK_OVERFLOW`.

When the current matrix mode is `TEXTURE`, the texture matrix stack of the active texture unit is pushed or popped.

The state required to implement transformations consists of a four-valued integer indicating the current matrix mode, one stack of at least two $4 \times 4$ matrices for each of `COLOR`, `PROJECTION`, each texture unit, `TEXTURE`, and a stack of at least 32 $4 \times 4$ matrices for `MODELVIEW`. Each matrix stack has an associated stack pointer. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial matrix mode is `MODELVIEW`. The initial value of `ACTIVE_TEXTURE_ARB` is `TEXTURE0_ARB`.

### F.2.7 Changes to Section 2.10.4 (Generating Texture Coordinates)

*Amend paragraph 4*

The state required for texture coordinate generation for each texture unit comprises a three-valued integer for each coordinate indicating coordinate generation mode, and a bit for each coordinate to indicate whether texture coordinate generation is enabled or disabled. In addition, four coefficients are required for the four coordinates for each of `EYE_LINEAR` and `OBJECT_LINEAR`. The initial state has the texture generation function disabled for all texture coordinates. The initial values of $p_i$ for $s$ are all 0 except $p_1$ which is one; for $t$ all the $p_i$ are zero except $p_2$, which is 1. The values of $p_i$ for $r$ and $q$ are all 0. These values of $p_i$ apply for both the EYE_LINEAR and OBJECT_LINEAR versions. Initially all texture generation modes are EYE_LINEAR.

For implementations which support more than one texture unit, there is texture coordinate generation state for each unit. The texture coordinate generation state which is affected by the **TexGen**, **Enable**, and **Disable** operations is set with **ActiveTextureARB**.

### F.2.8 Changes to Section 2.12 (Current Raster Position)

*Amend paragraph 2*

The state required for the current raster position consists of three window coordinates $x_w$, $y_w$, and $z_w$, a clip coordinate $w_c$ value, an eye coordinate distance, a valid bit, and associated data consisting of a color and multiple texture coordinate sets. It is set using one of the **RasterPos** commands:

void **RasterPos{234}{sifd}**( T *coords* );
void **RasterPos{234}{sifd}v**( T *coords* );

**RasterPos4** takes four values indicating $x$, $y$, $z$, and $w$. **RasterPos3** (or **RasterPos2**) is analogous, but sets only $x$, $y$, and $z$ with $w$ implicitly set to 1 (or only $x$ and $y$ with $z$ implicitly set to 0 and $w$ implicitly set to 1).

**Get**s of CURRENT_RASTER_TEXTURE_COORDS are affected by the setting of the state ACTIVE_TEXTURE_ARB.

*Modify figure 2.7*

*Amend paragraph 5*

The current raster position requires five single-precision floating-point values for its $x_w$, $y_w$, and $z_w$ window coordinates, its $w_c$ clip coordinate, and its eye coordinate distance, a single valid bit, a color (RGBA and color index), and texture coordinates for each texture unit. In the initial state, the coordinates and texture coordinates are all $(0, 0, 0, 1)$, the eye coordinate distance is 0, the valid bit is set, the associated RGBA color is $(1, 1, 1, 1)$ and the associated color index color is 1. In RGBA mode, the associated color index always has its initial value; in color index mode, the RGBA color always maintains its initial value.

### F.2.9 Changes to Section 3.8 (Texturing)

*Amend paragraphs 1 and 2*

Texturing maps a portion of one or more specified images onto each primitive for which texturing is enabled. This mapping is accomplished by using the color of an image at the location indicated by a fragment's $(s, t, r)$

APPENDIX T

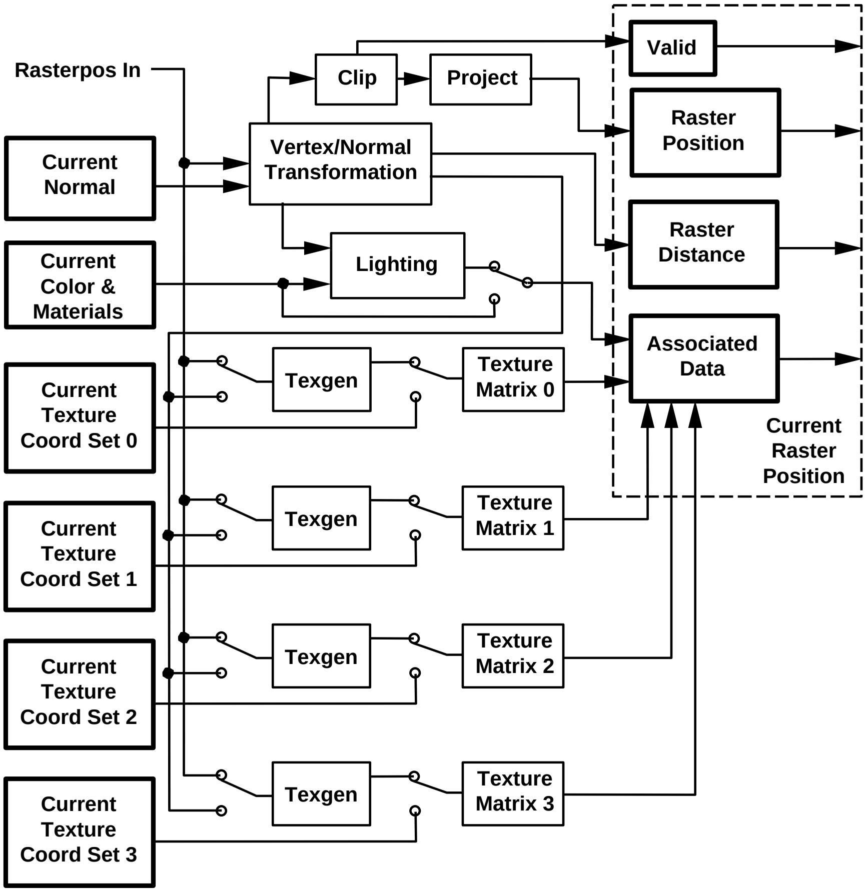


Figure F.2. The current raster position and how it is set. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation.

coordinates to modify the fragment's primary RGBA color. Texturing does not affect the secondary color.

An implementation may support texturing using more than one image at a time. In this case the fragment carries multiple sets of texture coordinates $(s, t, r)$ which are used to index separate images to produce color values which are collectively used to modify the fragment's RGBA color. Texturing is specified only for RGBA mode; its use in color index mode is undefined. The following subsections (up to and including Section 3.8.5) specify the GL operation with a single texture and Section 3.8.10 specifies the details of how multiple texture units interact.

### F.2.10 Changes to Section 3.8.5 (Texture Minification)

*Amend second paragraph under the* **Mipmapping** *subheading*

Each array in a mipmap is defined using **TexImage3D**, **TexImage2D**, **CopyTexImage2D**, **TexImage1D**, or **CopyTexImage1D**; the array being set is indicated with the level-of-detail argument *level*. Level-of-detail numbers proceed from `TEXTURE_BASE_LEVEL` for the original texture array through $p = \max\{n, m, l\} +$ `TEXTURE_BASE_LEVEL` with each unit increase indicating an array of half the dimensions of the previous one as already described. If texturing is enabled (and `TEXTURE_MIN_FILTER` is one that requires a mipmap) at the time a primitive is rasterized and if the set of arrays `TEXTURE_BASE_LEVEL` through $q = \min\{p,$ `TEXTURE_MAX_LEVEL`$\}$ is incomplete, then it is as if texture mapping were disabled for that texture unit. The set of arrays `TEXTURE_BASE_LEVEL` through $q$ is incomplete if the internal formats of all the mipmap arrays were not specified with the same symbolic constant, if the border widths of the mipmap arrays are not the same, if the dimensions of the mipmap arrays do not follow the sequence described above, if `TEXTURE_MAX_LEVEL` $<$ `TEXTURE_BASE_LEVEL`, or if `TEXTURE_BASE_LEVEL` $> p$. Array levels $k$ where $k <$ `TEXTURE_BASE_LEVEL` or $k > q$ are insignificant.

### F.2.11 Changes to Section 3.8.8 (Texture Objects)

*Insert following the last paragraph*

The texture object name space, including the initial one-, two-, and three-dimensional texture objects, is shared among all texture units. A texture object may be bound to more than one texture unit simultaneously. After a texture object is bound, any GL operations on that target object affect any other texture units to which the same texture object is bound.

Texture binding is affected by the setting of the state `ACTIVE_TEXTURE_ARB`.

**APPENDIX T**

If a texture object is deleted, it as if all texture units which are bound to that texture object are rebound to texture object zero.

### F.2.12 Changes to Section 3.8.10 (Texture Application)

*Amend second paragraph*

Each texture unit is enabled and bound to texture objects independently from the other texture units. Each texture unit follows the precendence rules for one-, two-, and three-dimensional textures. Thus texture units can be performing texture mapping of different dimensionalities simultaneously. Each unit has its own enable and binding states.

Each texture unit is paired with an environment function, as shown in figure F.3. The second texture function is computed using the texture value from the second texture, the fragment resulting from the first texture function computation and the second texture unit's environment function. If there is a third texture, the fragment resulting from the second texture function is combined with the third texture value using the third texture unit's environment function and so on. The texture unit selected by **ActiveTextureARB** determines which texture unit's environment is modified by **TexEnv** calls.

Texturing is enabled and disabled individually for each texture unit. If texturing is disabled for one of the units, then the fragment resulting from the previous unit, is passed unaltered to the following unit.

The required state, per texture unit, is three bits indicating whether each of one-, two-, or three-dimensional texturing is enabled or disabled. In the intial state, all texturing is disabled for all texture units.

### F.2.13 Changes to Section 5.1 (Evaluators)

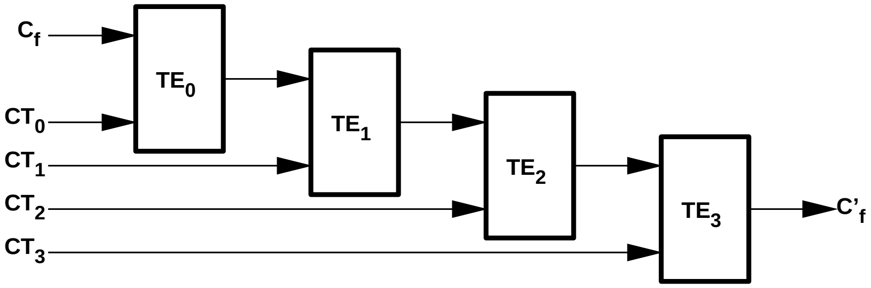
*Amend paragraph 7*

The evaluation of a defined map is enabled or disabled with **Enable** and **Disable** using the constant corresponding to the map as described above. The evaluator map generates only coordinates for texture unit `TEXTURE0_ARB`. The error `INVALID_VALUE` results if either *ustride* or *vstride* is less than $k$, or if $u_1$ is equal to $u2$, or if $v_1$ is equal to $v_2$. If the value of `ACTIVE_TEXTURE_ARB` is not `TEXTURE0_ARB`, calling **Map[12]** generates the error `INVALID_OPERATION`.

### F.2.14 Changes to Section 5.3 (Feedback)

*Amend paragraph 4*

APPENDIX T



$C_f$ = fragment color input to texturing

$C'_f$ = fragment color output from texturing

$CT_i$ = texture color from texture lookup *i*

$TE_i$ = texture environment *i*

Figure F.3. Multitexture pipeline. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation. The input fragment color is successively combined with each texture according to the state of the corresponding texture environment, and the resulting fragment color passed as input to the next texture unit in the pipeline.

APPENDIX T

The texture coordinates and colors returned are those resulting from the clipping operations described in Section 2.13.8. Only coordinates for texture unit TEXTURE0_ARB are returned even for implementations which support multiple texture units. The colors returned are the primary colors.

### F.2.15 Changes to Section 6.1.2 (Data Conversions)

*Insert following the last paragraph*

Most texture state variables are qualified by the value of ACTIVE_TEXTURE_ARB to determine which server texture state vector is queried. Client texture state variables such as texture coordinate array pointers are qualified by the value of CLIENT_ACTIVE_TEXTURE_ARB. Tables 6.5, 6.6, 6.7, 6.12, 6.14, and 6.25 indicate those state variables which are qualified by ACTIVE_TEXTURE_ARB or CLIENT_ACTIVE_TEXTURE_ARB during state queries.

### F.2.16 Changes to Section 6.1.12 (Saving and Restoring State)

*Insert following paragraph 3*

Operations on groups containing replicated texture state push or pop texture state within that group for all texture units. When state for a group is pushed, all state corresponding to TEXTURE0_ARB is pushed first, followed by state corresponding to TEXTURE1_ARB, and so on up to and including the state corresponding to TEXTURE$k$_ARB where $k + 1$ is the value of MAX_TEXTURE_UNITS_ARB. When state for a group is popped, the replicated texture state is restored in the opposite order that it was pushed, starting with state corresponding to TEXTURE$k$_ARB and ending with TEXTURE0_ARB. Identical rules are observed for client texture state push and pop operations. Matrix stacks are never pushed or popped with **PushAttrib**, **PushClientAttrib**, **PopAttrib**, or **PopClientAttrib**.

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| Modified state in table 6.5 | | | | | | |
| CURRENT_TEXTURE_COORDS | $1 * \times T$ | **GetFloatv** | 0,0,0,1 | Current texture coordinates | 2.7 | current |
| CURRENT_RASTER_TEXTURE_COORDS | $1 * \times T$ | **GetFloatv** | 0,0,0,1 | Texture coordinates associated with raster position | 2.12 | current |
| Modified state in table 6.6 | | | | | | |
| TEXTURE_COORD_ARRAY | $1 * \times B$ | **IsEnabled** | *False* | Texture coordinate array enable | 2.8 | vertex-array |
| TEXTURE_COORD_ARRAY_SIZE | $1 * \times Z^+$ | **GetIntegerv** | 4 | Coordinates per element | 2.8 | vertex-array |
| TEXTURE_COORD_ARRAY_TYPE | $1 * \times Z_4$ | **GetIntegerv** | FLOAT | Type of texture coordinates | 2.8 | vertex-array |
| TEXTURE_COORD_ARRAY_STRIDE | $1 * \times Z^+$ | **GetIntegerv** | 0 | Stride between texture coordinates | 2.8 | vertex-array |
| TEXTURE_COORD_ARRAY_POINTER | $1 * \times Y$ | **GetPointerv** | 0 | Pointer to the texture coordinate array | 2.8 | vertex-array |

Table F.1. Changes to State Tables

APPENDIX T

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| Modified state in table 6.7 | | | | | | |
| TEXTURE_MATRIX | $1* \times 2* \times M^4$ | **GetFloatv** | Identity | Texture matrix stack | 2.10.2 | – |
| TEXTURE_STACK_DEPTH | $1* \times Z^+$ | **GetIntegerv** | 1 | Texture matrix stack pointer | 2.10.2 | – |
| Modified state in table 6.12 | | | | | | |
| TEXTURE_$x$D | $1* \times 3 \times B$ | **IsEnabled** | *False* | True if $x$D texturing is enabled; $x$ is 1, 2, or 3 | 3.8.10 | texture/enable |
| TEXTURE_BINDING_$x$D | $1* \times 3 \times Z^+$ | **GetIntegerv** | 0 | Texture object bound to TEXTURE_$x$D | 3.8.8 | texture |
| Modified state in table 6.14 | | | | | | |
| TEXTURE_ENV_MODE | $1* \times Z_4$ | **GetTexEnviv** | MODULATE | Texture application function | 3.8.9 | texture |
| TEXTURE_ENV_COLOR | $1* \times C$ | **GetTexEnvfv** | 0,0,0,0 | Texture environment color | 3.8.9 | texture |
| TEXTURE_GEN_$x$ | $1* \times 4 \times B$ | **IsEnabled** | *False* | Texgen enabled ($x$ is S, T, R, or Q) | 2.10.4 | texture/enable |
| EYE_PLANE | $1* \times 4 \times R^4$ | **GetTexGenfv** | see 2.10.4 | Texgen plane equation coefficients (for S, T, R, and Q) | 2.10.4 | texture |
| OBJECT_PLANE | $1* \times 4 \times R^4$ | **GetTexGenfv** | see 2.10.4 | Texgen object linear coefficients (for S, T, R, and Q) | 2.10.4 | texture |
| TEXTURE_GEN_MODE | $1* \times 4 \times Z_3$ | **GetTexGeniv** | EYE_LINEAR | Function used for texgen (for S, T, R, and Q | 2.10.4 | texture |

Table F.2. Changes to State Tables (cont.)

Microsoft et al. Exhibit 1005

| Get value | Type | Get Cmnd | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| Added to table 6.6 | | | | | | |
| CLIENT_ACTIVE_TEXTURE_ARB | $Z_{1*}$ | **GetIntegerv** | TEXTURE0_ARB | Client active texture unit selector | 2.7 | vertex-array |
| Added to table 6.14 | | | | | | |
| ACTIVE_TEXTURE_ARB | $Z_{1*}$ | **GetIntegerv** | TEXTURE0_ARB | Active texture unit selector | 2.7 | texture |

Table F.3. New State Introduced by Multitexture

APPENDIX T

| Get value | Type | Get Cmnd | Minimum Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| Added to table 6.25 | | | | | | |
| MAX_TEXTURE_UNITS_ARB | $Z^+$ | **GetIntegerv** | 1 | Number of texture units (not to exceed 32) | 2.6 | – |

Table F.4. New Implementation-Dependent Values Introduced by Multitexture

APPENDIX T

# Index of OpenGL Commands

Microsoft et al. Exhibit 1005

APPENDIX T

APPENDIX T

APPENDIX T

APPENDIX T

APPENDIX T

Microsoft et al. Exhibit 1005

APPENDIX T

APPENDIX X

COMPUTING MILIEUX

# The Challenges of Mobile Computing

**George H. Forman and John Zahorjan**
**University of Washington**

**Computers are typically configured for use in a single location. The shift toward mobility and wireless communication is testing the abilities of designers to adapt traditional system structures.**

Recent advances in technology have provided portable computers with wireless interfaces that allow networked communication even while a user is mobile. Whereas today's first-generation notebook computers and personal digital assistants (PDAs) are self-contained, networked mobile computers are part of a greater computing infrastructure. Mobile computing — the use of a portable computer capable of wireless networking — will very likely revolutionize the way we use computers.

Wireless networking greatly enhances the utility of a portable computing device. It allows mobile users versatile communication with other people and expedient notification of important events, yet with much more flexibility than with cellular phones or pagers. It also permits continuous access to the services and resources of land-based networks. The combination of networking and mobility will engender new applications and services, such as collaborative software to support impromptu meetings, electronic bulletin boards whose contents adapt to the current viewers, lighting and heating that adjust to the needs of those present, and navigation software to guide users in unfamiliar places and on tours.[1]

The technical challenges that mobile computing must surmount to achieve this potential are hardly trivial, however. Some of the challenges in designing software for mobile computing systems are quite different from those involved in the design of software for today's stationary networked systems. In this article we focus on the issues pertinent to software designers without delving into the lower level details of the hardware realization of mobile computers. We look at some promising approaches under investigation and also consider their limitations.

The many issues to be dealt with stem from three essential properties of mobile computing: communication, mobility, and portability. Of course, special-purpose systems may avoid some design pressures by doing without certain desirable properties. For instance, portability would be less of a concern for mobile computers installed in the dashboards of cars than with hand-held mobile computers. However, we concentrate on the goal of large-scale, hand-held mobile computing as a way to reveal a wide assortment of issues.

## Wireless communication

Mobile computers require wireless network access, although sometimes — when in meeting rooms or at a user's desk — they may remain stationary long enough to be physically attached to the network for a better or cheaper connection.

Wireless networks communicate by modulating radio waves or pulsing infrared light. Wireless communication is linked to the wired network infrastructure by stationary transceivers. The area covered by an individual transceiver's signal is known as a cell. Cell sizes vary widely; for example, an infrared transceiver can cover a small meeting room, a cellular telephone transceiver has a range of a few miles, and a satellite beam can cover an area more than 400 miles in diameter.

Wireless communication faces more obstacles than wired communication because the surrounding environment interacts with the signal, blocking signal paths and introducing noise and echoes. As a result, wireless communication is characterized by lower bandwidths, higher error rates, and more frequent spurious disconnections. These factors can in turn increase communication latency resulting from retransmissions, retransmission time-out delays, error-control protocol processing, and short disconnections.

Mobility can also cause wireless connections to be lost or degraded. Users may travel beyond the coverage of network transceivers or enter areas of high interference. Unlike typical wired networks, the number of devices in a network cell varies dynamically, and large concentrations of mobile users, say, at conventions and public events, may overload network capacity.

The need for wireless communication leads to design challenges in several areas.

**Disconnection.** Today's computer systems often depend heavily on a network and may cease to function during network failures. For example, distributed file systems may lock up waiting for other servers, and application processes may fail altogether if the network stays down too long.

Network failure is a greater concern in mobile computing than in traditional computing because wireless communication is so susceptible to disconnection. Designers must decide whether to spend available resources on the network, trying to prevent disconnections, or to spend them trying to enable systems to cope with disconnections more gracefully and work around them where possible.

The more autonomous a mobile computer, the better it can tolerate network disconnection. For example, certain applications can reduce communication by running entirely locally on the mobile unit rather than by splitting the application and the user interface across the network. In environments with frequent disconnections, it is better for a mobile device to operate as a stand-alone computer than as a portable terminal.

In some cases, both round-trip latency and short disconnections can be hidden by asynchronous operation. The X11 Window System uses this technique to achieve good performance. With the synchronous remote procedure call paradigm, the client waits for a reply after each request; in asynchronous operation, a client sends multiple requests before asking for acknowledgment. Similarly, prefetching and delayed write-back also decouple the act of communication from the actual time a program consumes or produces data, allowing it to proceed during network disconnections. These techniques, therefore, have the potential to mask some network failures.

The Coda file system provides a good example of how to handle network disconnections, although it is designed for today's notebook computers in which disconnections may be less frequent, more predictable, and longer lasting than in mobile computing.[2] Information from the user's profile helps in keeping the best selection of files in an on-board cache. It is important to cache whole files rather than blocks of files so that entire files can be read during a disconnection. When the network reconnects, Coda attempts to reconcile the cache with the replicated master repository.

With Coda, files can be modified even during disconnections. More conservative file systems disallow this to prevent multiple users from making inconsistent changes. Coda's optimism is justified by studies showing that only rarely are files actually shared in a distributed system; fewer than 1 percent of all writes are followed by a write from a different user.[2] If strong consistency guarantees are needed, clients can ask for them explicitly. Hence, providing flexible consistency semantics can allow greater autonomy.

Of course, not all network disconnec-

## Portable terminal versus stand-alone computer

The role of a portable device has considerable latitude within the notion of mobile computing. Is it a terminal or an independent, stand-alone computer? How many purposes should the device serve? Should it incorporate a telephone like the AT&T EO? Should its work environment be that of a general-purpose workstation or of something more restrictive, say, the Apple Newton MessagePad? These design choices greatly affect the way we approach the important issues of communication, mobility, and portability. For example, a portable terminal such as Xerox PARC's Tab[1] is more dependent on a network but less prone to loss of storage media than a stand-alone computer. Such questions require careful consideration when designing software for mobile computing.

### Reference

1. M. Weiser, "Some Computer Science Issues in Ubiquitous Computing," *Comm. ACM*, Vol. 36, No. 7, July 1993, pp. 75-84.

APPENDIX X

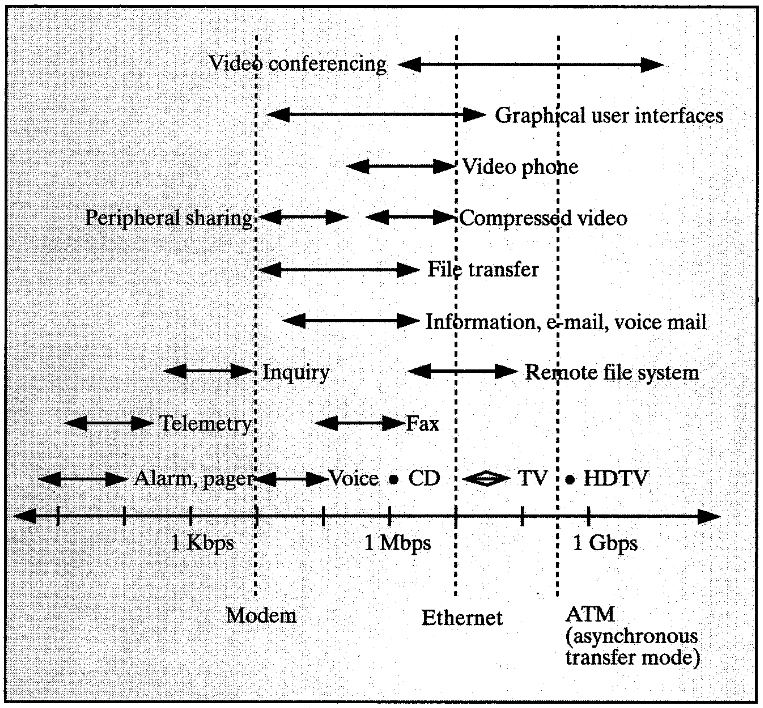


**Figure 1. Application bandwidth requirements in bits per second. The vertical dashed lines show the bandwidth capability of certain network technologies. Cellular modems are becoming fast enough for mobile users' everyday information needs, such as e-mail, and someday may be able to support remote file systems.**

tions can be masked. In these cases, good user interfaces can help by providing feedback about which operations are unavailable because of network disconnections.

**Low bandwidth.** Mobile computing designs need to reflect a greater concern for bandwidth consumption and constraints than do designs for stationary computing. Wireless networks deliver lower bandwidth than wired networks: Cutting-edge products for portable wireless communications achieve only 1 megabit per second for infrared communication, 2 Mbps for radio communication, and 9-14 kilobits per second for cellular telephony, while Ethernet provides 10 Mbps, fast Ethernet and FDDI 100 Mbps, and ATM (asynchronous transfer mode) 155 Mbps (see Figure 1). Even nonportable wireless networks, such as the Motorola Altair, barely achieve 5.7 Mbps.

Network bandwidth is divided among the users sharing a cell. The deliverable bandwidth *per user*, therefore, is an important measure of network capacity in addition to the raw transmission bandwidth. But because this measure depends on the size and distribution of a user population, Weiser and others recommend measuring a wireless network's capacity by its bandwidth per cubic meter.[1]

Improving network capacity means installing more wireless cells to service a user population. There are two ways to do this: Overlap cells on different wavelengths, or reduce transmission ranges so that more cells fit in a given area (see Figure 2).

The scalability of the first technique is limited because the electromagnetic spectrum available for public consumption is scarce. This technique is more flexible, however, because it allows (in fact, requires) software to allocate bandwidth among users.

The second technique is generally preferred. It is arguably simpler, reduces power requirements, and may decrease signal corruption because there are fewer objects in the environment to interact with. Also, it involves a hardware tradeoff between bandwidth and coverage area: Transceivers covering less area can achieve higher bandwidths.

Certain software techniques can also help cope with the low bandwidth of wireless links. Modems typically use compression to increase their effective bandwidth, sometimes almost doubling throughput. Because bulk operations are usually more efficient than many short transfers, buffering can improve bandwidth usage by making large requests out of many short ones. Buffering in conjunction with compression can further improve throughput because larger blocks compress better.

Certain software techniques for coping with disconnection can also help cope with low bandwidth. Network usage typically occurs in bursts, and disconnections are similar to bursts in that demand temporarily exceeds available bandwidth. For example, delayed write-back and prefetching use the periods of low network activity to reduce demand at the peaks. Delayed write-back can even reduce overall communication if the data to be transmitted is further mutated or deleted before it is transmitted. Prefetching involves knowing or guessing which files will be needed soon and downloading them over the network before they are demanded.[3] Bad guesses can waste network bandwidth, however.

System performance can be improved by scheduling communication intelligently. When available bandwidth does not satisfy the demand, processes the user is waiting for should be given priority. Backups should be performed only with "leftover" bandwidth. Mail can be trickle fed onto the mobile computer slowly before the user is notified. Although these techniques do not increase effective bandwidth, they improve user satisfaction just the same.

**High bandwidth variability.** Mobile computing designs also contend with much greater variation in network bandwidth than do traditional designs. Band-

APPENDIX X

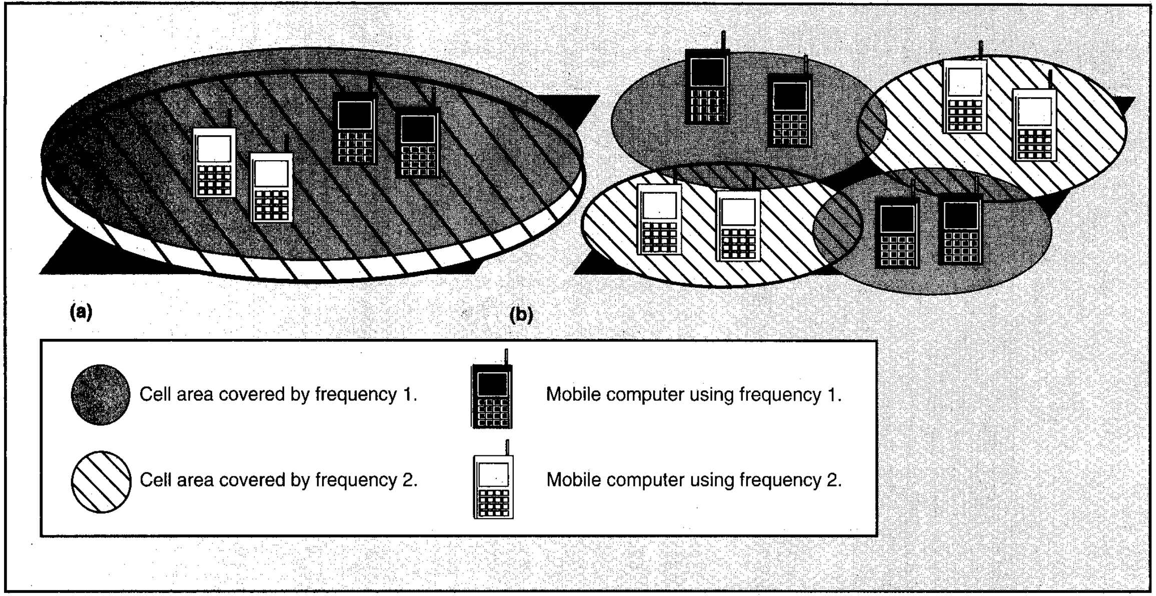

(a)
(b)
Cell area covered by frequency 1.
Mobile computer using frequency 1.
Cell area covered by frequency 2.
Mobile computer using frequency 2.

**Figure 2. Suppose that a single frequency provides only enough wireless bandwidth for two users. Then two frequencies can support (a) four users with two large coincident cells or (b) eight users with four small noninterfering cells that use the same frequency in nonadjacent cells. The latter scheme requires more transceivers and installation effort but is more scalable and allows higher bandwidth technology and lower transmission power.**

width can shift one to four orders of magnitude, depending on whether the system is plugged in or using wireless access. Fluctuant traffic load seldom causes this much variation in available bandwidth on today's networks.

An application can approach this variability in one of three ways: It can assume high-bandwidth connections and operate only while plugged in, it can assume low-bandwidth connections and not take advantage of higher bandwidth when it is available, or it can adapt to currently available resources, providing the user with a variable level of detail or quality. For example, a video-conferencing application could display only the current speaker or all the participants, depending on the available bandwidth. Different choices make sense for different applications.

**Heterogeneous network.** In contrast to most stationary computers, which stay connected to a single network, mobile computers encounter more heterogeneous network connections in several ways. First, as they leave the range of one network transceiver and switch to another, they may also need to change transmission speeds and protocols. Second, in some situations a mobile computer may have access to several network connections at once, for example, where adjacent cells overlap or where it can be plugged in for concurrent wired access.

Also, mobile computers may need to switch interfaces, for example, when going between indoors and outdoors. Infrared interfaces cannot be used outside because sunlight drowns out the signal. Even with radio frequency transmission, the interface may still need to change access protocols for different networks, for example, when switching from cellular coverage in a city to satellite coverage in the country. This heterogeneity makes mobile networking more complex than traditional networking.

**Security risks.** Precisely because connection to a wireless link is so easy, the security of wireless communication can be compromised much more easily than that of wired communication, especially if transmission extends over a large area. This increases pressure on mobile computing software designers to include security measures.

Security is further complicated if users are allowed to cross security domains. For example, a hospital may allow patients with mobile computers to use nearby printers but prohibit access to distant printers and resources designated for hospital personnel only.

Secure communication over insecure channels is accomplished by encryption, which can be done in software or, more quickly, by specialized hardware such as the recently proposed Clipper chip. Security depends on a secret encryption key known only to the authorized parties. Managing these keys securely is difficult, but it can be automated by software such as the Massachusetts Institute of Technology's Kerberos.[4]

Kerberos provides secure authentication services, as long as the Kerberos server itself is trusted. It authenticates users without exposing their passwords on the network and generates secret encryption keys that can be selectively shared between mutually suspicious parties. It also allows mobile units to authenticate themselves in domains where they are unknown, thus enhancing the scale of mobility. Methods have also been devised to use Kerberos for authorization control and accounting. Its security is limited, however. For example, the current version is susceptible to off-line password-guessing attacks and to replay attacks for a limited time window.

## Mobility

The ability to change locations while connected to the network increases the volatility of some information. Certain data considered static for stationary computing becomes dynamic for mobile computing. For example, a stationary computer can be configured statically to prefer the nearest server, but a mobile computer needs a mechanism for determining which server to use.

As volatility increases, cost-benefit trade-off points shift, calling for appropriate modifications in the design. For example, a highly volatile data object has fewer uses per modification. For such objects it makes little sense to cache the data. As another example, consider static information, which is often managed by hand; to handle higher rates of change, automated methods are required. However, even where such methods exist, they may be ill-suited for the dynamism of mobile computing.

Mobility introduces several problems: A mobile computer's network address changes dynamically, its current location affects configuration parameters as well as answers to user queries, and the communication path grows as it wanders away from a nearby server.

**Address migration.** As people move, their mobile computers will use different network access points, or "addresses." Today's networking is not designed for dynamically changing addresses. Active network connections usually cannot be moved to a new address. Once an address for a host name is known to a system, it is typically cached with a long expiration time and with no way to invalidate out-of-date entries. In the Internet Protocol, for example, a host IP name is inextricably bound with its network address; moving to a new location means acquiring a new IP name. Human intervention is commonly required to coordinate the use of addresses.

To communicate with a mobile computer, messages must be sent to its most recent address. Four basic mechanisms determine a mobile computer's current address: broadcast,[5,6] central services,[7] home base,[8] and forwarding pointers.[5] These are the building blocks of the current proposals for "mobile-IP" schemes.

**As people move, their mobile computers will use different network access points, or "addresses."**

*Selective broadcast.* With the broadcast method, a message is sent to all network cells asking the mobile computer sought to reply with its current address. This becomes too expensive for frequent use in a large network, but if the mobile computer is known to be in some small set of cells, selectively broadcasting in just those cells is workable. Hence, the methods described below can use selective broadcast to obtain the current address when only approximate location information is known. For example, a slightly out-of-date cell address may suffice if adjacent cells are known.

*Central services.* With the central service method, the current address for each mobile computer is maintained in a logically centralized database. Each time a mobile computer changes its address, it sends a message to update the database. Even with the database's centralized location, the common techniques of distribution, replication, and caching can be used to improve availability and response time.

*Home bases.* The home base method is essentially the limiting case of distributing a central service; that is, the location of a given mobile computer is known by a single server. This aggressive distribution without replication can lead to low availability of information. For example, if a home base is down or inaccessible, the mobile computers it tracks cannot be contacted. If users sometimes change home bases, the address migration problem arises again, albeit with much lower volatility.

*Forwarding pointers.* With the forwarding pointer method, each time a mobile computer changes its address, a copy of the new address is deposited at the old location. Each message is forwarded along the chain of pointers until it reaches the mobile computer. To avoid the inefficient routing that can result from long chains, pointers at message forwarders can be updated gradually to reflect more recent addresses.

Although the forwarding pointer method is among the fastest, it is prone to failures anywhere along the trail of pointers, and in its simplest form it does not allow forwarding pointers to be deleted unless all possible message sources have been updated. Hence, forwarding pointers are often used only to speed the common case, and another method is used to fall back on for failures and to allow reclamation of old pointers.

The forwarding pointer method requires an active entity at the old address to receive and forward messages. This does not fit standard networking models, where either a network address is a passive entity, such as an Ethernet cable, or it's specific to the mobile computer, which cannot remain to forward its own messages. This mismatch introduces subtle difficulties in implementing forwarding efficiently (for example, with intracell traffic or when multiple gateways are attached to a network address).

**Location-dependent information.** Because traditional computers do not move, information that depends on location, such as the local name server, available printers, and the time zone, is typically configured statically. One challenge for mobile computing is to factor out this information intelligently and provide mechanisms for obtaining configuration data appropriate to each location. Additionally, a mobile computer carried with a user is likely to be used in a wide variety of administrative domains. Dealing with the multitude of conventions that current computing systems rely on is another challenge to building mobile systems.

Besides this dynamic configuration problem, mobile computers need access to more location-sensitive information than stationary computers do. If they are to serve as guides in places unfamiliar to their users, mobile computers may need to answer queries such as "where is the fiction section (in this particular library)?" or "where is the nearest open gas station heading north?"

Queries of this sort require static location information about the world. Other information needs can be even more complex: Badrinath and Imielinski are studying a related class of queries that depend on the dynamic locations of other mobile objects, for example, the location of the nearest taxi.[6]

*Privacy.* Answering dynamic location queries requires knowing the location of another mobile user. In some cases this may be sensitive information, more so if given at a fine resolution. Even where it is not particularly sensitive, such information should be protected against misuse; for example, we do not want a burglar to be able to determine when the inhabitants of a house are far away.

Privacy can be ensured by denying users the ability to know another's location. The challenge for mobile computing is to allow more flexible access to this information without violating privacy. Legitimate uses of location information include contacting colleagues, routing telephone calls, logging meetings in personal diaries, and tailoring the content of electronic announcement displays to the current viewers.[1]

**Migrating locality.** Mobile computing engenders a new kind of locality that migrates as users move. Even if a mobile computer is equipped to find the nearest server for a given service, over time migration may alter this condition. Because the physical distance between two points does not necessarily reflect the network distance, the communication path can grow disproportionately to actual movement. For example, a small movement can result in a much longer path when crossing network administrative boundaries, and a longer network path means communication traverses more intermediaries, resulting in longer latency and greater risk of disconnection. A longer communication path also consumes more network capacity, even though the bandwidth between the mobile unit and the server may not degrade.

**Mobile computers need access to more location-sensitive information than do stationary computers.**

To avoid these disadvantages, service connections may be dynamically transferred to servers that are closer.[9] When many mobile units converge, during meetings, for example, load-balancing concerns may outweigh the importance of communication locality.

## Portability

Today's desktop computers are not meant to be carried, so designers take a liberal approach to space, power, cabling, and heat dissipation. In contrast, designers of hand-held mobile computers should strive for the properties of a wristwatch: small, light, durable, operational under wide environmental conditions, and requiring minimal power usage for long battery life. Concessions can be made in each of these areas to enhance functionality, but ultimately the user must receive value that exceeds the trouble of carrying the device. Similarly, any specialized hardware to offload such tasks as data compression or encryption from the CPU should justify its consumption of power and space.

Below, we describe the design pressures caused by portability constraints. These pressures are evident in the designs of the recent PDA products listed in Table 1.

**Low power.** Batteries are the largest single source of weight in a portable computer. While reducing battery weight is important, too small a battery can undermine the value of portability by causing users to recharge frequently, carry spare batteries, or use their mobile computers less. Minimizing power consumption can improve portability by reducing battery weight and lengthening the life of a charge.

Power consumption of dynamic components is proportional to $CV^2F$, where $C$ is the capacitance of the circuit, $V$ is the voltage swing, and $F$ is the clock frequency. This function suggests three ways to save power:

(1) Capacitance can be reduced by greater levels of VLSI integration and multichip module technology.
(2) Voltage can be reduced by redesigning chips to operate at lower voltages. Historically, chips operate at 5 volts, but some, like those in the Apple MessagePad, save power by operating at 3 volts. Manufacturers are rapidly developing a line of low-power chip sets for 2.5V and 3.3V operation.
(3) Clock frequency can be reduced, thereby trading computational speed for power savings. PDA products have adopted this concession, as shown in Table 1. In some notebook computers, the clock frequency can be changed dynamically, providing a flexible trade-off; for example, the Sharp PC 6785 can save power by dynamically shifting its clock from 25 MHz to 10 MHz or even 5 MHz, as shown in Table 2. To

**Table 1. Characteristics of personal digital assistant products and the AT&T EO tablet computer. Each has a pen interface and a black-and-white reflective LCD screen. The portable PC is included for comparison.**

| Product | RAM | MHz | CPU | Batteries | | Weight | Display | |
|---|---|---|---|---|---|---|---|---|
| | | | | (No. hours) | (type) | (lbs.) | (pixels) | (sq. in.) |
| Amstrad Pen Pad PDA600 | 128 Kbytes | 20 | Z-80 | 40 | 3 AA's | 0.9 | 240 × 320 | 10.4 |
| Apple Newton MessagePad | 640 Kbytes | 20 | ARM* | 6-8 | 4 AAA's | 0.9 | 240 × 336 | 11.2 |
| Apple Newton MessagePad 110 | 1 Mbyte | 20 | ARM* | 50 | 4 AA's | 1.25 | 240 × 320 | 11.8 |
| Casio Z-7000 PDA | 1 Mbyte | 7.4 | 8086 | 100 | 3 AA's | 1.0 | 320 × 256 | 12.4 |
| Sharp Expert Pad | 640 Kbytes | 20 | ARM* | 20 | 4 AAA's | 0.9 | 240 × 336 | 11.2 |
| Tandy Z-550 Zoomer PDA | 1 Mbyte | 8 | 8086 | 100 | 3 AA's | 1.0 | 320 × 256 | 12.4 |
| AT&T EO 440 Personal Communicator | 4-12 Mbytes | 20 | Hobbit | 1-6 | NiCad | 2.2 | 640 × 480 | 25.7 |
| Portable PC | 4-16 Mbytes | 33-66 | 486 | 1-6 | NiCad | 5-10 | 640 × 480 to 1,024 × 768 (color) | 40 |

*Advanced RISC microprocessor

retain more computational power at lower frequencies, processors are being designed that perform more work on each clock cycle.[10]

Power can be conserved not only by the design but also by efficient operation. Power management software can power down individual components when they are idle, for example, spinning down the internal disk or turning off screen lighting. Recently, Li et al. determined that for today's notebook computing it is worthwhile to spin down the internal disk drive after it has been idle for just a few seconds.[11]

Applications can conserve power by reducing their appetite for computation, communication, and memory, and by performing their periodic operations infrequently to amortize the start-up overhead. Since radio modem transmission typically requires about 10 times as much power as reception, power can be saved by trading transmission for reception. For example, base stations might periodically broadcast information that otherwise would have to be requested frequently. In this way, mobile computers can obtain this information without expending power to transmit a request.

The potential savings of these techniques can be evaluated using Tables 2 and 3, which break down power consumption in notebook computers by component and subsystem, respectively. Although screen lighting consumes a large amount of power, it has been found to greatly improve readability; for example, on EO models it enhances contrast from 6:1 to 13:1. Nevertheless, PDA products have elected to omit screen lighting in favor of longer battery life.

**Risks to data.** Making computers portable increases the risk of physical damage, unauthorized access, loss, and theft. Breaches of privacy or total loss of data become more likely. These risks can be reduced by minimizing the essential data kept on board. Obviously, a mobile device that serves only as a portable terminal is less prone to data loss than a stand-alone computer. This is the approach taken for Xerox PARC's Tabs and the portable multimedia terminal project at the University of California, Berkeley.[10]

To help prevent unauthorized disclosure of information, data stored on disks and removable memory cards can be encrypted. For this to be effective, users must not leave authenticated sessions (logins) unattended.

Keeping a copy that does not reside on the portable unit can safeguard against data loss. However, users often neglect to make backup copies, and even when they do, data modified between backups is not protected. With the addition of wireless networks to portable computers, new or modified data can be copied immediately to secure, remote media. This can be accomplished with replicated file systems such as Echo and Coda.[2]

**Table 2. Power consumption of portable-computer components and accessories.***

| Device | Power (watts) |
|---|---|
| Base system (2 Mbytes, 25-MHz CPU) | 3.650 |
| Base system (2 Mbytes, 10-MHz CPU) | 3.150 |
| Base system (2 Mbytes, 5-MHz CPU) | 2.800 |
| Screen backlight | 1.425 |
| Hard drive motor | 1.100 |
| Math coprocessor | 0.650 |
| Floppy drive | 0.500 |
| External keyboard | 0.490 |
| LCD screen | 0.315 |
| Hard drive active (head seeks) | 0.125 |
| IC card slot | 0.100 |
| Additional memory (per Mbyte) | 0.050 |
| Parallel port | 0.035 |
| Serial port | 0.030 |
| Accessories | |
| 1.8-inch PCMCIA hard drive | 0.7-3.0 |
| Cellular telephone (active) | 5.400 |
| Cellular telephone (standby) | 0.300 |
| Infrared network, 1 Mbit per second** | 0.250 |
| PCMCIA modem, 14,400 bits per second | 1.365 |
| PCMCIA modem, 9,600 bits per second | 0.625 |
| PCMCIA modem, 2,400 bits per second | 0.565 |
| Global positioning receiver** | 0.670 |

*Data for computer components was derived from the Sharp PC 6785 manual; data for accessories was obtained from the manufacturers.
**Estimate for soon-to-be-released product.

**Small user interface.** Size constraints on a portable computer require a small user interface. Desktop windowing environments may be sufficient for today's notebook computers, but for smaller, more portable devices, current windowing technology is inadequate. On small displays it is impractical to have several windows open at a time regardless of screen resolution, and it can be difficult to locate windows or icons deeply stacked one on another. Also, window title bars and borders either consume significant portions of screen space or, if reduced, become difficult to operate with the pointing device.

Duchamp, Feiner, and Maguire investigated the use of head-mounted virtual reality displays for portable computers.[9] As the user's head turns, the image displayed to the eye shifts to give the sensation of a surrounding screen. This effectively increases the screen area available for windowing systems; however, wearing head gear is cumbersome, resolution is low (one-tenth that of conventional displays), eyes succumb to fatigue, and dim lighting is required.

*Buttons versus analog input.* The shortage of surface area on a small computer leads designers to sacrifice buttons in favor of analog input devices for communicating user commands. These communication mechanisms include handwriting recognition, gesture recognition, and voice recognition. Although on average handwriting is about three times slower than typing, it allows the keyboard to be eliminated, thus reducing size and improving durability. This approach has been adopted by all the PDA products listed in Table 1.

Handwriting recognition rates for high-end systems are typically 96-98 percent accurate when trained to a specific user.[12] With context information, recognition rates can be enhanced effectively to 100 percent, but context constraints do not help for all kinds of input, for example, when entering words that are not in the on-line dictionary. The Apple Newton's handwriting recognition, while among the best of the PDAs, is nevertheless reputedly a source of frustration. Recognizing a user's intention in a general setting is inherently hard because the interpretation of pen strokes is ambiguous. For example, a user drawing a circle may intend to select an object or an area, or write a zero, a degree sign, or the letter *o*.

Speech generation and rec-

**Table 3. Power consumption of the major components in a portable computer.***

| System | Power (percent) |
|---|---|
| Display edge-light | 35 |
| CPU/memory | 31 |
| Hard disk | 10 |
| Floppy disk | 8 |
| Display | 5 |
| Keyboard | 1 |

*Data was obtained from the Compaq LTE 386/s20 manual.

APPENDIX X

ognition seem an ideal user interface for a mobile computer in that they require no surface area and allow hands-free and even eye-free operation. The voice-commanded VCR programmer by Voice Powered Technology demonstrates the feasibility of speech input to a hand-held device for a narrow domain. The Sphinx research project at Carnegie Mellon University has reported speaker-independent recognition rates of nearly 96 percent, and 98 percent for speaker-trained recognition. However, general-purpose speech input and output places substantial storage and processing demands on a mobile device. Also, speech may often be inappropriate: It disturbs others in quiet environments, it cannot be recognized clearly in noisy environments, and it can compromise privacy. Finally, because of its sequential nature, speech is ill-suited for skimming data.

*Pointing devices.* The mouse is the standard pointing device for desktop computers, but it doesn't suit mobile computers. Pens have become the standard input device for PDAs because of their ease of use while mobile, their versatility, and their ability to supplant the keyboard.

Switching to pens requires changing both the user interface and the software interface because a mouse and a pen are really quite different.[9] Users can jump to absolute screen positions and enter path information more easily with a pen than with a mouse, and it is nearly impossible to write with a mouse. Pen-positioning resolution on current tablet computers is several times more accurate than screen resolution; for example, pen resolution on the EO is 0.10 mm, while screen resolution is 0.23-0.30 mm. Also, parallax between the pen tip and the screen image can mislead when pointing; with a mouse there is no parallax because the mouse cursor provides feedback in the image plane. Finally, the mouse cursor obscures much less of the screen than the user's hand does when writing with a pen.

**Small storage capacity.** Storage space on a portable computer is limited by physical size and power requirements. Traditionally, disks provide large amounts of nonvolatile storage. In a mobile computer, however, disk drives are a liability. They consume more power than memory chips, except when off line, and they may not really be nonvolatile when subject to the indelicate treatment a portable device receives. Hence, none of the PDA products have disk drives.

Coping with limited storage is not a new problem. Solutions include compressing files automatically, accessing remote storage over the network, sharing code libraries, and compressing virtual memory pages. Although today's networked computers have had great success with distributed file systems and remote paging, mobile computers that regularly encounter network disconnections are less capable of relying on a network.

A novel approach to reducing the size of program code is to interpret script languages instead of executing compiled object codes, which are typically many times larger than the source code. This approach is embodied by General Magic's Telescript and Apple Technology Group's Dylan and NewtonScript. An equally important goal of such languages is to enhance portability by supporting a common programming model across different machines.

Mobile computing is a technology that enables access to digital resources at any time, from any location. From a narrow viewpoint, mobile computing represents a convenient addition to wire-based local area distributed systems. Taken more broadly, mobile computing represents the elimination of time-and-place restrictions imposed by desktop computers and wired networks.

In forecasting the impact of mobile technology, we would do well to observe recent trends in the use of the wired infrastructure, in particular, the Internet. In the past year, the advent of convenient mechanisms for browsing Internet resources has engendered an explosive growth in the use of those resources. The ability to access them at all times through mobile computing will allow their use to be integrated into all aspects of life and will accelerate the demand for network services. The challenge for computing designers is to adapt the system structures that have worked well for traditional computing so that mobile computing can be integrated as well. ■

## Acknowledgments

Support for this work was provided in part by the National Science Foundation (grants CCR-9123308 and CCR-9200832), Tektronix Inc. (a graduate fellowship), the Washington Technology Center, and Digital Equipment Corp. (Systems Research Center and External Research Program). We thank Robert Bedichek, Brian Bershad, Blake Hannaford, Marc Fiuczynski, Brian Pinkerton, and Stefan Savage for helpful pointers and clarifying discussions that significantly improved this article.

## References

1. M. Weiser, "Some Computer Science Issues in Ubiquitous Computing," *Comm. ACM*, Vol. 36, No. 7, July 1993, pp. 75-84.

2. J.J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," *ACM Trans. Computer Systems*, Vol. 10, No. 1, Feb. 1992, pp. 3-25.

3. C.D. Tait and D. Duchamp, "Detection and Exploitation of File Working Sets," *Proc. 11th Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 2144, 1991, pp. 2-9.

4. B.C. Neuman, "Protection and Security Issues for Future Systems," in *Workshop on Operating Systems of the 90s and Beyond*, Lecture Notes in Computer Science #563, Springer-Verlag, New York, 1991, pp. 184-201.

5. J. Ioannisdis, D. Duchamp, and G.Q. Maguire Jr., "IP-Based Protocols for Mobile Internetworking," *Proc. SIGComm 91 Symp.*, ACM, New York, 1991, pp. 235-245.

6. T. Imielinski and B.R. Badrinath, "Data Management for Mobile Computing," *SIGMOD Record*, Vol. 22, No. 1, Mar. 1993, pp. 34-39.

7. C. Ma, "On Building Very Large Naming Systems," *Proc. Fifth ACM SIGOPS Workshop Models and Paradigms for Distributed Systems Structuring*, ACM, New York, 1992, 5 pp.

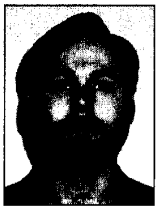
8. F. Teraoka and M. Tokoro, "Host Migration Transparency in IP Networks: The VIP Approach," *Computer Comm. Rev.*, Vol. 23, No. 1, Jan. 1993, pp. 45-65.

9. D. Duchamp, S.K. Feiner, and G.Q. Maguire Jr., "Software Technology for Wireless Mobile Computing," *IEEE Network Magazine*, Vol. 5, No. 6, Nov. 1991, pp. 12-18.

10. A. Chandrakasan, S. Sheng, and R.W. Brodersen, "Design Considerations for a Future Portable Multimedia Terminal," in *Third-Generation Wireless Information Networks*, S. Nanda and D.J. Goodman, eds., Kluwer Academic Publishers, Hingham, Mass., 1992, pp. 75-97.

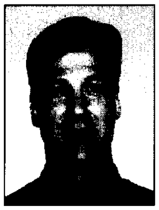
11. K. Li et al., "A Quantitative Analysis of Disk Drive Power Management in Portable Computers," tech. report, Computer Science Division, University of California, Berkeley, Calif., 1993.

12. C.C. Tappert, C.Y. Suen, and T. Wakahara, "On-Line Handwriting Recognition — A Survey," *Proc. Ninth Int'l Conf. Pattern Recognition*, Vol. 2, IEEE CS Press, Los Alamitos, Calif., Order No. 878, 1988, pp. 1,123-1,132.



**George Forman** is a PhD candidate in the Department of Computer Science and Engineering at the University of Washington. His research interests include mobile computing and compilers for parallel computers.

Forman received his BA in mathematics from Pomona College, California, in 1988. The following year he received a Fulbright fellowship for study at the Swiss Federal Institute of Technology, Zurich. He is a member of Sigma Xi and Phi Beta Kappa.



**John Zahorjan** is a professor of computer science and engineering at the University of Washington. His research interests include performance modeling and experimental evaluations, as well as issues in mobile computing, runtime support for parallel computing, and resource scheduling for continuous-media applications.

Zahorjan received an ScB in applied mathematics from Brown University in 1975 and MSc and PhD degrees in computer science from the University of Toronto in 1976 and 1980, respectively. In 1984 he received a Presidential Young Investigator Award from the National Science Foundation. He is a member of the IEEE Computer Society.

The authors can be contacted at the Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195, e-mail {forman, zahorjan}@cs.washington.edu.

# A Network Architecture for Mobile Computing[*]

Kevin Brown
Dept. of Computer Science
Univ. of South Carolina
Columbia, SC 29208
*kbrown@cs.sc.edu*

Suresh Singh
Dept. of Computer Science
Univ. of South Carolina
Columbia, SC 29208
*singh@cs.sc.edu*

September 7, 1995

**Submitted for publication**

**Abstract**

In this paper we report on an ongoing project to design and build the network and transport layers for mobile networking. The network architecture used is unique in that it separates the mobile network(s) from fixed networks and provides connectivity between the two via special gateways. These gateways provide QOS guarantees to mobile users for all their open connections. We provide summaries of all the protocols we are implementing (or have implemented) and discuss possible improvements.

```
Technical area : Wireless Networks and
                 Protocols

Corresponding author: Suresh Singh

Telephone: (803)777-2596
      Fax: (803)777-3767
```

## 1 Introduction

*Mobile computing* is an emerging new computing environment incorporating both wireless and high-speed networking technologies. Users equipped with *personal digital assistants* or PDAs

[*] Funding for this work was provided by the NSF under grant number NCR-9410357.

(palm-top computers) will have access to a wide variety of services that will be made available over national and international communication networks. Mobile users will be able to access their data and other services such as electronic mail, electronic news, yellow pages, map services, electronic banking and videotelephony services while on the move. To receive these services, mobile users will be connected to fixed networks via wireless networks (or mobile networks).

The **goal** of this paper is to present a comprehensive solution to the problem of wireless networking for mobile computing. We propose a mobile network architecture, network layer design and transport protocols that, we believe, will make it possible to offer all of the above services in an integrated manner. Such a system is currently being built at University of S. Carolina and, in this paper, we discuss the design in detail.

### 1.1 Challenges in Mobile Networking

Providing the type of services discussed above to mobile users requires high data rates on the wireless link and several authors (see for example Goodman[7, 8] Joseph[11]) have proposed an average data rate of between 1–2Mbps per mobile user. The third generation cellular system being developed in Europe (UMTS – Universal Mobile Communication System), for instance, also propose bandwidth in the same range (see DaSilva[4]). In order to support such high data rates, a *microcellular* network architecture has been proposed, see Goodman[7] and Duchamp[5]. Here, a geographical region such as a campus is divided into *microcells* with a diameter of the order of hundreds of meters. All mobile users within a microcell communicate with a central host machine within that cell who serves as a gateway to the wired networks; this machine is called a *mobile support station* (MSS).

What are some of the *networking issues* we need to address in order to provide the different types of service discussed above? Two broad issues that need to be considered are the following:

- Design of an efficient network architecture to support mobility and corresponding network layer protocols. The problems here include,
  - Tracking mobile users as they roam,
  - Routing messages and forwarding them to the current location of the mobile user,
  - Providing flow-control and buffering for open connections.
- Developing transport layer protocols that mesh easily with protocols that will be made available over high-speed networks. Some requirements here include,
  - Developing mobile network analogues of commonly used protocols such as TCP and UDP,
  - Developing protocols that support real-time applications such as voice and video or on-line data services,

- Maintaining quality of service guarantees for applications even in the presence of mobility.

Some of the problems mentioned above have been addressed by other researchers as we discuss below.

*Routing in Mobile Networks:* In mobile networks, since the hosts are mobile, routing is a problem. Ioanidis[9] proposes a solution called the **IPIP** ("IP-within-IP") protocol. Here every MH has a unique IP address called its 'home address'. To deliver a packet to a remote MH, the source MSS first broadcasts an ARP to all other MSS nodes to locate the MH. Eventually some MSS responds. The source MSS then encapsulates each packet from the source MH within another packet containing the IP address of the MSS in whose cell the destination MH is located. Upon receiving this packet the destination MSS extracts the original packet and attempts to deliver it to the destination MH. If the MH has moved away, the destination MSS attempts to locate it by broadcasting an ARP request. As discussed in Teraoka[15] this method is not easily scalable.

Teraoka[15] proposes a more flexible solution to the problem called – the Virtual Internet Protocol or **VIP**. Here every host has a *virtual network address* (VIP address) that is unchanging. In addition, hosts have associated *physical network addresses* (traditional IP addresses) that may change as the host moves around. At the transport layer, the target node is always specified by its VIP address only. The address resolution from the VIP address to the IP address takes place at either the network layer of the same machine or at a gateway. Both, the host machines and gateways, maintain a cache of VIP to IP mappings with associated time stamps. This information is in the form of a table and is called AMT (or *address mapping table*). Routing is achieved by referring to these AMT tables.

*Transport Protocols for Mobile Networks:* Since mobile hosts will expect the same services that are offered to fixed hosts, it is necessary to implement transport services in the mobile domain that are similar to those offered in the fixed networks. TCP is one such protocol. If we use TCP without any modification in mobile networks we have a serious problem of efficiency. This is because in TCP the sender begins retransmission of packets if they are not acknowledged within a short amount of time (hundreds of milliseconds). In a mobile environment, as a user moves between cells, there is a brief blackout period while the mobile unit performs a 'handshake' with the new MSS. These blackout periods may also be caused by physical obstacles in the environment that interfere with radio signals. These periods can be of the order of 1 second thus delaying the transmission of acknowledgements for packets received. This results in the TCP sender timing out and retransmitting the unacknowledged packets thus greatly reducing the efficiency of the connection. A solution to this problem is the I-TCP protocol (Indirect-TCP), implemented at Rutgers University as part of the DataMan project (see Barke[1]), that provides efficient reliable communication for the wireless environment. A benefit of the above implementation is that it allows mobile hosts to be connected over the Internet. We examine the I-TCP protocol in more detail in section 3.2.1 where we compare it against our own proposal.

### 1.2 Summary of Paper

We are currently building a mobile network from the ground up and this paper discusses our design and initial experience. Specifically, we propose a design for the network layer and transport layers for 3rd generation wireless systems and provide arguments in support of this design. The implementation of the protocol stack is been done under Unix (we use NetBSD) running on Pentium PCs.

- In section 2 we discuss our architecture for the mobile subnetwork that, we feel, best addresses the various issues raised in section 1.1.

- Our network layer design is presented in section 3.1. The sketch of our transport layer is presented in section 3.2. We also address management and control questions specific to the mobile environment (e.g., feedback of dynamic bandwidth changes to applications, etc.). Special protocols for notification applications (e.g., pager service) and continuous media are also incorporated.

## 2 Overview of our Proposed Architecture

The solutions discussed in section 1.1 use a microcellular architecture where the base station for each cell is a node on the internet. The solutions proposed for routing and TCP implementations thus assume that the underlying subnetwork is a datagram network. We believe that this assumption has only one justification – compatibility with existing technology – which, in our view, is insufficient. Some of the problems with this approach are,

- The base stations (or MSSs) are responsible for tracking mobile users and forwarding packets to their new locations. This adds to the *cost* and *complexity* of the base stations and since we expect cell sizes to be small (100m), in order to accommodate high-bandwidth applications, the total cost of a mobile network will be very high.

- As a user roams between cells, the bandwidth available in each cell may also vary. If the user has open connections, it will have to renegotiate QOS parameters frequently. This is clearly an undesirable situation.

- Cell latency times (staying time in a cell) are typically small (several seconds). This exacerbates both of the above problems.

Our architecture is discussed in detail in Singh[14]. We summarize the main points in this section. Our architecture may be viewed as a three-level hierarchy (see Figure 1). At the lowest level are the mobile hosts (MH) who communicate with MSS nodes in each cell. Several MSSs are controlled by a machine called the Supervisor Host (SH). The SH is connected to the wired network and it handles most of the routing and other protocol details for the mobile users. In addition it maintains connections for mobile users, handles flow-control and is responsible for maintaining the negotiated quality of service. A single SH may thus control all MSS nodes within a small building. Our architecture separates the mobile network from the high-speed wired network and provides connectivity between the two via supervisor hosts (SHs) who serve the function of a **gateway**.

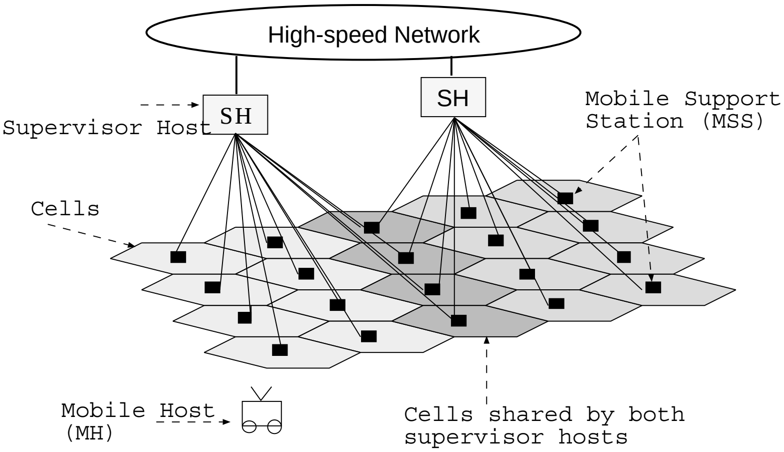


Figure 1: Proposed Architecture.

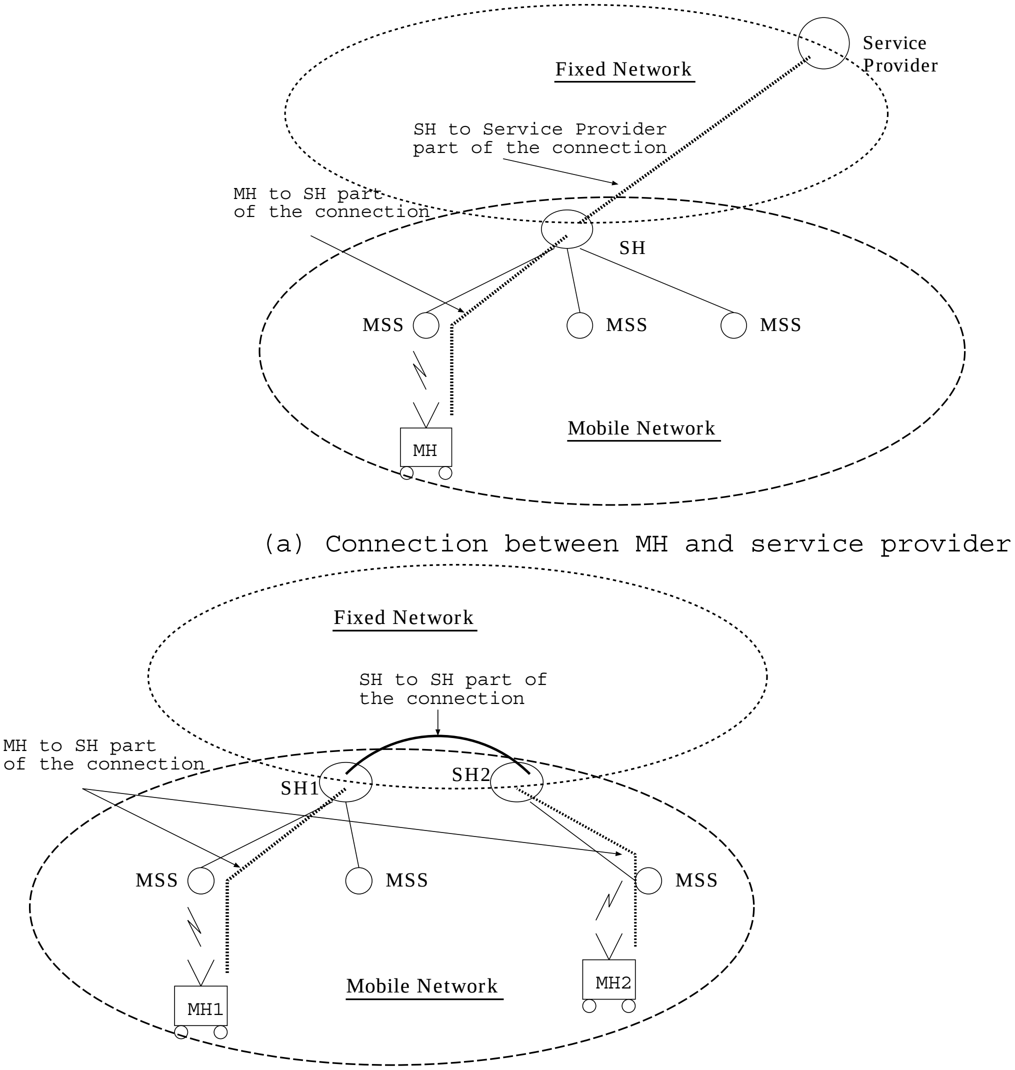
A mobile user may set up connections where the other end-point is either another mobile user or a fixed host (e.g., a service-provider) in the fixed network. In either case the connection is managed by the current SH of the mobile host(s) (see Figure 2). The reason for splitting the connection between the MH and the service-provider is to shield fixed nodes from the idiosyncrasies of the mobile environment. Thus, the service-provider sets up a connection with the SH assuming the SH is the other end-point of the connection. The SH sets up another connection to the MH. Thus for every **MH – service-provider** connection the QOS parameters are defined *separately* for the **MH – SH** part and for the **SH – service-provider** part of the connection. Connections between two MHs are broken in three (if the two MHs are controlled by different SHs) – **MH1 – SH1**, **SH1 – SH2** and **SH2 – MH2**. Note that the SH–SH part of any connection is established over the fixed network. The benefits of our architecture, thus, are:

- The MSSs are simple and cheap devices because they merely serve as a point of attachment for MHs.

- Since several MSSs are controlled by a single SH, the roaming MH remains within the domain of the same SH for long time periods[1]. This makes it easier to guarantee QOS

[1] e.g., a MH may roam frequently between rooms in an office building but remain for many hours within the building – each room is a cell and all cells in the building are controlled by one SH.

APPENDIX Y



(a) Connection between MH and service provider

(b) Connection between two MH nodes

Figure 2: Connections for MHs are managed by SHs.

parameters for MH connections.

## 3 Network and Transport Layer Protocols

Our view of the mobile network in relation to fixed wired networks is illustrated in Figure 3. The mobile network is actually composed of many sub-networks each of which is connected to the fixed networks via a SH node. SH nodes communicate with one another over the fixed network. Each SH controls several MSS nodes. Physical communication between a SH node and its MSS nodes is accomplished either over a dedicated network consisting of dedicated wiring (perhaps several MSSs are connected via twisted-pair to a hub and several hubs are connected directly to the SH) or over the fixed network itself – i.e., the MSS nodes are regular hosts on the fixed network and can be addressed with IP addresses. The latter is a cheaper solution and provides a *migration* path to having dedicated mobile networks.

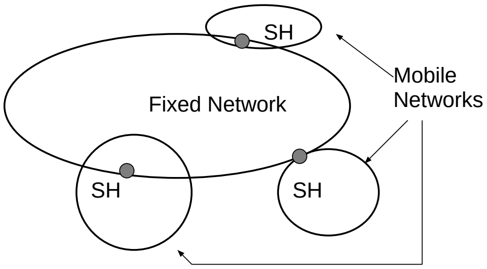


Figure 3: Relationship of mobile networks to fixed networks.

## 3.1 Network Layer Design

### 3.1.1 Routing and Tracking

Recall that all connections set up by a MH pass through its SH. For instance, a connection between MH $M_1$ and $M_2$ located within the same mobile network (i.e., same SH $S$) is set up as $M_1$–$S$–$M_2$. If the two MHs are in different mobile networks $S_1$ and $S_2$ the connection is $M_1$–$S_1$–$S_2$–$M_2$. The $S_1$–$S_2$ portion of the connection passes over the fixed network. A connection from $M_1$ to a fixed host $F$ is set up as, $M_1$–$S$–$F$ [2]. In all of these cases, notice that routing consists of two components – routing within a given mobile network from the SH to the MH and routing over the fixed network between SHs or between SHs and fixed hosts. Let us consider each of these two components separately.

In our design we implement *virtual circuits* at the network layer *in each mobile sub-network*. This means that the network layer will deliver all packets *in order* to the current MSS of the MH. Thus even if the MH moves between cells, so long as the MH remains within the same mobile sub-network, all packets will be delivered to its MSS in order. It is important to observe that the network layer *does not* guarantee delivery of packets *to the* MH; it only guarantees delivery to the MSS. This is because the wireless link is very unreliable and error recovery is best left to the transport layer which is responsible for implementing service guarantees. We discuss this point in greater detail in section 3.2. Moreover, if the MH moves into the domain of another SH, there are no guarantees made regarding the delivery of packets in transit.

To route within one mobile sub-network (i.e., within the domain of one SH), the network layer at the SH maintains a location table consisting of entries for each MH currently within its domain and the location of the MH (i.e., the identity of its MSS). This table is updated via control messages passed between the MSS and SH every time a MH moves. If the MSS nodes are

[2] The reason for routing all connections through a SH is that the SH can provide network level buffering and flow-control. If we set up a connection directly between two MHs, for instance, flow-control and retransmission of lost packets will require an exchange of control messages between the MHs consuming scarce wireless bandwidth.

connected via dedicated links to the SH, there is no need for them to have IP numbers. The SH simply transmits on the appropriate port. If the MSS nodes are nodes on the fixed network, then the SH needs to route messages to the MSS nodes on the fixed network. In our implementation all MH nodes have unique IP addresses (with some 'home' network as part of the address). Here the SH nodes route messages to the MH by using the IP *loose source routing option* (following the work of Johnson[10]). The destination address in the header is set to the IP address of the MSS and the MH IP address is contained as the first IP address in the option part of the header. The MSS examines the datagram and delivers it to the MH (if it is present in its cell). If it has moved away, the MSS discards the datagram.

To implement reliable delivery (in the sense discussed earlier), every datagram is given a sequence number. A MSS sends an ACK for each datagram transmitted to the MH (note that the datagram may not be received by the MH because of fading or other interference). Until a datagram is ACKed, it is buffered at the network layer of the SH. If a MH moves away from its current MSS to a new MSS, the old MSS discards all messages but simultaneously informs the SH of the sequence numbers of these discarded messages. These messages are then retransmitted to the new location of the MH. A detailed protocol is presented in Gahi[6] (though our current implementation contains many changes).

For routing over the fixed network (e.g., between SHs or between an SH and a service provider), the existing routing protocol provided over the fixed network is used. Thus the network layer shown in Figure 4 is local to the mobile sub-networks only. The network layer shown in Figure 4 consists of two sub-layers – a tracking and VC maintenance sub-layer sitting on top of IP. The tracking sub-layer is responsible for maintaining location information for each MH currently in that mobile sub-network. VC maintenance refers to the task of guaranteeing reliable delivery of datagrams. It is noteworthy that we currently use IP for routing purposes. This choice is dictated more by budgetary constraints than scientific ones.

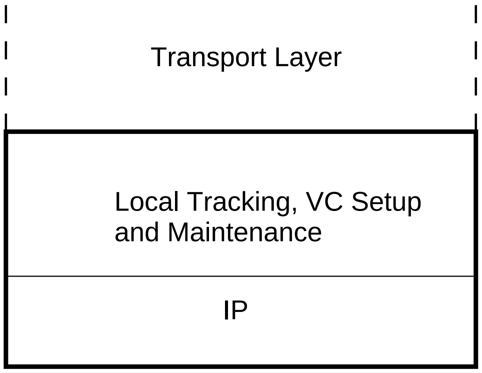


Figure 4: Network Layer.

### 3.2 Transport Layer Design

When developing transport layer protocols for the mobile environment, it is important to keep the following constraints in mind:

- The wireless link is very fragile and error-prone. This means that any reliable protocol must perform a significant amount of error-recovery.
- The bandwidth of the wireless link will always be a limited resource. Thus, all protocols need to be 'lean'. For instance, in the case of TCP-like protocols, it is not a good idea to have end-to-end flow control (where one end is mobile) because of the high number of control messages that will be sent.
- Available bandwidth within a cell may change dynamically (because it is impossible to control the number of users per cell). This leads to all kinds of problems in guaranteeing QOS parameters such as delay bounds and bandwidth guarantees.
- Mobile hosts frequently encounter extended periods of disconnection (caused due to handshake or due to physical interference with the signal) and this will result in significant losses of data for UDP-like protocols. We need to redefine best-effort service for such cases.
- Mobile hosts may move between SHs after opening transport connections. Should the old SH continue to be responsible for these connections or should control be transferred to the new SH?

All of the above issues can be summarized in the form of two questions. The answer to these questions will determine the transport layer design.

**(1) Should the transport layer be aware of the mobility of MHs?**

**(2) Should the transport layer be aware of bandwidth fluctuations at the wireless link?**

If we were to strictly follow the layering idea of the OSI hierarchy, mobility and bandwidth fluctuations will have to be concealed from the transport layer. For mobile networks, however, even though we can adhere to this philosophy, it will result in degraded performance and increased message overhead. To see why this is the case, let us consider two scenarios that will be commonplace in a mobile environment. In the first, a MH with open data connection(s) moves into a cell where there are many other MHs. It is likely that the negotiated QOS for the connections of this MH can no longer be satisfied and will have to be renegotiated. Since renegotiation of QOS parameters involves the transport layer (and the network layer), it is not possible to conceal bandwidth fluctuations from the transport layer. We discuss this problem further in section 3.2.4.

For the second scenario, consider what happens if a MH with an open TCP connection moves from its current SH (where the TCP connection was established) to another SH. In this case it is not necessary to re-establish the TCP connection (because the network layer can forward datagrams from the old SH to the new SH). However, the efficiency of the TCP connection will be degraded for reasons discussed in section 1.1 (recall that I-TCP attempts to alleviate this efficiency problem by breaking the TCP connection in two). In our architecture, as we discuss in section 3.2.1 and section 4, since we perform a great deal of bandwidth management within each mobile sub-network (controlled by a single SH), it becomes necessary to re-establish connections when a MH moves from the sub-network of one SH into the sub-network of another. Thus, we believe, the transport layer will need to be aware of the mobility of MHs and bandwidth fluctuations for each open connection.

Our view for the transport layer of the mobile sub-network is shown in Figure 5. The prefix M stands for 'mobile'. Thus we have a version of mobile-TCP and mobile-UDP. M-CM refers to the mobile-continuous media protocol and is useful for implementing real-time services such as voice or video. It is loosely based on the continuous media protocol of Moran[12]. The mobility management module deals with the problems of mobility (i.e., re-establishing transport connections when a MH moves between sub-networks and renegotiating QOS parameters).

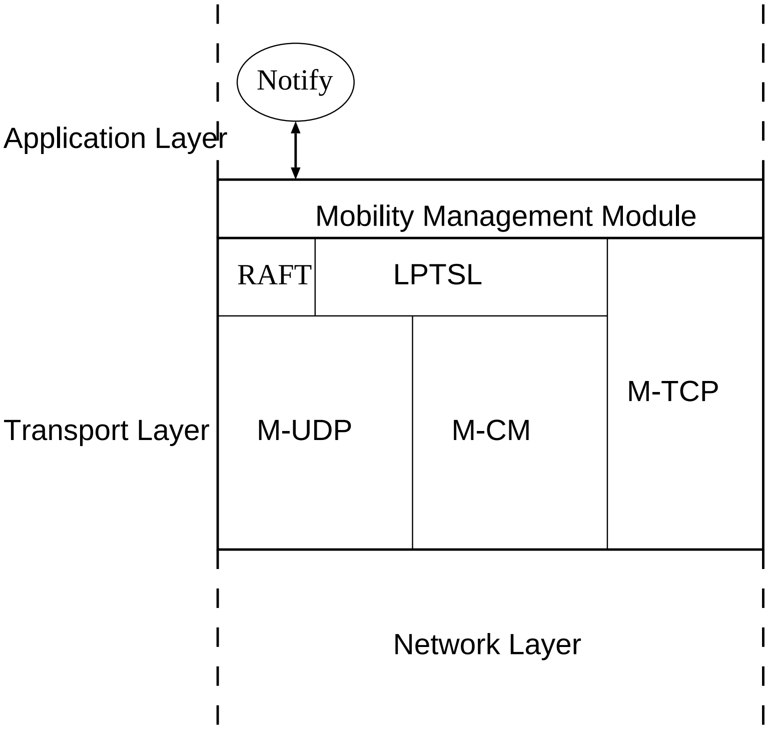


Figure 5: Transport Layer.

### 3.2.1 M-TCP

In section 1.1 we touched upon the problems associated with providing TCP-like service over mobile networks. Specifically, if the receiver is a MH, the TCP sender times out frequently because of the 'blackout' periods in mobile networks. The solution of Barke[1] breaks the connection in two (similar to our idea with the difference that they consider mobile networks to be part of the internet) – from the fixed host to the MSS and from the MSS to the MH. The MSS effectively serves as the TCP connection end-point from the point of view of the fixed host. The MSS is then responsible for forwarding all data reliably to the MH. Note that the *semantics* of this implementation differ from TCP semantics – it is possible for the sender to think that all data has been correctly delivered to the MH (since the MSS has sent ACKs) even if this is not the case.

In our design, the TCP connection is broken in two – fixed host to SH and SH to MH. The fixed host to SH part of the connection uses regular TCP while M-TCP is used for the SH to MH part of the connection. Since M-TCP is layered on top of a reliable virtual circuit connection (reliable within one mobile sub-network), its design is relatively simplified. Even though this design looks similar to I-TCP, unlike Brake[1], we implement almost TCP-like semantics that make it easier for the sending application to recover from the type of error described above. The TCP client on the SH always ACKs *all but the last byte* of data received from the sender. The last byte is ACKed only after it has been successfully sent to the MH by M-TCP at the SH. In this scenario, if the MH disconnects before receiving all data from the SH, the sender will never receive an ACK for the last byte. Thus, the sending application will know that the connection has failed and can take remedial action. In addition to the above change, we ensure that the buffers at the SH are not exhausted (which can happen, for instance, if the MH has been temporarily disconnected) by linking buffer availability (i.e., receiver window size) to the *available bandwidth on the wireless link*. This causes the TCP client to automatically choke the sender when the MH is in a crowded cell! This linkage between expected wireless bandwidth and TCP buffers is another unique feature of our design.

It is important to note, however, that M-TCP semantics are still slightly different from TCP semantics. This is best illustrated by considering a *talk* application. The application displays data on the user screen as and when it is received over the TCP connection. If the SH crashes, it is possible for the sender to think that almost all the lines it has typed thus far have appeared on the receiver's screen (because ACKs have been received for all but the last byte) even though this may not be the case (i.e., the SH crashed before that data was sent to the MH). This situation is not completely hopeless, however, because the sender will eventually realize that the SH has crashed and it can then take remedial action.

What happens when a MH moves from the sub-network of one SH into the sub-network of another? In our implementation this is handled in a manner similar to I-TCP. The old SH transfers TCP state to the new SH after it has been informed by the mobility management module about the move (recall that whenever a MH enters a new cell, it performs a handshake procedure with the new MSS. The information exchanged during this handshake includes the identity of the previous SH and information about all open connections, see Ghai[6]). Meanwhile all datagrams arriving at the old SH are sent on to the new SH via IP loose source routing.

In our implementation, we require the MH to maintain the identity of the original SH (who first set up the TCP connection). If the MH moves from $SH_{original}$ to $SH_1$ to $SH_2$, after $SH_1$ has transferred TCP-state to $SH_2$, $SH_{original}$ is given the IP address of $SH_2$ so that it can route datagrams directly to $SH_2$ without going through $SH_1$ first. This method keeps the cost of forwarding datagrams small. In the future, if IP is replaced by a protocol like VIP, we will not have to worry about this particular problem. This is because the intermediate routers in the fixed network will automatically associate the IP address of $SH_2$ with the VIP address of the MH (see Singh[14] for more details) and route datagrams accordingly.

### 3.2.2 M-UDP

If we were to implement UDP in the mobile network without any changes, the performance seen by a MH would be very poor. This is because packets transmitted while the MH is moving between cells or is blocked by some physical obstruction are lost (note that in TCP these packets would be retransmitted). Only a small percentage of loss is due to lack of bandwidth. A high-level view of our M-UDP protocol as implemented is the following (a detailed protocol may be found in Brown[2]).

- Every UDP packet is buffered at the SH.
- The SH discards a packet if it has run out of buffer space or if it has been transmitted $n$ times to the MSS.

The semantics of M-UDP are almost identical to UDP in the sense that delivery is not guaranteed. However, M-UDP attempts a 'best effort' service that is constrained only by buffer space availability at the SH and by bandwidth availability on the wireless link. In our experiments we observed a 2 to 3 fold improvement in the number of packets delivered by M-UDP in comparison to UDP (for mobile hosts).

### 3.2.3 M-CM

A large percentage of future applications will require transmission of data at regular intervals. This kind of data is referred to as continuous media and some examples include voice communication, video communication, etc. Continuous media applications have severe time deadlines and bandwidth requirements and thus cannot be implemented on top of message-based transport protocols such as M-TCP or M-UDP. Following Moran[12] we propose a separate transport protocol suite called M-CM (mobile-continuous media) that will provide the functionality required by such applications. Unfortunately, however, the protocols proposed in Moran[12] and other solutions

APPENDIX Y

proposed for the high-speed network domain cannot be adapted to the mobile networks because of two reasons:

- bandwidth availability varies in an unpredictable manner as hosts roam,
- fading and handoff cause periods of disconnection.

All proposals for CM-type protocols require the network layer to provide strict guarantees regarding bandwidth availability and delay bounds for each connection. Because of the above reasons, however, this is not possible in the mobile domain.

Our approach is to provide a "best-possible" service to M-CM connections. Intuitively, this means that the SH will arbitrate between various MH connections to determine how much bandwidth must be allocated to each open connection. Thus, if a MH has an open ftp connection (via M-TCP) and an open video connection (via M-CM), a *scheduler* process (CS process, see section 4) will starve the ftp connection in favor of the video connection if the available bandwidth, within the current cell of the MH, gets reduced. The scheduler also interfaces with LPTSL in case some fraction of data along the M-CM connection(s) need to be discarded. This M-CM protocol has not yet been fully specified.

### 3.2.4 LPTSL (*Loss Profile Transport Sub-Layer*)

Future applications to be provided to mobile users will include audio (e.g., telephone, audio conferencing, etc.) and video applications (e.g., map information, viewing movies, etc.). These applications have real-time constraints and therefore need to be implemented over M-CM. However, we have a unique problem of *dynamic bandwidth changes during the lifetime of a connection* caused due to the **unpredictable mobility** of mobile hosts.

To illustrate a consequence of this unpredictable mobility, consider a situation where several mobile users have opened high-bandwidth connections. When these connections are set up, the network ensures that the users receive some guaranteed bandwidth. Since these users are all mobile, it is possible that many of them could move into the same cell. In such a situation, it is very likely that the requested bandwidth of the cell will exceed available bandwidth resulting in the original QOS (quality of service) parameters being violated. This situation does not arise in high-speed networks because users are not mobile during the life-time of a connection.

To deal with this situation, we propose that most open connections, *than can tolerate losses*, within the choked cell be penalized (either equally or differentially, based on some need-based policy). Thus, a penalized connection will see a $x\%$ reduction in available bandwidth. The question now is – does the MH renegotiate the QOS parameters to force the sender to reduce the connection bandwidth (e.g., use a higher compression ratio for a video connection) or is data for that connection discarded by the SH? The first option sounds attractive but is not necessarily the right choice for the following reasons:

Microsoft et al. Exhibit 1005

- The bandwidth crunch is probably temporary and will be alleviated as soon as a MH roams out of the cell. When this happens, the QOS parameters will have to be renegotiated.
- The cell latency of a MH is of the order of several seconds. The end-to-end renegotiation process is time consuming and thus the bandwidth available may change even before this renegotiation is completed.
- While the renegotiation is going on, data will continue to be sent at the old rate. Since the wireless bandwidth is small, the buffers at the SH will possibly overflow.

We propose that the SH *judiciously* discard data for each penalized connection. Since the SH operates at the transport layer (it is a gateway), it can do this. Note, however, that indiscriminate discarding of data may result in garbage at the MH (e.g., if data is randomly thrown out of a compressed video stream, no video can be reconstructed). To solve this problem, we have proposed a new sub-layer called the Loss Profile Transport Sub-Layer. The sending application puts *flags* in the data stream by making calls to LPTSL. All data between a pair of flags represents a *logical segment* (e.g., one logical segment may be a single compressed frame in JPEG-video). The LPTSL at the SH discards entire logical segments in the event of a bandwidth crunch to ensure a $x\%$ reduction in bandwidth (note that the application at the MH can be informed of the location of the discarded segments in case that information is required – as in MPEG-2 video). A detailed protocol is presented in Seal[13]. See Figure 6 for an explanation of the operation of this layer. Here the data stream at the sender is broken into data segments (logical segments) separated by special flags. These flags are inserted by the LPTSL layer at the sender. The LPTSL layer at the SH is informed of the available bandwidth in the current cell of the MH and determines if any data needs to be discarded. If so, it discards entire logical segments (all data between consecutive flags). The LPTSL at the MH passes up the arriving data to its receiving application *and indicates the location and size of the discarded segments.*

A reason for discarding data at the LPTSL is because LPTSL provides different *discarding functions.* When a connection is set up, a QOS parameter negotiated is the discarding function(s) to be used in the event of a bandwidth crunch. For instance, if the connection is an audio connection, the user may prefer uniform random loss as opposed to bursty loss. On the other hand, if the data is compressed video, random loss will prove to be disastrous. In this case the user may opt for bursty loss (i.e., discard entire frames rather than random bytes from several frames). These discarding functions are provided in the form of a indexed library of sub-routines at the LPTSL layer of the SH.

#### 3.2.5 RAFT (*Repetitive Almost-reliable Fast Transport*)

In addition to applications discussed in section 2, PDAs will be used as devices whereby critical data can be transmitted to the user quickly – much like today's pagers or beepers but with a

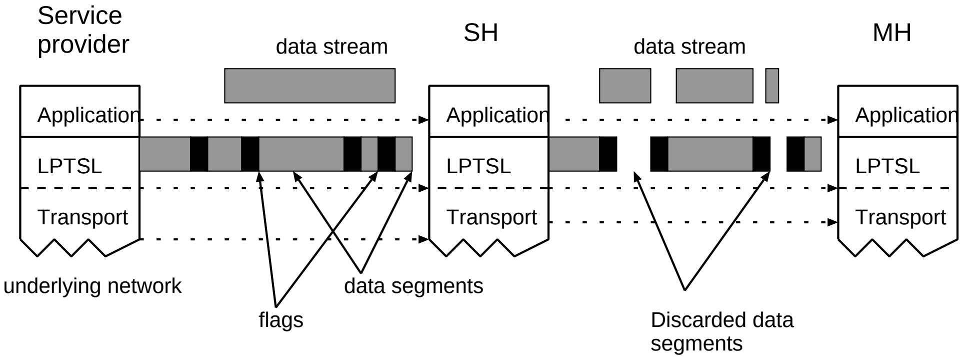


Figure 6: Loss Profile Transport Sub-Layer.

greater degree of sophistication. These type of *notification* applications need to be able to transmit data *quickly* and *reliably* to the mobile user. To facilitate the development of such applications we propose a transport sub-layer called RAFT that is built on top of M-UDP. The approach we use is the following – RAFT at the SH sends data several times. When the MH eventually receives all data correctly it sends a *shut_off* message to RAFT at the SH. RAFT data takes precedence over all other connections.

## 4 Management and Control

In Figure 7, we show the transport and network layer control and management functions at the SH. The SH needs to ensure that QOS parameters for open connections are maintained. This implies appropriate bandwidth management within each cell and buffer allocation for each connection at the SH. In Figure 7 the management and control functions of the protocol stack for the mobile sub-network is shown in detail. The different arrows indicate control paths, management paths, data paths and QOS negotiation paths.

At the network layer we have a process (CBM) that monitors the bandwidth utilization within each cell controlled by the SH and passes this information up to a transport layer process (CS) via a management path. The CS process arbitrates bandwidth within each cell between all open connections. Thus, if there is a real-time connection (e.g., audio) and a data connection into a cell, the CS may choose to starve the data connection in order to ensure the delay bounds for the real-time connection are met. Another control process at the network layer (NL_QOS) monitors the QOS being delivered to each VC and sends this information to the CS as well. This management path is required to ensure that QOS parameters (such as bandwidth usage or delays) for M-CM connections are met. If some M-CM connection has not been receiving its negotiated QOS, the CS process allocates more bandwidth to that connection.

CS is the process which is responsible for maintaining QOS for all M-CM connections and ensuring that data connections (M-UDP or M-TCP) do not get starved in the process. The CS receives information for each M-CM connection's QOS contract. Note that this contract may be renegotiated during the lifetime of the connection and thus CS needs to be informed of this change via the management path. M-UDP connections are subject to data discarding (via LPTSL) in the event of a bandwidth crunch. Thus, we assume that some QOS negotiation also takes place for UDP connections and this information is passed on to the CS process as well. CS *periodically* informs M-CM, M-UDP and M-TCP of the available bandwidth for each connection via control paths. It is up to these protocols to ensure that they control each open connection (either choke the sender as in M-TCP, or discard data via LPTSL as in M-CM) adequately.

QOS negotiations take place between M-CM's QOS process, NL_QOS and possibly the QOS process on the fixed network (in case the connection is to a fixed host) via QOS negotiation paths. QOS control information is also exchanged between LPTSL and the QOS processes of M-UDP and M-CM. Finally, the buffer manager processes at both, the network layer and the transport layer, are responsible for buffer allocation to different connections. The data paths followed for some typical connections are illustrated in the figure as well.

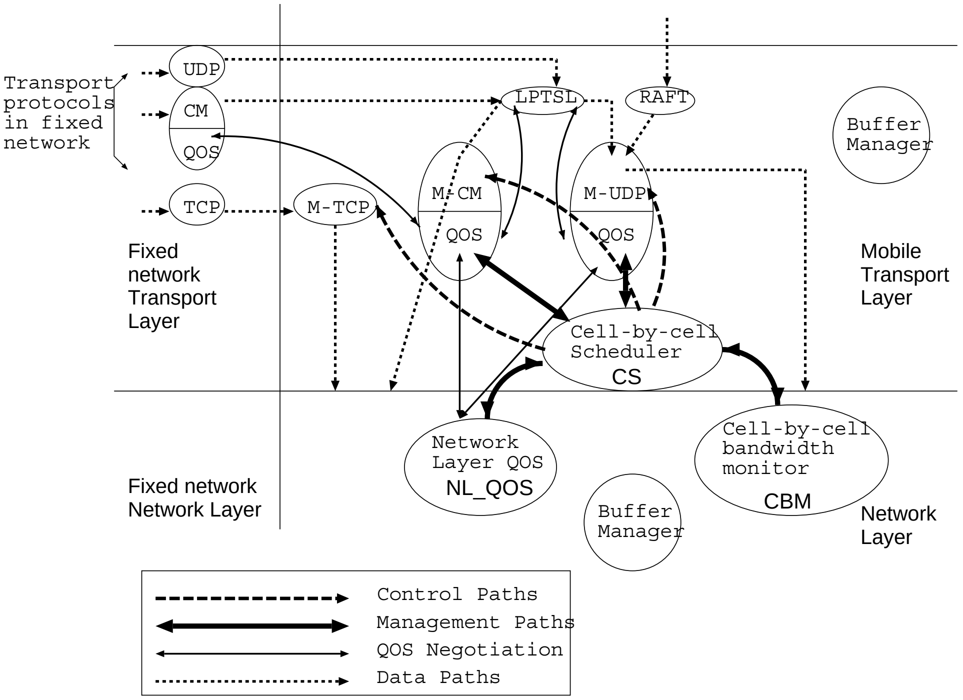


Figure 7: Management and Data flow at the transport and network layers of the SH.

## 5 Conclusions

In this paper we have proposed a complete design of the network layer and transport layer in a manner that best addresses the problems of the mobile environment. Our design is a radical departure from other researchers in that we propose that mobile networks be viewed as separate

APPENDIX Y

from wired networks with connectivity being provided by special gateway nodes. These nodes provide a variety of transport level services that best meet the constraints of the mobile environment. A complete implementation of this architecture is underway and Figure 8 indicates the current status of this implementation.

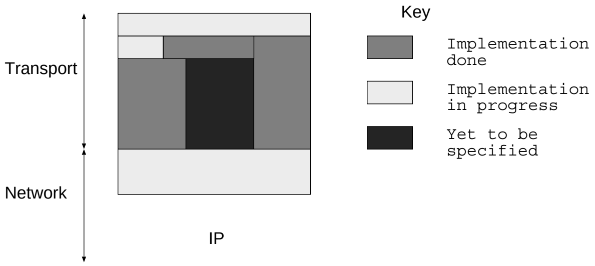


Figure 8: Current status of implementation.

## References

[1] A. Bakre and B. R. Badrinath,"I-TCP: Indirect TCP for Mobile Hosts", *Technical Report* DCS-TR-314, Rutgers University, Piscataway, NJ 08855.

[2] K. Brown and S. Singh,"M-UDP: Mobile UDP", Manuscript.

[3] I. Arieh Cimet, "How to Assign Service Areas in a Cellular Mobile Telephone System", *IEEE ICC'94*, pp. 197-200, May 1994.

[4] J. S. DaSilva and B. E. Fernandes,"The European Research Program for Advanced Mobile Systems", *IEEE Personal Communications Magazine*, February 1995, pp. 14-19.

[5] D. Duchamp, Steven K. Feiner and G. Q. Maguire, "Software technology for wireless Mobile computing" *IEEE Network Mag.*, pp 12-18, November 1991.

[6] R. Ghai and S. Singh,"An Architecture and Communication Protocol for Picocellular Networks",*IEEE Personal Communications Magazine*, Vol. 1(3), 1994, pp. 36-46.

[7] David J. Goodman, "Cellular Packet Communications", *IEEE Trans. on Comm.*, vol. 38, no. 8, pp 1272-1280, August 1990.

[8] David J. Goodman, "Trends in Cellular and Cordless Communications",*IEEE Communications Magazine*, pp 31-40, June 1991.

[9] J. Ioanidis, D. Duchamp and G. Q. Maguire, "IP-based protocols for mobile internetworking" *Proc. of ACM SIGCOMM'91*, pp 235-245, September 1991.

[10] D. B. Johnson,"Mobile Host Internetworking Using IP Loose Source Routing", Technical Report CMU-CS-93-128, Carnegie Mellon University, Pittsburgh, PA 15213, 1993.

[11] C.S. Joseph, *et al*, "Propagation Measurement to Support Third Generation Mobile Radio Network Planning", *43rd IEEE Vehicular Tech. Conf.*, May 1993, pp. 61-64.

[12] M. Moran and B. Wolfinger,"Design of a Continuous Media Data Transport Service and Protocol", Technical Report TR-92-019, Computer Science Division, University of California Berkeley, April 1992.

[13] K. Seal and S. Singh,"Loss Profiles: A Quality of Service Measure in Mobile Computing", *J. Wireless Networks*, (submitted).

[14] S. Singh,"Quality of Service Guarantees in Mobile Computing", *Journal Computer Communications*, (to appear).

[15] F. Teraoka and M. Tokoro, "Host Migration Transparency in IP Networks: The VIP Approach", *SIGCOMM*, Vol. 23, No. 1, Jan 1993, pp. 45-65.

APPENDIX Z

Abstract

The ongoing European ACTS project *OnTheMove* provides support services for distributed mobile multimedia applications. The project defines, implements, and demonstrates a mobile middleware called a Mobile Application Support Environment (MASE) which is based on UMTS concepts. The mobile application programming interface (mobile API) of MASE, which will be submitted for standardization, allows common access to the underlying operating systems and network infrastructure, and facilitates the development of new, mobile-aware, multimedia applications.

# *UMTS: A Middleware Architecture and Mobile API Approach*

BIRGIT KRELLER, SIEMENS AG

ANTHONY SANG-BUM PARK AND JENS MEGGERS, AACHEN UNIVERSITY OF TECHNOLOGY

GUNNAR FORSGREN, ERICSSON RADIO SYSTEMS AB

ERNÖ KOVACS AND MICHAEL ROSINUS, SONY INTERNATIONAL (EUROPE) GMBH

Any information at any time, at any place, in any form. This promise of mobile multimedia will be realized through third-generation mobile communication networks, which will offer high-bit-rate data services, guaranteed on-demand bandwidth, and low delays. The European Telecommunication Standards Institute (ETSI) is working on the Universal Mobile Telecommunications System (UMTS) [1, 2], which belongs to the family of similar or compatible standards developed within the International Telecommunication Union (ITU) called International Mobile Telecommunication in the year 2000 (IMT-2000) [3].

Today, mobile users already utilize a wide variety of mobile terminals ranging from simple mobile phones and personal digital assistants (PDA) to high-end multimedia notebooks. UMTS and the Mobile Broadband System (MBS) will offer suitable bandwidth and global connectivity to enable true mobile multimedia. As an early contribution to the UMTS service specifications, and in order to provide a smooth evolution path from second- to third-generation communication systems, the Advanced Communications Technologies and Services (ACTS) project *OnTheMove* has developed a mobile middleware system called the Mobile Application Support Environment (MASE). Along with the MASE, an application programming interface (API) has been defined that allows applications to access the MASE components. This API, called the *mobile API*, will be submitted for standardization. The purpose of the MASE middleware is to ease the development of mobile-aware applications by providing a common underlying platform. Furthermore, the middleware approach enables a smooth transition from current wireless networks, such as Global System for Mobile Communications (GSM) and Digital European Cordless Telecommunications (DECT) to future UMTS networks. Instead of directly accessing the operating system, mobile-aware applications make use of the *mobile* API and benefit from simple access to MASE services, which hide the complexity of heterogeneous networks and operating systems from the applications. Thus, MASE simplifies the development of mobile-aware applications and frees them from the complex processing caused by additional needs when accessing heterogeneous networks and running on mobile devices. Furthermore, MASE eases the evolution of multimedia applications toward UTMS [4] and enables legacy applications to benefit from a subset of its functionality.

In this article we focus on the services provided by MASE and their relation to UMTS. We demonstrate the interworking between different parts of MASE through a typical mobile-aware application, the CityGuide. The CityGuide uses geographical information provided by the MASE Location Manager module to display a map of the current geographical surroundings of the mobile user. Several different layers of interesting places (e.g., public transportation, administration buildings, accommodation, restaurants, and much more) are shown on the map and are linked to corresponding Web pages.

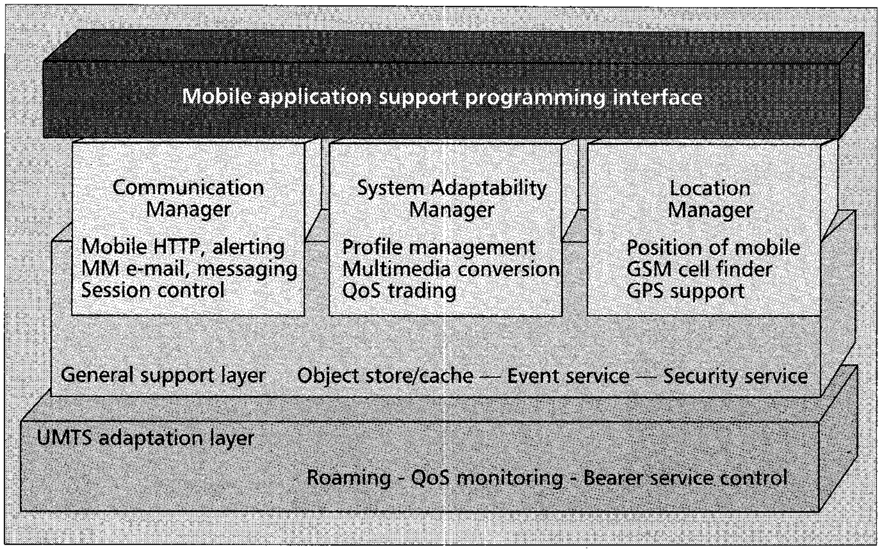
We will first elaborate on the generic MASE architecture. We will outline a possible scenario for the ongoing UMTS service definitions and then explain, step by step, the MASE components accessed by the CityGuide implementation.

## *MASE*

The Mobile Application Support Environment is a distributed system that runs on both the mobile device and the so-called mobility gateway. The latter acts as a mediator between the wireless and fixed network infrastructures. It works as an agent for mobile clients which are typically connected over unreliable wireless access networks with low bandwidth. MASE enables access to the UMTS adaptation layer (UAL), which provides applications and middleware components unified access to all possible underlying networks. An additional general support layer provides the functionality required for distributed systems. On top of both layers several manager components are installed, providing different dedicated services. Figure 1 shows the overall MASE architecture with its corresponding building blocks. All the components shown have been implemented.

MASE is built around the concepts of awareness, adapta-

1070-9916/98/$10.00 © 1998 IEEE



■ **Figure 1.** *Overall architecture of MASE.*

tion, and abstraction. MASE applications are aware of the current network quality of service (QoS) through sophisticated monitoring and management facilities which are provided by the UAL. The UAL hides network specific details by selecting the appropriate bearer service and protocol stack according to the requested QoS. The general functional features of the UAL are:

- Selecting appropriate transport protocols and configuring them for efficient use (e.g., adjusting packet size and timers to the bearer service parameters)
- Selecting and configuring an appropriate bearer service
- Managing roaming between different networks and bearer services as well as bearer service switching

Details of the UAL have been published elsewhere (e.g., [5]). The General support layer implements object storing and caching as well as event handling and security services. These services will also not be described in detail here.

***Awareness*** – End terminal characteristics and user preferences are stored in profiles. They are managed by the Profile Manager, a part of the System Adaptability Manager (SAM), and are available on demand at all nodes of the system.

***Adaptation*** – Profile information and monitored QoS are used by the MASE communication facilities to adapt their usage to the current QoS situation and user requirements. This adaptation is transparent to the application.

***Abstraction*** – The MASE provides high-level abstractions, for example, an alerting function or location management. An alert is an abstraction of an important short message which has to be sent to the user. Depending on the current network situation, the alert manager maps an alert to different network services. If the user is connected to the network over TCP/IP, the message will be delivered directly to an alert server on the mobile device messaging service, or as a GSM short message service (SMS) notification if no such connection is available. The final version of MASE will deal with adaptation to varying degrees of QoS, robustness in the face of disconnected links, roaming between different operators and network types, personalized information filtering, and location-aware applications using various location trackers.

MASE also integrates legacy communication applications and improves communication over wireless networks. In this way, the MASE takes up the challenges of mobile multimedia and prepares for UMTS.

## *The CityGuide Application Scenario*

The CityGuide is part of a set of mobile-aware applications (Fig. 2). It is a typical mobile user application providing access to a map of the surroundings of the mobile user. This map displays several information layers, such as hotels, restaurants, automated teller machines, bus stops, and phone booths. These information sources are linked to Web pages to allow instant access to further information about a particular location.

The CityGuide runs as a Java applet within a Web browser and provides access to maps describing the current surroundings. This application uses the *mobile* API to ask the MASE Location Manager for the actual coordinates (longitude and latitude) of the user. This information will be checked against the coordinates stored in the server. The most suitable map and the associated geographical objects will be loaded using HTTP. Since the browser is a legacy application and normal HTTP is not well suited to mobile communication, the MASE integrates these calls using an HTTP proxy system. In the following we describe the MASE components participating in this process.

## *Location Manager*

The Location Manager (LM) helps users navigate in new environments. It enables applications and other MASE components to determine the parameters of the current geographical position of a mobile device as well as the accuracy of these values. This is another example of the abstractions provided by MASE because the geographical information is accessible through a simple uniform API and independent of the mechanism used by the LM to gather location data. A subset of the LM API calls is shown in Table 1.

The current LM implementation supports the Global Positioning System (GPS), a satellite-based radio navigation system developed and operated by the U.S. Department of Defense [6], and uses WaveLAN (a wireless LAN device from Lucent Technology [7]) cell identifier information. GPS provides latitude and longitude coordinates, velocity, and the user's moving direction with an accuracy of about 10–300 m.

| Method | Description |
|---|---|
| getAccuracy | Returns the position error of the device |
| getLastDateOfUpdate | Returns information about last date of update of GPS location information |
| getLastTimeOfUpdate | Returns information about last time of update of GPS location information |
| getLatitude | Returns the latitude in WGS84 format |
| getLongitude | Returns the longitude in WGS84 format |
| getVelocity | Returns the speed of the device |

■ **Table 1.** *A subset of the Location Manager API.*

APPENDIX Z

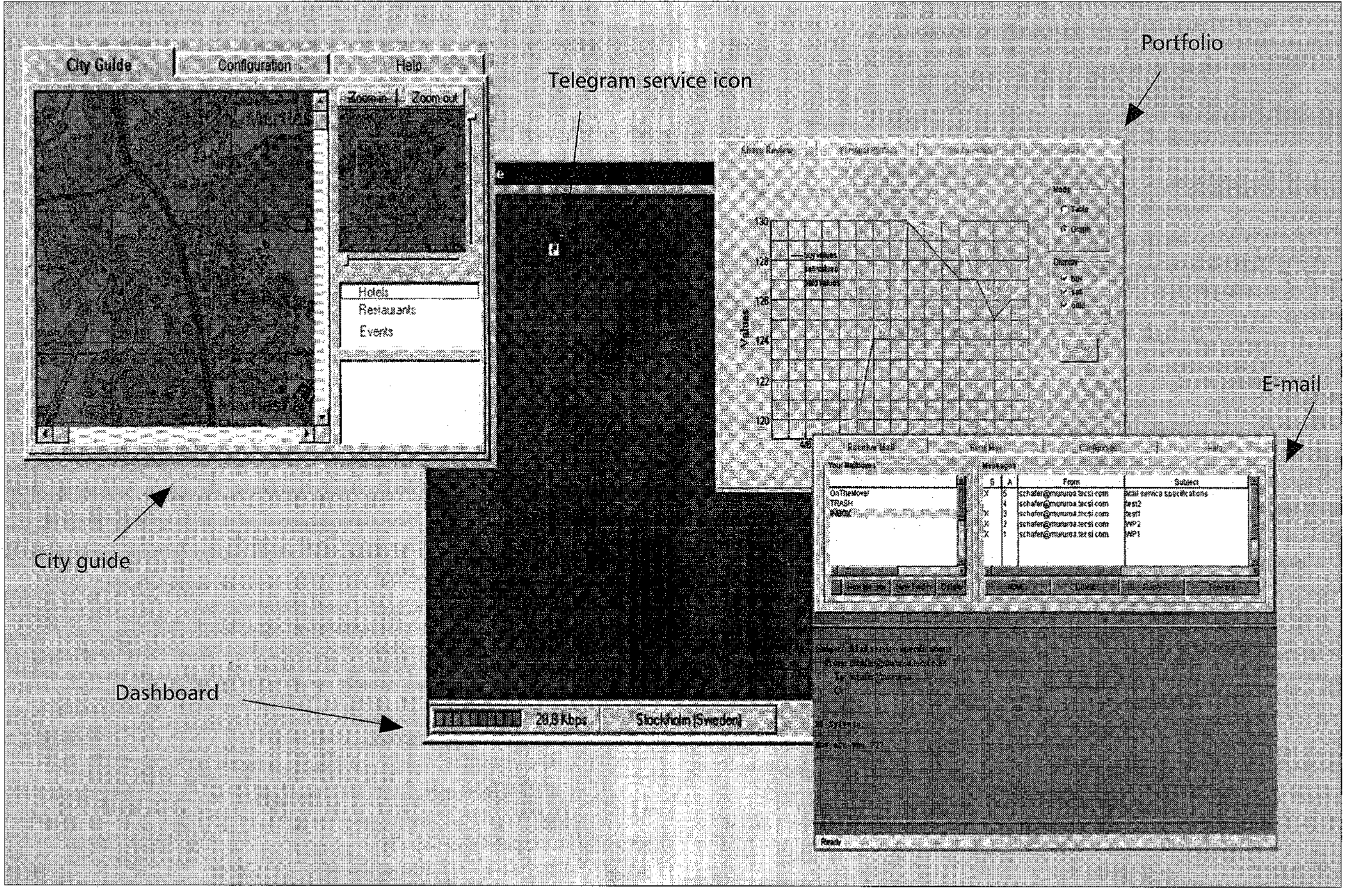


**Figure 2.** *OnTheMove desktop and mobile-aware applications.*

In wireless communication systems cell identifiers can help determine the position of the mobile device. The accuracy depends on the network cell size and varies from about 30 m (wireless LAN) up to a few kilometers using GSM. A table lookup maps these cell identifiers to the physical position of the terminal. A central LM process communicates with the available location information sources. The retrieved values are made available to MASE-aware applications and other MASE components by stub interfaces to the central process. This architecture is shown in Fig. 3.

## *Communication Manager*

The Communication Manager (CM) of the MASE architecture supports HTTP and e-mail communication, HTTP prefetching, alerting, messaging, and disconnected operations. In the following we will focus on the HTTP proxy. The HTTP proxy system improves the performance and usability of HTTP over low-bandwidth network connections (cellular, radio LAN, modem). Requested multimedia objects can be processed/converted by MASE to match the current network bandwidth, terminal characteristics (display capability, storage capacity, etc.), and user requirements. This adaptation is performed at the mobility gateway before transmitting the data to the mobile terminal. The distribution of the HTTP proxy functionality is shown in Fig. 4.

Multiple users simultaneously access Web-based information using the HTTP protocol. Each mobile terminal runs a client-side HTTP proxy (CSP) that requests Web objects from the server-side HTTP proxy (SSP). SSP can serve multiple client connections in parallel, with each connection being handled by a separate connection handler thread. Each such handler communicates with a peer process on the CSP over a multiplexed logical communication session. Multiple sessions run on top of a single TCP/IP connection between a mobile terminal and the mobility gateway. The multiplexing scheme allows data transfer for all logical sessions from one terminal to be transmitted in parallel over a single full-duplex TCP/IP connection.

In a typical scenario, an HTTP client on the mobile terminal (such as the CityGuide application) will request HTML pages and related graphic objects from a remote HTTP server. The HTTP Proxy system intercepts these calls and communicates with the SAM to adapt the requested objects to the current QoS and user requirements. The application (in this case the browser) is configured to use the CSP as an HTTP proxy server.

As shown in Fig. 4, for each HTTP request the client will

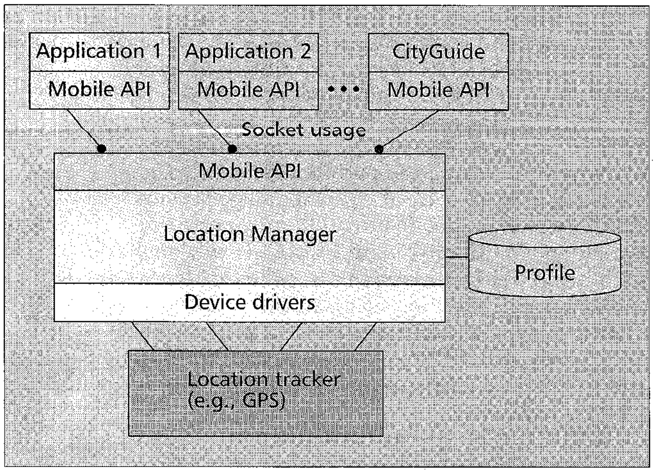


**Figure 3.** *Location Manager architecture.*

establish a TCP/IP connection to the CSP ①. The CSP checks whether the requested object is available locally by calling the `checkCache()` methods of the SAM ②. See Table 2 for the description of the SAM API. If the SAM indicates local availability, the object will be returned by the CSP to the requesting client.

Otherwise, the CSP will create a logical communication session to the SSP side and will pass the HTTP request to the SSP ③. Another `checkCache()` call ④ checks whether the requested object is available on the Mobility Gateway. If not, the object is fetched from the HTTP server ⑤ and inserted into the local cache using the `createMMO()` method. The SSP now calls the `trade()` method to adapt the multimedia object. This trading process is described in the next section. The SAM creates a variant of the original object and inserts it into the family of related objects managed by the local cache. The variant together with instructions for the post trading phase (e.g., which decompression method to use) is returned and transferred to the CSP. Here a post trading phase is initiated and the resulting object passed to the requesting client.

| Method | Description |
|---|---|
| trade() | Performs trading |
| postTrade() | Performs required processing steps on the receiver side (e.g., decompression) |
| checkCache() | Searches cache for a local available MMObject |
| createMMO() | Creates an initial MMObject |
| insertVariant() | Inserts an MMObject variant into the family of related objects |
| removeVariant() | Removes variant from the family |

■ **Table 2.** *A subset of the QoS trader API.*

| | |
|---|---|
| user.image.reductionAllowed<br>user.image.maxTransmissionSize<br>user.image.maxWaitTime<br>user.image.resolutionReduction | True<br>20,000 bytes<br>5 s<br>Yes |
| system.terminal.display<br>system.terminal.memory<br>system.terminal.diskSize | Black and white<br>2 MB<br>20 MB |
| network.currentBearer<br>network.currentBearer.expThroughput | GSM<br>9600 b/s |

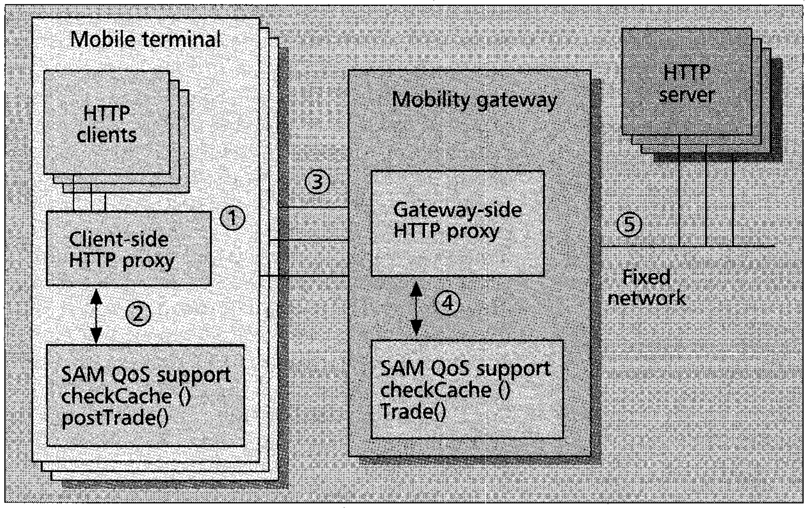
■ **Table 3.** *Profile values.*

## *System Adaptability Manager*

The System Adaptability Manager (SAM) is responsible for the provision of optimized and personalized mobile services. User-, network-, and terminal-specific QoS parameters are managed in profiles handled by the Profile Manager. These profiles are used to compute the best adaptation of the MASE communication services, taking into account the current network and end system QoS parameters as well as to the user's personal preferences. The SAM has several adaptation possibilities:

- It can make a choice between different available networks based on the available bandwidth, bandwidth guarantees, and cost.
- It can compress, convert, transcode, or reduce the multimedia objects prior to transmission.

For example, an image is supposed to be transmitted to a terminal with a black and white screen. In this case color information can be eliminated by the mobility gateway. Other reasons for adaptation can arise, for example, from the available network bandwidth and the cost involved. These decisions are made by the QoS trader within the SAM.



■ **Figure 4.** *The partitioning of the HTTP proxy.*

### *Profile Manager*

The terminal characteristics of the mobile device are stored in a *terminal profile*. The network characteristics of the mobile device's current (wireless) connections are stored in the *network profile*. The network profile is constantly updated by the UAL. User-specific preferences are stored in a *user profile*. A profile generally contains a hierarchical tree of properties (name/value pairs), each describing a certain property. Profiles are replicated on demand and stored persistently throughout the MASE system.

A MASE-aware application can access the profile values at either the mobile device or the mobility gateway. Consistency can be enforced independently at every host and for each node to reduce the communication overhead. Table 3 shows examples of profile values for the user, system and network tree.

### *QoS Trader*

The QoS trader adapts the MASE communication services according to the user's preferences and the current terminal/network situation as reflected in the profile. Instead of transmitting multimedia objects directly to the mobile client, the HTTP proxy calls the QoS Trader to perform a trading process. This process consists of the steps illustrated in Fig. 5.

During the *QoS gathering* phase the QoS trader accesses the current terminal and network QoS which are stored in the profile. The profile also contains the user's preferences, which are later used to compute filters and preferences for the planning process. It further examines the properties of the current multimedia object, and then enters the *planning* phase, during which it decides about the actions to perform on the current multimedia object. This process generates a "new" plan from a knowledge base (*plan generation*) and the gathered QoS parameters.

A plan consists of one or more compression, conversion or reduction steps. During the *plan* *filtering* phase the static properties of the new plan are matched against the user preferences and terminal requirements. Small devices, for instance, might have implemented only one or two decompression methods. The plan filtering phase will only select plans suitable for this particular device.

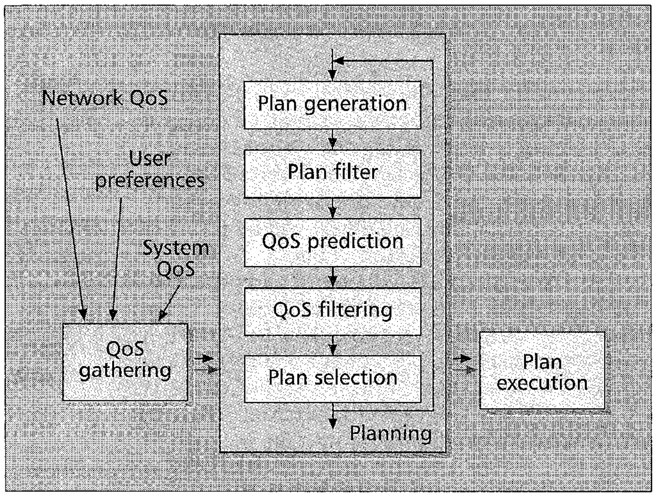
Subsequently, the QoS trader predicts the prospective outcome of the current plan using knowledge provided by the multimedia conversion (MMC) routines. MMC works *online*. It offers lookup functions that enable it to estimate conversion time, and execute conversions if appropriate. MMC provides some general-purpose methods for image manipulation, such as conversion of images to other image formats by means of scaling, graining, and color reductions.

Two methods that support the QoS prediction and filtering phases of the QoS trader, by estimating the required conversion time and the size of the reduced objects, are important for the SAM, which checks whether varying the reduction $R$ of an object $x$ can fulfill Eq. 1.

$$T_{MaxWait} > T_{Seek}(x_{org}) + T_{Reduce}(x_{org},R) + T_{Trans}(x_{red},B) \quad (1)$$

$T_{Trans}(x,B)$ is the transmission time of an object $x$ at bandwidth $B$ (b/s), $T_{Reduce}(x,R)$ is the estimated reduction time of the original object $x$ at reduction $R$. $T_{Seek}(x)$ is the time used to estimate the reduction time of an object $x$ to the requested file size (planning process). $T_{MaxWait}$ (stored in the profile under `user.image.maxWaitTime`) is a user preference parameter which defines the maximal time the mobile client wants to wait for an image. If $T_{MaxWait} > T_{Trans}(x_{org}, B)$, there is no need to reduce the file size, and the SAM transmits the original image.

A set of suitable conversion commands have been selected for our purpose. For images we use scaling, reducing colors, dithering, and converting to black and white to reduce image sizes. Some formats like JPEG allow conversion by scaling and reducing the overall quality. All those commands are "lossy"; they reduce the quality of an image and hence reduce the file size. To obtain a table of relative values for conversion predictions we have measured a set of images with all available conversion commands.



■ **Figure 5.** *The QoS trading process.*

Using this knowledge the trader can estimate the QoS parameters resulting from converting and transmitting the converted multimedia object (e.g., the overall processing time). These predicted QoS parameters are matched against the filters derived from the profile setting. During the *plan selection* phase a preference index is computed for each plan which has passed the filters. The used preference function is selected according to a user-defined criterion (e.g., the smallest resulting object size, the shortest time required for converting and transmitting the result, the best quality remaining).

When the planning is finished, the best plan with the highest preference index is executed. The resulting object is either returned to the application (local trading) or stored locally and will be marshaled for transmission.

| Method | Description |
|---|---|
| compress | Compresses a file |
| decompress | Decompresses a file |
| estimateReductionTime | Returns estimated time to perform a desired reduction |
| reduceFileSize | Reduces file size by the desired reduction |
| convert | Converts an image into a given type |
| colorRedQuant | Reduces the number/depth of colors |
| colorRedDepth | Reduces the depth of color |
| colorRedDither | Reduces the number of shades per color |
| getHeigth | Returns the height of an image |
| getMaxVal | Returns the max value of shades per color |
| getSize | Returns the file size in bytes |
| getType | Returns the image type as String |
| getWidth | Returns the width of the image |
| isCompressed | Returns true if the image is compressed |
| isImage | Returns true if the instance is an image |

■ **Table 4.** *Subset of multimedia object conversion API.*

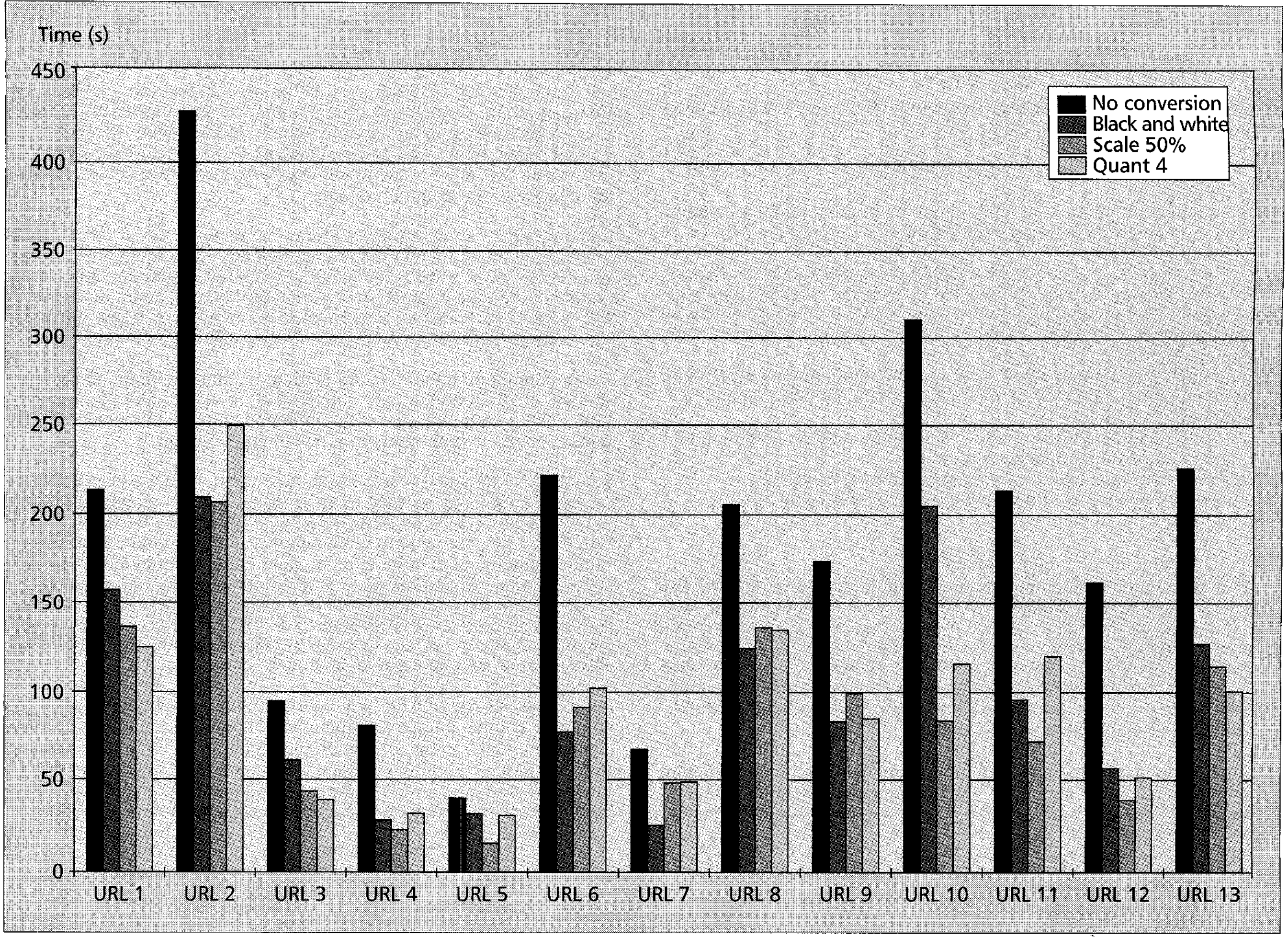
***Example*** – The *plan generation* phase generated the plan "SCF(0.5); TRA; SCF(2)," which defines a scaling operation by the factor of 0.5, the transfer of the image, and the rescaling of the image to the original size during the `postTrading()` call on the mobile terminal. During the *plan filtering* phase this plan is compared with user preferences stored in the profile. For example, the parameter user.image.resolutionReduction defines whether or not the user accepts resolution reductions (scaling). If the user does not allow scaling, the filter derived from that value will prevent the above plan to be selected.

During the *QoS prediction* phase, the time required to execute the above plan will be computed by predicting the time required for the conversion SCF (0.5), for the transmission of the reduced image, and for the reconversion (SCF (2)) on the mobile device. The time and resulting image size for both conversions are computed by using average results derived from former conversions which were initially performed offline. Furthermore, the computing power of the mobile device is taken into account by using relative computing indices from the terminal profile for this device. In addition, the time to transfer the original image and a user-specific quality factor are computed.

During the *QoS filtering* phase, these results are used in Eq. 1 to check whether the user's

Microsoft et al. Exhibit 1005

APPENDIX Z



■ **Figure 6.** *Measurements of a MASE-supported browser over GSM.*

requirement maxTransmissionTime is fulfilled and whether the reduction will result in a faster transmission. If the plan is still valid, a preference index will be computed. Several preference functions are possible, ranging from a selection based purely on the processing time to a mixture of processing time and the quality of the reduced image. Finally, the plan with the best preference index will be executed.

Figure 6 shows the achieved measurements over the actual cellular system, GSM, that has data transport capabilities of up to 9600 b/s. The results show the advantage of the QoS trading and multimedia conversion. Depending on the automatically chosen conversion method the interaction of CM, SAM, and UAL achieve up to 70 percent acceleration of the HTTP transmission time. Table 4 represents a small subset of the SAM API that application programmers may use to realize conversions offline, or even during online connections.

## Future Work

Future work will deal with the downsizing of the current architecture and implementation to fit the requirements of small mobile devices (e.g., PDAs and PICs), which usually have only limited memory and CPU power. The implementation of the Location Manager will be extended by supporting more input devices, such as an interface to a DECT system which holds information concerning the actual interworking unit. Further development will be done on a Session Manager that provides a higher abstraction of connections than TCP/IP does. The Session Manager controls the ongoing communication sessions and allows scheduling of different MASE operations like prefetching and caching. Results of the project's field trials will be used to improve the current status.

## Conclusions

The data services of future third-generation mobile telecommunication systems play a critical role in facilitating new and innovative mobile-aware applications and services. UMTS will define a set of services enabling seamless roaming between different networks, QoS monitoring, and bandwidth on demand. In this article we have introduced the OnTheMove approach, which employs middleware to support the special needs of mobile-aware applications. This middleware not only allows the development of mobile-aware applications in an easy way, it also shields today's application developer from the ongoing developments toward UMTS. The MASE mobile middleware will pave the road to future mobile telecommunication systems.

### Acknowledgments

The authors would like to acknowledge all contributing partners of the OnTheMove project [8] which is sponsored partially by the European Commission in the ACTS program under contract AC034. The project participants are Ericsson Radio Systems AB, T-Mobil GmbH, Ericsson Eurolab GmbH, Siemens AG, Rheinisch-Westfälische Technische Hochschule Aachen (RWTH), IBM France, Tecsi, BT, Bonnier Business Press, Royal Institute of Technology (KTH), Sony International (Europe) GmbH, Burda Com Media Solutions GmbH, Centre for Wireless Communications (CWC), and Iona.

Microsoft et al. Exhibit 1005

APPENDIX Z

## References

[1] EC DG XIII/B (1/3/96), "UMTS Task Force Final Report," Brussels,Belgium, ACTS InfoWin, http://www.infowin.org/ACTS/.

[2] J. Schwarz da Silva *et al.*, "Evolution Towards UMTS," ACTS InfoWin,http://www.infowin.org/ACTS/IENM/CONCERTATION/MOBILITY/umts0.htm.

[3] M. H. Callendar, Ed., *IEEE Pers. Commun.*, Special Issue on International Mobile Telecommunications-2000: Standards Efforts of the ITU, vol. 4, no. 4, Aug. 1997.

[4] J. Meggers and A. S. Park, "Mobile Middleware: Additional Functionality to Cover Wireless Terminals," *Proc. 3rd Int'l. Wkshp. Mobile Multimedia Commun.* (MoMuC-3), Princeton, NJ, Sept. 1996; D. J. Goodman and D. Raychaudhuri, Eds., *Mobile Multimedia Communications*, Plenum, 1997, pp. 151–57.

[5] J. Meggers, A. S. Park, and R. Ludwig, "Roaming between GSM and Wireless LAN," *ACTS Mobile Commun. Summit*, Granada, Spain, Nov. 1996, pp. 828–34.

[6] U.S. Coast Guard Navigation center, "GPS, DGPS, LORAN, OMEGA, LNM," http://www.navcen.uscg.mil/gps/gps.htm.

[7] WaveLAN Wireless Computing, http://www.wavelan.com.

[8] OnTheMove home page, http://www.sics.se/~onthemove.

## Biographies

BIRGIT KRELLER (birgit.kreller@mchp.siemens.de) received her Dipl.-Inform. (Master's in computer science) from the University of Magdeburg, Germany, in 1996 on the subject of mobile agents for load balancing in large telecommunication systems. She joined the Siemens Corporate Research and Development Department in March 1996, where she is working in the ACTS project OnTheMove. Her current research interests include mobile computing architectures, wireless networks, geographical positioning systems for mobile devices, and mobile agents.

ANTHONY SANG-BUM PARK received his Dipl.-Inform. from Aachen University of Technology (RWTH), Germany, in 1995, previously studying at the University of Koblenz. From 1989 to 1992 he was with Philips Communication Industry AG in quality control, and with Parsytec Computer GmbH focusing on massive parallel computing. Since 1995 he has been a researcher and Ph.D. candidate at RWTH, Department of Computer Science, responsible for agent technology research projects and working in ACTS projects. Research topics are mobile computing and personal multimedia communications. Activities concerning distributed systems and middleware architectures are mainly in the area of mobile agent technology.

JENS MEGGERS received his Dipl.-Inform. in computer science from RWTH, Germany, in 1995. He joined the Department of Computer Science of the same university as a Ph.D. candidate in computer science in 1995. He participates in various internal and external research activities, including the ACTS project OnTheMove. His main research focus lies in QoS supporting network and transport protocols for mobile multimedia communications.

GUNNAR FORSGREN received his B.S.E.E degree from Härnösand Gymnasium, Sweden, in 1978 and has been with Ericsson in various R&D positions since 1980. His research interests include information/communication services on wireless devices and their interaction with agent services.

ERNÖ KOVACS received his Dipl.-Inform.from the University of Kaiserslautern, Germany in 1991. During 1986–1990 he worked at IBM's European Networking Centre (ENC) in various research projects concerning multimedia e-mail, multimedia documents, and distributed hypermedia systems. From 1991 to 1996 he worked at the Institute of Parallel and Distributed High-Performance Systems (IPVR) of the University of Stuttgart. He conducted several projects in the area of middleware for distributed systems. In 1997 he joined Sony's Research and Development Department in Stuttgart and worked in the ACTS project OnTheMove. His current research interests include mobile multimedia, quality-of-service trading, and mobile agent systems.

MICHAEL ROSINUS received his Dipl.-Inform. at the Universität des Saarlandes, Germany, in 1996 and worked at the German Research Center for Artificial Intelligence (DFKI) in the area of intelligent agents. In May 1996 he joined the Sony Research and Development Department where he is working in the OnTheMove project. His current research interests include mobile and intelligent agents, multimedia data processing, and wireless networks.

Microsoft et al. Exhibit 1005

APPENDIX AA

# Real-time synthetic vision cockpit display for general aviation

Andrew J. Hansen, W. Garth Smith, and Richard M. Rybacki

MetaVR, Inc.
http://www.metavr.com[1]

## ABSTRACT

Low cost, high performance graphics solutions based on PC hardware platforms are now capable of rendering synthetic vision of a pilot's out-the-window view during all phases of flight. When coupled to a GPS navigation payload the virtual image can be fully correlated to the physical world. In particular, differential GPS services such as the Wide Area Augmentation System WAAS will provide all aviation users with highly accurate 3D navigation. As well, short baseline GPS attitude systems are becoming a viable and inexpensive solution. A glass cockpit display rendering geographically specific imagery draped terrain in real-time can be coupled with high accuracy (7m 95% positioning, sub degree pointing), high integrity (99.99999% position error bound) differential GPS navigation/attitude solutions to provide both situational awareness and 3D guidance to (auto) pilots throughout en route, terminal area, and precision approach phases of flight.

This paper describes the technical issues addressed when coupling GPS and glass cockpit displays including the navigation/display interface, real-time 60Hz rendering of terrain with multiple levels of detail under demand paging, and construction of verified terrain databases draped with geographically specific satellite imagery. Further, on-board recordings of the navigation solution and the cockpit display provide a replay facility for post-flight simulation based on live landings as well as synchronized multiple display channels with different views from the same flight. PC-based solutions which integrate GPS navigation and attitude determination with 3D visualization provide the aviation community, and general aviation in particular, with low cost high performance guidance and situational awareness in all phases of flight.

**Keywords:** situational awareness, real-time visualization, correlated terrain databases, geographically specific satellite imagery

## 1. INTRODUCTION

A remarkable transition in state-of-the-art image generation is taking place as single purpose, specialized rendering hardware is being replaced with off the shelf components driven by PC-based processors. The dramatic performance improvements realized by the PC graphics industry in the last two years has leveraged the broad base of innovation across the industry. The rich mix of focused development efforts in chip design, bus architecture, software driver standards, and processor technology feeds the continuous improvement in PC graphics capability that is reaching the upper echelons of visualization-simulation (VizSim) performance standards. This paper focuses on the underlying capabilities needed to render the virtual environment in a mobile platform such as an aircraft. Primarily these are the image generator hardware and software implementation and the generation of a three-dimensional database of the environment including terrain, aircraft, and cultural features. It does not address symbology and information content that should be displayed and interested readers are referred to [1,2].

These low cost, high performance graphics solutions based on PC hardware platforms are now capable of rendering both moving map displays and synthetic out-the-window views of a moving vehicle with an extremely high degree of realism. By coupling with a GPS navigation/attitude payload the virtual image can be fully correlated to the physical world in real time. In particular, differential GPS services such as the FAA's Wide Area Augmentation System (WAAS) [4] will provide users with highly accurate 3D navigation information. The WAAS position solution is specified to have accuracy better than 7.5m 95% and a guaranteed (99.99999%) confidence interval [5]. Prototype implementations of WAAS are achieving nominal accuracy of about 1-2m 1-sigma in all three dimensions [6]. Carrier phase GPS based attitude heading references system (AHRS) prototypes are also being implemented [7,8,9] which can provide sub degree accuracy in all three axes, roll/pitch/yaw. Integration of accurate position/velocity/attitude state information and a highly capable rendering engine enables synthetic image generation of the physical scene.

[1] Correspondence: Email: {ahansen,wgsmith,rmrybacki}@metavr.com; Telephone: (617)739-2667

Part of the SPIE Conference on Enhanced and Synthetic Vision 1999
Orlando, Florida • April 1999 SPIE Vol. 3691 • 0277-786X/99/$10.00

The underlying resource that ties these two pieces together is an accurate and reliable geographic database that describes the physical environment. It must be accessed efficiently to serve the real time application but also have the fidelity to enhance rather than detract from the pilot's situational awareness. Our solution incorporates geographically specific satellite imagery and cultural features into an efficient terrain database. The satellite imagery provides contextual information for situational awareness while specific cultural features such as runways can be inserted as additional objects with higher resolution and special properties. For run time flexibility, the resulting database can be stored in memory or demand paged off of a storage device using a "look ahead" algorithm.. Real time performance for extremely high-resolution terrain (100m post) and imagery (5m) is supported using hierarchical level of detail switching. In addition the render engine has variable field of view and far horizon clipping plane parameters so that the necessary display refresh rates can be maintained. The solution we describe below borrows heavily from the visual simulation concepts developed in distributed interactive simulation (DIS) research with the new twist that high fidelity high performance can be achieved on PC platforms. The economies of scale in this arena provide low cost systems and the impetus to move toward embedded solutions. While all of the features currently supported by 3D VizSim applications may not be appropriate in the cockpit, we identify them here as candidates for use and leave their designation to the community at large.

The remainder of this paper first touches briefly on the linkage of position/velocity/attitude state information and the virtual environment in the computer. We then focus on the visualization hardware and the innovations in the graphics industry which now provide the power to render one or more 3D out-the-window scenes or 2D top down moving maps (so called plan view displays). The next section focuses on our database construction process, database storage requirements, and its correlation to truth. We close with some comments on the opportunity to extend the cockpit display to a networked solution where, given a low bandwidth communication channel, information from multiple entities could be included in the display. In this mode the display could provide additional situational awareness vis-a-vis TCAS I/II systems that have a cockpit display of traffic information (CDTI).

## 2. LINKING AIRCRAFT STATE TO VIRTUAL ENVIRONMENT

### 2.1 Navigation: Position, Velocity, and Attitude

In order to place the virtual aircraft at the appropriate position and orientation in the virtual environment, sensor system outputs of the aircraft's position, velocity, and attitude must be available to the graphical render engine in real time. Differential GPS navigation and attitude determination is a low cost option for obtaining these states in the aircraft. The geographical extents covered in aviation applications are well served by wide area differential GPS (WADGPS) systems for real time positioning. Likewise the global coverage of GPS allows a user with multiple antennas to compute an attitude solution at any position within aviation capability.

The FAA is specifically developing the Wide Area Augmentation System (WAAS) for seamless, high integrity navigation in all phases of flight. Successful prototype signal-in-space flight tests have already been implemented and carried out by the FAA Technical Center with Stanford Telecommunications [13] and Stanford University [5]. The WAAS uses a geosynchronous satellite broadcast channel for continental scale coverage and high data link availability. In cooperation with the FAA and industrial representatives, RTCA, Inc. has written the WAAS Minimum Operational Performance Standards [12] (WAAS MOPS) to specify the WAAS signal structure and the application of the differential corrections to stand alone GPS measurements. The WAAS navigation payload includes a GPS receiver capable of receiving an additional 250 bps WAAS data stream from a geosynchronous satellite. The WAAS message stream is unpacked to form differential corrections for satellite clock and ephemeris errors as well as a differential ionospheric correction. These corrections are then applied to the standard GPS measurements for each satellite in view. The differentially corrected signals form the basis for the navigation solution and its associated confidence interval. This navigation solution, which contains both position and velocity, is fed directly to the image generator in the form of WGS84 coordinates.

Synthetic vision applications are very sensitive to errors in attitude determination because the entire field of view is controlled by the orientation of the viewpoint. Low cost AHRS based on carrier phase GPS are now incorporating rate or inertial aiding [7,8] to provide the accuracy and noise performance necessary to drive cockpit displays. Strapdown AHRSs are also shrinking the antenna baselines needed to achieve sub degree accuracy in all three directions [9]. The resulting attitude solution in body coordinates can be input directly to the image generator. Adequate systems require an update rate on navigation and attitude of at least 10 Hz [1] in order to reach suspension of disbelief for the operator. Of course the faster the better, but in any case if the sensor inputs do not update at the frame rate of the display system a model of the aircraft is propagated forward to update the synthetic vision viewpoint to maintain the 60 Hz visual update rate.

Microsoft et al. Exhibit 1005

In flight, updating the aircraft model which must reside in the same coordinate system as the terrain database requires a transformation of the navigation solution from WGS84 coordinates to the local coordinates of the database. This does require some additional but necessary computation. The virtual environment database needs sufficient fidelity to fully correlate the sensor states with the terrain in the rendered image. Spheroidal coordinate systems such as WGS84 cannot provide that level of fidelity and the database must use local coordinate system based on a geoid.

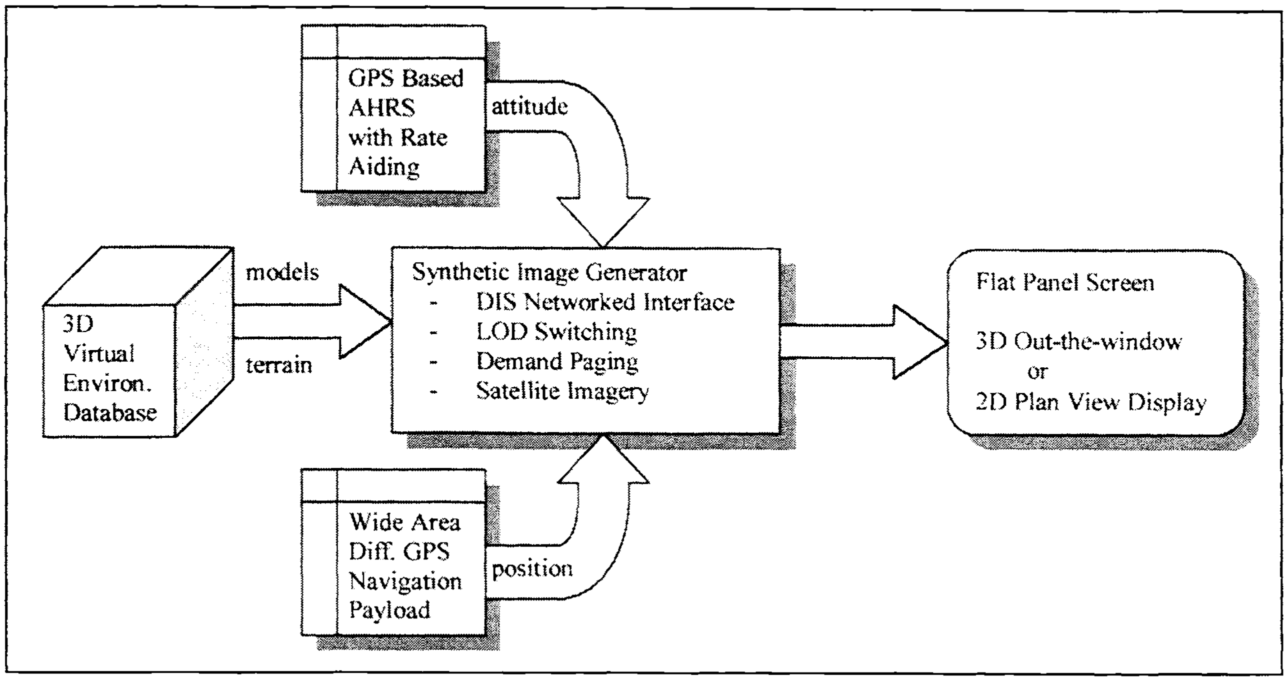


**Figure 1. The image generator ingests the 3D-database and real time data from position/attitude sensors to determine the viewpoint for the synthetic scene and pushed onto graphics card for rendering.**

### 2.1 Distributed Interactive Simulation Protocol

As briefly mentioned above the sensor states may need to be propagated forward some number of epochs as the render engine my update faster than the navigation and attitude updates are available. Our implementation abstracts the input linkage from the sensor systems to the render engine. We utilize the IEEE standardized [14] Distributed Interactive Simulation (DIS) protocol for inserting new sensor updates into the virtual environment. This DIS protocol exists at the application layer of the communications stack. It is built upon User Datagram Protocol (UDP) packets called Protocol Data Units (PDUs). These PDUs are well defined in the DIS standard and include necessary elements such as kinematic model parameters as well as graphical information in the form of texture and polygonal models.

One added benefit to the DIS approach is that multiple views from the same entity can be added simply by plugging in another render engine. Another, and we believe more powerful benefit, is that multiple entities can appear in the same virtual environment exactly as they do in the physical environment. An entire suite of functionality including multicasting, loss tolerance, forward state prediction, and communication protocol is already defined and implemented by the VizSim community. DIS is abstracted from the physical layer so that the network could be a high speed wired intranet or just as easily a low bandwidth wireless LAN so far as the application is concerned. In fact, DIS is expressly designed for a heterogeneous network where some paths have much greater bandwidth than others. This flexibility has direct benefits for future applications that include multiple entities (other air traffic).

An additional feature of the DIS network solution is the ability to log PDU packets being transmitted. By logging the state information in PDU form, the entire flight can be captured for playback. This is particularly useful for experimental or testing purposes in early development of operational systems to replay and view from any vantage point using a six-degree-of-freedom pointing device.

## 3. IMAGE GENERATION

The image generator is the core of the cockpit display. As shown in Figure 1 it ingests the terrain database and aircraft models and accepts input from the navigation and attitude payloads. One or more graphics engines slaved to the image generator can then render images to the screen with one viewpoint for each engine. The baseline mode is a single 3D out-the-window view out the front of the aircraft showing the terrain and cultural features in the environment. Adding an additional graphics engine or switching modes to the plan view display provides the pilot with a top down moving map. This approach is very appealing as it can immediately utilize advances in hardware performance offered by the industry as it continues to improve.

The image generator is currently hosted on a PC platform. It requires a graphics card that supports the DirectX API and can utilize either the PCI or AGP bus as the graphics pipeline. A host platform consisting of a 450MHz Pentium II processor with 512Mb of RAM and a Canopus Spectra 2500 with 16Mb of VRAM consistently maintains 60Hz frame rates for fields of view covering a 50 km radius at velocities up to Mach 3. Note that these frame rates are significantly higher than the update rates that thé navigation/attitude sensors support. An important consideration in the software development of the image generator was the graceful degradation in performance as either the host platform is scaled back or the fidelity of the database is scaled up. We have already implemented laptop PCs rendering 3D out-the-window views of the virtual world.

DirectX and OpenGL capable graphics cards are designed to render polygonal shapes as triangular patches in hardware. Textures may also be stored in memory and applied to these polygons as part of the hardware processing of the render engine. The image generator is responsible for pushing the textures up into video memory and then pipelining the polygonal shapes from the terrain database up to the graphics card using in our case either the PCI or AGP bus under the DirectX protocol. As such there is a balance that needs to be struck to ensure that the central processor and the graphics chip set are reasonably well matched in performance. The CPU must index and arrange the terrain polygons based on the current aircraft state and the graphics card is responsible for rendering the textured polygons.

The importance of the level of detail (LOD) switching and demand paging now becomes clear. LOD switching aids in balancing the load between the CPU and graphics card. If the current field of view has too many textured polygons for the graphics card to handle then the CPU can switch some of the far field regions to lower resolution and thereby reduce the number of polygons being rendered. This LOD switching is an improvement over the simplest form of switching which is the insertion of a clipping plane that limits the field of view. To accomplish LOD switching the image generator and database must be intimately coupled as not only does the polygon resolution switch but also the textures applied to them. For platforms that have memory limitations the image generator can invoke demand paging of regions of the database that are coming into the field of view. Knowing velocity states allows the image generator to look ahead in the database to see if upcoming regions are loaded into memory and ingest them in the event that they are not.

The core process in the image generator is to continuously update the viewpoint of the virtual aircraft at each epoch. Under the DIS paradigm the aircraft state is propagated forward from the last PDU update. In the host platform this is nominally never longer than 20 msec. The local region of the terrain database which is stored in memory is then interrogated to assemble the textured polygons for pipelining up to the graphics renderer. If other entities besides the host aircraft are in view they are also updated and their virtual representation is pushed up the graphics pipeline.

There are many other features available from VizSim applications that are probably not useful in the cockpit such as variable visible spectrum, DirectSound output, atmospheric emulation of fog and clouds, and six degree of freedom input device compatibility. However, the existence of these features demonstrates the head room available in this implementation which can be converted into other more pertinent features such as situational awareness symbology, tunnel-in-the-sky guidance [2], and traffic information.

Neglecting the navigation and attitude platforms, the hardware necessary for the image generator is currently on the order of $4000. Once available, projections for WAAS and AHRS system prices are in the ones of thousands of dollars. This places hardware costs for first generation integrated cockpit displays at around $10-15k plus installation. Our expectation on cost trends is that they would follow the precedence set in the rest of the VizSim market: continual improvement in the price/performance point at market. There are also certification and recurring costs associated with any avionics systems which we do not have sufficient experience to comment on with the exception of generating and updating the 3D databases.

Because of the proliferation of the underlying resources (elevation data and satellite imagery) and competing utilities for database construction we also expect the database generation costs to decline. The ultimate goal for a cockpit display is the integration of the attitude/navigation/renderer into a single embedded system which could drive a flat panel display

## 4. TERRAIN AND IMAGERY RESOURCES

Terrain information is typically available in raw form as digital elevation maps (DEMs) or digital terrain elevation data (DTED) at various levels of resolution. In order to create a terrain database that can be rendered on a computer this information must be converted into polygonal surfaces that represent the surface of the terrain. These polygonal elements are well suited to image generation as the industry has optimized graphics chip sets to handle them in hardware. This optimized hardware is now readily available on PC platforms. One of the most important considerations in this conversion is the need for extreme efficiency in both the size and accessibility of the resulting database so that the image generator can ingest and render the virtual scene. For even medium fidelity terrain information, say 125m, post, and a reasonable coverage area for aviation, say 5° x 5° cells, the raw data for elevation information alone can run into the hundreds of megabytes in size. We defer the discussion of our conversion process and the resulting database to the next section.

Another important part of generating a convincing image for the user is the texture overlay on the terrain surface. Our approach is to apply geographically specific satellite imagery on the terrain polygons. By overlaying real imagery that is coordinated directly to the terrain data, the scene that is eventually rendered by the visualization engine has a very high degree of realism. The sources for satellite imagery are increasing rapidly and we anticipate that the vast majority if not all of the national air space (NAS) will be covered and in fact frequently and continuously renewed with world wide coverage soon to follow. Even at this time custom imagery for any particular location can be ordered directly off of the World Wide Web from commercial vendors.

Our secondary approach to overlays follows the standard approach of texture mapping each surface. Here synthetic textures are created and used in place of the satellite imagery where it is not available. In either case, real imagery or synthetic textures, rendering of terrain polygons is treated exactly the same by the render engine as the database is responsible for arbitrating the virtual world including the overlays.

There are several sources of elevation maps in digital form. NIMA outputs Digital Terrain Elevation Data (DTED) at various levels of resolution, typically only the lowest level is openly available. The USGS supplies elevation data in the form of Digital Elevation Maps (DEMs) which can be purchased on the web. ERDAS Imagine, CTDB, and other formats commonly used for GIS applications are also available commercially. We utilize primarily DTED, CTDB, and soon DEM data formats as the raw elevation resource for constructing the terrain database.

Satellite an aerial imagery is the other commodity we rely on for generating high fidelity databases. The commercial availability of high-resolution geographically specific imagery is growing. Individual providers are already offering photo to order imagery purchases over the web. We have already worked with products from ImageLinks in the 10-50m range. Although not openly available classified customers also have access to high resolution (1m) satellite imagery from NIMA. The important and necessary condition of the imagery is that it be geographically specific, that is ortho-rectified and pin-pointed to a reference in a standard coordinate system. This real imagery can then be draped onto the terrain surface and replace older approaches using synthetic textures.

For application specific information such as that needed for aviation, cultural features may be inserted into the database as explicit objects. An example is the runway and markings at a specific airport. These types of features can be created in a number of different formats. The industry standard is the OpenFlight format from MultiGen, Inc. but many other new and heritage formats such as VMAP, AGRD formats. The importance of the OpenFlight format is that it also supports models for dynamic entities such as aircraft. We invoke the OpenFlight format to ingest cultural features generated from graphics and modeling tools available in the industry as well as the ARDG format for cultural features on the plan view display. Although some models are already available commercially, most of the creation of cultural features and object models is carried out on a specific project basis. We see this approach eventually converging to a pool of models openly available to the community.

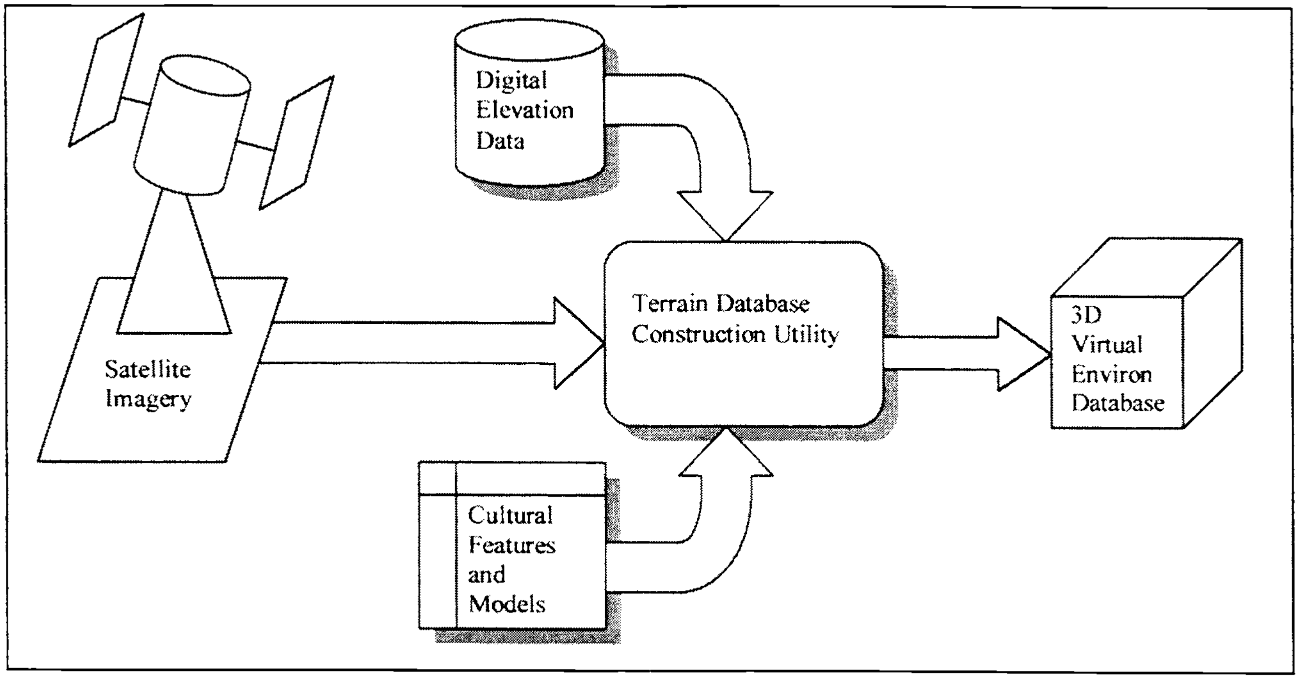

Digital Elevation Data
Satellite Imagery
Terrain Database Construction Utility
3D Virtual Environ Database
Cultural Features and Models

**Figure 2. The database construction process assembles a 3D environmental database suitable for input to the image generator from three basic resources: digital elevation data, geospecific imagery, and designated cultural features.**

## 5. DATABASE CONSTRUCTION

The elegance of our terrain database construction which uses geographically specific imagery is that it provides a real world source for constructing cultural features such as buildings, road networks, vegetation, bridges, even power grids if the image resolution is high enough. We are currently developing a palette base utility for constructing 3D terrain databases that integrate these three elements, digital elevation maps, geographically specific imagery, and automated cultural feature generation into one umbrella application. This utility will be capable of directly feeding the image generator during database design for viewing the construction on the fly. As such it will provide a suitable facility for mission planning, mission briefing, and mission rehearsal. The current database construction utility functions as a wizard type application which allows the user to enter raw data resource file names and then automatically generates the resulting database.

The three fundamental components of our database construction are the ingest of the digital elevation information, ingest of bitmap based geographically specific imagery, recognition and conversion of imagery details into cultural features (not yet available), and export to any of four database formats: MDB, MDX (both MetaVR specific), CTDB, and OpenFlight. To support the highest levels of image rendering, the MDX database format supports hierarchical levels of detail with switching controlled by range thresholds on both terrain and textures. This allows the central processor in the image generator to match the rendering capabilities of the highest end graphics cards that have 180Mpixel fill rates. The database format also allows terrain information to be loaded by the render engine incrementally using look-ahead demand paging.

The most important consideration is full correlation between entity state coordinates and the terrain database. As noted by Barrows [2] and Ourston [10], lack of correlation in some current implementations is unacceptable, particularly in exercises as the virtual scene is not convincing and detracts heavily from the qualitative performance of the simulation. MetaVR's

MDB and MDX database formats are fully correlated terrain databases that eliminate any such anomalous behavior. By using a local geoid coordinate system and transforming the WGS84 coordinate aircraft states in the image generator the entities are guaranteed to be consistent with the terrain. This of course does not mitigate errors due to the level of resolution for the terrain elevation data.

The raw elevation data is tessellated into triangular patches [11] using a Delaunay triangulation where the vertices of the triangluation are the locations of the elevation data in the local coordinate system. This is a fast routine and is computed repeatedly on sub sampled intervals to generate various levels of detail. The geographically specific imagery is then cut into patches corresponding to the levels and indexed to the appropriate triangular surfaces.

The incorporation of cultural features is currently supported on an internal format (MetaVR CLT). Development of a VMAP capable module is in process to support interoperable ingest/export of cultural features. This will provide a clear path for an imagery to cultural feature format that is sharable. We envision the downstream capability, given adequate imagery, to professionally construct and modify scenarios for training and planning.

The 3D virtual environment database also contains the models of physical entities, e.g. aircraft. These are critical for the image generator as they provide the mechanism by which the virtual scene can be propagated at very high rates (60 Hz) for rendering to the screen. At each epoch, if new state information is not available from the navigation or attitude subsystem, the states are propagated according to the properties specified in the aircraft model. This of course leads to models which are specific to the type of aircraft being simulated, e.g. Cessna 152 versus Boeing 737, in order to capture the pertinent physical properties. The inclusion of such models is particularly important if one desires to render other aircraft in the virtual scene as we describe in the next section.

The following four figures depict the underlying terrain database and its full rendering with the satellite imagery overlay. Figures 3 and 4 show a relatively flat region with the satellite imagery capturing the road network and surrounding buildings. The contextual information in the satellite imagery provides very strong situational awareness that is not available in texture mapping and extremely user intensive to design by hand in graphical models. The pair of images in Figure 5 is the wire frame and full imagery of a coastal region in Alaska, Prince William Sound, and demonstrate the LOD capability. The geographically specific satellite imagery is 25m at highest resolution. Figure 6 is a screen capture of the 2D plan view display mode of the graphical render engine. It is most useful in applications that require a moving map display with a great deal of cultural information and possibly other traffic information for situational awareness.

## 6. POTENTIAL FOR COLLISION AVOIDANCE APPLICATIONS

The network capable image generator described in Sections 3 and 4 provides the possibility of displaying other entities on the cockpit display to realize a CDTI. That is, given a low bandwidth communications channel upon which other aircraft could transmit time tagged state information the display could render those aircraft on the display. The DIS protocol mentioned above is well suited for such an application because it codifies the packet format and content necessary to propagate entities in the virtual environment. Indeed this was its designated purpose in simulated training applications for the U.S. military where it originated.

In the current application each aircraft would broadcast its identity, type, and state information over a given communication channel, eventually say the automatic dependent surveillance ADS-B data link. Gazit [11] gives general overview of this type of improved aircraft tracking and avoidance as well as the data link implications. The image generator would have an internal model of all other aircraft types or at least the capability to request and incorporate such a model. An instantiation of any one of these models may be propagated by the image generator for each unique aircraft broadcasting state information intermittently in the local region. The high level of fidelity already available in aircraft models reduces the bandwidth burden of updating other entities in the virtual environment. Unlike the host aircraft whose state information must be tightly coupled to the image generator, other traffic would require much lower update rates to realistically render their modeled entities.

APPENDIX AA

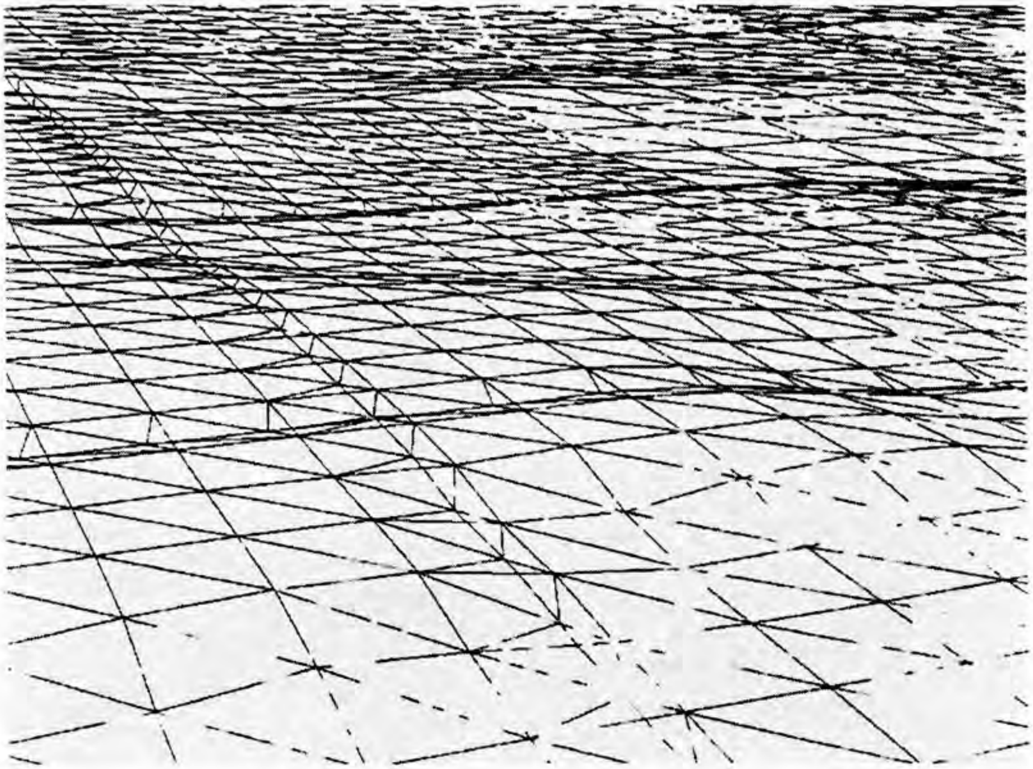


Figure 3. The wire frame image of the underlying terrain polygons shows the varying levels of detail that are stored within the virtual environment database. This image is a screen capture from the real time image generator.

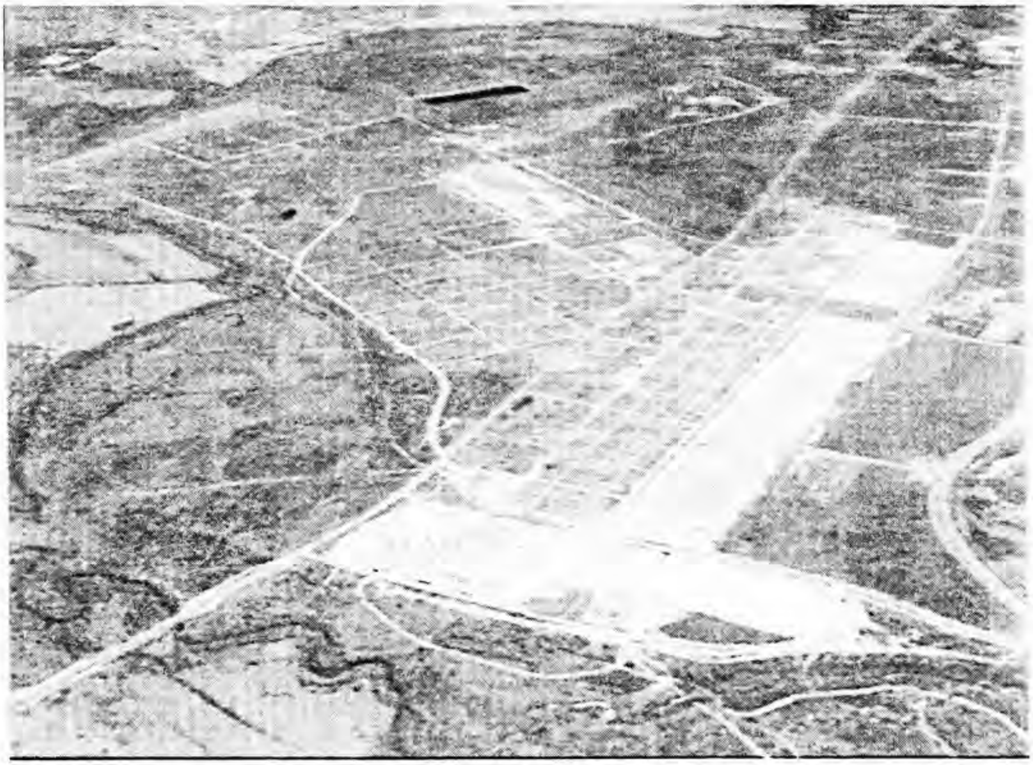


Figure 4. Geographically specific satellite imagery is applied to the terrain polygons in patches. Multiple patch sizes are encoded into the database for varying levels of detail. This image is a B/W screen capture from the real time image generator.

Microsoft et al. Exhibit 1005

APPENDIX AA

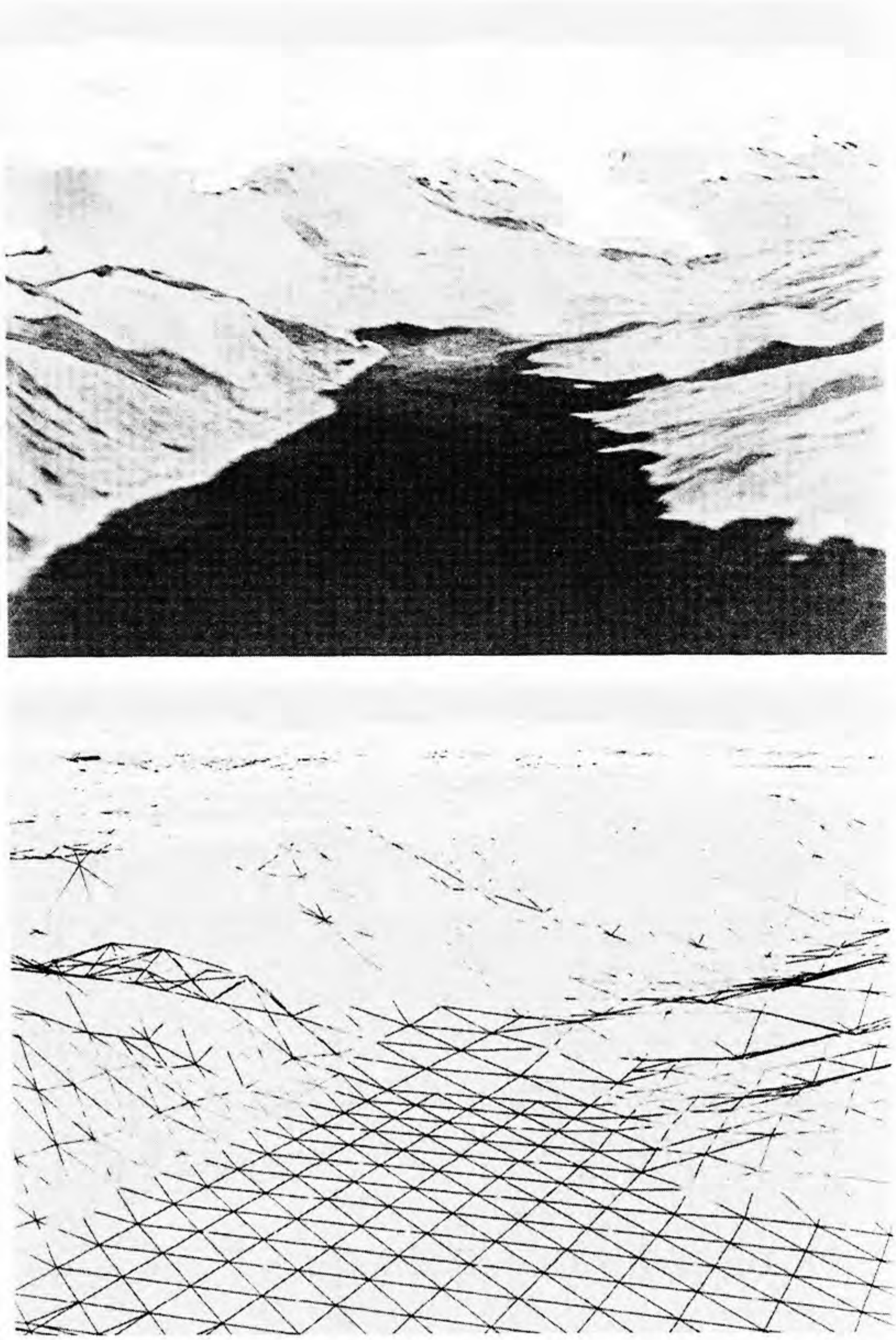


Figure 5. Terrain with extreme amounts of structure can be accommodated with high fidelity. The bottom graphic is a wire frame image of the Alaskan coastline. On the top is the fully rendered scene with imagery.

Microsoft et al. Exhibit 1005

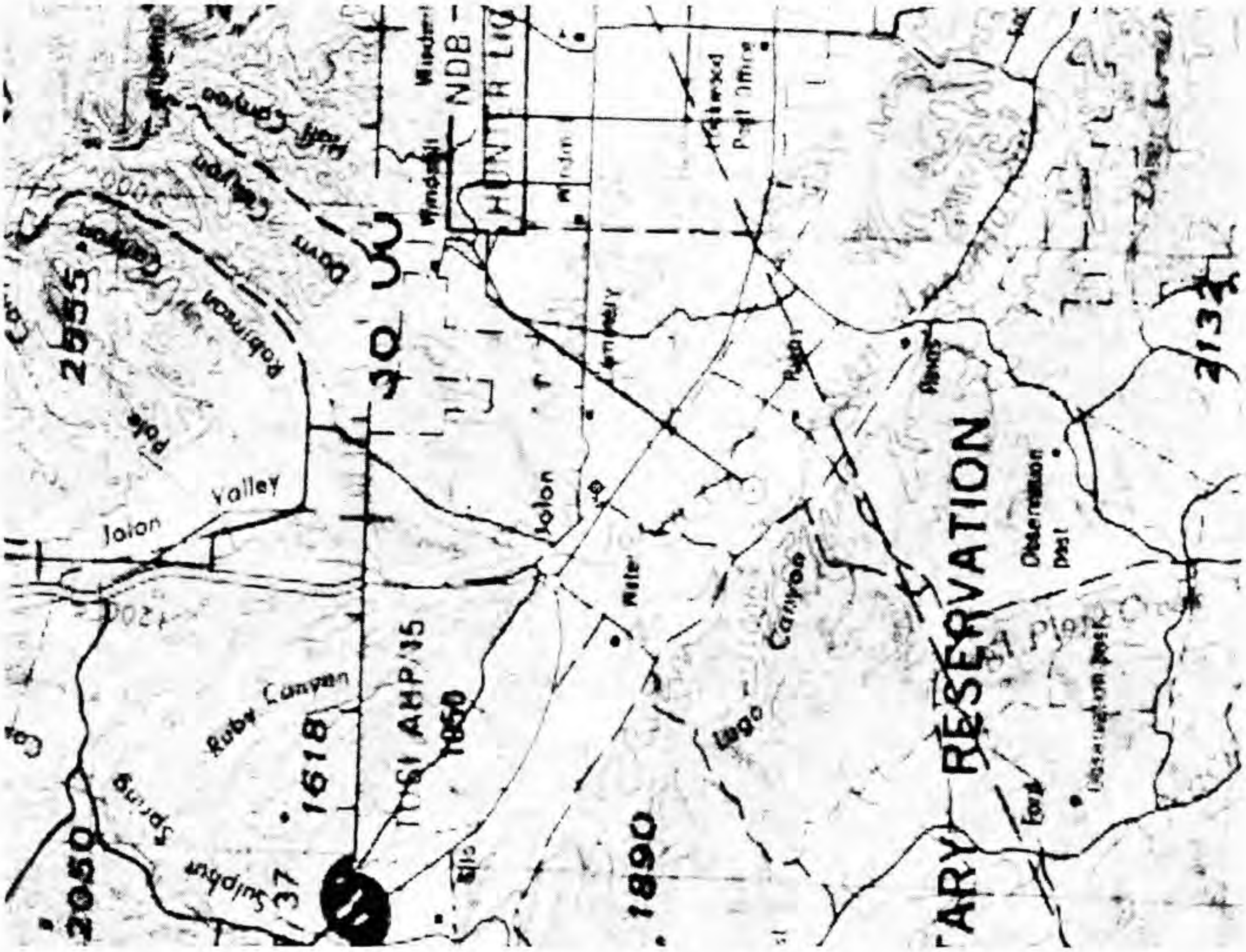


**Figure 6. Top down views of additional cultural features are also possible using the plan view display mode. This image displays a moving map that is shifting underneath the aircraft's viewpoint.**

## 7. CONCLUSIONS

The integration of fully capable low-cost image generators, high fidelity terrain databases, and differential GPS navigation/attitude determination provides a viable path to the production of 3D glass cockpit displays for aviation applications, and general aviation in particular. The end goal of such a system is ultimately to aid the pilot by providing enhanced situational awareness. We have described here the basic components of the underlying system: low cost/high accuracy navigation and attitude sensors that are reliable, fully capable image generators that degrade gracefully, and high fidelity virtual environment databases that have complete correlation with the navigation system.

In the fullness of time the FAA's WAAS will provide high accuracy navigation solution with integrity to all equipped aircraft. The continuous, incremental improvement in PC graphics capability will, we predict, push this type of prototype implementation into the realm of an embedded system. At that point the economies of scale would again dramatically reduce the cost of an integrated solution.

## 8. REFERENCES

1. Barrows, A., K. Alter, P. Enge, B. Parkinson, and J. Powell, "Operational experience with and improvements to a tunnel-in-the-sky display for light aircraft", ION GPS'97, September 1997.
2. Barrows, A., P. Enge, B. Parkinson, and J. Powell, "Flying curved approaches and missed approaches: 3-D display trials onboard light aircraft", ION GPS '96, September 1996.
3. Alter, K., A. Barrows, C. Jennings, P. Enge, and D. Powell, "3-D cockpit displays for general aviation in mountainous terrain", ION GPS'98, September 1998.
4. Enge, P., et. al., "Wide area augmentation of the Global Positioning System", *Proceedings of the IEEE*, **84**, #8, 1996.
5. Comp, C., et. al., "Demonstration of WAAS aircraft approach and landing in Alaska", ION GPS'98, September 1998.
6. Walter, T., P. Enge, and A. Hansen, "A proposed integrity equation for WAAS MOPS", ION GPS'98, September 1998.
7. Teague, H., "Low-cost GPS/inertial attitude and heading reference system (AHRS) for EV/SV applications", *Proceedings of SPIE*, **3691**, Aerosense/Enhanced and Synthetic Vistion, April 1999.
8. Gebre-Egziabher, D., R. Hayward, and J. Powell, "A low-cost GPS/inertial attitude heading reference system (AHRS) for general aviation applications", IEEE PLANS '98, April 1998.
9. Hayward, R., and J. Powell, "Real time calibration of antenna phase errors for ultra short baseline attitude systems", ION GPS'98, September 1998.
10. Ourston and Reece, "Issues Involved with Integrating Live and Artificial Virtual Individual Combatants", Simulation Interoperability Workshop, March 1998.
11. Hansen, A., and P. Levin, "On conforming Delaunay mesh generation", *Advances in Engineering Software*, **4**, #2, 1991.
12. Gazit, R., *Aircraft Surveillance and Collision Avoidance Using GPS*, PhD thesis, Stanford University, 1996.
13. RTCA Special Committee 159, *Minimum Operational Performance Standards for Airborne Equipment Using Global Positioning System/Wide Area Augmentation*, RTCA/DO-229 Change 3, RTCA, Inc., November 1997.
14. Pogorelc, S., et. al., "Flight and static test results for NSTB", ION GPS '96, September 1996.
15. IEEE, "DIS Exercise Management and Feedback—Recommended Practice", IEEE Standard 1278.3, Institute of Electrical and Electronic Engineers, Inc., 1995.

APPENDIX BB

US005760783A

# United States Patent [19]

**Migdal et al.**

[11] **Patent Number: 5,760,783**

[45] **Date of Patent: Jun. 2, 1998**

[54] **METHOD AND SYSTEM FOR PROVIDING TEXTURE USING A SELECTED PORTION OF A TEXTURE MAP**

[75] Inventors: **Christopher Joseph Migdal**, Mt. View; **James L. Foran**, Milpitas; **Michael Timothy Jones**, Los Altos; **Christopher Clark Tanner**, San Jose, all of Calif.

[73] Assignee: **Silicon Graphics, Inc.**, Mountain View, Calif.

[21] Appl. No.: **554,047**

[22] Filed: **Nov. 6, 1995**

[51] **Int. Cl.$^6$** .......... **G06T 11/00**

[52] **U.S. Cl.** .......... **345/430**

[58] **Field of Search** .......... 395/128–132, 395/125–127; 345/430

[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 4,727,365 | 2/1988 | Bunker et al. | 340/728 |
| 4,974,176 | 11/1990 | Buchner et al. | 364/522 |
| 5,097,427 | 3/1992 | Lathrop et al. | 395/130 |
| 5,490,240 | 2/1996 | Foran et al. | 395/130 |

FOREIGN PATENT DOCUMENTS

| | | |
|---|---|---|
| 0 447 227 A2 | 9/1991 | European Pat. Off. . |
| 0 513 474 A1 | 11/1992 | European Pat. Off. . |

OTHER PUBLICATIONS

Blinn, Jim, "Jim Blinn's Corner: The Truth About Texture Mapping," *IEEE Computer Graphics & Applications*, Mar., 1990, pp. 78–83.

Foley et al., "17.4.3 Other Pattern Mapping Techniques," *Computer Graphics: Principles and Practice*, 1990, pp. 826–828.

Cosman, M., "Global Terrain Texture: Lowering the Cost," *Proceedings of the 1994 Image VII Conference*, Tempe, Arizona: The Image Society, pp. 53–64.

Dungan, W. et al., "Texture Tile Considerations for Raster Graphics," *Siggraph '78 Proceedings* (1978) pp. 130–134.

Economy, R. et al., "The Application of Aerial Photography and Satellite Imagery to Flight Simulation," pp. 280–287.

Foley et al., *Computer Graphics Principles and Practice*, Second Edition, Addison–Wesley Publishing Company, Reading, Massachusetts (1990), pp. 742–743 and 826–828.

Watt, A., *Fundamentals of Three–Dimensional Computer Graphics*, Addison–Wesley Publishing Company, USA (1989), pp. 227–250.

Williams, L., "Pyramidal Parametrics," *Computer Graphics*, vol. 17, No. 3, Jul. 1983, pp. 1–15.

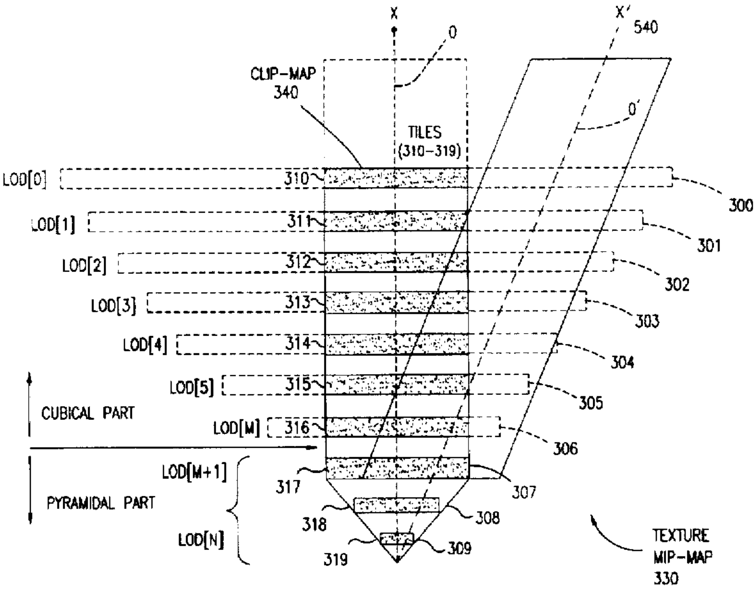
*Primary Examiner*—Almis R. Jankus

*Attorney, Agent, or Firm*—Sterne, Kessler, Goldstein & Fox P.L.L.C.

[57] **ABSTRACT**

An apparatus and method for quickly and efficiently providing texel data relevant for displaying a textured image. A large amount of texture source data, such as photographic terrain texture, is stored as a two-dimensional or three-dimensional texture MIP-map on one or more mass storage devices. Only a relatively small clip-map representing selected portions of the complete texture MIP-map is loaded into faster, more expensive memory. These selected texture MIP-map portions forming the clip-map consist of tiles which contain those texel values at each respective level of detail that are most likely to be mapped to pixels being rendered for display based upon the viewer's eyepoint and field of view. To efficiently update the clip-map in real-time, texel data is loaded and discarded from the edges of tiles. Attempts to access a texel lying outside of a particular clip-map tile are accommodated by utilizing a substitute texel value obtained from the next coarser resolution clip-map tile which encompasses the sought texel.

**28 Claims, 14 Drawing Sheets**



APPENDIX BB

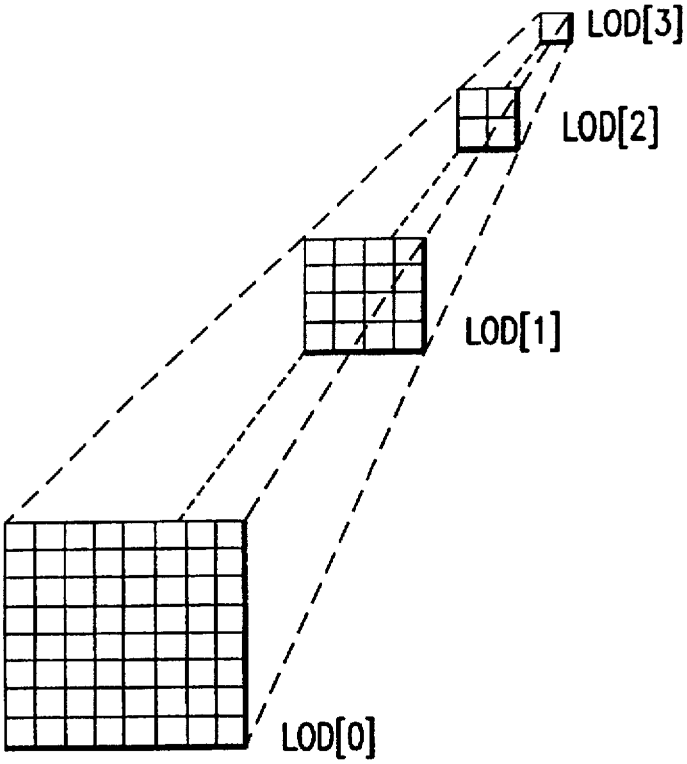


FIG.1A

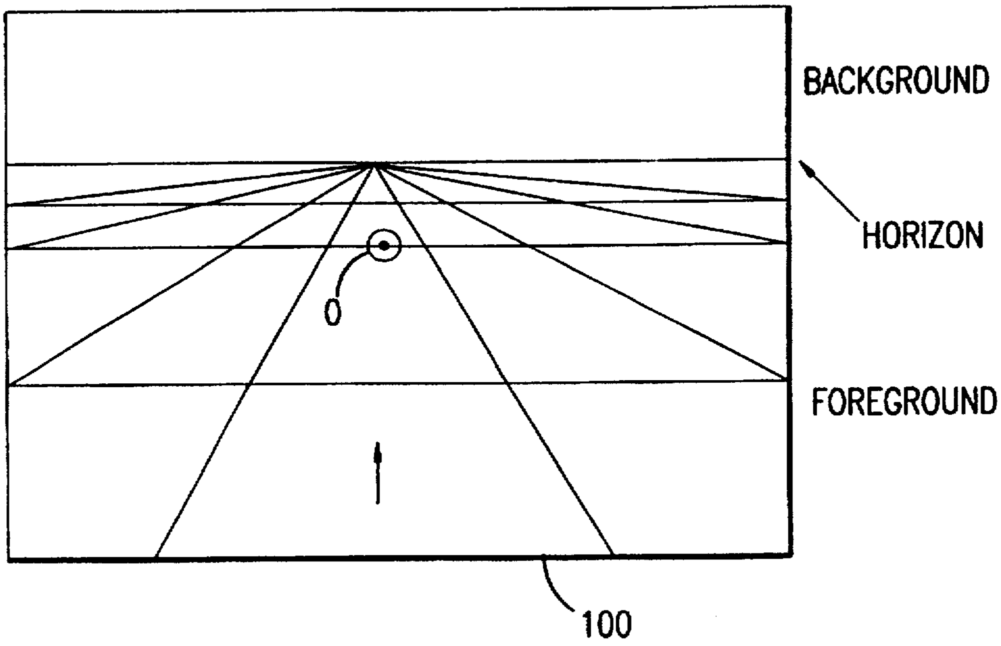


FIG.1B

APPENDIX BB

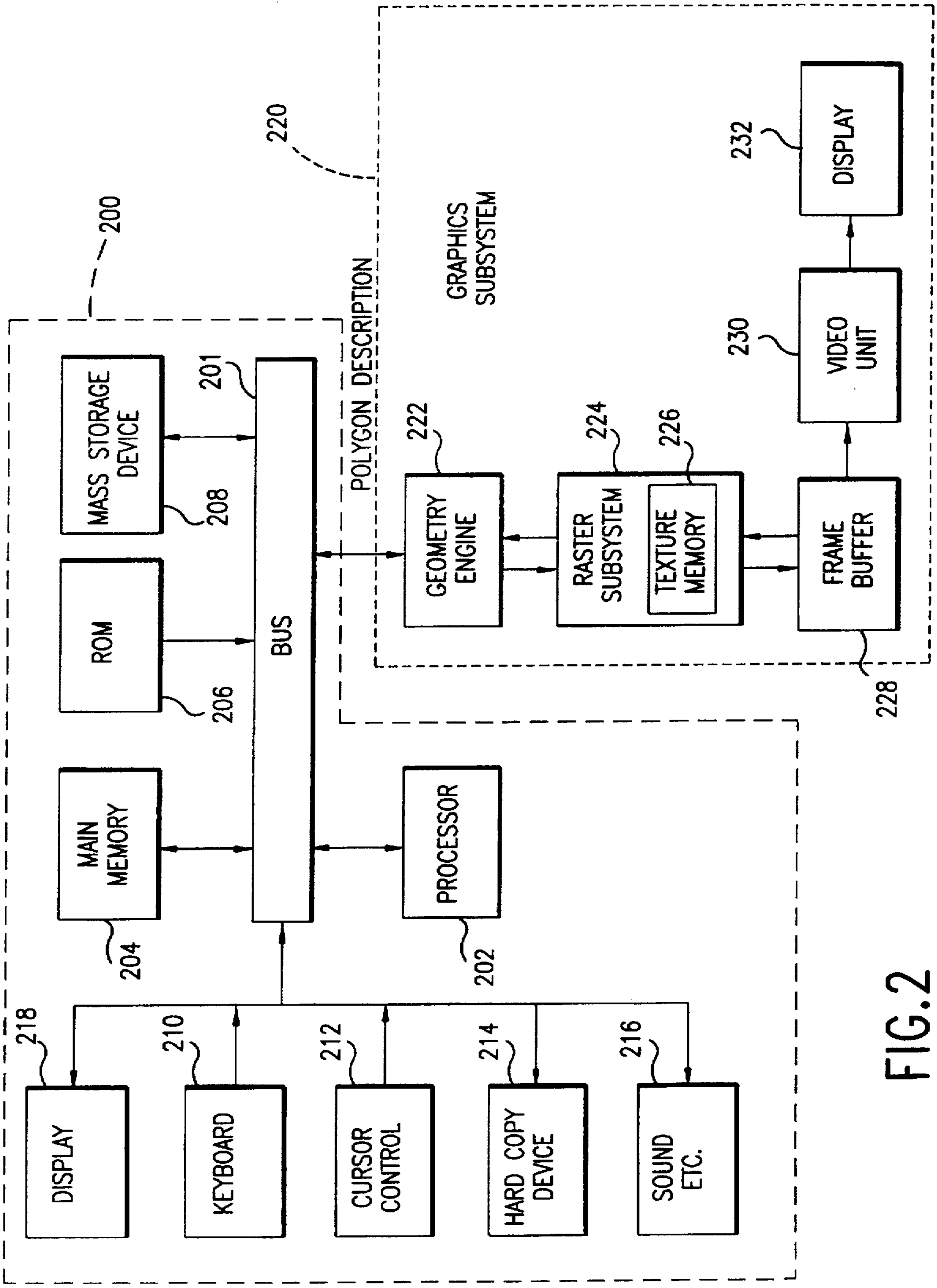


FIG.2

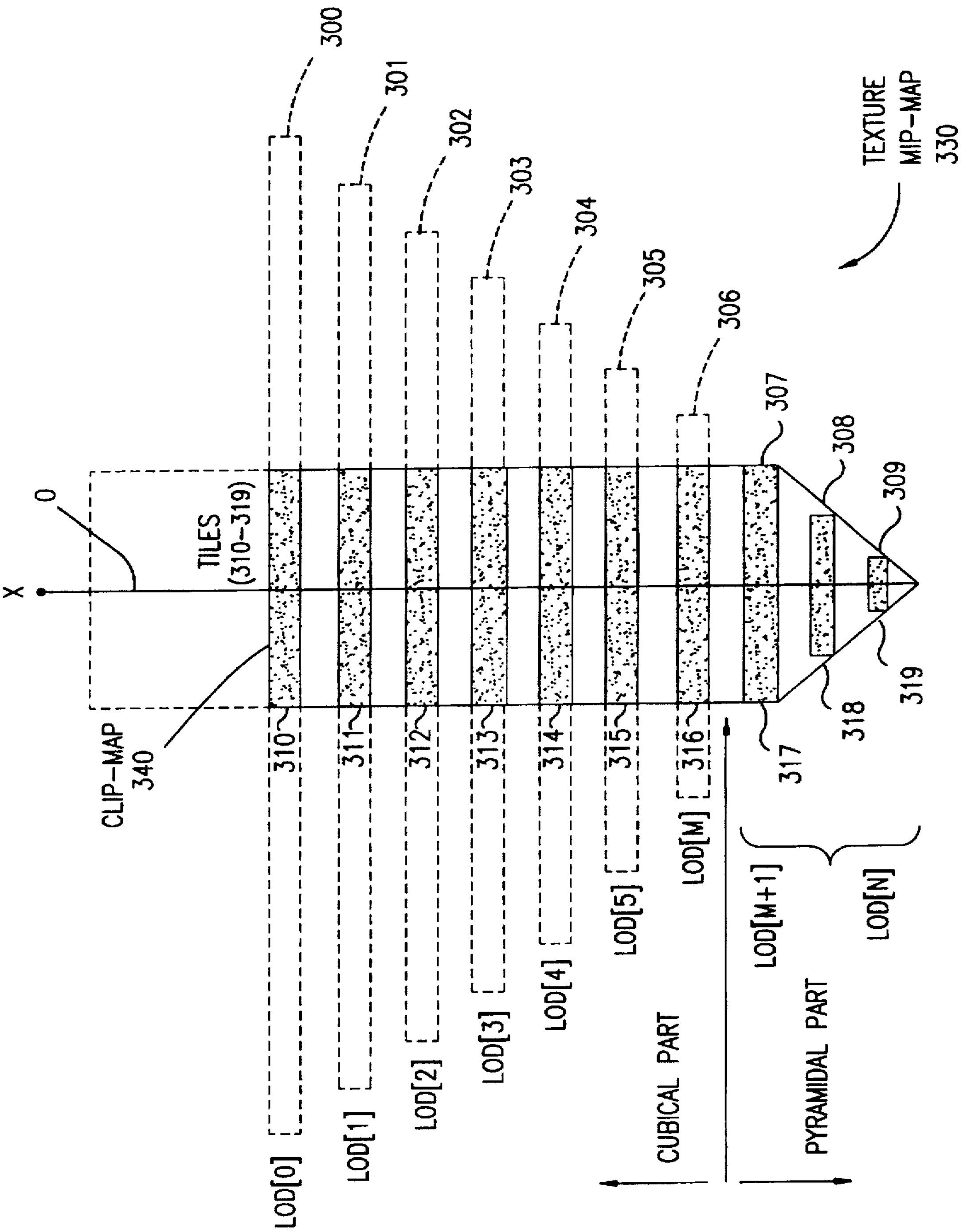

X
0
CLIP-MAP
340
TILES
(310-319)
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
TEXTURE
MIP-MAP
330
LOD[0]
LOD[1]
LOD[2]
LOD[3]
LOD[4]
LOD[5]
LOD[M]
LOD[M+1]
LOD[N]
CUBICAL PART
PYRAMIDAL PART

FIG.3

APPENDIX BB

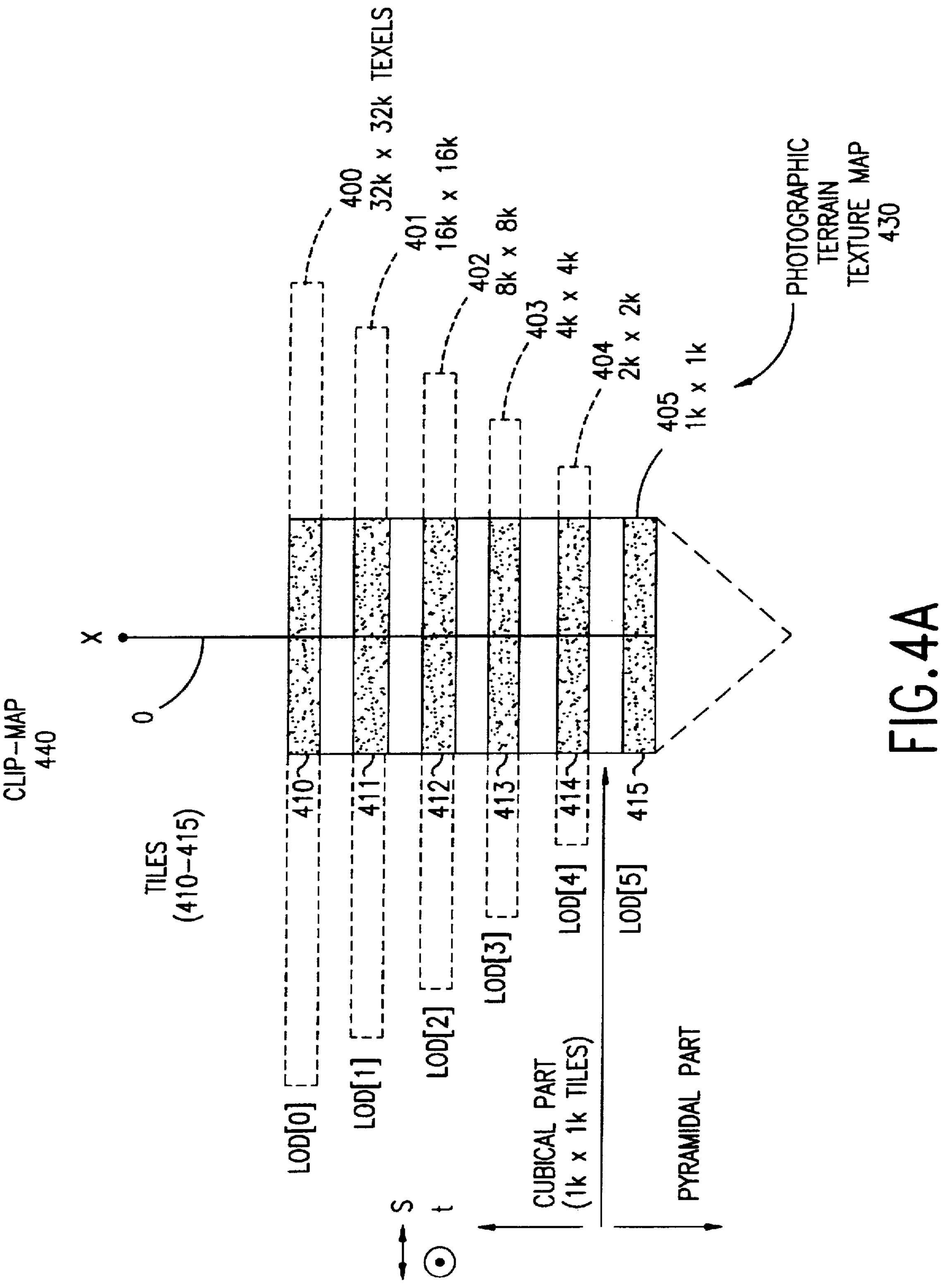


FIG.4A

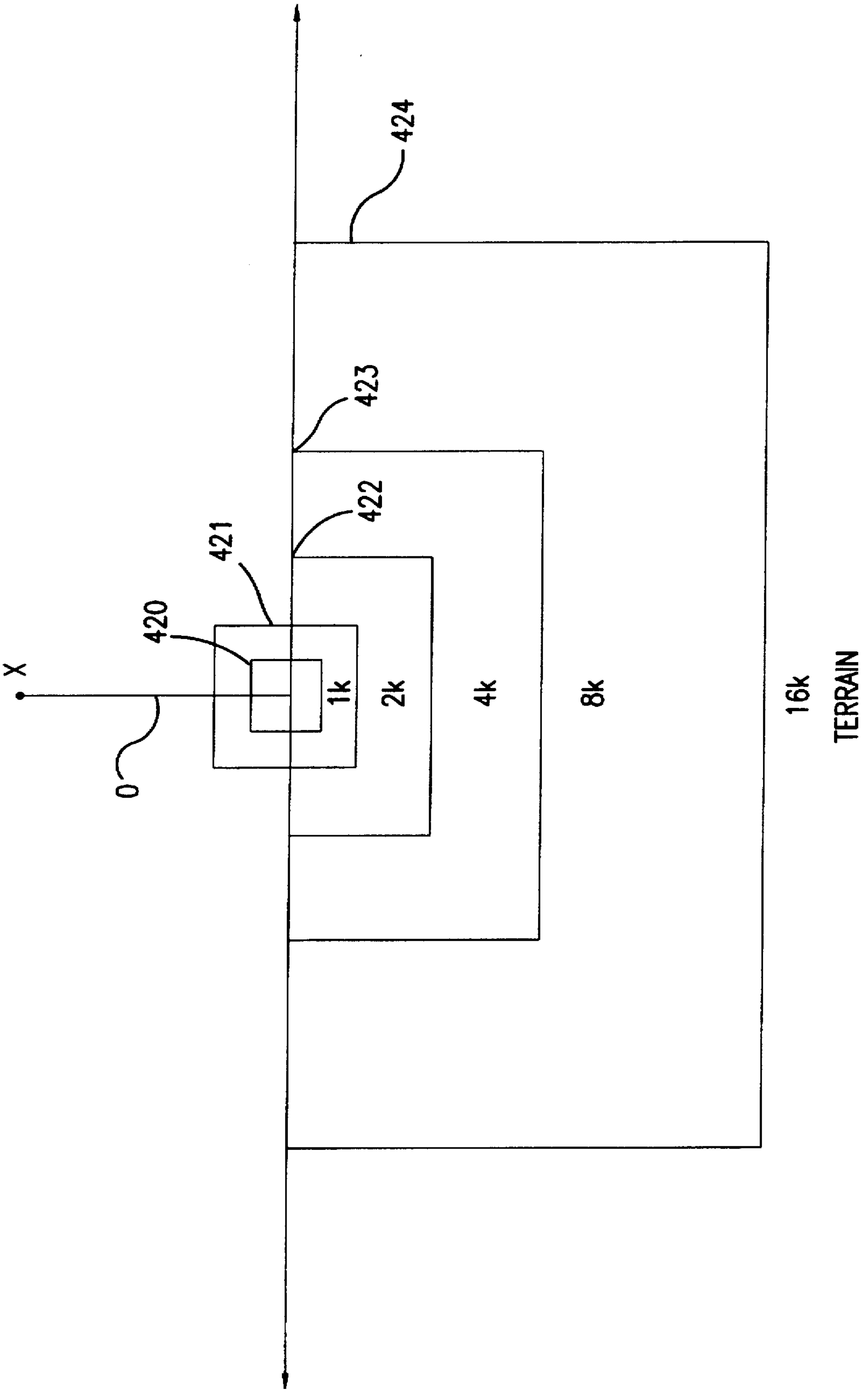


FIG.4B

APPENDIX BB

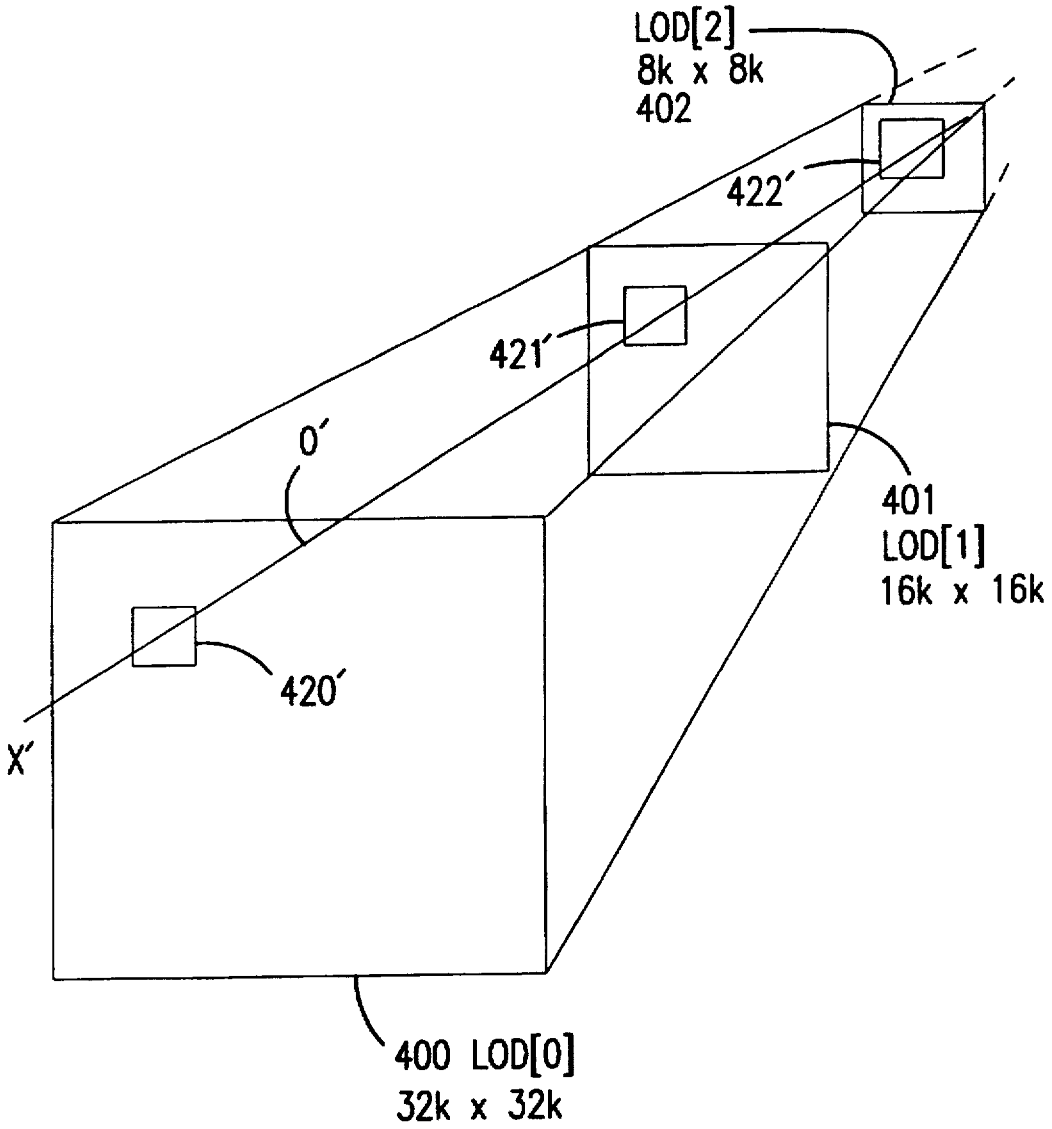


FIG.4C

APPENDIX BB

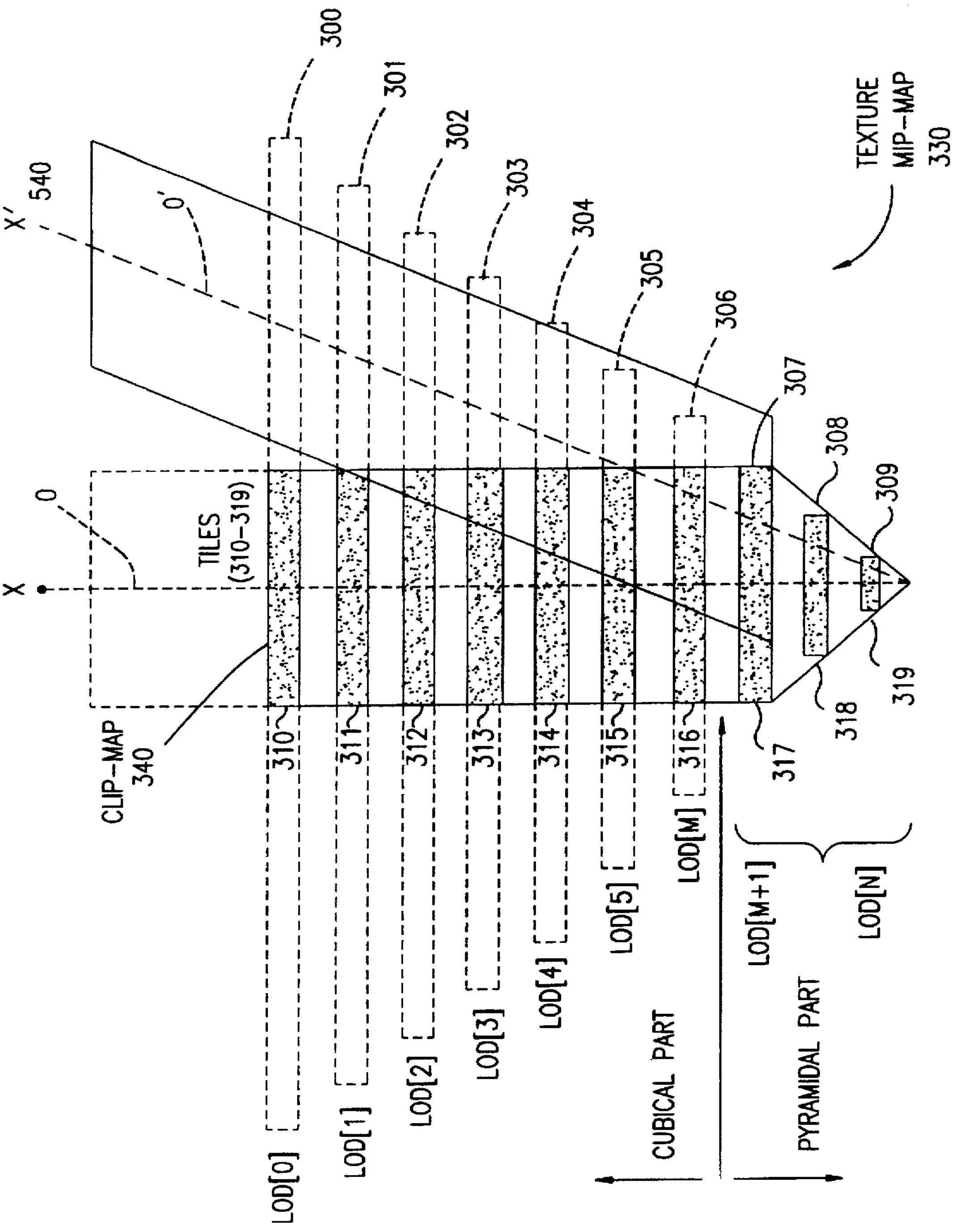


FIG.5

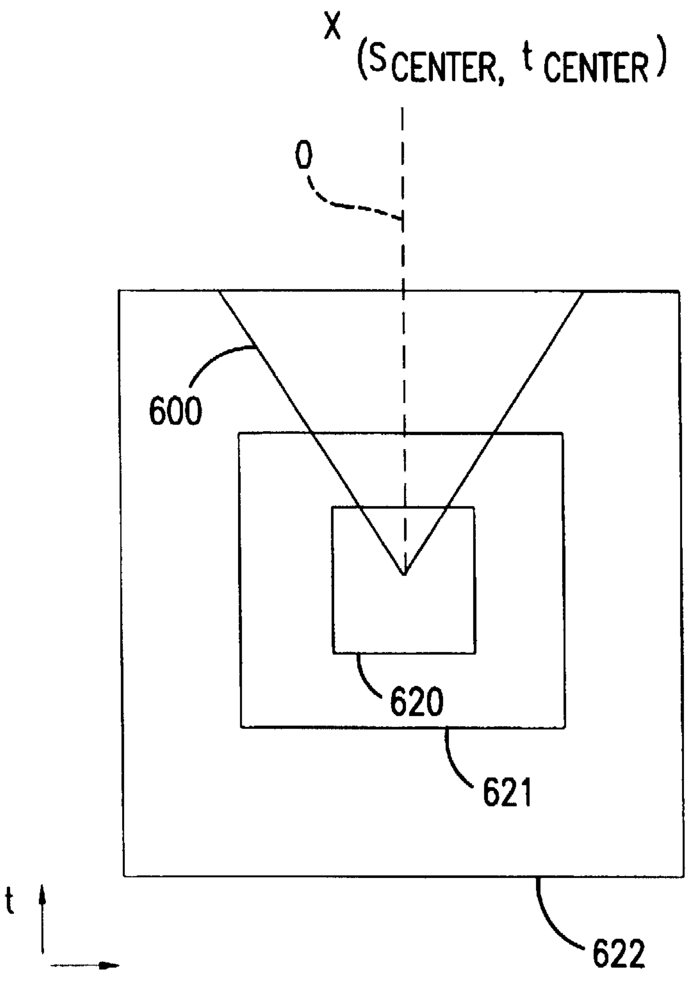

X
(S_CENTER, t_CENTER)
0
600
620
621
622
t
s

FIG.6A

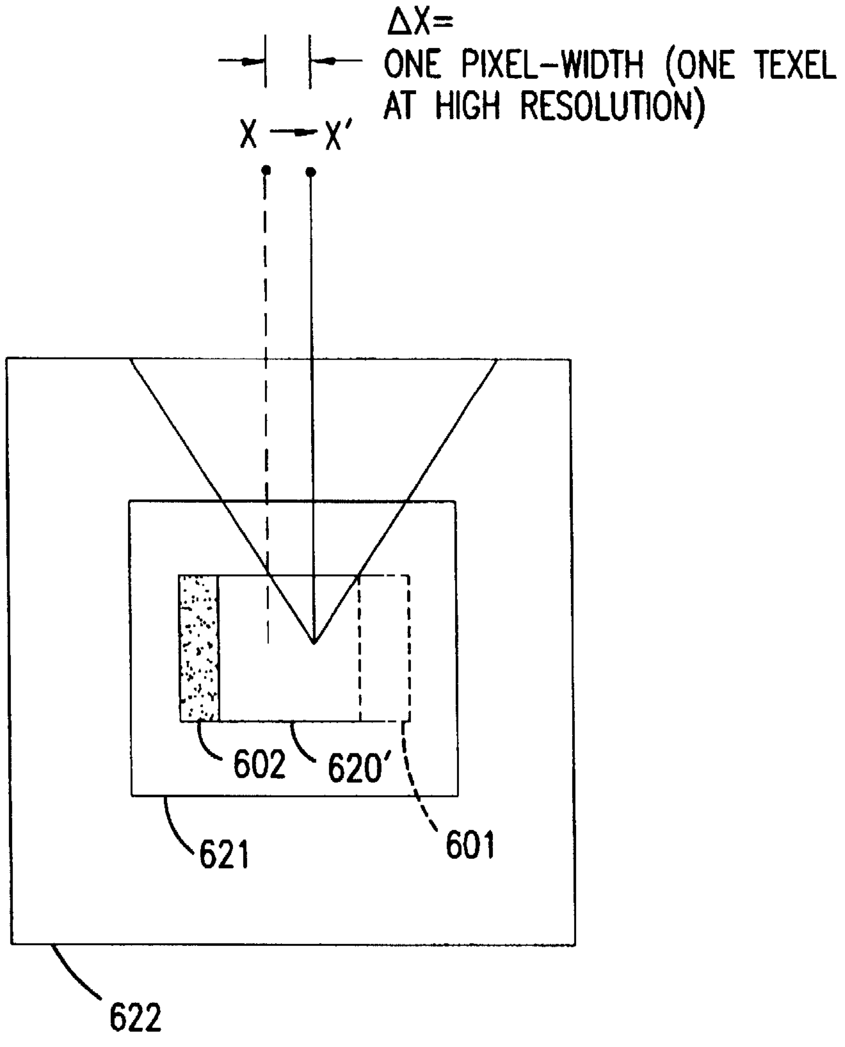

ΔX=
ONE PIXEL-WIDTH (ONE TEXEL
AT HIGH RESOLUTION)
X → X'
602
620'
601
621
622

FIG.6B

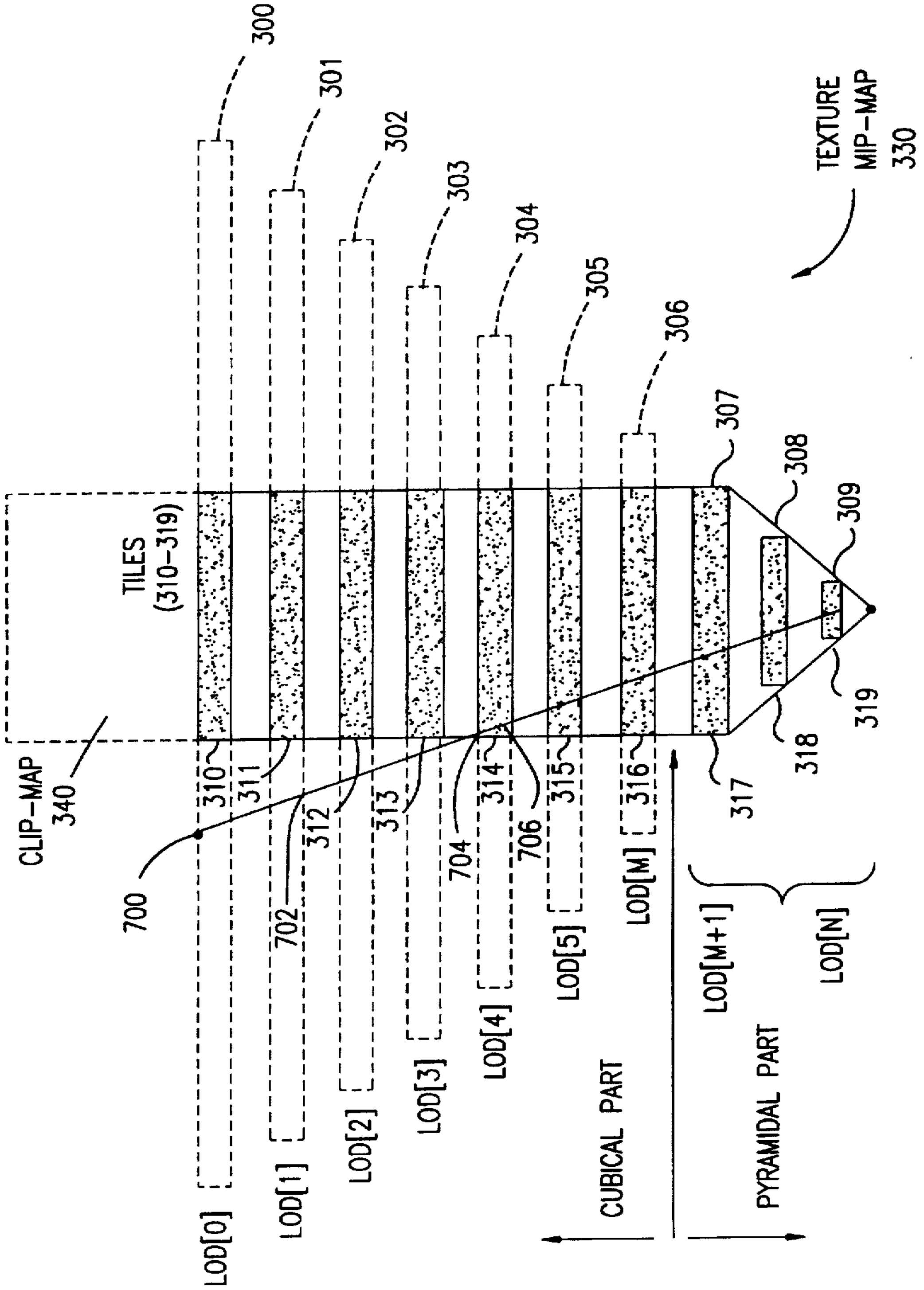

300
301
302
303
304
305
306
307
308
309
TEXTURE
MIP-MAP
330
CLIP-MAP
340
TILES
(310-319)
310
311
312
313
314
315
316
317
318
319
700
702
704
706
LOD[0]
LOD[1]
LOD[2]
LOD[3]
LOD[4]
LOD[5]
LOD[M]
LOD[M+1]
LOD[N]
CUBICAL PART
PYRAMIDAL PART

FIG.7

APPENDIX BB

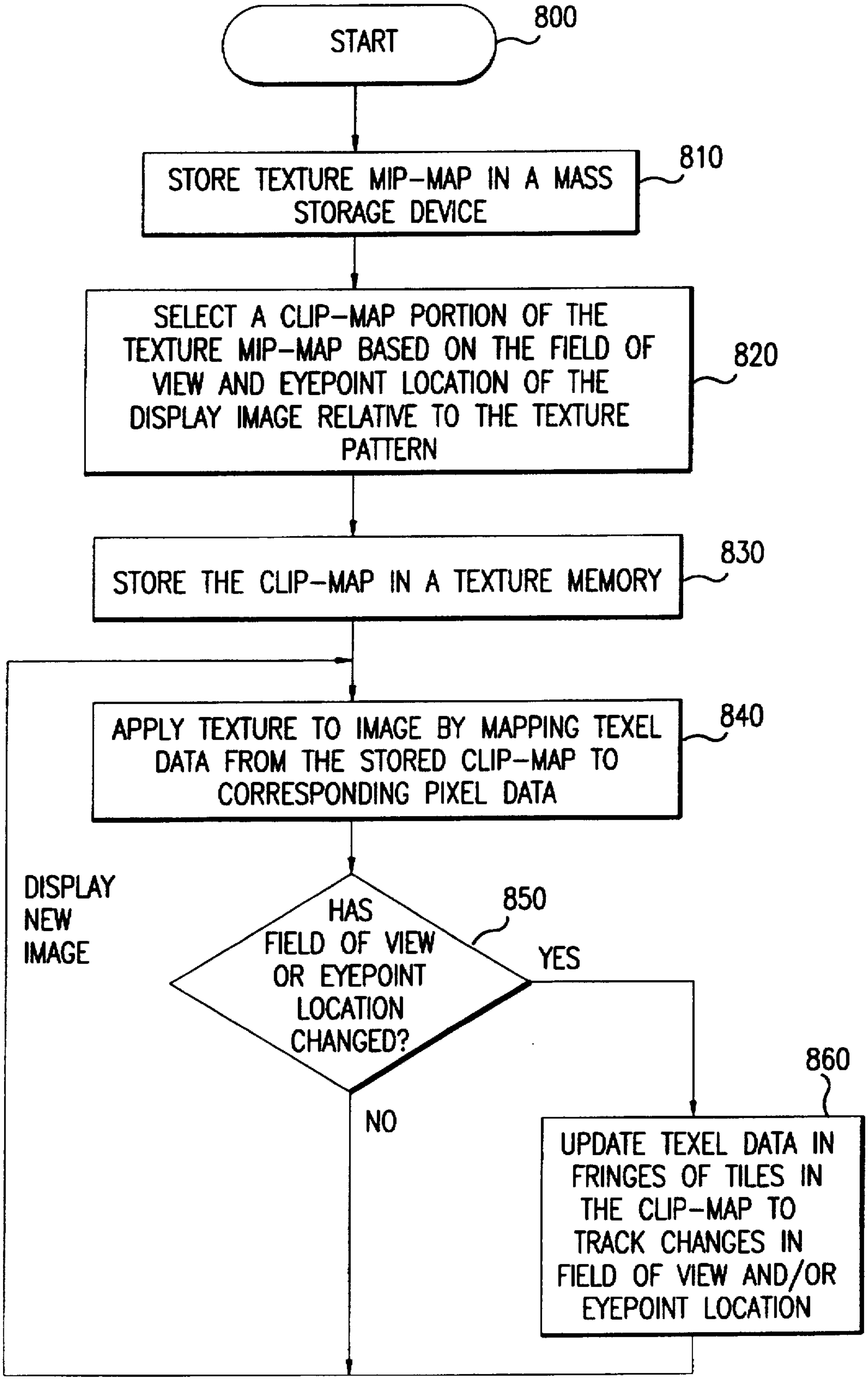

START
800
STORE TEXTURE MIP-MAP IN A MASS STORAGE DEVICE
810
SELECT A CLIP-MAP PORTION OF THE TEXTURE MIP-MAP BASED ON THE FIELD OF VIEW AND EYEPOINT LOCATION OF THE DISPLAY IMAGE RELATIVE TO THE TEXTURE PATTERN
820
STORE THE CLIP-MAP IN A TEXTURE MEMORY
830
APPLY TEXTURE TO IMAGE BY MAPPING TEXEL DATA FROM THE STORED CLIP-MAP TO CORRESPONDING PIXEL DATA
840
DISPLAY NEW IMAGE
HAS FIELD OF VIEW OR EYEPOINT LOCATION CHANGED?
850
YES
NO
860
UPDATE TEXEL DATA IN FRINGES OF TILES IN THE CLIP-MAP TO TRACK CHANGES IN FIELD OF VIEW AND/OR EYEPOINT LOCATION

FIG.8A

APPENDIX BB

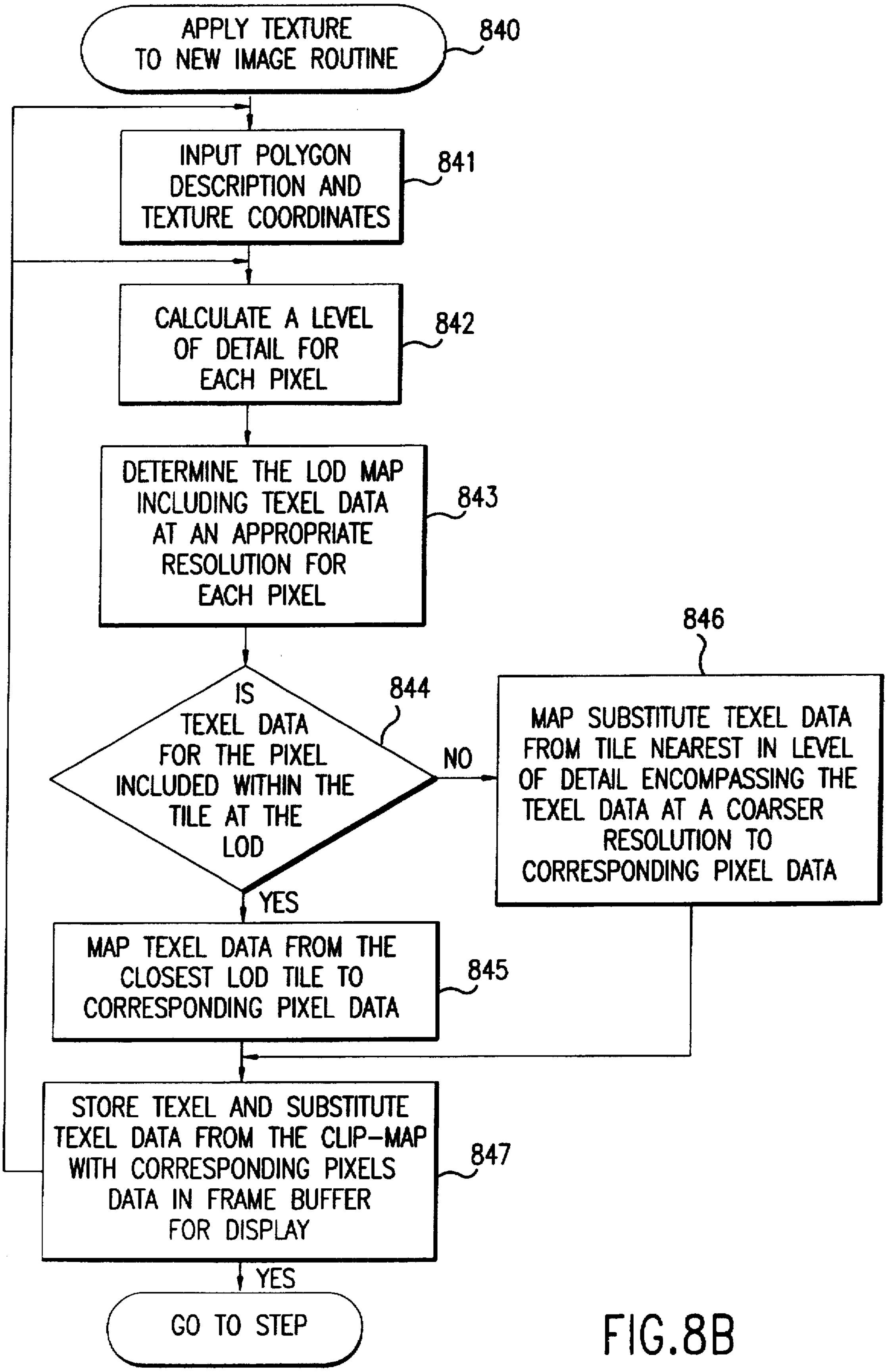

APPLY TEXTURE TO NEW IMAGE ROUTINE
840
INPUT POLYGON DESCRIPTION AND TEXTURE COORDINATES
841
CALCULATE A LEVEL OF DETAIL FOR EACH PIXEL
842
DETERMINE THE LOD MAP INCLUDING TEXEL DATA AT AN APPROPRIATE RESOLUTION FOR EACH PIXEL
843
IS TEXEL DATA FOR THE PIXEL INCLUDED WITHIN THE TILE AT THE LOD
844
NO
846
MAP SUBSTITUTE TEXEL DATA FROM TILE NEAREST IN LEVEL OF DETAIL ENCOMPASSING THE TEXEL DATA AT A COARSER RESOLUTION TO CORRESPONDING PIXEL DATA
YES
MAP TEXEL DATA FROM THE CLOSEST LOD TILE TO CORRESPONDING PIXEL DATA
845
STORE TEXEL AND SUBSTITUTE TEXEL DATA FROM THE CLIP-MAP WITH CORRESPONDING PIXELS DATA IN FRAME BUFFER FOR DISPLAY
847
YES
GO TO STEP

FIG.8B

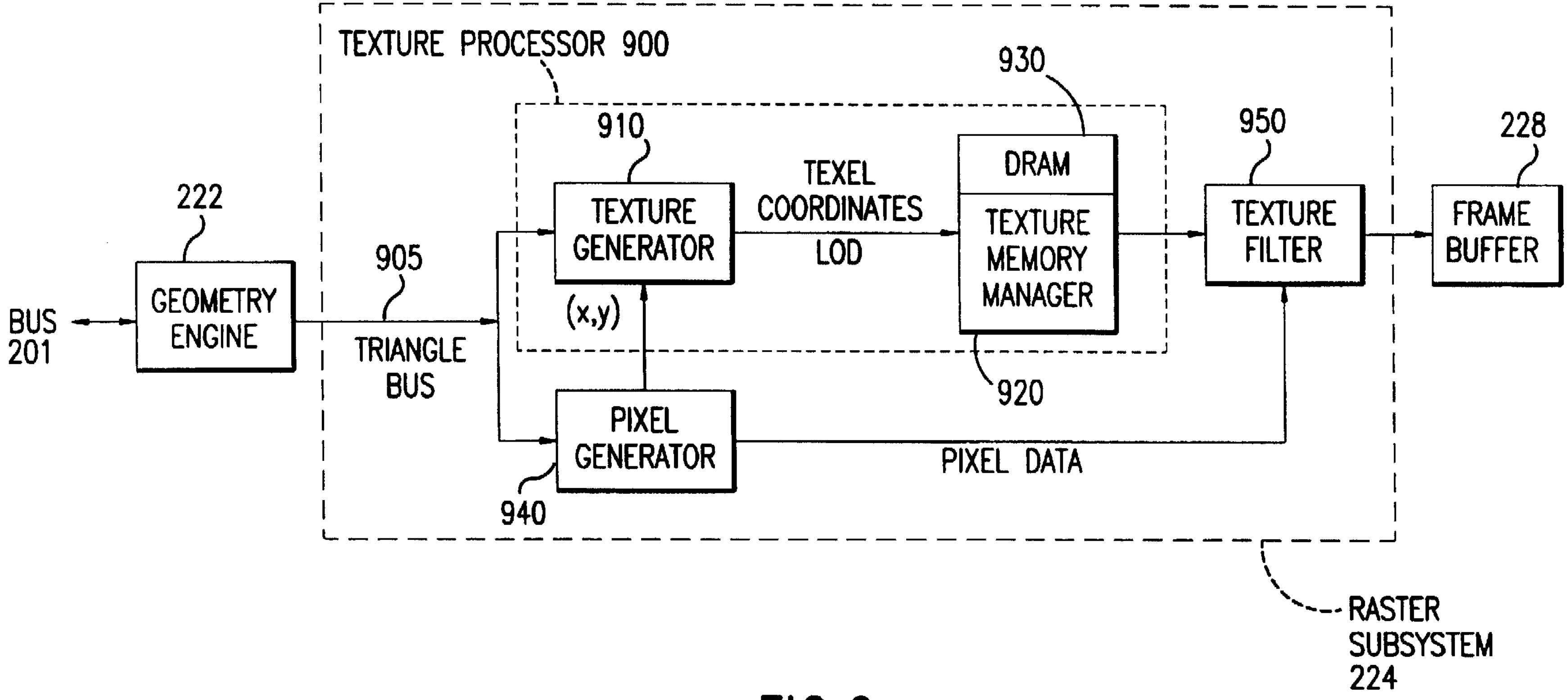

TEXTURE PROCESSOR 900
930
910
950
228
222
TEXEL
COORDINATES
LOD
DRAM
TEXTURE
MEMORY
MANAGER
TEXTURE
GENERATOR
TEXTURE
FILTER
FRAME
BUFFER
905
BUS
201
GEOMETRY
ENGINE
TRIANGLE
BUS
(x,y)
920
PIXEL
GENERATOR
PIXEL DATA
940
RASTER
SUBSYSTEM
224

FIG.9

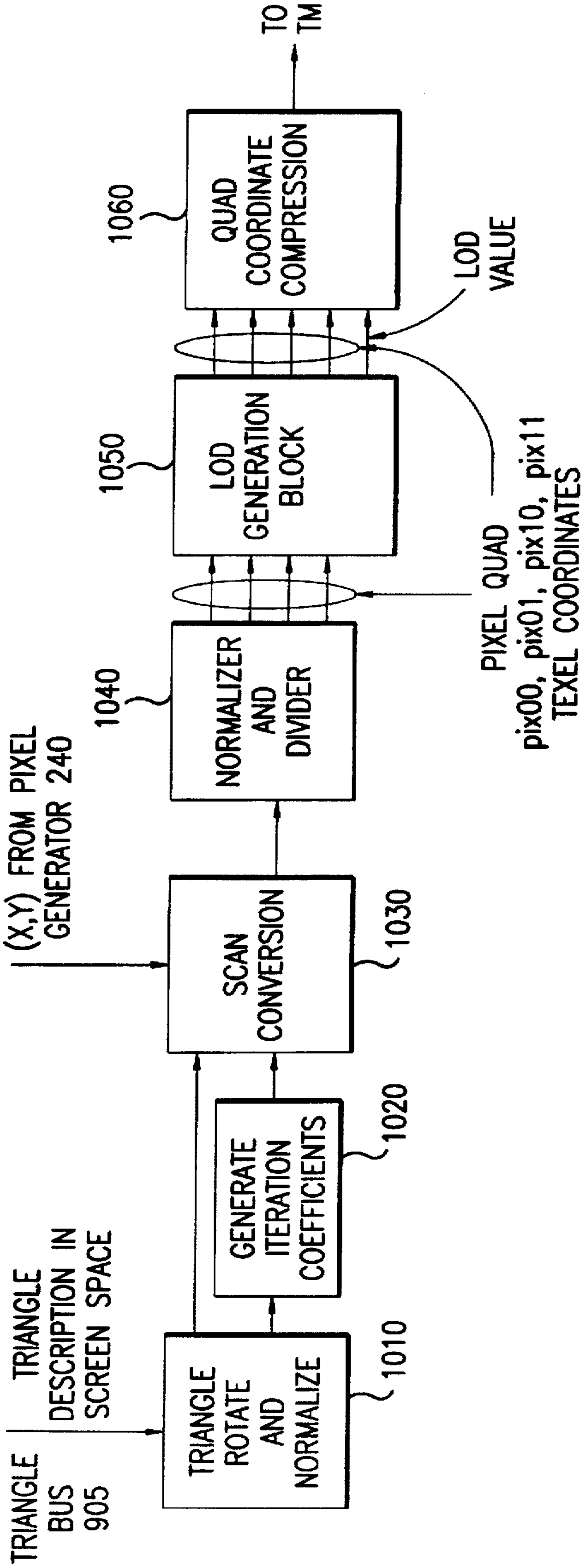

TRIANGLE BUS 905
TRIANGLE DESCRIPTION IN SCREEN SPACE
TRIANGLE ROTATE AND NORMALIZE
1010
GENERATE ITERATION COEFFICIENTS
1020
(X,Y) FROM PIXEL GENERATOR 240
SCAN CONVERSION
1030
NORMALIZER AND DIVIDER
1040
PIXEL QUAD pix00, pix01, pix10, pix11 TEXEL COORDINATES
LOD GENERATION BLOCK
1050
LOD VALUE
QUAD COORDINATE COMPRESSION
1060
TO TM

FIG.10

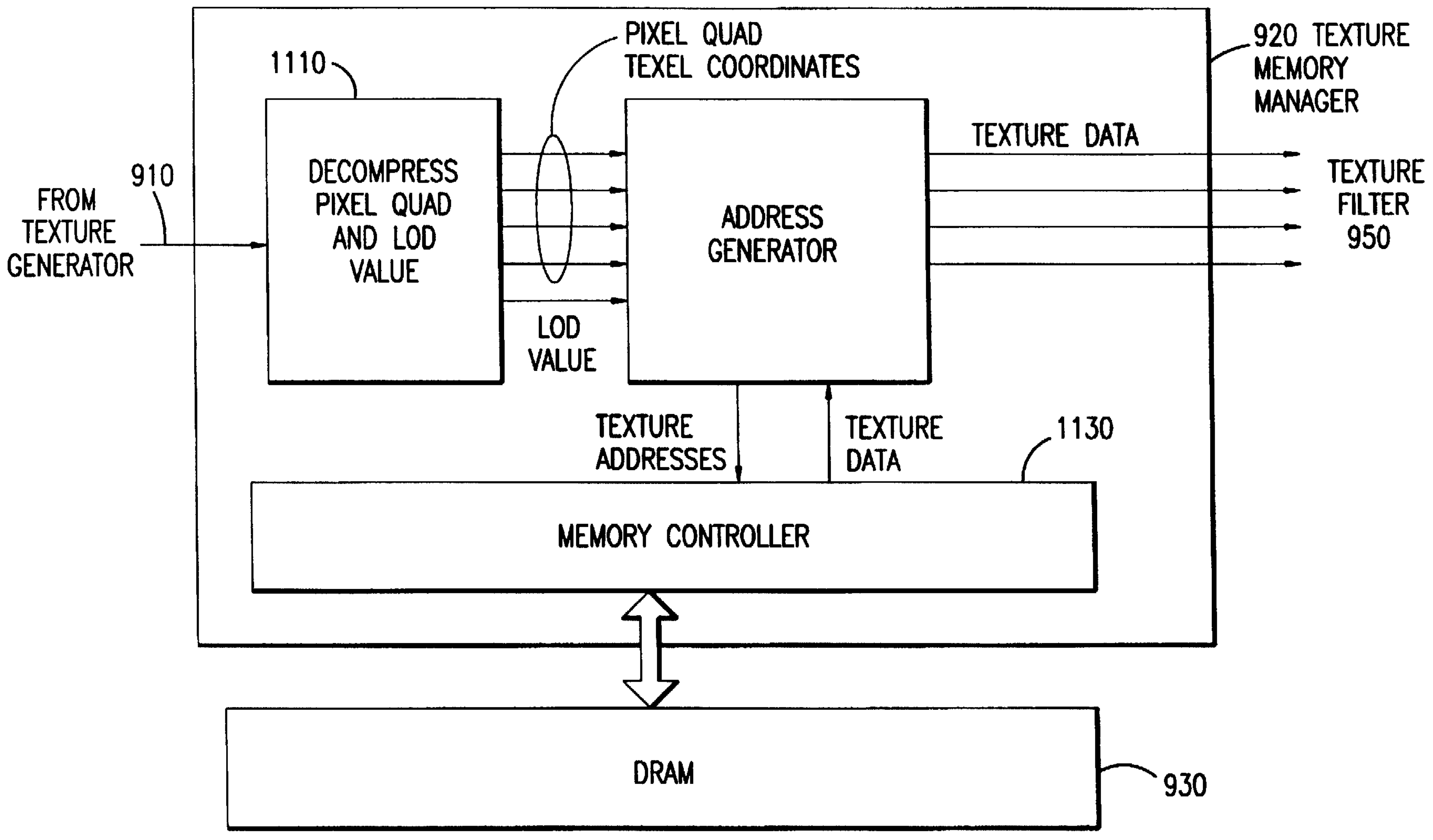

FROM
TEXTURE
GENERATOR
910
1110
DECOMPRESS
PIXEL QUAD
AND LOD
VALUE
PIXEL QUAD
TEXEL COORDINATES
LOD
VALUE
ADDRESS
GENERATOR
TEXTURE DATA
920 TEXTURE
MEMORY
MANAGER
TEXTURE
FILTER
950
TEXTURE
ADDRESSES
TEXTURE
DATA
1130
MEMORY CONTROLLER
DRAM
930

FIG.11

# METHOD AND SYSTEM FOR PROVIDING TEXTURE USING A SELECTED PORTION OF A TEXTURE MAP

## BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention pertains to the field of computer graphics. More particularly, the present invention relates to an apparatus and method for providing texel data from selected portions of a texture MIP-map (referred to herein as a clip-map).

2. Related Art

Computer systems are commonly used for displaying graphical objects on a display screen. These graphical objects include points, lines, polygons, and three dimensional solid objects. By utilizing texture mapping techniques, color and other details can be applied to areas and surfaces of these objects. In texture mapping, a pattern image, also referred to as a "texture map," is combined with an area or surface of an object to produce a modified object with the added texture detail. For example, given the outline of a featureless cube and a texture map defining a wood grain pattern, texture mapping techniques can be used to "map" the wood grain pattern onto the cube. The resulting display is that of a cube that appears to be made of wood. In another example, vegetation and trees can be added by texture mapping to an otherwise barren terrain model. Likewise, labels can be applied onto packages or cans for visually conveying the appearance of an actual product. Textures mapped onto geometric surfaces provide motion and spatial cues that surface shading alone might not provide. For example, a sphere rotating about its center appears static until an irregular texture or pattern is affixed to its surface.

The resolution of a texture varies, depending on the viewpoint of the observer. The texture of a block of wood displayed up close has a different appearance than if that same block of wood were to be displayed far away. Consequently, there needs to be some method for varying the resolution of the texture (e.g., magnification and minification). One approach is to compute the variances of texture in real time, but this filtering is too slow for complex textures and/or requires expensive hardware to implement.

A more practical approach first creates and stores a MIP-map (multum in parvo meaning "many things in a small place"). The MIP-map consists of a texture pattern pre-filtered at progressively lower or coarser resolutions and stored in varying levels of detail (LOD) maps. See, e.g., the explanation of conventional texture MIP-mapping in Foley et al., *Computer Graphics Principles and Practice,* Second Edition, Addison-Wesley Publishing Company, Reading, Mass. (1990), pages 742–43 and 826–828 (incorporated by reference herein).

FIG. 1A shows a conventional set of texture LOD maps having pre-filtered texel data associated with a particular texture. Four different levels of detail (LOD[O]-LOD[3]m) are shown. Each successive coarser texture LOD has a resolution half that of the preceding LOD until a unitary LOD is reached representing an average of the entire high resolution base texture map LOD[0]. Thus, in FIG. 1A, LOD[O] is an 8×8 texel array; LOD[1] is a 4×4 texel array; LOD[2] is a 2×2 texel array; and LOD [3] is a single 1×1 texel array. Of course, in practice each LOD can contain many more texels, for instance, LOD[0] can be 8k×8k, LOD[1] 4k×4k, and so forth depending upon particular hardware or processing limits.

The benefit of MIP-mapping is that filtering is only performed once on texel data when the MIP-map is initially created and stored in LOD maps. Thereafter, texels having a dimension commensurate with pixel size are obtained by selecting the closest LOD map having an appropriate resolution. By obtaining texels from the pre-filtered LOD maps, filtering does not have to be performed during run-time. More sophisticated filtering operations can be executed beforehand during modeling without delaying real-time operation speed.

To render a display at the appropriate image resolution, a texture LOD is selected based on the relationship between the smallest texel dimension and the display pixel size. For a perspective view of a landscape **100**, as shown in FIG. **1B**, the displayed polygonal image is "magnified" in a foreground region relative to polygonal regions located closer to the center horizon and background along the direction indicated by the arrow. To provide texture for pixels in the closest foreground region, then, texels are mapped from the finest resolution map LOD[0]. Appropriate coarser LODs are used to map texel data covering pixels located further away from the viewer's eyepoint. Such multi-resolution texture MIP-mapping ensures that texels of the appropriate texture LOD gets selected during pixel sampling. To avoid discontinuities between images at varying resolutions, well-known techniques such as linear interpolation are used to blend the texel values of two LODs nearest a particular image pixel.

One significant drawback to conventional MIP-mapping, however, is the amount of memory consumed by the various texture LOD maps. Main memory in the form of a dynamic random access memory (DRAM) or a static random access memory (SRAM) is an expensive and inefficient site for a large texture MIP-map. Each additional level of detail map at a higher level of detail requires four times more memory. For example, a 16×16 texture array having 256 texture picture elements (texels), is four times bigger than an 8×8 texture array which has 64 texels. To put this increase in perspective, a texture MIP-map having six levels of detail requires over 4,096 times more memory than the texture map at the finest resolution. Implementing large texture MIP-maps quickly becomes an expensive luxury. In addition, for large texture MIP-maps, many portions of the stored MIP-map are not used in a display image.

Memory costs become especially prohibitive in photographic texture applications where the source texture, such as, satellite data or aerial photographs, occupy a large storage area. Creating a pre-filtered MIP-map representation of such source texture data further increases memory consumption.

This problem is further exacerbated by the fact that in order to increase the speed at which images are rendered for display, many of the high-performance computer systems contain multiple processors. A parallel, multiple processor architecture typically stores individual copies of the entire MIP-map in each processor memory.

Thus, there is a need to efficiently implement large texture maps for display purposes so as to minimize attendant memory and data retrieval costs. Visual quality must not be sacrificed for memory savings. Final images in an improved texture mapping system need to be virtually indistinguishable from that of images generated by a traditional MIP-map approach.

There is also a need to maintain real-time display speeds even when navigating through displays drawn from large texture maps. For example, flight simulations must still be performed in real-time even when complex and voluminous source data such as satellite images of the earth or moon, are used to form large texture motifs.

## SUMMARY OF THE INVENTION

The present invention pertains to an apparatus and method for providing texture by using selected portions of a texture MIP-map. The selected portions are referred to herein as a clip-map. Texel data relevant to a display image is stored, accessed, and updated efficiently in a clip-map in texture memory.

Entire texture MIP-maps are stored onto one or more mass storage devices, such as hard disk drives, optical disk drives, tape drives, CD drives, etc. According to the present invention, however, only a clip-map needs to be loaded into a more expensive but quicker texture memory (e.g., DRAM). Two dimensional or three dimensional texture data can be used. The clip-map is identified and selected from within a texture MIP-map based upon the display viewer's current eyepoint and field of view. The clip-map is composed of a set of selected tiles. Each tile corresponds to the respective portion of a texture level of detail map at or near the current field of view being rendered for display.

Virtually unlimited, large amounts of texture source data can be accommodated as texture MIP-maps in cheap, mass storage devices while the actual textured image displayed at any given time is readily drawn from selected tiles of corresponding clip-maps stored in one or more texture memories. In one example, the clip-map consists of only 6 million texels out of a total of 1.365 billion texels in a complete texture MIP-map—a savings of 1.36 billion texels! Where texture information is represented as a 8-bit color value, a texture memory savings of 10.9 gigabits (99.6%) is obtained.

According to another feature of the present invention, real-time flight over a large texture map is obtained through efficient updating of the selected clip-maps. When the eyepoint of a viewer shifts, the edges of appropriate clip-map tiles stored in the texture memory are updated along the direction of the eyepoint movement. New texel data for each clip-map tile is read from the mass storage device and loaded into the texture memory to keep the selected clip-map tiles in line with the shifting eyepoint and field of view. In one particularly efficient embodiment, when the eyepoint moves a distance equal to one texel for a particular LOD, one texel row of new texture LOD data is added to the respective clip-map tile to keep pace with the direction of the eyepoint movement. The texel row in the clip-map tile which encompasses texel data furthest from the moving eyepoint is discarded.

In a further feature of the present invention, a substitute texel value is used when an attempt is made to access a texel lying outside of a particular clip-map tile at the most appropriate resolution. The substitute texel value is obtained from the next coarser resolution clip-map tile which encompasses the texel being sought. The substitution texel that is chosen is the one closest to the location of the texel being accessed. Thus, this approach returns usable texel data from a clip-map even when mapping wayward pixels lying outside of a particular clip-map tile. Of course, for a given screen size, the tile size and tile center position can be calculated to guarantee that there would be no wayward pixels.

Finally, in one specific implementation of the present invention, texture processing is divided between a texture generator and a texture memory manager in a computer graphics raster subsystem. Equal-sized square tiles simplify texel addressing. The texture generator includes a LOD generation block for generating an LOD value identifying a clip-map tile for each pixel quad. A texture memory manager readily accesses the texel data from the clip-map using tile offset and update offset information.

Further embodiments, features, and advantages of the present inventions, as well as the structure and operation of the various embodiments of the present invention, are described in detail below with reference to the accompanying drawings.

## BRIEF DESCRIPTION OF THE FIGURES

The accompanying drawings, which are incorporated herein and form a part of the specification, illustrate the present invention and, together with the description, further serve to explain the principles of the invention and to enable a person skilled in the pertinent art to make and use the invention.

In the drawings:

FIG. 1A shows a conventional multi-resolution MIP-map covering four levels of detail.

FIG. 1B shows a conventional example of a polygon perspective of a landscape to which texture MIP-mapping can be applied.

FIG. 2 shows a block diagram of an example computer graphics system implementing the present invention.

FIG. 3 shows a side view of a ten-level texture MIP-map and the selected tiles that constitute a clip-map according to the present invention.

FIG. 4A shows a side view of the first six levels of a clip-map for photographic terrain texture in one example of the present invention.

FIG. 4B shows the progressively larger areas of a terrain texture covered by coarser tiles in the present invention.

FIG. 4C shows three LOD-maps and associated clip-map tile areas relative to an observer's field of view.

FIG. 5 shows the shifting of selected tiles in a clip-map to track a change in the viewer eyepoint.

FIGS. 6A and 6B illustrate an efficient updating of clip-map tiles according to the present invention to follow eyepoint changes.

FIG. 7 shows obtaining a substitute texel value from the next closest clip-map tile having the highest resolution according to the present invention.

FIGS. 8A and 8B are flowcharts describing steps for obtaining a textured display image using a texture clip-map according to the present invention.

FIGS. 9 to 11 are block diagrams illustrating one example of a computer graphics subsystem implementing the present invention.

FIG. 9 shows a raster subsystem including a texture processor having a texture generator and a texture memory manager according to the present invention.

FIG. 10 shows a block diagram of the texture generator in FIG. 9.

FIG. 11 shows a block diagram of the texture memory manager in FIG. 9.

The present invention will now be described with reference to the accompanying drawings. In the drawings, like reference numbers indicate identical or functionally similar elements. Additionally, the left-most digit(s) of a reference number identifies the drawing in which the reference number first appears.

## DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

I. Overview and Discussion

II. Terminology

## I. Overview and Discussion

The present invention provides an apparatus and method for efficiently storing and quickly accessing texel data relevant for displaying a textured image. A large amount of texture source data is stored as a multi-resolution texture MIP-map on one or more mass storage devices. Only a relatively small clip-map representing selected portions of the complete texture MIP-map is loaded into faster, more expensive texture memory. These selected texture MIP-map portions include tiles which contain those texel values at each respective level of detail that are most likely to be mapped to pixels being rendered for display.

When the eyepoint or field of view is changed, the tiles stored in the texture memory are updated accordingly. In one efficient embodiment for updating the clip-map in real-time, new texel data is read from the mass storage device and loaded into the fringes of tiles to track shifts in the eyepoint. To maintain the size of the clip-map, tile texel data corresponding to locations furthest away from a new eyepoint is discarded. Anomalous attempts to access a texel lying outside of a particular clip-map tile are accommodated by utilizing a substitute texel value obtained from the next coarser resolution clip-map tile which encompasses the texel being sought.

## II. Terminology

To more clearly delineate the present invention, an effort is made throughout the specification to adhere to the following term definitions as consistently as possible.

The term "texture map" refers to source data representing a particular texture motif at its highest resolution. A "texture MIP-map" and equivalents thereof such as a "MIP-map of texel data" are used to refer to conventional multum in parvo MIP-map representations of a texture map at successive multiple levels of details (LOD), that is, varying degrees of resolution.

On the other hand, "clip-map" is used to refer to portions of a MIP-map selected according to the present invention. Thus, a clip-map is a multi-resolution map made up of a series of tiles wherein at least some of the tiles represent smaller, selected portions of different levels of detail in a MIP-map. When two-dimensional texture data sets are used, these tiles are two-dimensional texel arrays. When three-dimensional texture data sets are used, these tiles are three-dimensional texel arrays, i.e. cubes.

Finally, texel array dimensions are given in convenient 1k×1k, 2k×2k, etc., shorthand notation. In actual implementations using digital processing, 1k equals 1,024, 2k equals 2,048, etc.

## III. Example Environment

The present invention is described in terms of a computer graphics display environment for displaying images having textures drawn from multi-resolution texture MIP-maps. Moreover, sophisticated texture motifs covering a large area, such as satellite data and aerial photographs, are preferred to fully exploit the advantages of the clip-map system and method described herein. As would be apparent to a person skilled in the pertinent art, the present invention applies generally to different sizes and types of texture patterns limited only by the imagination and resources of the user.

Although the present invention is described herein with respect to two-dimensional texture mapping, the present invention can be extended to three-dimensional texture mapping when the requisite additional software and/or hardware resources are added. See e.g. the commonly-assigned, U.S. patent application Ser. No. 08/088,716, now U.S. Pat. No. 5,490,240 (Attorney Docket No. 15-4-99.00 (1452.0140000), filed Jul. 9, 1993, entitled "A System and Method of Generating Interactive Computer Graphic Images Incorporating Three Dimensional Textures," by James L. Foran et al. (incorporated herein by reference in its entirety).

## IV. Computer Graphics System

Referring to FIG. 2, a block diagram of a computer graphics display system **200** is shown. System **200** drives a graphics subsystem **220** for generating textured display images according to the present invention. In a preferred implementation, the graphics subsystem **220** is utilized as a high-end, interactive computer graphics workstation.

System **200** includes a host processor **202** coupled through a data bus **201** to a main memory **204**, read only memory (ROM) **206**, and mass storage device **208**. Mass storage device **208** is used to store vast amounts of digital data relatively cheaply. For example, the mass storage device **208** can consist of one or more hard disk drives, floppy disk drives, optical disk drives, tape drives, CD ROM drives, or any number of other types of storage devices having media for storing data digitally.

Different types of input and/or output (I/O) devices are also coupled to processor **202** for the benefit of an interactive user. An alphanumeric keyboard **210** and a cursor control device **212** (e.g., a mouse, trackball, joystick, etc.) are used to input commands and information. The output devices include a hard copy device **214** (e.g., a laser printer) for printing data or other information onto a tangible medium. A sound recording or video option **216** and a display screen **218** can be coupled to the system **200** to provide for multimedia capabilities.

Graphics data (i.e. a polygonal description of a display image or scene) is provided from processor **202** through data bus **201** to the graphics subsystem **220**. Alternatively, as would be apparent to one skilled in the art, at least some of the functionality of generating a polygonal description could be transferred to the computer graphics subsystem as desired.

Processor **202** also passes texture data from mass storage device **208** to texture memory **226** to generate and manage a clip-map as described below. Including the software and/or hardware in processor **202** for generating and managing clip-maps is one example for implementing the present invention. Separate modules or processor units for generating and managing a texture clip-map could be provided along data bus **201** or in graphics subsystem **220**, as would be apparent to one skilled in the art considering this description.

The graphics subsystem **220** includes a geometry engine **222**, a raster subsystem **224** coupled to a texture memory **226**, a frame buffer **228**, video board **230**, and display **232**.

Processor **202** provides the geometry engine **222** with a polygonal description (i.e. triangles) of a display image in object space. The geometry engine **222** essentially transforms the polygonal description of the image (and the objects displayed therein) from object space (also known as world or global space) into screen space.

Raster subsystem **224** maps texture data from texture memory **226** to pixel data in the screen space polygonal description received from the geometry engine **222**. Pixel data and texture data are eventually filtered, accumulated, and stored in frame buffer **228**. Depth comparison and other display processing techniques can be performed in either the raster subsystem **224** or the frame buffer **228**. Video unit **230** reads the combined texture and pixel data from the frame buffer **228** and outputs data for a textured image to screen display **232**. Of course, as would be apparent to one skilled in the art, the output image can be displayed on display **218**, in addition to or instead of display **232**. The digital data representing the output textured display image can also be saved, transmitted over a network, or sent to other applications.

The present invention is described in terms of this example high-end computer graphics system environment. Description in these terms is provided for convenience only. It is not intended that the invention be limited to application in this example environment. In fact, after reading the following description, it will become apparent to a person skilled in the relevant art how to implement the invention in alternative environments.

## V. Texture MIP-Mapping

In the currently preferred embodiment of the present invention, large texture maps are stored on one or more mass storage devices **208**. A MIP-map representation of the texture maps can either be pre-loaded onto the mass storage device **208** or can be computed by the processor **202** and then stored onto mass storage device **208**. Two-dimensional or three-dimensional texture data sets are accommodated.

As is well-known in computer graphics design, the filtering and MIP-structure development necessary to derive and efficiently store the successive levels of detail for a texture MIP-map can be effectuated off-line prior to run-time operation. In this way, high-quality filtering algorithms can be utilized over a large texture map or database without hindering on-line image display speed and performance. Alternatively, if a less flexible but fast approach is acceptable, hardware can be used to produce the successive coarser levels of detail directly from input texture source data.

Under conventional texture mapping techniques, even if texture data were to be accessed from a remote, large texture MIP-map, the rendering of a textured image for display in real-time would be impractical, if not impossible. The present invention, however, realizes the advantages of accommodating large texture MIP-maps in one or more mass storage devices **208** without reducing texture access time. A relatively small clip-map representing only selected portions of a complete texture MIP-map is stored in a texture memory **226** having a fast rate of data return. In this way, texture memory **226** acts as a cache to provide texture rapidly to the raster subsystem **224**.

This hierarchical texture mapping storage scheme allows huge texture MIP-maps to be stored rather inexpensively on the mass storage device **208**. Based on the viewer eye point and/or field of view, only selected portions of a texture MIP-map corresponding to the texture motif to be rendered for display need to be loaded into the texture memory **226**. In this manner, large 2-D or 3-D texture MIP-maps can be used to provide texture rather inexpensively, yet the textured images can be rendered in real-time.

## VI. Selecting Portions of a Texture MIP-Map

The process for determining which portions of a complete texture MIP-map are to be loaded from mass storage devices **208** into texture memory **226** to form a clip-map will now be described in more detail. FIG. **3** shows a side view of a complete texture MIP-map **330** having ten conventional levels of details (not shown to actual geometric scale). The levels of detail **300** to **309** each correspond to successively coarser resolutions of a texture map. The highest level of detail LOD[0] corresponds to the finest resolution texel map **300**. Each subsequent level of detail map **301** to **309** are filtered to have half the resolution of the preceding level of detail. Thus, each coarser level of detail covers an area of the texture map four times greater than the preceding level of detail.

FIG. **3** further illustrates the selected portions of the texture MIP-map **330** constituting a clip-map **340** according to the present invention. Clip-map **340** consists of relatively small tiles **310–319** which are regions of the levels of detail maps **300–309**. The actual size and shape of these tiles depends, inter alia, on the eye point and/or field of view of the display viewer. Each of these tiles must substantially encompass a potential field of view for a display view. To simplify addressing and other design considerations, equal-sized square tiles, i.e. square texel arrays, are used which can each be addressed relative to a common, fixed central eye point X and a center line O running through the clip map. For 3-D texture, square cubes consisting of a 3-D texel array are used.

Clip-map **340** essentially consists of a set of tiles, including a cubical part (**310–316**) and a pyramidal part (**317–319**). The cubical part consists of a shaft of tiles (**310–316**) of equal size. In the pyramidal part, the tiles consist of the actual level of detail maps (LOD[M+1]-LOD[N]). The pyramidal part begins at the first level of detail map (LOD[M+1]) which is equal to or smaller than a tile in the cubical part and extends down to a 1×1 texel (LOD[N]).

The reduced memory requirements for storing a clip-map instead of a complete texture MIP-map are clear. A complete, conventional 2-D texture MIP-map having dimensions given by "size in s" and "size in t" uses at least the following memory M:

$$M_{Texture\ MIP\text{-}map}=4/3 * (\text{size in s}) * (\text{size in t}) * \text{texel size (in bytes)}.$$

The smaller, clip-map example having equal-sized, square tiles in the cubical part only uses the following memory M:

$$M_{Texture\ clip\text{-}map}=[(\text{number of levels in cubical part} * (\text{tile size })^2)+ 4/3 * (\text{size in s of pyramidal part}) * (\text{size in t of pyramidal part})] * \text{texel size (in bytes)}.$$

Ten levels of detail are shown in FIG. **3** to illustrate the principle of the present invention. However, a smaller or greater number of levels of detail can be utilized. In a preferred example of the present invention, **16** levels of detail are supported in a high-end interactive computer graphics display workstation.

## VII. Photographic Terrain Texture

Substantial reductions in memory costs and great improvements in real-time display capability are immedi-

APPENDIX BB

ately realized by using a clip-map to render textured images. These advantages are quite pronounced when large texture maps such as a photographic terrain texture are implemented.

For example, source data from satellites covering 32 or more square kilometers of a planet or lunar surface is available. Such terrain can be adequately represented by a photographic texture MIP-map **430** having sixteen level of detail maps. The six highest resolution LOD maps **400–405** and tiles **410–415** are shown in FIG. 4A. The highest resolution level LOD[0] consists of a 32k×32k array of texels. Successive level of details LOD[1]-LOD[5] correspond to the following texel array sizes: 16k×16k, 8k×8k, 4k×4k, 2k×2k, and 1k×1k. The remaining pyramidal part not shown consists of texel arrays 512×512, 256×256, . . . 1X1. Thus, a total of 1.365 billion texels must be stored in mass storage device **208**.

The size of the clip-map, however, is a function of the field of view and how close the observer is to the terrain. Generally, a narrow field of view requires a relatively small tile size increasing the memory savings. For example, the higher resolutions in the cubical part of clip-map **440** need only consist of 1k×1k tiles **410–414** for most close perspective images. The entire clip-map **440** then contains 6 million texels—a savings of 1.36 billion texels! Where texture information is represented as a 8-bit color value, a memory savings of 10.9 gigabits (99.6%) is obtained.

By storing the smaller clip-map **440** in texture memory **226**, further advantages inherent in a hierarchial memory system can be realized. The complete texture MIP-map **430** of 1.365 billion texels can be stored in cheap mass storage device **208** while the small clip-map **440** is held in a faster texture memory **226**, such as DRAM or SRAM. Sophisticated texel data can then be used to render rich textured images in real-time from the easily-accessed clip-map **440**. For example, a screen update rate of 30 to 60 Hz, i.e. 1/30 to 1/60 sec., is realized. The transport delay or latency of 1 to 3 frames of pixel data is approximately 10 to 50 msec. The above screen update rates and latency are illustrative of a real-time display. Faster or slower rates can be used depending on what is considered real-time in a particular application.

As shown in FIG. 4B, the texel data stored in clip-map **440** can provide texture over a large display image area. For example, each high resolution texel of LOD[0] in a 32k×32k texel array can cover one square meter of geographic area in the display image. The 1k×1k tile **410** contains texel data capable of providing texture for a display image located within one square kilometer **420**.

Moreover, typical images are displayed at a perspective as described with respect to FIG. 1B. The highest texel resolution included in tile **410** need only be used for the smaller areas magnified in a foreground region. Because of their progressively coarser resolution, each successive tile **411–414** covers the following broader areas 4 square kilometers (**421**), 16 square kilometers (**422**), 64 square kilometers (**423**), and 256 square kilometers (**424**), respectively. The complete 1,024 square kilometer area covered by texel data in tile **415** is not shown due to space limitations.

Even though the tiles may be equal-sized texel arrays, each tile covers a geometrically large area of a texture map because of filtering, albeit at a coarser level of detail. FIG. 4C shows a perspective view of regions **420'** to **422'** covered by tiles **410** to **412** within the respective first three level of detail maps **400** to **402**. Each of the regions **420'** to **422'** are aligned along a center line O' stemming from an eyesight location X' to the center of the coarsest 11X1tile (not shown) in the pyramidal part of the clip-map **440**.

Thus, the clip-map **440** contains sufficient texel data to cover larger minified areas in the background of a display where coarser texture detail is appropriate. As a result, high quality textured display images, in perspective or warped, are still obtained for large texture patterns by using texel data from a clip-map.

## VIII. Updating the Clip-Map During Real-Time Operation

The discussion thus far has considered only stationary eyepoints (X or X'). Many graphics display applications, such as flight applications over a textured terrain, present a constantly changing display view. As is well-known in graphics design, the display view can simulate flight by constantly moving the eyepoint along a terrain or landscape being viewed. Such flight can be performed automatically as part of a program application, or manually in response to user input such as mouse or joystick movement. Hyperlinks or jumps can be selected by a user to abruptly select a new viewpoint and/or field of view.

Regardless of the type of movement, today's user demands that new views be displayed in real-time. Delays in mapping texture data directly from large texture maps are intolerable. Reloading an entire new texture MIP-map for a new display viewpoint is often impractical.

As shown in FIG. 5, when the eyepoint X shifts to a new point X' for a new display view, the texel data forming the clip-map **340** must similarly shift to track the new field of view along the axis O'. According to one feature of the present invention, portions of the texture MIP-map **330** forming a new "slanted" clip-map **540** are loaded into the texture memory **226**. The "slanted" clip-map **540** is necessarily drawn in a highly stylized and exaggerated fashion in FIG. **5** in the interest of clarity. Actual changes from one display view to the next are likely less dramatic. The actual tiles in the slanted clip-map would also be more staggered if the level of detail maps were drawn in true geometric proportion.

New tiles can be calculated and stored when the eyepoint and/or field of view changes to ensure that clip-map **540** contains the texel data which is most likely to be rendered for display. Likewise, the size and/or shape of the tiles can be altered to accommodate a new display view.

Texel data can be updated by loading an entire new slanted clip-map **540** from mass storage device **208** into texture memory **226**. Full texture loads are especially helpful for dramatic changes in eyepoint location and/or field of view.

## IX. Efficiently Updating the Clip-Map

According to a further feature of the present invention, subtexture loads are performed to efficiently update texel data at the edges of tiles on an on-going basis. For example, as the eyepoint shifts, a new row of texel data is added to a tile in the direction of the eyepoint movement. A row located away from a new eyepoint location is discarded. Coarser tiles need not be updated until the eyepoint has moved sufficiently far to require a new row of texel data. Thus, the relatively small amount of texel data involved in a subtexture loads allows clip-map tiles to be updated in real-time while maintaining alignment with a moving eyepoint X.

For example, FIGS. **6**A and **6**B illustrate, respectively, the areas of texel data covered by clip-map tiles before and after a subtexture load in the highest resolution tile **410**. Each of the areas **620** to **622** correspond to the regions of a texture map covered by 1k×1k tiles **410**–**412**, as described earlier with respect to the terrain of FIG. **4B**. A field of view **600** along the direction O marks the display area which must be covered by texture detail. Hence, only those texels residing within triangle **600** need to be stored or retained in the texture memory **226**. Texels around the fringes of triangle **600**, of course, can be added to provide additional texture data near the edges of a display image.

As shown in FIG. **6B**, each time the eyepoint advances one pixel-width (the pixel-width is exaggerated relative to the overall tile size to better illustrate the updating operation), a new texel row **601** located forward of the eyepoint is loaded from mass storage device **208** into the highest resolution tile **410** in texture memory **226**. The texel row **602** furthest from the new eyepoint X' is then discarded. In this way, tile **410** contains texel data for an area **620'** covering the new display area **600**. For small changes, then, coarser tiles (**411**, **412**, etc.) do not have to be updated. Because an equal amount of texels are discarded and loaded, the tile size (and amount of texture memory consumed by the clip-map) remains constant.

When the eyepoint moves a greater distance, texture data is updated similarly for the tiles at coarser LODs. Because two texels from an LOD are filtered to one texel in each direction s or t in texture space to form a successive LOD, the minimum resolution length for each LOD[n] is $2^n$ pixels, where n=0 to N. Accordingly, the tiles for LOD[1], LOD[2] . . . LOD[4] in the cubical part of a clip-map **440** are only updated when the eyepoint has moved two, four, eight, and sixteen pixels respectively. Because each level of detail in the pyramidal part is already fully included in the tile **415**, no updating is necessary in theory. To simplify an updating algorithm, however, when tiles in either the cubical part or the pyramidal part reach the end of a level of detail map, garbage or useless data can be considered to be loaded. Substitute texel data drawn from a coarser tile would be used instead of the garbage data to provide texture detail in those regions.

According to the present invention, then, the amount of texel data which must be loaded at any given time to update clip-map **540** is minimal. Real-time display operation is not sacrificed.

Texel data can be updated automatically and/or in response to a user-provided interrupt. Subtexture loads are further made in advance of when the texel data is actually rendered for display.

Finally, a check can be made to prevent attempts to draw an image using texel data which is being updated. Fringe regions are defined at the edges of tiles in the cubical part of the clip-map. The fringes include at least those texels being updated. To better accommodate digital addressing, it is preferred that the fringes consist of a multiple of eight texels. For example, in a 1k×1k tile having 1,024 texels on a side, eight texels at each edge form the fringe regions leaving 1,008 texels available to provide texture. Any attempt to access a texel in the fringe is halted and a substitute texel from the next coarsest level of detail is used instead. In this way, accesses by the raster subsystem **224** to specific texel data do not conflict with any texel updating operation.

## X. Substitute Texel Data

According to a further feature of the present invention, substitute texel data is returned for situations where pixel data lying outside of a clip-map tile at a desired level of detail is to be mapped. When the raster subsystem **224** seeks to map a pixel not included in a clip-map tile corresponding to the desired level of detail, there is a problem in that the texture memory **226** cannot provide texel data at the most appropriate resolution at that particular pixel. The likelihood of such a situation arising can be minimized by brutishly mandating larger tile sizes. Of course, for a given screen size, the tile size and center position can be calculated to guarantee that there would be no wayward pixels.

The inventors, however, have discovered a more elegant solution which does not require an unnecessary expansion of the clip-map to accommodate wayward pixels. As shown in FIG. **7**, substitute texel data is derived for a pixel **700** lying outside of a clip-map **340**. A line **702** is first determined between the out-of-bounds pixel **700** and the apex of the pyramid part (center of the coarsest 1×1 texel tile LOD[N]). At some point **704**, this line **702** intersects the shaft of the clip-map. Substitute texel data **706**, covering pixel **700**, is then drawn from the nearest, coarser tile **314**.

In practice, when the resolution between levels of detail varies by a factor of 2, substitute texel data is easily drawn from the next coarser level of detail by shifting a texel address one bit. Operations for obtaining memory addresses for a texel located in a clip-map tile at the desired level of detail are further described below. Arithmetic and shifting operations to obtain a substitute texel memory address from the tile at the next nearest level of detail which covers the sought pixel is also described below.

By returning substitute texel having the next best level of detail, the overall texture detail remains rich as potential image degradation from pixels lying outside the clip-map is reduced. Moreover, by accommodating wayward pixels, greater latitude is provided in setting tile size, thereby, reducing the storage capacity required of texture memory **226**.

## XI. Overall Clip-Map Operation

FIGS. **8A** and **8B** are flowcharts illustrating the operation of the present invention in providing texture data from a clip-map to display images.

First, a texture MIP-map representation of a texture map is stored in an economical memory such as, a mass storage device **208** (step **810**). Processor **202** can perform pre-filtering to calculate a texture MIP-map based on a texture map supplied by the user. Alternatively, the texture MIP-map can be loaded and stored directly into the mass storage device **208**.

Portions of the MIP-maps are then selected based on a particular field of view and/or eyepoint location to form a clip-map (step **820**). The clip-map is stored in a faster texture memory **226** (step **830**). Using texel data stored in the clip-map, texture can be mapped quickly and efficiently to corresponding pixel data to display a new textured image (step **840**).

To track changes in the field of view and/or eyepoint location of a display view, only the fringes of the tiles in the clip-map are updated (steps **850** and **860**). Such subtexture loads can be performed in real-time with minimal processor overhead. Unlike conventional systems, an entire texture load operation need not be performed.

FIG. **8B** shows the operation in step **840** for processing texture for a new image in greater detail. In step **841**, a description of polygonal primitives and texture coordinates is input. Triangle vertices are typically provided in screen space by a geometry engine **222**. A raster subsystem **224** then maps texture coordinates at the vertices to pixels.

Texture coordinates can be calculated for two-dimensional or three-dimensional texture LOD maps.

In step **842**, an appropriate level of detail is calculated for each pixel according to standard LOD calculation techniques based on the pixel dimension and texel dimension. A level of detail map closest to this appropriate level of detail is determined for the pixel (step **843**).

Texture closest to the appropriate level of detail is then obtained from the finest resolution tile in the clip-map which actually encompasses a texel corresponding to the pixel (steps **844** to **847**). First, a check is made to determine whether texel data for the pixel is included within a tile corresponding to the appropriate level of detail map determined in step **843**. Because the tiles are determined based on eyepoint location and/or field of view, texel data for a pixel is likely found within a tile at an appropriate level of detail. In this case, a texel is accessed from the corresponding tile and mapped to a corresponding pixel (step **845**).

As described earlier with respect to FIG. **7**, when a texel at the appropriate level of detail is not included within a corresponding tile, a coarser substitute texel is accessed. The substitute texel is chosen from the tile at the nearest level of detail which encompasses the originally-sought texel (step **846**). Texels mapped to pixels in step **845** and substitute texels mapped to corresponding pixels in step **846** are accumulated, filtered, and stored in a frame buffer **228** for subsequent display (step **847**).

Steps **841** to **847** are repeated for each input polygon description until a complete display image has been mapped to texel data and stored in the frame buffer **228**.

As would be apparent to one skilled in computer-generated textured graphics, the "clip-map" process described with respect to FIGS. **8**A and **8**B, can be carried out through firmware, hardware, software executed by a processor, or any combination thereof.

## XII. Specific Implementation

FIGS. **9** to **11** illustrate one preferred example of implementing texture processing using a clip-map within computer graphics subsystem **220** according to the present invention. FIG. **9** shows a block diagram of a texture processor **900** within raster subsystem **224**. Texture processor **900** includes a texture generator **910** and a texture memory manager **920**. FIG. **10** shows a block diagram of the texture generator **910**. FIG. **11** shows a block diagram of a texture memory manager **920**. The operation of texture processor **900** in managing a clip-map to provide a texture display image will be made even more clear by the following description.

As shown in FIG. **9**, raster subsystem **224** includes texture processor **900**, pixel generator **940**, and texture filter **950**. The texture processor **900** includes a texture generator **910** coupled to a texture memory manager **920**. Texture memory manager **920** is further coupled to texture memory (DRAM) **930**.

Both the texture generator **910** and the pixel generator **940** are coupled to the geometry engine **222** via a triangle bus **905**. As explained earlier with respect to FIG. **2**, polygonal primitives (i.e. triangles) of an image in screen space (x,y), are output from the geometry engine **222**. Texture generator **910** outputs specific texel coordinates for a pixel quad and an appropriate LOD value based on the triangle description received from geometry engine **222** and the (x,y) screen space coordinates of the pixel quad received from pixel generator **940**. The LOD value identifies the clip-map tile in DRAM **930** which includes a texel at the desired level of detail. When a substitute texel must be used as described above, the LOD value identifies the clip-map tile at the closest coarser level of detail which covers the sought pixel data.

Texture memory manager **920** retrieves the texel or substitute texel from the clip-map stored in DRAM **930**. The retrieved texel data is then sent to a texture filter **950**.

Texture filter **950** filters texel data sent by the texture memory according to conventional techniques. For example, bi-linear and higher order interpolations, blending, smoothing, and texture sharpening techniques can be applied to textures to improve the overall quality of the displayed image. Texture filter **950** (or alternatively the frame buffer **228**) further combines and accumulates the texel data output from texture memory manager **930** and the corresponding pixel data output by the pixel generator **940** for storage in frame buffer **228**.

FIG. **10** shows component modules **1010–1060** forming texture generator **910**. Blocks **1010** to **1040** represent graphics processing modules for scan converting primitives. For purposes of this example, it is presumed that the primitive description consists of triangles and that pixel data is processed as 2×2 pixel quads. Module **1010** rotates and normalizes the input triangles received across triangle bus **905**. Module **1020** generates iteration coefficients. Scan conversion module **1030** then scan converts the triangles based on the outputs of modules **1010** and **1020** and the (x,y) coordinates output from a stepper (not shown) in pixel generator **940**. Texture coordinates for each pixel quad are ultimately output from scan conversion module **1030**. Normalizer and divider **1040** outputs normalized texture coordinates for the pixel quad. Such scan conversion processing is well-known for both two-dimensional and three-dimensional texture mapping and need not be described in further detail.

LOD generation block **1050** determines an LOD value for texture coordinates associated with a pixel quad. Compared to LOD generation blocks used in conventional texture MIP-mapping, LOD generation block **1050** is tailored to consider the contents of the clip-map and whether a substitute texel is used. The LOD value output from LOD generation block **1050** identifies the clip-map tile which includes texels or substitute texels covering a pixel quad.

LOD generation block **1050** essentially performs two calculations to derive the LOD value, as described previously with respect to steps **842** to **846**. LOD generation block **1050** first calculates an appropriate level of detail for the pixel quad (or pixel) according to standard LOD generation methods based on the individual pixel size and texel dimension. A level of detail map closest to the calculated level of detail is determined.

According to the present invention, then, a check is made to determine whether texel data for the pixel quad is included within a tile corresponding to the appropriate level of detail. When a texel is included within a tile at the appropriate level of detail, a LOD value corresponding to this tile is output. Otherwise, a LOD value is output identifying a tile at a lower level of detail which includes a substitute texel, as described earlier.

The above discussion largely refers to a pixel quad (i.e. a 2×2 pixel array). However, as would be apparent to one skilled in the art, the invention is not limited to use of a pixel quad. Using a pixel quad merely reduces the number of calculations and the amount of data which must be tracked. If desired, a separate LOD value could be calculated by LOD block **1050** for each pixel.

Other modules (not shown) can further use the pixel quad and LOD value output from LOD generation block **1050** to

APPENDIX BB

perform supersampling, clamping, or other graphics display optimizing processes.

The present invention takes further advantage of the use of a pixel quad to reduce the amount of data required to be sent from the texture generator **910** to the texture memory manager **920**. Quad coordinate compression module **1060** compresses the texture coordinates of a pixel quad and the LOD value data sent to one or more texture memory managers **920**. In particular, in a 2×2 pixel quad, texture coordinates for one pixel are needed but the other three pixels can be defined relative to the first pixel. In this way, only the differences (i.e. the offsets) between the centers of the other three pixels relative to the center of the first pixel need to be transmitted.

FIG. 11 shows component modules forming texture memory manager **920**. Module **1110** decompresses texture coordinates of a pixel quad and LOD value information received from texture generator **910**. Texture coordinates for one pixel are received in full. Texture coordinates for the other three pixels can be determined by the offsets to the first pixel texture coordinates. The LOD value is associated with each pixel.

The texture coordinates sent from texture generator **910** are preferably referenced to the global texture MIP-map stored and addressed in mass storage device **208**. Address generator **1120** translates the texture coordinates for the pixel quad from the global texture MIP-map space of mass storage device **208** to texture coordinates specific to the clip-map stored and addressed in DRAM **930**. Alternatively, such translation could be carried out in the texture generator **910** depending upon how processing was desired to be distributed.

Address generator **1120** first identifies a specific tile at the level of detail indicated by the LOD value. To translate texture coordinates from global texture MIP-map space to the specific tile, the address generator **1120** considers both (1) the offset of the specific tile region relative to a complete level of detail map (i.e. tile offset) and (2) the center eyepoint location of a tile (i.e. update offset).

Memory addresses corresponding to the specific texture coordinates are then sent to a memory controller **1130**. Memory controller **1130** reads and returns texture data from DRAM **930**, through address generator **1120**, to texture filter **950**.

As would be apparent to one skilled in the art from the foregoing description, a conventional LOD value can be sent from the LOD generation block **1050** without regard to the selected portions of the texture MIP-map stored in the clip-map. The steps for determining whether a texel is within a tile and for determining a LOD value for a substitute texel would then be carried out at the texture memory manager **920**.

## XIII. Square Clip-Map Tiles Example

Selecting, managing and accessing texel data in a clip-map will now be discussed with respect to the specific square clip-map **440**. According to another feature of the present invention, each tile in DRAM **930** is configured as a square centered about a common axis stemming from the eyepoint location. Each tile has a progressively coarser resolution varying by factor of 2. These restrictions simplify texel addressing for each tile in the cubical part of a clip-map considerably. The additional cost in hardware and/or software to address the tiles is minimal both when the tiles are initially selected and after any subtexture loads update the tiles.

By selecting square tiles to form the initial clip-map stored in DRAM **930**, the work of a processor **202** (or a dedicated processor unit in the graphics subsystem **220**) is straightforward. First, the center of the finest level (LOD[0]) is chosen. Preferably the center ($s_{center}$,$t_{center}$) is defined as integers in global (s,t) texture coordinates. A finest resolution tile **410** is then made up from the surrounding texel data in LOD[0] map **400** within a predetermined distance d from the center point. All the other tiles for the levels of the cubical part (LOD[1]-LOD[M]) are established by shifting the center position down along the eyepoint. Thus, the texture coordinates ($s_{centern}$,$t_{centern}$) for a tile at level of detail LOD[n] are given by:

$$s_{centern}=s_{center>>n}$$

$$t_{centern}=t_{center>>n}.$$

where >>n denotes n shift operations. Texel data for the other tiles **401–404** is likewise swept in from regions a predetermined distance d in the s and t direction surrounding each center point.

Simple subtraction and comparison operations are carried out in texture generator **910** to determine whether a texel for a pixel quad is within a tile at an appropriate LOD. The finest, appropriate level of detail is determined by conventional techniques (see steps **842** and **843**). The additional step of checking whether the desired texel for a pixel quad is actually included within a tile at that LOD (step **844**) can be performed by calculating the maximum distance from four sample points in a pixel quad to the center of the tile. To be conservative, s and t distances for each of four sample points ($s_0$, $t_0$) . . . ($s_3$, $t_3$) can be calculated within a LOD generation block **1050** as follows:

$$\begin{aligned} s_0\,dist &= |s_{centern}-s_0| \\ t_0\,dist &= |t_{centern}-t_0| \\ &\dots \\ s_3\,dist &= |s_{centern}-s_3| \\ t_3\,dist &= |t_{centern}-t_3| \end{aligned}$$

where the tile center point at a LOD value n is given by ($s_{centern}$, $t_{centern}$)

Because the four samples of a pixel quad are strongly related, performing only two of the above subtractions is generally sufficient. Maximum distances $s_{max}$ and $t_{max}$ for a pixel quad are then determined by comparison. The use of square tiles means only one maximum distance in s or t needs to be calculated.

Based on the maximum distances in s and t, the finest available tile including texel or substitute texel data for the pixel quad is determined in a few arithmetic operations. If the constant size of the tiles is defined by $s_{tile}$ and $t_{tile}$ where the tile size equals ($2^{s\ tile}$, $2^{t\ tile}$) the finest available LOD value is given by the number of significant bits (sigbits) as follows:

$$\text{LOD } s \text{ finest=sigbits } (s_{max})-s_{tile};$$

$$\text{LOD } t \text{ finest=sigbits } (t_{max})-t_{tile}.$$

As would be apparent to one skilled in the art, LOD generation block **1050** can perform the above calculations and output the greater of the two numbers as the LOD value identifying the appropriate finest resolution tile containing texel or substitute data.

Finally, in addition to reducing the work of LOD generation block **1050**, the restrictive use of equal-sized square

APPENDIX BB

tiles and power of two changes in resolution between tiles simplifies the work of the texture memory manager **920**. Address generator **1120** can translate global texture coordinates referencing LOD maps to specific tile texture coordinates in the cubical part of the clip-map by merely subtracting a tile offset and an update offset. The tile offset represents the offset from the corner of an LOD map to the corner of a tile. The update offset accounts for any updates in the tile regions which perform subtexture loads to track changes in the eyepoint location and/or field of view.

Thus, an address generator **1120** can obtain specific s and t tile coordinates ($s_{fine}$, $t_{fine}$) for a fine tile having a level of detail equal to the LOD value provided by the texture generator **910** as follows:

$$s_{fine}=s_{TG}-s\text{ tile offset }_{LOD\ value}-s\text{ update offset}_{LOD\ value}$$

$$t_{fine}=t_{TG}-t\text{ tile offset}_{LOD\ value}-t\text{ update offset}_{LOD\ value},$$

where $s_{TG}$ and $t_{TG}$ represent the global s and t texture coordinates provided by a texture generator **910**, s and t tile offset $_{LOD\ value}$ represent tile offset values in s and t for a tile at the LOD value provided by the texture generator **910**, and s and t update offset$_{LOD\ value}$ represent update offset values in s and t for a tile at the LOD value.

Some texture filters and subsequent processors also use texel data from the next coarser level of detail. In this case, texture memory manager **920** needs to provide texel data from the next coarser tile as well. Once the specific $s_{fine}$ or $t_{fine}$ coordinates are calculated as described above s and t coordinates ($s_{coarse}$, $t_{coarse}$) for the next coarser tile are easily calculated.

In particular, the global texture coordinates ($s_{TG}$, $t_{TG}$) are shifted one bit and reduced by 0.5 to account for the coarser resolution. The tile offset and update offset values (tile offset $_{LODcoarse}$ and update offset $_{LODcoarse}$) for the next coarser tile are also subtracted for each s and t coordinate. Thus, address generator **1120** determines specific texture coordinate in the next coarser tile as follows:

$$s_{coarse}=\text{trunc}[(s_{TG}>>1)-0.5)]-s\text{ tile offset }_{LODcoarse}-s\text{ update offset}_{LOD\ coarse};\text{ and}$$

$$t_{coarse}=\text{trunc}[(t_{TG}>>1)-0.5)]-t\text{ tile offset}_{LOD\ coarse}-t\text{ update offset}_{LOD\ coarse}.$$

## XIV. Conclusion

While specific embodiments of the present invention have been described above, it should be understood that they have been presented by way of example only, and not limitation. It will be understood by those skilled in the art that various changes in form and details may be made therein without departing from the spirit and scope of the invention as defined in the appended claims. Thus, the breadth and scope of the present invention should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.

What is claimed is:

1. A computer graphics raster subsystem for providing texture from a texture pattern to an image to be rendered for display in real-time comprising:

texture memory for storing a select portion of a texture map representation of said texture pattern, said select texture map portion containing texture data at multiple levels of detail to substantially cover said image in a display view;

texture mapping means for mapping texture data from said select texture map portion stored in said texture memory to corresponding pixel data defining said display image; and

clip-map updating means for updating edges of said select texture map portion to track changes in the location of the eyepoint in real-time.

2. The system of claim 1, wherein said texture memory stores said texture data in a two-dimensional or a three-dimensional texel array.

3. A computer graphics processing system for providing texture from a texture pattern in a display image, comprising:

first texture memory for storing a texture map, said texture map having multiple level of detail maps for storing texels representing the texture pattern filtered at successively coarser resolutions;

clip-map selecting means for selecting a clip-map within said texture map, said clip-map consisting of a set of tiles corresponding to regions of said level of detail maps substantially covering the display image;

second texture memory for storing said clip-map;

texture processor means for retrieving texels from said clip-map which map to pixel data forming the display image; and

clip-map updating means for updating texels at edges of said tiles stored in said second texture memory to track changes in the location of an eyepoint.

4. The computer graphics system of claim 3, wherein said first and second texture memory constitute a hierarchial memory arrangement relative to said texture processor means, said texture processor means accesses texels stored in said second texture memory faster than said first texture memory.

5. The computer graphics system of claim 3, wherein:

said first texture memory comprises a mass storage device, and

said second texture memory comprises at least one of a static random access memory (SRAM) device and a dynamic random access memory (DRAM) device.

6. The computer graphics system of claim 3, wherein said clip-map selecting means determines the area of the texture pattern covered by each tile based on the location of an eyepoint of the display image.

7. The computer graphics system of claim 3, wherein said clip-map updating means updates texels at edges of said tiles stored in said second texture memory to track changes in the location of a new eyepoint for a new display image.

8. The computer graphics system of claim 7, wherein said clip-map updating means discards texels from said second texture memory and loads texels from said first texture memory into said second texture memory; said texels being discarded from at least one tile edge located furthest from said new eyepoint and said texels being loaded into at least one tile edge located closer to said new eyepoint.

9. The computer graphics subsystem of claim 3, wherein, said texture processor means further retrieves coarser substitute texels from said clip-map.

10. The computer graphics system of claim 3, wherein textured display images are output for display in real-time.

11. The computer graphics system of claim 3, wherein said clip-map contains over 99% less texels than said texture map.

12. The computer graphics system of claim 3, wherein said texture processor means comprises:

a texture generator; and

a texture memory manager coupled between said texture generator and said second texture memory;

wherein said texture generator includes a texture coordinate generator for generating texel coordinates for each pixel and a LOD generator for generating a LOD value for each pixel, said LOD value identifies the tile which covers the pixel and has texel dimensions closest in size to the pixel, said texel coordinates and LOD value being output for each pixel to said texture memory manager; and

wherein said texture memory manager retrieves texels from said clip-map for combining with said pixels to form a textured display image.

13. The system of claim 3, wherein said second texture memory stores texels in said clip-map in a two-dimensional or a three-dimensional texel array.

14. A computer graphics processing system for providing texture from a texture pattern in a display image, comprising:

first texture memory for storing a texture map said texture map having multiple level of detail maps for storing texels representing the texture pattern filtered at successively coarser resolutions;

clip-map selecting means for selecting a clip-map within said texture map, said clip-map consisting of a set of tiles corresponding to regions of said level of detail maps substantially covering the display image;

second texture memory for storing said clip-map; and

texture processor means for retrieving texels from said clip-map which map to pixel data forming the display image, wherein said texture processor means comprises:

texture coordinate generator for generating at least one texture coordinate identifying where a pixel in the display image maps to the texture pattern,

LOD generator for generating a LOD value, and

a memory controller for retrieving at least one texel from said clip-map stored in said second texture memory based on said at least one texture coordinate and said LOD value, wherein, said LOD generator includes:

LOD identifying means for identifying an appropriate level of detail representing the level of detail map amongst said multiple level of detail maps where texel dimension is closest in size to said pixel,

texel determining means for determining whether a texel at said at least one texture coordinate is included in a first tile at said appropriate level of detail, said LOD value being set to said appropriate level of detail when said texel is included in said first tile, and

substitute texel determining means for determining a substitute texel in a second tile which includes said at least one texture coordinate, said LOD value being set to the level of detail of said second tile.

15. A computer graphics processing system for providing texture from a texture pattern in a display image, comprising:

first texture memory for storing a texture map, said texture map having multiple level of detail maps for storing texels representing the texture pattern filtered at successively coarser resolutions;

clip-map electing means for selecting a clip-map within said texture map, said clip-map consisting of a set of tiles corresponding to regions of said level of detail maps substantially covering the display image;

second texture memory for storing said clip-map; and

texture processor means for retrieving texels from said clip-map which map to pixel data forming the display image, wherein said set of tiles comprises a cubical part and a pyramidal part;

said one or more tiles in said cubical part comprising one or more arrays of texels, respectively, within said regions of said level of detail maps, and

said one or more tiles in said pyramidal part comprising one or more arrays of texels, respectively, said one or more arrays texels in said pyramidal part consisting of said level of detail maps which are equal to or smaller than said one or more tiles in said cubical part.

16. A method for providing texture from a texture pattern in a display image comprising the steps of:

storing a texture map in a first texture memory, said texture map having multiple level of detail maps for storing texels representing the texture pattern filtered at successively coarser resolutions;

selecting a clip-map within said texture map, said clip-map consisting of a set of tiles corresponding to regions of said level of detail maps, said set of tiles being centered in a field of view extending from an eyepoint location of the display image to substantially cover the display image;

storing said clip-map in a second texture memory;

retrieving texels from said clip-map stored in said second texture memory which map to pixels forming the display image; and

updating texels at edges of said tiles stored in said second texture memory to track changes in the location of the eyepoint location.

17. The method of claim 16, wherein said updating step updates said clip-map to track a change in said eyepoint location for a new display image.

18. The method of 17, further comprising the step of checking whether a texel to be accessed is located in a fringe portion of a tile, said fringe portion including the edges where texels are updated, and using a substitute texel when said texel is located in said fringe portion.

19. The method of claim 16, wherein said updating step update texels at edges of said tiles stored in said second texture memory to track a change to a new eyepoint location for a new display image.

20. The method of claim 19, wherein said updating step includes the steps of:

discarding texels from an edge of a tile located furthest from said new eyepoint location; and

loading texels from said first texture memory to said second texture memory, said texels being loaded next to an edge of a tile closer to the new eyepoint location wherein the number of texels loaded equals the number of texels discarded to maintain the size of said tile constant.

21. The method of claim 16, further comprising the step of retrieving coarser substitute texels from said clip-map stored in said second texture memory.

22. The method of claim 16, wherein said storing step stores texels in said clip-map in a two-dimensional or a three-dimensional texel array.

23. A method for providing texture from a texture pattern in a display image comprising the steps of:

storing a texture map in a first texture memory, said texture map having multiple level of detail maps for storing texels representing the texture pattern filtered at successively coarser resolutions;

selecting a clip-map within said texture map, said clip-map consisting of a set of tiles corresponding to regions of said level of detail maps, said set of tiles being centered in a field of view extending from an eyepoint location of the display image to substantially cover the display image;

storing said clip-map in a second texture memory; and

retrieving texels from said clip-map stored in said second texture memory which map to pixels forming the display image, wherein said retrieving step includes the following steps:

generating at least one texture coordinate identifying where a pixel in the display image maps to the texture pattern,

generating a LOD value, and

retrieving at least one texel from said clip-map stored in said second texture memory based on said at least one texture coordinate and said LOD value, wherein, said LOD value generating step includes:

identifying an appropriate level of detail representing the level of detail map amongst said multiple level of detail maps where texel dimension is closest in size to said pixel,

determining whether a texel at said at least one texture coordinate is included in a first tile at said appropriate level of detail and setting said LOD value to said appropriate level of detail when said texel is included in said first tile, and

when said texel in not included in said first tile, determining a substitute texel in a coarser second tile which includes said at least one texture coordinate and setting said LOD value to the level of detail of said second tile.

**24**. A texture processor for mapping texels from a clip-map to corresponding pixels, wherein the clip-map consists a set of tiles representing a selected portion of a texture MIP-map which substantially covers the pixels, said texture processor comprising:

a texture memory storing the clip-map;

a texture generator; and

a texture memory manager coupled between said texture generator and said texture memory, wherein

said texture generator includes a texture coordinate generator for generating texel coordinates for each pixel and a LOD generator for generating a LOD value for each pixel, said LOD value identifies the tile which covers the pixel and has texel dimensions closest in size to the pixel, said texel coordinates and LOD value being output for each pixel to said texture memory manager; and

said texture memory manager retrieves texels from the clip-map stored in said texture memory for combining with said pixels to form a textured display image.

**25**. The texture processor of claim **24**, wherein said texture memory stores said texels in two-dimensional or three-dimensional texel arrays.

**26**. The texture processor of claim **24**, wherein said texture memory manager includes an address generator for subtracting at least one of a tile offset and an update offset.

**27**. The texture processor of claim **24**, wherein 2×2 groups of pixels are processed as pixel quads.

**28**. The texture processor of claim **27**, wherein said texture generator includes a quad coordinate compression block for compressing texel coordinates for each pixel quad sent to said texture memory manager, and said texture memory manager includes a quad coordinate decompression block to decompress the compressed texel coordinates for each pixel quad.

* * * * *

APPENDIX EE

This report appeared in a special issue of the International Journal of Geographic Information Sciences, issue 4 of volume 13, in 1999.

# A Commentary on GeoVRML: A Tool for 3D Representation of GeoReferenced Data on the Web

**Theresa-Marie Rhyne**
**ACM SIGGRAPH Director at Large**
**Lockheed Martin Technical Services**
**US EPA Scientific Visualization Center**
**86 Alexander Drive**
**Research Triangle Park, North Carolina 27711**
**(trhyne@vislab.epa.gov)**

## Abstract:

GeoVRML techniques have the potential to provide functional and transparent communication between geographic information and 3D Web visualization tools. This report outlines recommended practices and modifications to the VRML 97 standard to consider pre-existing cartographic projections and georeferenced data. The concepts outlined for incorporating georeferenced coordinate systems in VRML worlds have generic applicability to 3D Web technologies like MPEG-4, Java3D and Chrome.

## Introduction:

The interactive three dimensional (3D) representation of georeferenced data on the World Wide Web (Web) is achieved with tools like the Virtual Reality Modeling Language (VRML). VRML97 is the approved International Standard (ISO/IEC 14772) file format for describing interactive multimedia on the Internet. In general, a VRML file is also called a "world". Users explore these "worlds" with Web browsers that support the viewing of VRML files. More information on VRML can be found at the Web3D Consortium Web pages, see: (http://www.web3d.org).

The VRML97 standard was designed primarily by the computer graphics community. Typical computer graphics imagery focuses on locally bounded regions and small screen sizes where maximum pixel ranges are approximately 1600 by 1280 pixels. As a result, VRML97 relies on single-precision (32 bit) IEEE floating point data values. The coordinate system for VRML97 is based on the simple Cartesian local (X,Y,Z) coordinate system with the orgin being at (0,0,0) and Y representing up. This coordinate system is often sufficient for many computer graphics problems.

These two parameters of the VRML 97 standard provide limitations for the representation of geographic and cartographic data as well as georeferenced computational modeling simulations in VRML. For example, since the earth's diameter approximates 12 million meters, it is not possible to present geographic data resolutions greater than 10 to 100 meters with single-precision data values. This means that data obtained from global positioning systems (GPS) with absolute locations within 1

Microsoft et al. Exhibit 1005

APPENDIX EE

meter resolution cannot be accurately presented in VRML97. The heavy reliance on Cartesian coordinates also poses difficulties with data in Geodetic (GDC or lattitude/longitude), Universal Transverse Mercator (UTM), Lambert Conformal Conic (LCC) or other pre-existing cartographic projections. In February 1998, the VRML Consortium approved the formation of the GeoVRML Working Group to discuss and develop tools, recommended practices and standards necessary to generate, display and exchange georeferenced data in VRML, (Iverson & GeoVRML, 1998). In December 1998, the VRML Consortium expanded its charter and renamed itself as the "Web 3D Consortium".

This report reviews the major recommended practices and modifications to the VRML standard under consideration and development by the GeoVRML Working Group. Additional emerging 3D Web technologies and their relation to geospatial data visualization will also be highlighted. GeoVRML techniques have the potential to provide functional and transparent communication between geographic information and 3D visualization tools, (Rhyne, 1997).

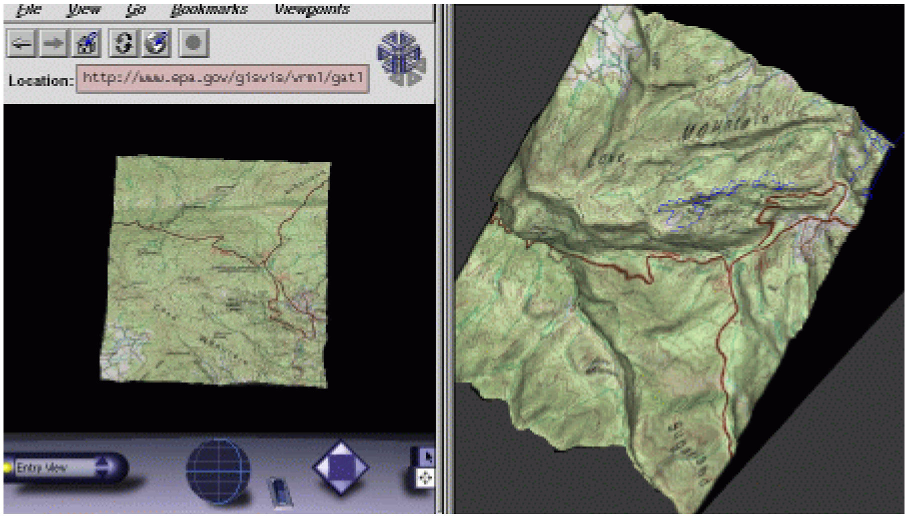


Figure #1: Example VRML world with a TIFF image of a USGS map drapped over the 3D surface is shown on the left. On the right is a similar image made with a visualization toolkit package. Notice how the map is inverted in the VRML browser. We hope to improve this situation with GeoVRML Coordinate systems. Images developed by Theresa-Marie Rhyne and Thomas Fowler of Lockheed Martin Technical Services at the United States Environmental Protection Agency's Scientific Visualization Center. See: (http://www.epa.gov/gisvis).

**Moving towards GeoVRML Coordinate Systems:**

The geographic information systems, cartographic and military simulation communities have developed a number of standards for the representation of geospatial information and georeferencing of arbitrary data, (Rhyne, 1998). The Open GIS Consortium is presently moving forward with efforts to support the full integration of geospatial data into mainstream computing and the widespread usage of interoperable commercial geoprocessing software, see: (http://www.opengis.org/). There are also International Organization for Standardization (ISO) efforts in the Geographic information/Geomatics arenas, see: (http://www.iso.ch/meme/TC211.html).

In order to attempt to include a methodology for supporting georeferenced data in the upcoming (1999) revision to the VRML97 standard, the GeoVRML Working Group decided to base its efforts on a currently existing reference model and software package entitled the SEDRIS Geographic

Microsoft et al.   Exhibit 1005

APPENDIX EE

Reference Model (GRM). The Synthetic Environment Data Representation & Interchange Specification (SEDRIS) is a project funded by the United States' Defense Modeling and Simulation Office. The SEDRIS GRM supports twelve different coordinate systems and provides tools to automatically convert reference marks between them. The software source for GRM is publically available and is currently implemented in the C programming language. More information on the SEDRIS GRM can be found on the SEDRIS Web site at: (http://www.sedris.org/).

The GeoVRML Working Group is thus recommending a Level 1 practice whereby geographical coordinates based on the SEDRIS GRM are converted into a local Cartesian coordinate system for improved level of detail in GeoVRML visualizations. The GeoVRML Working Group is also exploring a Level 2 practice whereby geo-referenced data is transparently and seamlessly converted from a wider and multiple variety of sources, (Iverson & GeoVRML, 1998).

Researchers at the SRI International - Artificial Intelligence Center, have recently developed, for public release, the GeoTransform Java class file hierarchy based on the SEDRIS GRM. With the GeoTransform Java package, it is possible to perform efficient and accurate geographic coordinate transformations for the Geodetic Coordinate System (GDC), GeoCentric Coordinate System (GCC), and Universal Transverse Mercator (UTM) System. GeoTransform allows for authoring VRML worlds that read coordinates in any of these systems and tranparently convert the geographic data into Cartesian Coordinates for display in a VRML browser. More information on the GeoTranform Java package can be found at: (http://www.ai.sri.com/~reddy/geovrml/geotransform/) .

**Defining the GeoOrigin Node:**

In order to build a georeferenced VRML world, a GeoOrigin node is defined in the VRML file. This GeoOrigin allows for converting coordinates from cartographic earth-based coordinate systems into the existing VRML97 Cartesian reference frame, (Iverson & GeoVRML, 1998). A single GeoOrigin node, representing a single georeferenced point, becomes the reference frame identified with the VRML world's zero-based (0,0,0) origin.

GeoOrigin

```
EXTERNPROTO GeoOrigin [
  field MFString geoSystem ["GDC"]
  field SFString geoCoords ""
] "urn:geovrml:protos#GeoOrigin"
```

The geoSystem field selects a geographic reference system from the naming conventions based on the SEDRIS GRM. Some of these georeference coordinate systems require additional arguments to fully designate the coordinates. As an example, the Geodetic (GDC) system involves the selection of ellipsoid, geoid, and datum references. Additional strings in the geoSystem field support this requirement.

The geoCoords field is a sequence of 64-bit precision values seperated by spaces that define an absolute location using the coordinate system selected in the geoSystem field. Optional strings in the geoSystem field determine the interpretation of the geoCoords field. As an example, "DMS" can specify that the geoCoords string will include degree, minute and second fields for each latitute and longitude value in a GDC coordinate. Every geospatial location determined by a geoSystem and geoCoords pair defines an implicit orthogonal Cartesian reference frame indexed by x,y,z in meters

Microsoft et al. Exhibit 1005

APPENDIX EE

with the designated geospatial location at the origin and with y being the up direction. This allows for conformance with the VRML97 standard.

A more detailed discussion about the GeoOrigin node can be found in the Request for Comment document on GeoVRML Coordinate Systems. This discussion is located at the GeoVRML Working Group's Web site at: (http://www.ai.sri.com/~leei/geovrml/).

During the past year, SRI International developed a series of VRML97 nodes for improved support of terrain visualization. These contributions were developed as part of the GeoVRML Working Group and are in the public domain. A comprehensive discussion of these efforts can be found in the SRI International - Artificial Intelligence Center Report No. 559, which is cited the the references below, (Reddy, et. al., 1998). These new GeoVRML nodes can be accessed on the Web at: (http://www.ai.sri.com/geovrml/protos) .

**Integrating Spatial Data Repositories and GeoVRML Visualizations:**

There are a number of efforts underway to examine the use of the Virtual Reality Modeling Language (VRML) for the interactive exploration of geospatial data repositories (Rhyne & Fowler, 1996). In the United States, some of this work is being done in conjunction with the Federal Geographic Data Committee (FGDC) 's National Geospatial Data Clearinghouse (see: (http://fgdc.er.usgs.gov/)). In addition to the use of intelligent agents, data mining techniques are being employed to assist with the retrieval of spatial data. The development of GeoOrgin nodes in VRML will support the use of agent and data mining technology for rapid creation of interactive web-based visualizations. This will greatly facilitate visual information retrieval of geospatial data.

**Reaching out to other Interactive 3D Web Technologies:**

In addition to VRML, there are other 3D Web technologies under development. Three examples include (a) the development of the MPEG-4 standard; (b) Java 3D and (c) Chrome. In early 1998, the International Organization for Standardization (ISO) announced that it will use Apple Computer's QuickTime file format as the basis for a unified digital media storage format for the MPEG-4 standard for graphics content on the Web. The VRML Consortium has established a Working Group to examine MPEG-4 and VRML integration. Java3D, from Sun Microsystems, supports the development of 3D computer graphics applications in the Java programming language. This includes the development of VRML browsers with Java 3D. Another emerging 3D Web technology is Chrome from Microsoft Corporation. Chrome is a Windows 98 add-on that uses the Extensible Markup Language (XML) to access Windows 98 multimedia capabilities for creating 3D content on the Web. The concepts outlined above for incorporating georeferenced coordinate systems in VRML worlds have generic applicability to 3D Web technologies like MPEG-4, Java3D and Chrome. Details about the QuickTime file format and its adoption by ISO as the starting point for MPEG-4 can be found at the Apple Computer web site, see: (http://www.apple.com/quicktime/). More information on Java and Java 3D can be found at the Javasoft Web site, see: (http://www.javasoft.com/products/java-media/3D/). Additional information on Chrome can be found by searching the Microsoft web site at: (http://www.microsoft.com).

**Concluding Remarks:**

The use of VRML for cartographic and geographic presentation is currently being examined by research groups participating in the International Cartographic Association's Commission on Visualization, (Fairburn and Parsley, 1997). Preliminary definitions of the needs for geofunctions in

Microsoft et al. Exhibit 1005

APPENDIX EE

virtual reality and VRML were done at Leicester University in July 1997, (Moore, et. al.). The Commission has also explored other multimedia and web-based technologies for developing mapping products, (Cartwright, 1998) and (Andrienko & Andrienko, 1998). The Association for Computing Machinery's Special Interest Group on Graphics (ACM - SIGGRAPH)'s collaboration with the ICA Commission on Visualization has attempted to examine how computer graphics technology can be effectively adapted to meet cartographic needs and requirements. This project, entitled the ACM SIGGRAPH Carto Project, is pleased that the VRML Consortium chose to create the GeoVRML Working Group to actualize effective exchange of georeferenced data in VRML. We anticipate GeoVRML techniques expanding to address many 3D Web Technologies as the VRML Consortium redefines itself as the Web 3D Consortium. The issues discussed here are important steps toward functional integration of geographic information and 3D visualization tools. We hope similar efforts will continue to emerge in the future.

**Acknowledgements:**

We would like to acknowledge the efforts of Lee Iverson, founding Chair of the GeoVRML Working Group of the Web 3D Consortium, Don Brutzman, Vice President for Technology of the Web 3D Consortium, and Martin Reddy (who built many of the new GeoVRML nodes for VRML97). We are also appreciative to Judy Brown, Past Chair of Special Projects for ACM SIGGRAPH, for all the encouragement she provided during the first two years of the ACM SIGGRAPH Carto Project.

**References:**

ACM SIGGRAPH Carto Project Web Site: (http://www.siggraph.org/~rhyne/carto/).

Andrienko & Andrienko. 1998, Descartes -Intelligent Mapping and Visual Data Exploration on the Internet, Proceedings of the 1998 Polish Spatial Information Association Conference, May 1998, Warsaw Poland, : 339 - 340.

Cartwright, W. 1997. New media and their application to the production of map products. Computers &; Geosciences, special issue on Exploratory Cartographic Visualization 23(4) : 447-456.

Fairbairn, D. and Parsley, S. 1997. The use of VRML for cartographic presentation. Computers &; Geosciences, special issue on Exploratory Cartographic Visualization 23(4): 475-482.

GeoVRML Working Group of the VRML Consortium Web Site: (http://www.ai.sri.com/geovrml/).

Iverson, Lee & the GeoVRML Working Group of the VRML Consortium. 1998, GeoVRML RFC1: Coordinate Systems, (http://www.ai.sri.com/geovrml/rfc1.html).

ICA Commission on Visualization Web Site: (http://www.geog.psu.edu/ica/ICAvis.html).

Moore, K., Dykes, J., Wood, J., Bastin, L., Fisher, P. 1997, VR Geofunctions, (http://www.geog.le.ac.uk/mek/VRGeoFunctions.html).

Reddy, M., Leclerc, Y. G., Iverson, L., Bletter, N., and Vidimce, K. 1998, Modeling the Digital Earth in VRML, AIC Technical Report No. 559. SRI International, Menlo Park, CA. November 1998.

Rhyne, T.-M. and Fowler, T. 1996, Examining Dynamically Linked Geographic Visualization, Proceedings of the 1996 Computing in Environmental Resource Management Speciality Conference

Microsoft et al. Exhibit 1005

APPENDIX EE

sponsored by the Air & Waste Management Association, Dec. 1996, Research Triangle Park, North Carolina (USA), : 571 - 573.

Rhyne, T.-M. 1997. Going virtual with geographic information and scientific visualization. Computers & Geosciences, special issue on Exploratory Cartographic Visualization 23(4): 489-492.

Rhyne, T.-M. 1998, Open Spatial Data Standards for the Information Highway (Examining Dynamically Linked Geographic Visualization), Proceedings of the 1998 Polish Spatial Information Association Conference, May 1998, Warsaw Poland, : 297 - 299.

**Biography of the Author:**

Theresa-Marie Rhyne is a Director at Large of the ACM SIGGRAPH Executive Committee and is the Project Director of the ACM SIGGRAPH Carto Project. She is a lead scientific visualization researcher for Lockheed Martin Technical Services at the United States Environmental Protection Agency's Scientific Visualization Center.

--------------------------------------------------------------------------

Microsoft et al. Exhibit 1005

APPENDIX GG

# GeoTIFF Format Specification

# GeoTIFF Revision 1.0

+----------------------------------------------------------------------+

Specification Version: 1.8.1
Last Modified: 31 October, 1995

## Authors:

Niles Ritter, Jet Propulsion Laboratory
Cartographic Applications Group
4800 Oak Grove Dr.
Pasadena, CA 91109
email:ndr@tazboy.jpl.nasa.gov

Mike Ruth, SPOT Image Corp
Product Development Group
1897 Preston White Dr.
Reston, VA 22091
email:ruth@spot.com

## Acknowledgments:

GeoTIFF Working Group:
Mike Ruth, Niles Ritter, Ed Grissom, Brett Borup, George Galang,
John Haller, Gary Stephenson, Steve Covington, Tim Nagy,
Jamie Moyers, Jim Stickley,Joe Messina, Yves Somer.

Additional advice from discussions with Tom Lane, Sam Leffler regarding TIFF implementations.

Roger Lott, Fredrik Lundh, and Jarle Land provided valuable information regarding projections, projection code databases and geodetics.

GeoTIFF Mailing list:
Posting: geotiff@tazboy.jpl.nasa.gov
Subscription: geotiff-request@tazboy.jpl.nasa.gov
(send message "subscribe geotiff your-name-here").

## Disclaimers and Notes for This Version:

This proposal has not been approved by SPOT, JPL, or any other organization. This represents a proposal, which derives from many discussions between an international body of TIFF users and developers.

APPENDIX GG

The authors and their sponsors assume no liability for any special, incidental, indirect or consequences of any kind, or any damages whatsoever resulting from loss of use, data or profits, whether or not advised of the possibility of damage, and on any theory of liability, arising out of or in connection with the use of this specification.

# Copyright

# Licenses and Trademarks

# Concurrence

The following members of the GeoTIFF working group have reviewed and approved of this revision.

| Name | Organization | Representing |
|---|---|---|
| Niles Ritter | Jet Propulsion Labs | JPL Carto Group |
| Mike Ruth | SPOT Image Corp. (USA) | SPOT Image Corp. (USA) |

+------------------------------------------------------------+

# Table of Contents

+------------------------------------------------------------+

+------------------------------------------------------------+

# 1 Introduction

+------------------------------------------------------------+

+-------------------------------+

## 1.1 About this Specification

This is a description of a proposal to specify the content and structure of a group of industry-standard tag sets for the management of georeference or geocoded raster imagery using Aldus-Adobe's public domain Tagged-Image File Format (TIFF).

This specification closely follows the organization and structure of the TIFF specification document.

+----------------------------------+

### 1.1.1 Background

TIFF has emerged as one of the world's most popular raster file formats. But TIFF remains limited in cartographic applications, since no publicly available, stable structure for conveying geographic information presently exists in the public domain.

Several private solutions exist for recording cartographic information in TIFF tags. Intergraph has a mature and sophisticated geotie tag implementation, but this remains within the private TIFF tagset registered exclusively to Intergraph. Other companies (such as ESRI, and Island Graphics) have geographic solutions which are also proprietary or limited by specific application to their software's architecture.

Many GIS companies, raster data providers, and their clients have requested that the companies concerned with delivery and exploitation of raster geographic imagery develop a publicly available, platform interoperable standard for the support of geographic TIFF imagery. Such TIFF imagery would originate from satellite imaging platforms, aerial platforms, scans of aerial photography or paper maps, or as a result of geographic analysis. TIFF images which were supported by the public "geotie" tagset would be able to be read and positioned correctly in any GIS or digital mapping system which supports the "GeoTIFF" standard, as proposed in this document.

The savings to the users and providers of raster data and exploitation softwares are potentially significant. With a platform interoperable GeoTIFF file, companies could stop spending excessive development resource in support of any and all proprietary formats which are invented. Data providers may be able to produce off-the-shelf imagery products which can be delivered in the "generic" TIFF format quickly and possibly at lower cost. End-users will have the advantage of developed software that exploits the GeoTIFF tags transparently. Most importantly, the same raster TIFF image which can be read and modified in one GIS environment may be equally exploitable in another GIS environment without requiring any file duplication or import/export operation.

+----------------------------------+

### 1.1.2 History

The initial efforts to define a TIFF "geotie" specification began under the leadership of Ed Grissom at Intergraph, and others in the early 1990's. In 1994 a formal GeoTIFF mailing-list was created and maintained by Niles Ritter at JPL, which quickly grew to over 140 subscribers from government and industry. The purpose of the list is to discuss common goals and interests in developing an industry-wide GeoTIFF standard, and culminated in a conference in March of 1995 hosted by SPOT Image, with representatives from USGS, Intergraph, ESRI, ERDAS, SoftDesk, MapInfo, NASA/JPL, and others, in which the current working proposal for GeoTIFF was outlined. The outline was condensed into a prerelease GeoTIFF specification document by Niles Ritter, and Mike Ruth of SPOT Image.

Following discussions with Dr. Roger Lott of the European Petroleum Survey Group (EPSG), the GeoTIFF projection parametrization method was extensively modified, and brought into compatibility with both the POSC Epicentre model, and the Federal Geographic Data Committee (FGDC) metadata approaches.

+--------------------------------+

### 1.1.3 Scope

The GeoTIFF spec defines a set of TIFF tags provided to describe all "Cartographic" information associated with TIFF imagery that originates from satellite imaging systems, scanned aerial photography, scanned maps, digital elevation models, or as a result of geographic analyses. Its aim is to allow means for tying a raster image to a known model space or map projection, and for describing those projections.

GeoTIFF does not intend to become a replacement for existing geographic data interchange standards, such as the USGS SDTS standard or the FGDC metadata standard. Rather, it aims to augment an existing popular raster-data format to support georeferencing and geocoding information.

The tags documented in this spec are to be considered completely orthogonal to the raster-data descriptions of the TIFF spec, and impose no restrictions on how the standard TIFF tags are to be interpreted, which color spaces or compression types are to be used, etc.

+--------------------------------+

### 1.1.4 Features

GeoTIFF fully complies with the TIFF 6.0 specifications, and its extensions do not in any way go against the TIFF recommendations, nor do they limit the scope of raster data supported by TIFF.

GeoTIFF uses a small set of reserved TIFF tags to store a broad range of georeferencing information, catering to geographic as well as projected coordinate systems needs. Projections include UTM, US State Plane and National Grids, as well as the underlying projection types such as Transverse Mercator, Lambert Conformal Conic, etc. No information is stored in private structures, IFD's or other mechanisms which would hide information from naive TIFF reading software.

GeoTIFF uses a "MetaTag" (GeoKey) approach to encode dozens of information elements into just 6 tags, taking advantage of TIFF platform-independent data format representation to avoid cross-platform interchange difficulties. These keys are designed in a manner parallel to standard TIFF tags, and closely follow the TIFF discipline in their structure and layout. New keys may be defined as needs arise, within the current framework, and without requiring the allocation of new tags from Aldus/Adobe.

GeoTIFF uses numerical codes to describe projection types, coordinate systems, datums, ellipsoids, etc. The projection, datums and ellipsoid codes are derived from the EPSG list compiled by the Petrotechnical Open Software Corporation (POSC), and mechanisms for adding further international projections, datums and ellipsoids has been established. The GeoTIFF information content is designed to be compatible with the data decomposition approach used by the National Spatial Data Infrastructure (NSDI) of the U.S. Federal Geographic Data Committee (FGDC).

While GeoTIFF provides a robust framework for specifying a broad class of existing Projected coordinate systems, it is also fully extensible, permitting internal, private or proprietary information storage. However, since this standard arose from the need to avoid multiple proprietary encoding systems, use of private implementations is to be discouraged.

+-----------------------------------+

## 1.2 Revision Notes

This is the final release of GeoTIFF Revision 1.0, supporting the new EPSG 2.x codes.

Changes from 1.8 document: minor spelling and typo corrections.

+-----------------------------------+

### 1.2.1 Revision Nomenclature

A Revision of GeoTIFF specifications will be denoted by two integers separated by a decimal, indicating the Major and Minor revision numbers. GeoTIFF stores most of its information using a "Key-Code" pairing system; the Major revision number will only be incremented when a substantial addition or modification is made to the list of information Keys, while the Minor Revision number permits incremental augmentation of the list of valid codes.

+-----------------------------------+

### 1.2.2 New Features

Revision 1.0 New Transformation Matrix Tag.

Index Table added in Section 6.4 to assist in looking up geodesy codes.

+-----------------------------------+

### 1.2.3 Clarifications

```
Revision 1.0:

 o The former ModelTransformationTag (33920) conflicts with
   an internal Intergraph implementation and is being deprecated,
   in favor of a new tag (34264, registered to JPL).

 o The "Origin" keys have been renamed with "Natural" or "Nat"
   prefixes, to distinguish from "False" origins, and to have
   a closer match to EPSG/POSC terminology. All Revision 0.2
   names shall be recognized in a backward-compatible fashion.

 o The GeoTIFF/Cartlab web page addresses have been moved out
   of the author's ~ndr/ personal directory, and may now be found at:
```

http://www-mipl.jpl.nasa.gov/cartlab/geotiff/geotiff.html

Revision 0.2:

- o South Oriented Gauss Conformal is Transverse Mercator with South pointing up, and so has been given a distinct code, rather than aliased to Transverse Mercator.

Revision 0.1:

- o GeoTIFF-writers shall store the GeoKey entries in key-sorted order within the GeoKeyDirectoryTag. This is a change from preliminary discussions which permitted arbitrary order, and more closely follows the TIFF discipline.
- o The third value "ScaleZ" in ModelPixelScaleTag = (ScaleX, ScaleY, ScaleZ) shall by default be set to 0, not 1, as suggested in preliminary discussions. This is because most standard model spaces are 2-dimensional (flat), and therefore its vertical shape is independent of the pixel-value.
- o The code 32767 shall be used to imply "user-defined", rather than 16384. This avoids breaking up the reserved public GeoKey code space into two discontiguous ranges, 0-16383 and 16385-32767.
- o If a GeoKey is coded "undefined", then it is exactly that; no parameters should be provided (e.g. EllipsoidSemiMajorAxis, etc). To provide parameters for a non-coded attribute, use "user-defined".

+----------------------------------+

### 1.2.4 Organizational changes

None.

+----------------------------------+

### 1.2.5 Changes in Requirements

Changes to this preliminary revision:

o Support for new transformation matrix tag (34264) required.

+----------------------------------+

### 1.2.6 Agenda for Future Development

Revision 1.0, which is the first true "Baseline" revision, is proposed to support well-documented, public, relatively simple Projected Coordinate Systems (PCS), including most commonly used and supported in the international public domains today, together with their underlying map-projection systems. Following the critiques of the 0.x Revision phase, the 1.0 Revision spec is hereby released in Sept '95.

In the coming year, incremental 1.x augmentations to the "codes" list will be established, as well as discussions regarding the future "2.0" requirements.

The Revision 2.0 phase is proposed to extend the capability of the GeoTIFF tagsets beyond PCS projections into more complex map projection geometries, including single-project, single-vendor, or proprietary cartographic solutions.

TBD: Sounding Datums and related parameters for Digital Elevation Models (DEM's) and bathymetry -- Revision 2?

+----------------------------------+

## 1.3 Administration

+----------------------------------+

### 1.3.1 Information and Support:

The most recent version of the GeoTIFF spec, EPSG/POSC tables, and source code is available via anonymous FTP at:

```
       ftp://mtritter.jpl.nasa.gov/pub/tiff/geotiff/
```

and is mirrored at the USGS:

```
     ftp://ftpmcmc.cr.usgs.gov/release/geotiff/jpl_mirror/
```

There are several subdirectories called spec/ tables/ and code/.

The USGS also has an archive of prototype GeoTIFF images at:

```
     ftp://ftpmcmc.cr.usgs.gov/release/geotiff/images/
```

Information and a hypertext version of the GeoTIFF spec is available via WWW at the following site:

```
     http://www-mipl.jpl.nasa.gov/cartlab/geotiff/geotiff.html
```

A mailing-list is currently active to discuss the on-going development of this standard. To subscribe to this list, send e-mail to:

```
    GeoTIFF-request@tazboy.jpl.nasa.gov
```

with no subject and the body of the message reading:

```
  subscribe geotiff  your-name-here
```

To post inquiries directly to the list, send email to:

```
    geotiff@tazboy.jpl.nasa.gov
```

+----------------------------------+

### 1.3.2 Private Keys and Codes:

As with TIFF, in GeoTIFF private "GeoKeys" and codes may be used, starting with 32768 and above. Unlike the TIFF spec, however, these private key-spaces will not be reserved, and are only to be used for private, internal purposes.

+-----------------------------------+

### 1.3.3 Proposed Revisions to GeoTIFF

Should a feature arise which is not currently supported, it should be formally proposed for addition to the GeoTIFF spec, through the official mailing-list.

The current maintainer of the GeoTIFF specification is Niles Ritter, though this may change at a later time. Projection codes are maintained through EPSG/POSC, and a mechanism for change/additions will be established through the GeoTIFF mailing list.

+----------------------------------------------------------------------+

## 2 Baseline GeoTIFF

+----------------------------------------------------------------------+

+-----------------------------------+

### 2.1 Notation

This spec follows the notation remarks of the TIFF 6.0 spec, regarding "is", "shall", "should", and "may"; the first two indicate mandatory requirements, "should" indicates a strong recommendation, while "may" indicates an option.

+-----------------------------------+

### 2.2 GeoTIFF Design Considerations

Every effort has been made to adhere to the philosophy of TIFF data abstraction. The GeoTIFF tags conform to a hierarchical data structure of tags and keys, similar to the tags which have been implemented in the "basic" and "extended" TIFF tags already supported in TIFF Version 6 specification. The following are some points considered in the design of GeoTIFF:

o Private binary structures, while permitted under the TIFF spec, are in general difficult to maintain, and are intrinsically platform- dependent. Whenever possible, information should be sorted into their intrinsic data-types, and placed into appropriately named tags. Also, implementors of TIFF readers would be more willing to honor a new tag specification if it does not require parsing novel binary structures.

o Any Tag value which is to be used as a "keyword" switch or modifier should be a SHORT type, rather than an ASCII string. This avoids common mistakes of mis-spelling a keyword, as well as facilitating an implementation in code using the "switch/case" features of most languages. In general, scanning ASCII strings for keywords (CaseINSensitiVE?) is a hazardous (not to mention slower and more complex) operation.

o True "Extensibility" strongly suggests that the Tags defined have a sufficiently abstract definition so that the same tag and its values may be used and interpreted in different ways as more complex information spaces are developed. For example, the old SubFileType tag (255) had to be obsoleted and replaced with a NewSubFileType tag, because images began appearing which could not fit into the narrowly defined classes for that Tag. Conversely, the YCbCrSubsampling Tag has taken on new meaning and importance as the JPEG compression standard for TIFF becomes finalized.

+-----------------------------------+

## 2.3 GeoTIFF Software Requirements

GeoTIFF requires support for all documented TIFF 6.0 tag data-types, and in particular requires the IEEE double-precision floating point "DOUBLE" type tag. Most of the parameters for georeferencing will not have sufficient accuracy with single-precision IEEE, nor with RATIONAL format storage. The only other alternative for storing high-precision values would be to encode as ASCII, but this does not conform to TIFF recommendations for data encoding.

It is worth emphasizing here that the TIFF spec indicates that TIFF-compliant readers shall honor the 'byte-order' indicator, meaning that 4-byte integers from files created on opposite order machines will be swapped in software, and that 8-byte DOUBLE's will be 8-byte swapped.

A GeoTIFF reader/writer, in addition to supporting the standard TIFF tag types, must also have an additional module which can parse the "Geokey" MetaTag information. A public-domain software package for performing this function is now available; see the "References" in section 5 for the location.

+-----------------------------------+

## 2.4 GeoTIFF File and "Key" Structure

This section describes the abstract file-format and "GeoKey" data storage mechanism used in GeoTIFF. Uses of this mechanism for implementing georeferencing and geocoding is detailed in section 2.6 and section 2.7 .

A GeoTIFF file is a TIFF 6.0 file, and inherits the file structure as described in the corresponding portion of the TIFF spec. All GeoTIFF specific information is encoded in several additional reserved TIFF tags, and contains no private Image File Directories (IFD's), binary structures or other private information invisible to standard TIFF readers.

The number and type of parameters that would be required to describe most popular projection types would, if implemented as separate TIFF tags, likely require dozens or even hundred of tags, exhausting the limited resources of the TIFF tag-space. On the other hand, a private IFD, while providing thousands of free tags, is limited in that its tag-values are invisible to non-savvy TIFF readers (which don't know that the IFD_OFFSET tag value points to a private IFD).

To avoid these problems, a GeoTIFF file stores projection parameters in a set of "Keys" which are virtually identical in function to a "Tag", but has one more level of abstraction above TIFF. Effectively, it is a sort of "Meta-Tag". A Key works with formatted tag-values of a TIFF file the way that a TIFF file deals with the raw bytes of a data file. Like a tag, a Key has an ID number ranging from 0 to 65535, but unlike TIFF tags, all key ID's are available for use in GeoTIFF parameter definitions.

The Keys in GeoTIFF (also call "GeoKeys") are all referenced from the GeoKeyDirectoryTag, which defined as follows:

```
GeoKeyDirectoryTag:
      Tag = 34735 (87AF.H)
      Type = SHORT (2-byte unsigned short)
      N = variable, >= 4
      Alias: ProjectionInfoTag, CoordSystemInfoTag
      Owner: SPOT Image, Inc.
```

This tag may be used to store the GeoKey Directory, which defines and references the "GeoKeys", as described below.

The tag is an array of unsigned SHORT values, which are primarily grouped into blocks of 4. The first 4 values are special, and contain GeoKey directory header information. The header values consist of the following information, in order:

```
Header={KeyDirectoryVersion, KeyRevision, MinorRevision, NumberOfKeys}
```

where

"KeyDirectoryVersion" indicates the current version of Key implementation, and will only change if this Tag's Key structure is changed. (Similar to the TIFFVersion (42)). The current DirectoryVersion number is 1. This value will most likely never change, and may be used to ensure that this is a valid Key-implementation.

"KeyRevision" indicates what revision of Key-Sets are used.

"MinorRevision" indicates what set of Key-codes are used. The complete revision number is denoted <KeyRevision>.<MinorRevision>

"NumberOfKeys" indicates how many Keys are defined by the rest of this Tag.

This header is immediately followed by a collection of <NumberOfKeys> KeyEntry sets, each of which is also 4-SHORTS long. Each KeyEntry is modeled on the "TIFFEntry" format of the TIFF directory header, and is of the form:

```
KeyEntry = { KeyID, TIFFTagLocation, Count, Value_Offset }
```

where

"KeyID" gives the key-ID value of the Key (identical in function to TIFF tag ID, but completely independent of TIFF tag-space),

"TIFFTagLocation" indicates which TIFF tag contains the value(s) of the Key: if TIFFTagLocation is 0, then the value is SHORT, and is contained in the "Value_Offset" entry. Otherwise, the type (format) of the value is implied by the TIFF-Type of the tag containing the value.

"Count" indicates the number of values in this key.

"Value_Offset" Value_Offset indicates the index-offset *into* the TagArray indicated by TIFFTagLocation, if it is nonzero. If TIFFTagLocation=0, then Value_Offset contains the actual (SHORT) value of the Key, and Count=1 is implied. Note that the offset is not a byte-offset, but rather an index based on the natural data type of the specified tag array.

Following the KeyEntry definitions, the KeyDirectory tag may also contain additional values. For example, if a Key requires multiple SHORT values, they shall be placed at the end of this tag, and the KeyEntry will set TIFFTagLocation=GeoKeyDirectoryTag, with the Value_Offset pointing to the location of the value(s).

All key-values which are not of type SHORT are to be stored in one of the following two tags, based on their format:

GeoDoubleParamsTag:
    Tag = 34736 (87BO.H)
    Type = DOUBLE (IEEE Double precision)
    N = variable
    Owner: SPOT Image, Inc.

This tag is used to store all of the DOUBLE valued GeoKeys, referenced by the GeoKeyDirectoryTag. The meaning of any value of this double array is determined from the GeoKeyDirectoryTag reference pointing to it. FLOAT values should first be converted to DOUBLE and stored here.

GeoAsciiParamsTag:
    Tag = 34737 (87B1.H)
    Type = ASCII
    Owner: SPOT Image, Inc.
    N = variable

This tag is used to store all of the ASCII valued GeoKeys, referenced by the GeoKeyDirectoryTag. Since keys use offsets into tags, any special comments may be placed at the beginning of this tag. For the most part, the only keys that are ASCII valued are "Citation" keys, giving documentation and references for obscure projections, datums, etc.

Note on ASCII Keys:

Special handling is required for ASCII-valued keys. While it is true that TIFF 6.0 permits multiple NULL-delimited strings within a single ASCII tag, the secondary strings might not appear in the output of naive "tiffdump" programs. For this reason, the null delimiter of each ASCII Key value shall be converted to a "|" (pipe) character before being installed back into the ASCII holding tag, so that a dump of the tag will look like this.

```
AsciiTag="first_value|second_value|etc...last_value|"
```

A baseline GeoTIFF-reader must check for and convert the final "|" pipe character of a key back into a NULL before returning it to the client software.

GeoKey Sort Order:

**APPENDIX GG**

In the TIFF spec it is required that TIFF tags be written out to the file in tag-ID sorted order. This is done to avoid forcing software to perform N-squared sort operations when reading and writing tags.

To follow the TIFF philosophy, GeoTIFF-writers shall store the GeoKey entries in key-sorted order within the CoordSystemInfoTag.

Example:

```
GeoKeyDirectoryTag=(   1,     1, 2,     6,
                    1024,     0, 1,     2,
                    1026, 34737,12,     0,
                    2048,     0, 1, 32767,
                    2049, 34737,14,    12,
                    2050,     0, 1,     6,
                    2051, 34736, 1,     0 )
GeoDoubleParamsTag(34736)=(1.5)
GeoAsciiParamsTag(34737)=("Custom File|My Geographic|")
```

The first line indicates that this is a Version 1 GeoTIFF GeoKey directory, the keys are Rev. 1.2, and there are 6 Keys defined in this tag.

The next line indicates that the first Key (ID=1024 = GTModelTypeGeoKey) has the value 2 (Geographic), explicitly placed in the entry list (since TIFFTagLocation=0). The next line indicates that the Key 1026 (the GTCitationGeoKey) is listed in the GeoAsciiParamsTag (34737) array, starting at offset 0 (the first in array), and running for 12 bytes and so has the value "Custom File" (the "|" is converted to a null delimiter at the end). Going further down the list, the Key 2051 (GeogLinearUnitSizeGeoKey) is located in the GeoDoubleParamsTag (34736), at offset 0 and has the value 1.5; the value of key 2049 (GeogCitationGeoKey) is "My Geographic".

The TIFF layer handles all the problems of data structure, platform independence, format types, etc, by specifying byte-offsets, byte-order format and count, while the Key describes its key values at the TIFF level by specifying Tag number, array-index, and count. Since all TIFF information occurs in TIFF arrays of some sort, we have a robust method for storing anything in a Key that would occur in a Tag.

With this Key-value approach, there are 65536 Keys which have all the flexibility of TIFF tag, with the added advantage that a TIFF dump will provide all the information that exists in the GeoTIFF implementation.

This GeoKey mechanism will be used extensively in section 2.7, where the numerous parameters for defining Coordinate Systems and their underlying projections are defined.

+--------------------------------+

## 2.5 Coordinate Systems in GeoTIFF

Geotiff has been designed so that standard map coordinate system definitions can be readily stored in a single registered TIFF tag. It has also been designed to allow the description of coordinate system definitions which are non-standard, and for the description of transformations between coordinate systems, through the use of three or four additional TIFF tags.

However, in order for the information to be correctly exchanged between various clients and providers of GeoTIFF, it is important to establish a common system for describing map projections.

In the TIFF/GeoTIFF framework, there are essentially three different spaces upon which coordinate systems may be defined. The spaces are:

```
1) The raster space (Image space) R, used to reference the pixel values
   in an image,
2) The Device space D, and
3) The Model space, M, used to reference points on the earth.
```

In the sections that follow we shall discuss the relevance and use of each of these spaces, and their corresponding coordinate systems, from the standpoint of GeoTIFF.

+----------------------------------+

## 2.5.1 Device Space and GeoTIFF

In standard TIFF 6.0 there are tags which relate raster space R with device space D, such as monitor, scanner or printer. The list of such tags consists of the following:

```
 ResolutionUnit (296)
 XResolution    (282)
 YResolution    (283)
 Orientation    (274)
 XPosition      (286)
 YPosition      (287)
```

In Geotiff, provision is made to identify earth-referenced coordinate systems (model space M) and to relate M space with R space. This provision is independent of and can co-exist with the relationship between raster and device spaces. To emphasize the distinction, this spec shall not refer to "X" and "Y" raster coordinates, but rather to raster space "J" (row) and "I" (column) coordinate variables instead, as defined in section 2.5.2.2.

+----------------------------------+

## 2.5.2 Raster Coordinate Systems

+----------------------------------+

### 2.5.2.1 Raster Data

Raster data consists of spatially coherent, digitally stored numerical data, collected from sensors, scanners, or in other ways numerically derived. The manner in which this storage is implemented in a TIFF file is described in the standard TIFF specification.

Raster data values, as read in from a file, are organized by software into two dimensional arrays, the indices of the arrays being used as coordinates. There may also be additional indices for multispectral data, but these indices do not refer to spatial coordinates but spectral, and so of not of concern here.

Many different types of raster data may be georeferenced, and there may be subtle ways in which the nature of the data itself influences how the coordinate system (Raster Space) is defined for raster data. For example, pixel data derived from imaging devices and sensors represent aggregate values collected over a small, finite, geographic area, and so it is natural to define coordinate systems in which the pixel value is thought of as filling an area. On the other hand, digital elevations models may consist of discrete "postings", which may best be considered as point measurements at the vertices of a grid, and not in the interior of a cell.

### 2.5.2.2 Raster Space

The choice of origin for raster space is not entirely arbitrary, and depends upon the nature of the data collected. Raster space coordinates shall be referred to by their pixel types, i.e., as "PixelIsArea" or "PixelIsPoint".

Note: For simplicity, both raster spaces documented below use a fixed pixel size and spacing of 1. Information regarding the visual representation of this data, such as pixels with non-unit aspect ratios, scales, orientations, etc, are best communicated with the TIFF 6.0 standard tags.

+-----------------------------------+

### "PixelIsArea" Raster Space

The "PixelIsArea" raster grid space R, which is the default, uses coordinates I and J, with (0,0) denoting the upper-left corner of the image, and increasing I to the right, increasing J down. The first pixel-value fills the square grid cell with the bounds:

top-left = (0,0), bottom-right = (1,1)

and so on; by extension this one-by-one grid cell is also referred to as a pixel. An N by M pixel image covers an are with the mathematically defined bounds (0,0),(N,M).

```
  (0,0)
   +---+---+-> I
   | * | * |
   +---+---+        Standard (PixelIsArea) TIFF Raster space R,
   | (1,1)  (2,1)      showing the areas (*) of several pixels.
   |
   J
```

+-----------------------------------+

### "PixelIsPoint" Raster Space

The PixelIsPoint raster grid space R uses the same coordinate axis names as used in PixelIsArea Raster space, with increasing I to the right, increasing J down. The first pixel-value however, is realized as a point value located at (0,0). An N by M pixel image consists of points which fill the mathematically defined bounds (0,0),(N-1,M-1).

```
  (0,0)   (1,0)
   *-------*------> I
   |       |
   |       |      PixelIsPoint TIFF Raster space R,
   *-------*        showing the location (*) of several pixels.
   |     (1,1)
   J
```

If a point-pixel image were to be displayed on a display device with pixel cells having the same size as the raster spacing, then the upper-left corner of the displayed image would be located in raster space at (-0.5, -0.5).

+-----------------------------------+

## 2.5.3 Model Coordinate Systems

The following methods of describing spatial model locations (as opposed to raster) are recognized in Geotiff:

    Geographic coordinates
    Geocentric coordinates
    Projected coordinates
    Vertical coordinates

Geographic, geocentric and projected coordinates are all imposed on models of the earth. To describe a location uniquely, a coordinate set must be referenced to an adequately defined coordinate system. If a coordinate system is from the Geotiff standard definitions, the only reference required is the standard coordinate system code/name. If the coordinate system is non-standard, it must be defined. The required definitions are described below.

Projected coordinates, local grid coordinates, and (usually) geographical coordinates, form two dimensional horizontal coordinate systems (i.e., horizontal with respect to the earth's surface). Height is not part of these systems. To describe a position in three dimensions it is necessary to consider height as a second one dimensional vertical coordinate system.

To georeference an image in GeoTIFF, you must specify a Raster Space coordinate system, choose a horizontal model coordinate system, and a transformation between these two, as will be described in section 2.6

+-----------------------------------+

### 2.5.3.1 Geographic Coordinate Systems

Geographic Coordinate Systems are those that relate angular latitude and longitude (and optionally geodetic height) to an actual point on the earth. The process by which this is accomplished is rather complex, and so we describe the components of the process in detail here.

+--------------------------------+

## Ellipsoidal Models of the Earth

The geoid - the earth stripped of all topography - forms a reference surface for the earth. However, because it is related to the earth's gravity field, the geoid is a very complex surface; indeed, at a detailed level its description is not well known. The geoid is therefore not used in practical mapping.

It has been found that an oblate ellipsoid (an ellipse rotated about its minor axis) is a good approximation to the geoid and therefore a good model of the earth. Many approximations exist: several hundred ellipsoids have been defined for scientific purposes and about 30 are in day to day use for mapping. The size and shape of these ellipsoids can be defined through two parameters. Geotiff requires one of these to be

the semi-major axis (a),

and the second to be either

the inverse flattening (1/f)

or

the semi-minor axis (b).

Historical models exist which use a spherical approximation; such models are not recommended for modern applications, but if needed the size of a model sphere may be defined by specifying identical values for the semimajor and semiminor axes; the inverse flattening cannot be used as it becomes infinite for perfect spheres.

Other ellipsoid parameters needed for mapping applications, for example the square of the eccentricity, can easily be calculated by an application from the two defining parameters. Note that Geotiff uses the modern geodesy convention for the symbol (b) for the semi-minor axis. No provision is made for mapping other planets in which a tri-dimensional (triaxial) ellipsoid might be required, where (b) would represent the semi-median axis and (c) the semi-minor axis.

Numeric codes for ellipsoids regularly used for earth-mapping are included in the Geotiff reference lists.

+--------------------------------+

## Latitude and Longitude

The coordinate axes of the system referencing points on an ellipsoid are called latitude and longitude. More precisely, **geodetic** latitude and longitude are required in this Geotiff standard. A discussion of the several other types of latitude and longitude is beyond the scope of this document as they are not required for conventional mapping.

Latitude is defined to be the angle subtended with the ellipsoid's equatorial plane by a perpendicular through the surface of the ellipsoid from a point. Latitude is positive if north of the equator, negative if south.

Longitude is defined to be the angle measured about the minor (polar) axis of the ellipsoid from a prime meridian (see below) to the meridian through a point, positive if east of the prime meridian and negative if west. Unlike latitude which has a natural origin at the equator, there is no feature on the ellipsoid which forms a natural origin for the measurement of longitude. The zero longitude can be any defined meridian. Historically, nations have used the meridian through their national astronomical observatories, giving rise to several prime meridians. By international convention, the meridian through Greenwich, England is the standard prime meridian. Longitude is only unambiguous if the longitude of its prime meridian relative to Greenwich is given. Prime meridians other than Greenwich which are sometimes used for earth mapping are included in the Geotiff reference lists.

+-----------------------------------+

## Geodetic Datums

As well as there being several ellipsoids in use to model the earth, any one particular ellipsoid can have its location and orientation relative to the earth defined in different ways. If the relationship between the ellipsoid and the earth is changed, then the geographical coordinates of a point will change.

Conversely, for geographical coordinates to uniquely describe a location the relationship between the earth and the ellipsoid must be defined.  This relationship is described by a geodetic datum. An exact geodetic definition of geodetic datums is beyond the current scope of Geotiff. However the Geotiff standard requires that the geodetic datum being utilized be identified by numerical code. If required, defining parameters for the geodetic datum can be included as a citation.

+-----------------------------------+

## Defining Geographic Coordinate Systems

In summary, geographic coordinates are only unique if qualified by the code of the geographic coordinate system to which they belong. A geographic coordinate system has two axes, latitude and longitude, which are only unambiguous when both of the related prime meridian and geodetic datum are given, and in turn the geodetic datum definition includes the definition of an ellipsoid. The Geotiff standard includes a list of frequently used geographic coordinate systems and their component ellipsoids, geodetic datums and prime meridians. Within the Geotiff standard a geographic coordinate system can be identified either by

the code of a standard geographic coordinate system

or by

a user-defined system.

The user is expected to provide geographic coordinate system code/name, geodetic datum code/name, ellipsoid code (if in standard) or ellipsoid name and two defining parameters (a) and either (1/f) or (b), and prime meridian code (if in standard) or name and longitude relative to Greenwich.

+-----------------------------------+

### 2.5.3.2 Geocentric Coordinate Systems

A geocentric coordinate system is a 3-dimensional coordinate system with its origin at or near the center of the earth and with 3 orthogonal axes. The Z-axis is in or parallel to the earth's axis of rotation (or to the axis around which the rotational axis precesses). The X-axis is in or parallel to the plane of the equator and passes through its intersection with the Greenwich meridian, and the Y-axis is in the plane of the equator forming a right-handed coordinate system with the X and Z axes.

Geocentric coordinate systems are not frequently used for describing locations, but they are often utilized as an intermediate step when transforming between geographic coordinate systems. (Coordinate system transformations are described in section 2.6 below).

In the Geotiff standard, a geocentric coordinate system can be identified, either

through the geographic code (which in turn implies a datum),

or

through a user-defined name.

+------------------------------------+

### 2.5.3.3 Projected Coordinate Systems

Although a geographical coordinate system is mathematically two dimensional, it describes a three dimensional object and cannot be represented on a plane surface without distortion. Map projections are transformations of geographical coordinates to plane coordinates in which the characteristics of the distortions are controlled. A map projection consists of a coordinate system transformation method and a set of defining parameters. A projected coordinate system (PCS) is a two dimensional (horizontal) coordinate set which, for a specific map projection, has a single and unambiguous transformation to a geographic coordinate system.

In GeoTIFF PCS's are defined using the POSC/EPSG system, in which the PCS planar coordinate system, the Geographic coordinate system, and the transformation between them, are broken down into simpler logical components. Here are schematic formulas showing how the Projected Coordinate Systems and Geographic Coordinates Systems are encoded:

```
  Projected_CS  =  Geographic_CS + Projection
  Geographic_CS =  Angular_Unit + Geodetic_Datum + Prime_Meridian
  Projection    =  Linear Unit + Coord_Transf_Method + CT_Parameters
  Coord_Transf_Method   = { TransverseMercator | LambertCC | ...}
  CT_Parameters = {OriginLatitude + StandardParallel+...}
```

(See also the Reference Parameters documentation in section 2.5.4).

Notice that "Transverse Mercator" is not referred to as a "Projection", but rather as a "Coordinate Transformation Method"; in GeoTIFF, as in EPSG/POSC, the word "Projection" is reserved for particular, well-defined systems in which both the coordinate transformation method, its defining parameters, and their linear units are established.

Several tens of coordinate transformation methods have been developed. Many are very similar and for practical purposes can be considered to give identical results. For example in the Geotiff standard Gauss-Kruger and Gauss-Boaga projection types are considered to be of the type Transverse Mercator. Geotiff includes a listing of commonly used projection defining parameters.

Different algorithms require different defining parameters. A future version of Geotiff will include formulas for specific map projection algorithms recommended for use with listed projection parameters.

To limit the magnitude of distortions of projected coordinate systems, the boundaries of usage are sometimes restricted. To cover more extensive areas, two or more projected coordinate systems may be required. In some cases many of the defining parameters of a set of projected coordinate systems will be held constant.

The Geotiff standard does not impose a strict hierarchy onto such zoned systems such as US State Plane or UTM, but considers each zone to be a discrete projected coordinate system; the ProjectedCSTypeGeoKey code value alone is sufficient to identify the standard coordinate systems.

Within the Geotiff standard a projected coordinate system can be identified either by

the code of a standard projected coordinate system

or by

a user-defined system.

User-define projected coordinate systems may be defined by defining the Geographic Coordinate System, the coordinate transformation method and its associated parameters, as well as the planar system's linear units.

### 2.5.3.4 Vertical Coordinate Systems

Many uses of Geotiff will be limited to a two-dimensional, horizontal, description of location for which geographic coordinate systems and projected coordinate systems are adequate. If a three-dimensional description of location is required Geotiff allows this either through the use of a geocentric coordinate system or by defining a vertical coordinate system and using this together with a geographic or projected coordinate system.

In general usage, elevations and depths are referenced to a surface at or close to the geoid. Through increasing use of satellite positioning systems the ellipsoid is increasingly being used as a vertical reference surface. The relationship between the geoid and an ellipsoid is in general not well known, but is required when coordinate system transformations are to be executed.

+-----------------------------------+

### 2.5.4 Reference Parameters

# APPENDIX GG

Most of the numerical coding systems and coordinate system definitions are based on the hierarchical system developed by EPSG/POSC. The complete set of EPSG tables used in GeoTIFF is available at:

ftp://ftpmcmc.cr.usgs.gov/release/geotiff/jpl-mirror/tables

or:

ftp://mtritter.jpl.nasa.gov/pub/tiff/geotiff/tables

Appended below is the README.TXT file that accompanies the tables of defining parameters for those codes:

```
+-----------------------------------+
|      EPSG Geodesy Parameters      |
|    version 2.1, 2nd June 1995.    |
+-----------------------------------+
```

The European Petroleum Survey Group (EPSG) has compiled and is distributing this set of parameters defining various geodetic and cartographic coordinate systems to encourage standardisation across the Exploration and Production segment of the oil industry. The data is included as reference data in the Geotiff data exchange specification, in Iris21 the Petroconsultants data model, and in Epicentre, the POSC data model. Parameters map directly to the POSC Epicentre model v2.0, except for data item codes which are included in the files for data management purposes. Geodetic datum parameters are embedded within the geographic coordinate system file. This has been done to ease parameter maintenance as there is a high correlation between geodetic datum names and geographic coordinate system names. The Projected Coordinate System v2.0 tabulation consists of systems associated with locally used projections. Systems utilising the popular UTM grid system have also been included.

Criteria used for material in these lists include:

- information must be in the public domain: "private" data is not included.
- data must be in current use.
- parameters are given to a precision consistent with coordinates being to a precision of one centimetre.

The user assumes the entire risk as to the accuracy and the use of this data. The data may be copied and distributed subject to the following conditions:

1) All data must then be copied without modification and all pages must be included;

2) All components of this data set must be distributed together;

3) The data may not be distributed for profit by any third party; and

4) Acknowledgement to the original source must be given.

INFORMATION PROVIDED IN THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES

```
OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.

Data is distributed on MS-DOS formatted diskette in comma-
separated record format.  Additional copies may be obtained
from Jean-Patrick Girbig at the address below at a cost of
US$100 to cover media and shipping, payment to be made in
favour of Petroconsultants S.A at Union Banque Suisses,
1211 Geneve 11, Switzerland (compte number 403 458 60 K).

The data is to be made available on a bulletin board shortly.


Shipping List
-------------

This data set consists of 8 files:

PROJCS.CSV  Tabulation of Projected Coordinate Systems to
            which map grid coordinates may be referenced.

GEOGCS.CSV  Tabulation of Geographic Coordinate Systems to
            which latitude and longitude coordinates may be
            referenced.  This table includes the equivalent
            geocentric coordinate systems and also the
            geodetic datum, reference to which allows latitude
            and longitude or geocentric XYZ to uniquely
            describe a location on the earth.

VERTCS.CSV  Tabulation of Vertical Coordinate Systems to
            which heights or depths may be referenced. This
            table is currently in an early form.

PROJ.CSV    Tabulation of transformation methods and
            parameters through which Projected Coordinate
            Systems are defined and related to Geographic
            Coordinate Systems.

ELLIPS.CSV  Tabulation of reference ellipsoids upon which
            geodetic datums are based.

PMERID.CSV  Tabulation of prime meridians upon which geodetic
            datums are based.

UNITS.CSV   Tabulation of length units used in Projected and
            Vertical Coordinate Systems and angle units used
            in Geographic Coordinate Systems.

README.TXT  This file.
```

```
+----------------------------------------------------------------------+
```

## 2.6 Coordinate Transformations

The purpose of Geotiff is to allow the definitive identification of georeferenced locations within a raster dataset. This is generally accomplished through tying raster space coordinates to a model space coordinate system, when no further information is required. In the GeoTIFF nomenclature, "georeferencing" refers to tying raster space to a model space M, while "geocoding" refers to defining how the model space M assigns coordinates to points on the earth.

The three tags defined below may be used for defining the relationship between R and M, and the relationship may be diagrammed as:

```
           ModelPixelScaleTag
           ModelTiepointTag
    R  ----------- OR -------------> M
  (I,J,K)  ModelTransformationTag   (X,Y,Z)
```

The next section describes these Baseline georeferencing tags in detail.

```
+----------------------------------+
```

### 2.6.1 GeoTIFF Tags for Coordinate Transformations

For most common applications, the transformation between raster and model space may be defined with a set of raster-to-model tiepoints and scaling parameters. The following two tags may be used for this purpose:

```
ModelTiepointTag:
      Tag = 33922 (8482.H)
      Type = DOUBLE (IEEE Double precision)
      N = 6*K,  K = number of tiepoints
      Alias: GeoreferenceTag
      Owner: Intergraph
```

This tag stores raster->model tiepoint pairs in the order

```
     ModelTiepointTag = (...,I,J,K, X,Y,Z...),
```

where (I,J,K) is the point at location (I,J) in raster space with pixel-value K, and (X,Y,Z) is a vector in model space. In most cases the model space is only two-dimensional, in which case both K and Z should be set to zero; this third dimension is provided in anticipation of future support for 3D digital elevation models and vertical coordinate systems.

A raster image may be georeferenced simply by specifying its location, size and orientation in the model coordinate space M. This may be done by specifying the location of three of the four bounding corner points. However, tiepoints are only to be considered exact at the points specified; thus defining such a set of bounding tiepoints does **not** imply that the model space locations of the interior of the image may be exactly computed by a linear interpolation of these tiepoints.

However, since the relationship between the Raster space and the model space will often be an exact, affine transformation, this relationship can be defined using one set of tiepoints and the "ModelPixelScaleTag", described below, which gives the vertical and horizontal raster grid cell size, specified in model units.

If possible, the first tiepoint placed in this tag shall be the one establishing the location of the point (0,0) in raster space. However, if this is not possible (for example, if (0,0) is goes to a part of model space in which the projection is ill-defined), then there is no particular order in which the tiepoints need be listed.

For orthorectification or mosaicking applications a large number of tiepoints may be specified on a mesh over the raster image. However, the definition of associated grid interpolation methods is not in the scope of the current GeoTIFF spec.

Remark: As mentioned in section 2.5.1, all GeoTIFF information is independent of the XPosition, YPosition, and Orientation tags of the standard TIFF 6.0 spec.

The next two tags are optional tags provided for defining exact affine transformations between raster and model space; baseline GeoTIFF files may use either, but shall never use both within the same TIFF image directory.

```
ModelPixelScaleTag:
      Tag = 33550
      Type = DOUBLE (IEEE Double precision)
      N = 3
      Owner: SoftDesk
```

This tag may be used to specify the size of raster pixel spacing in the model space units, when the raster space can be embedded in the model space coordinate system without rotation, and consists of the following 3 values:

ModelPixelScaleTag = (ScaleX, ScaleY, ScaleZ)

where ScaleX and ScaleY give the horizontal and vertical spacing of raster pixels. The ScaleZ is primarily used to map the pixel value of a digital elevation model into the correct Z-scale, and so for most other purposes this value should be zero (since most model spaces are 2-D, with Z=0).

A single tiepoint in the ModelTiepointTag, together with this tag, completely determine the relationship between raster and model space; thus they comprise the two tags which Baseline GeoTIFF files most often will use to place a raster image into a "standard position" in model space.

Like the Tiepoint tag, this tag information is independent of the XPosition, YPosition, Resolution and Orientation tags of the standard TIFF 6.0 spec. However, simple reversals of orientation between raster and model space (e.g. horizontal or vertical flips) may be indicated by reversal of sign in the corresponding component of the ModelPixelScaleTag. GeoTIFF compliant readers must honor this sign-reversal convention.

This tag must not be used if the raster image requires rotation or shearing to place it into the standard model space. In such cases the transformation shall be defined with the more general ModelTransformationTag, defined below.

```
ModelTransformationTag
      Tag  =  34264  (85D8.H)
      Type =  DOUBLE
      N    =  16
      Owner: JPL Cartographic Applications Group
```

This tag may be used to specify the transformation matrix between the raster space (and its dependent pixel-value space) and the (possibly 3D) model space. If specified, the tag shall have the following organization:

```
   ModelTransformationTag = (a,b,c,d,e....m,n,o,p).
```

where

```
     model                              image
     coords =          matrix     *     coords

     |-   -|      |-                 -|  |-   -|
     |  X  |      |   a   b   c   d   |  |  I  |
     |     |      |                   |  |     |
     |  Y  |      |   e   f   g   h   |  |  J  |
     |     |   =  |                   |  |     |
     |  Z  |      |   i   j   k   l   |  |  K  |
     |     |      |                   |  |     |
     |  1  |      |   m   n   o   p   |  |  1  |
     |-   -|      |-                 -|  |-   -|
```

By convention, and without loss of generality, the following parameters are currently hard-coded and will always be the same (but must be specified nonetheless):

```
    m = n = o = 0,  p = 1.
```

For Baseline GeoTIFF, the model space is always 2-D, and so the matrix will have the more limited form:

```
     |-   -|      |-                 -|  |-   -|
     |  X  |      |   a   b   0   d   |  |  I  |
     |     |      |                   |  |     |
     |  Y  |      |   e   f   0   h   |  |  J  |
     |     |   =  |                   |  |     |
     |  Z  |      |   0   0   0   0   |  |  K  |
     |     |      |                   |  |     |
     |  1  |      |   0   0   0   1   |  |  1  |
     |-   -|      |-                 -|  |-   -|
```

Values "d" and "h" will often be used to represent translations in  X and Y, and so will not necessarily be zero. All 16 values should be specified, in all cases. Only the raster-to-model transformation is defined; if the inverse transformation is required it must be computed by the client, to the desired accuracy.

This matrix tag should not be used if the ModelTiepointTag and the ModelPixelScaleTag are already defined. If only a single tiepoint (I,J,K,X,Y,Z) is specified, and the ModelPixelScale = (Sx, Sy, Sz) is specified, then the corresponding transformation matrix may be computed from them as:

```
     |-                            -|
     |   Sx    0.0   0.0   Tx       |
|                           |     Tx = X - I/Sx
|  0.0  -Sy    0.0   Ty     |     Ty = Y + J/Sy
|                           |     Tz = Z - K/Sz  (if not 0)
|  0.0   0.0   Sz    Tz     |
|                           |
|  0.0   0.0   0.0   1.0    |
|-                         -|
```

where the -Sy is due the reversal of direction from J increasing- down in raster space to Y increasing-up in model space.

Like the Tiepoint tag, this tag information is independent of the XPosition, YPosition, and Orientation tags of the standard TIFF 6.0 spec.

Note: In Revision 0.2 and earlier, another tag was used for this matrix, which has been renamed as follows:

```
IntergraphMatrixTag
      Tag  =  33920  (8480.H)
      Type =  DOUBLE
      N    =  17 (Intergraph implementation) or 16 (GeoTIFF 0.2 impl.)
      Owner: Intergraph
```

This tag conflicts with an internal software implementation at Intergraph, and so its use is no longer encouraged. A GeoTIFF reader should look first for the new tag, and only if it is not found should it check for this older tag. If found, it should only consider it to be contain valid GeoTIFF matrix information if the tag-count is 16; the Intergraph version uses 17 values.

```
+----------------------------------+
```

## 2.6.2 Coordinate Transformation Data Flow

The dataflow of the various GeoTIFF parameter datasets is based upon the EPSG/POSC configuration. Here is the text of the description accompanying the EPSG parameter tables:

The data files (.CSV) have a hierarchical structure:

```
 +---------------------------+   +----------------------------+
 |           VERTCS          |   |           PROJCS           |
 +---------------------------+   +----------------------------+
 |Vertical Coordinate Systems|   |Projected Coordinate Systems|
 +-------------+-------------+   +------------+---------------+
               |                              |
      +--------+                              |
      |                                       |
      |            +--------------------------+
      |            |                          |
      |            |            +-------------+---------------+
      |            |            |            GEOGCS           |
      |            |            +-----------------------------+
      |            |            |Geographic Coordinate Systems|
      |            |            |Geocentric Coordinate Systems|
      |            |            +-----------------------------+
      |            |            |       Geodetic Datums       |
      |            |            +-------------+---------------+
      |            |                          |
      |            |                 +--------+-------+
                  |            |                 |                |
                  |     +------+-----+    +------+-----+   +------+-------+
                  |     |    PROJ    |    |   ELLIPS   |   |    PMERID    |
                  |     +------------+    +------------+   +--------------+
                  |     | Projection |    | Ellipsoid  |   |Prime Meridian|
                  |     | Parameters |    | Parameters |   |  Parameters  |
                  |     +------+-----+    +------+-----+   +------+-------+
                  |            |                 |                |
                  +------------+-----------+-----+----------------+
                                           |
                             +-------------+-------------+
                             |           UNITS           |
                             +---------------------------+
                             | Linear and Angular Units  |
                             +---------------------------+
```

The parameter listings are "living documents" and will be updated by the EPSG from time to time. Any comment or suggestions for improvements should be directed to:

Jean-Patrick Girbig,  
Manager Cartography,  
Petroconsultants S.A.,  
PO Box 152,  
24 Chemin de la Marie,  
1258 Perly-Geneva,  
Switzerland.

or

Roger Lott,  
Head of Survey,  
BP Exploration,  
Uxbridge One,  
Harefield Road,  
Uxbridge,  
Middlesex UB8 1PD,  
England.

Internet:  
lottrj@txpcap.hou.xwh.bp.com

Requests for the inclusion of new data should include supporting documentation. Requests for changing existing data should include reference to both the name and code of the item.

```
+------------------------------------+
```

### 2.6.3 Cookbook for Defining Transformations

Here is a 4-step guide to producing a set of Baseline GeoTIFF tags for defining coordinate transformation information of a raster dataset.

Step 1: Establish the Raster Space coordinate system used: RasterPixelIsArea or RasterPixelIsPoint.

Step 2: Establish/define the model space Type in which the image is to be georeferenced. Usually this will be a Projected Coordinate system (PCS). If you are geocoding this data set, then the model space is defined to be the corresponding geographic, geocentric or Projected coordinate system (skip to the "Cookbook" section 2.7.3 first to do determine this).

Step 3: Identify the nature of the transformations needed to tie the raster data down to the model space coordinate system:

Case 1: The model-location of a raster point (x,y) is known, but not the scale or orientations:

Use the ModelTiepointTag to define the (X,Y,Z) coordinates of the known raster point.

Case 2: The location of three non-collinear raster points are known exactly, but the linearity of the transformation is not known.

Use the ModelTiepointTag to define the (X,Y,Z) coordinates of all three known raster points. Do not compute or define the ModelPixelScale or ModelTransformation tag.

Case 3: The position and scale of the data is known exactly, and no rotation or shearing is needed to fit into the model space.

Use the ModelTiepointTag to define the (X,Y,Z) coordinates of the known raster point, and the ModelPixelScaleTag to specify the scale.

Case 4: The raster data requires rotation and/or lateral shearing to fit into the defined model space:

Use the ModelTransformation matrix to define the transformation.

Case 5: The raster data cannot be fit into the model space with a simple affine transformation (rubber-sheeting required).

Use only the ModelTiepoint tag, and specify as many tiepoints as your application requires. Note, however, that this is not a Baseline GeoTIFF implementation, and should not be used for interchange; it is recommended that the image be geometrically rectified first, and put into a standard projected coordinate system.

Step 4: Install the defined tag values in the TIFF file and close it.

+----------------------------------+

## 2.7 Geocoding Raster Data

+----------------------------------+

### 2.7.1 General Approach

A geocoded image is a georeferenced image as described in section 2.6, which also specifies a model space coordinate system (CS) between the model space M (to which the raster space has been tied) and the earth. The relationship can be diagrammed, including the associated TIFF tags, as follows:

```
     ModelPixelScaleTag
     ModelTiepointTag                  GeoKeyDirectoryTag CS
 R  -------- OR ---------------> M  --------- AND  -----------> Earth
     ModelTransformationTag            GeoDoubleParamsTag
                                       GeoAsciiParamsTag
```

The geocoding coordinate system is defined by the GeoKeyDirectoryTag, while the Georeferencing information (T) is defined by the ModelTiepointTag and the ModelPixelScale, or ModelTransformationTag. Since these two systems are independent of each other, the tags used to store the parameters are separated from each other in the GeoTIFF file to emphasize the orthogonality.

+----------------------------------+

## 2.7.2 GeoTIFF GeoKeys for Geocoding

As mentioned above, all information regarding the Model Coordinate System used in the raster data is referenced from the GeoKeyDirectoryTag, which stores all of the GeoKey entries. In the Appendix, section 6.2 summarizes all of the GeoKeys defined for baseline GeoTIFF, and their corresponding codes are documented in section 6.3. Only the Keys themselves are documented here.

+--------------------------------+

## Common Features

+--------------------------------+

## Public and Private Key and Code Ranges

GeoTIFF GeoKey ID's may take any value between 0 and 65535. Following TIFF general approach, the GeoKey ID's from 32768 and above are available for private implementations. However, no registry will be established for these keys or codes, so developers are warned to use them at their own risk.

The Key ID's from 0 to 32767 are reserved for use by the official GeoTIFF spec, and are broken down into the following sub-domains:

| | |
|---|---|
| [ 0, 1023] | Reserved |
| [ 1024, 2047] | GeoTIFF Configuration Keys |
| [ 2048, 3071] | Geographic/Geocentric CS Parameter Keys |
| [ 3072, 4095] | Projected CS Parameter Keys |
| [ 4096, 5119] | Vertical CS Parameter Keys |
| [ 5120, 32767] | Reserved |
| [32768, 65535] | Private use |

GeoKey codes, like keys and tags, also range from 0 to 65535. Following the TIFF approach, all codes from 32768 and above are available for private user implementation. There will be no registry for these codes, however, and so developers must be sure that these tags will only be used internally. Use private codes at your own risk.

The codes from 0 to 32767 for all public GeoKeys are reserved by this GeoTIFF specification.

## Common Public Code Values

For consistency, several key codes have the same meaning in all implemented GeoKeys possessing a SHORT numerical coding system:

```
       0 = undefined
   32767 = user-defined
```

The "undefined" code means that this parameter is intentionally omitted, for whatever reason. For example, the datum used for a given map may be unknown, or the accuracy of a aerial photo is so low that to specify a particular datum would imply a higher accuracy than is in the data.

The "user-defined" code means that a feature is not among the standard list, and is being explicitly defined. In cases where this is meaningful, Geokey parameters have been supplied for the user to define this feature.

"User-Defined" requirements: In each section below a specification of the additional GeoKeys required for the "user-defined" option is given. In all cases the corresponding "Citation" key is strongly recommended, as per the FGDC Metadata standard regarding "local" types.

+--------------------------------+

## GeoTIFF Configuration GeoKeys

+--------------------------------+

These keys are to be used to establish the general configuration of this file's coordinate system, including the types of raster coordinate systems, model coordinate systems, and citations if any.

+----------------------------------------------------------------+

### GTModelTypeGeoKey

Key ID = 1024
Type: SHORT (code)
Values: Section 6.3.1.1 Codes

This GeoKey defines the general type of model Coordinate system used, and to which the raster space will be transformed:unknown, Geocentric (rarely used), Geographic, Projected Coordinate System, or user-defined. If the coordinate system is a PCS, then only the PCS code need be specified. If the coordinate system does not fit into one of the standard registered PCS'S, but it uses one of the standard projections and datums, then its should be documented as a PCS model with "user-defined" type, requiring the specification of projection parameters, etc.

GeoKey requirements for User-Defined Model Type (not advisable):

GTCitationGeoKey

+----------------------------------------------------------------+

### GTRasterTypeGeoKey

Key ID = 1025
Type = Section 6.3.1.2 codes

This establishes the Raster Space coordinate system used; there are currently only two, namely RasterPixelIsPoint and RasterPixelIsArea. No user-defined raster spaces are currently supported. For variance in imaging display parameters, such as pixel aspect-ratios, use the standard TIFF 6.0 device-space tags instead.

+-----------------------------------------------------------------------+

## GTCitationGeoKey

Key ID = 1026
Type = ASCII

As with all the "Citation" GeoKeys, this is provided to give an ASCII reference to published documentation on the overall configuration of this GeoTIFF file.

+-----------------------------------------------------------------------+
+-----------------------------------+

## Geographic CS Parameter GeoKeys

+-----------------------------------+
+-----------------------------------------------------------------------+

In general, the geographic coordinate system used will be implied by the projected coordinate system code. If however, this is a user-defined PCS, or the ModelType was chosen to be Geographic, then the system must be explicitly defined here, using the Horizontal datum code.

+-----------------------------------------------------------------------+

## GeographicTypeGeoKey

Key ID = 2048
Type = SHORT (code)
Values = Section 6.3.2.1 Codes

This key may be used to specify the code for the geographic coordinate system used to map lat-long to a specific ellipsoid over the earth.

GeoKey Requirements for User-Defined geographic CS:

GeogCitationGeoKey
GeogGeodeticDatumGeoKey
GeogAngularUnitsGeoKey (if not degrees)
GeogPrimeMeridianGeoKey (if not Greenwich)

+-----------------------------------------------------------------------+

## GeogCitationGeoKey

Key ID = 2049
Type = ASCII
Values = text

General citation and reference for all Geographic CS parameters.

+-----------------------------------------------------------------------+

## GeogGeodeticDatumGeoKey

Key ID = 2050
Type = SHORT (code)
Values = Section 6.3.2.2 Codes

This key may be used to specify the horizontal datum, defining the size, position and orientation of the reference ellipsoid used in user-defined geographic coordinate systems.

```
GeoKey Requirements for User-Defined Horizontal Datum:
       GeogCitationGeoKey
       GeogEllipsoidGeoKey

+-----------------------------------------------------------------------+
```

## GeogPrimeMeridianGeoKey

```
Key ID = 2051
Type = SHORT (code)
Units: Section 6.3.2.4 code
```

Allows specification of the location of the Prime meridian for user-defined geographic coordinate systems. The default standard is Greenwich, England.

```
+-----------------------------------------------------------------------+
```

## GeogPrimeMeridianLongGeoKey

```
Key ID = 2061
Type = DOUBLE
Units =  GeogAngularUnits
```

This key allows definition of user-defined Prime Meridians, the location of which is defined by its longitude relative to Greenwich.

```
+-----------------------------------------------------------------------+
```

## GeogLinearUnitsGeoKey

```
Key ID = 2052
Type = DOUBLE
Values: Section 6.3.1.3 Codes
```

Allows the definition of geocentric CS linear units for user-defined GCS.

```
+-----------------------------------------------------------------------+
```

## GeogLinearUnitSizeGeoKey

```
Key ID = 2053
Type = DOUBLE
Units: meters
```

Allows the definition of user-defined linear geocentric units, as measured in meters.

```
+-----------------------------------------------------------------------+
```

## GeogAngularUnitsGeoKey

```
Key ID = 2054
Type = SHORT (code)
Values =  Section 6.3.1.4  Codes
```

Allows the definition of **geocentric** CS Linear units for user-defined GCS and for ellipsoids.

```
GeoKey Requirements for "user-defined" units:
    GeogCitationGeoKey
```

GeogAngularUnitSizeGeoKey

+---------------------------------------------------------------------+

## GeogAngularUnitSizeGeoKey

Key ID = 2055
Type = DOUBLE
Units: radians

Allows the definition of user-defined angular geographic units, as measured in radians.

+---------------------------------------------------------------------+

## GeogEllipsoidGeoKey

Key ID = 2056
Type = SHORT (code)
Values = Section 6.3.2.3 Codes

This key may be used to specify the coded ellipsoid used in the geodetic datum of the Geographic Coordinate System.

GeoKey Requirements for User-Defined Ellipsoid:

```
   GeogCitationGeoKey
   [GeogSemiMajorAxisGeoKey,
           [GeogSemiMinorAxisGeoKey | GeogInvFlatteningGeoKey] ]
```

+---------------------------------------------------------------------+

## GeogSemiMajorAxisGeoKey

Key ID = 2057
Type = DOUBLE
Units: Geocentric CS Linear Units

Allows the specification of user-defined Ellipsoid Semi-Major Axis (a).

+---------------------------------------------------------------------+

## GeogSemiMinorAxisGeoKey

Key ID = 2058
Type = DOUBLE
Units: Geocentric CS Linear Units

Allows the specification of user-defined Ellipsoid Semi-Minor Axis (b).

+---------------------------------------------------------------------+

## GeogInvFlatteningGeoKey

Key ID = 2059
Type = DOUBLE
Units: none.

Allows the specification of the **inverse** of user-defined Ellipsoid's flattening parameter (f). The eccentricity-squared e^2 of the ellipsoid is related to the non-inverted f by:

```
      e^2  = 2*f  - f^2
```

Note: if the ellipsoid is spherical the inverse-flattening becomes infinite; use the GeogSemiMinorAxisGeoKey instead, and set it equal to the semi-major axis length.

+---------------------------------------------------------------------+

## GeogAzimuthUnitsGeoKey

Key ID = 2060
Type = SHORT (code)
Values =  Section 6.3.1.4 Codes

This key may be used to specify the angular units of measurement used to defining azimuths, in geographic coordinate systems. These may be used for defining azimuthal parameters for some projection algorithms, and may not necessarily be the same angular units used for lat-long.

+---------------------------------------------------------------------+

+----------------------------------+

## Projected CS Parameter GeoKeys

+----------------------------------+

The PCS range of GeoKeys includes the projection and coordinate transformation keys as well. The projection keys are included in this block since they can only be used to define projected coordinate systems.

+---------------------------------------------------------------------+

## ProjectedCSTypeGeoKey

Key ID = 3072
Type = SHORT (codes)
Values: Section 6.3.3.1 codes

This code is provided to specify the projected coordinate system.

GeoKey requirements for "user-defined" PCS families:
PCSCitationGeoKey
ProjectionGeoKey

+---------------------------------------------------------------------+

## PCSCitationGeoKey

Key ID = 3073
Type = ASCII

As with all the "Citation" GeoKeys, this is provided to give an ASCII reference to published documentation on the Projected  Coordinate System particularly if this is a "user-defined" PCS.

+---------------------------------------------------------------------+

+----------------------------------+

## Projection Definition GeoKeys

+-----------------------------------+

+----------------------------------------------------------------------+

With the exception of the first two keys, these are mostly  projection-specific parameters, and only a few will be required for any particular projection type. Projected coordinate systems automatically imply a specific projection type, as well as specific parameters for that projection, and so the keys below will only be necessary for user-defined projected coordinate systems.

+----------------------------------------------------------------------+

## ProjectionGeoKey

Key ID = 3074
Type = SHORT (code)
Values:  Section 6.3.3.2 codes

Allows specification of the coordinate transformation method and projection zone parameters.  Note : when associated with an appropriate Geographic Coordinate System, this forms a Projected Coordinate System.

GeoKeys Required for "user-defined" Projections:

PCSCitationGeoKey
ProjCoordTransGeoKey
ProjLinearUnitsGeoKey
(additional parameters depending on ProjCoordTransGeoKey).

+----------------------------------------------------------------------+

## ProjCoordTransGeoKey

Key ID = 3075
Type = SHORT (code)
Values:  Section 6.3.3.3 codes

Allows specification of the coordinate transformation method used. Note: this does not include the definition of the corresponding Geographic Coordinate System to which the projected CS is related; only the transformation method is defined here.

GeoKeys Required for "user-defined" Coordinate Transformations:

PCSCitationGeoKey
<additional parameter geokeys depending on the Coord. Trans. specified).

+----------------------------------------------------------------------+

## ProjLinearUnitsGeoKey

Key ID = 3076
Type = SHORT (code)
Values: Section 6.3.1.3 codes

Defines linear units used by this projection.

+----------------------------------------------------------------------+

## ProjLinearUnitSizeGeoKey

Key ID = 3077
Type = DOUBLE
Units: meters

Defines size of user-defined linear units in meters.

+---------------------------------------------------------------------+

## ProjStdParallel1GeoKey

Key ID = 3078
Type = DOUBLE
Units: GeogAngularUnit
Alias: ProjStdParallelGeoKey (from Rev 0.2)

Latitude of primary Standard Parallel.

+---------------------------------------------------------------------+

## ProjStdParallel2GeoKey

Key ID = 3079
Type = DOUBLE
Units: GeogAngularUnit

Latitude of second Standard Parallel.

+---------------------------------------------------------------------+

## ProjNatOriginLongGeoKey

Key ID = 3080
Type = DOUBLE
Units: GeogAngularUnit
Alias: ProjOriginLongGeoKey

Longitude of map-projection Natural origin.

+---------------------------------------------------------------------+

## ProjNatOriginLatGeoKey

Key ID = 3081
Type = DOUBLE
Units: GeogAngularUnit
Alias: ProjOriginLatGeoKey

Latitude of map-projection Natural origin.

+---------------------------------------------------------------------+

## ProjFalseEastingGeoKey

Key ID = 3082
Type = DOUBLE
Units: ProjLinearUnit

Gives the easting coordinate of the map projection Natural origin.

+---------------------------------------------------------------------+

## ProjFalseNorthingGeoKey

Key ID = 3083
Type = DOUBLE

```
Units: ProjLinearUnit

   Gives the northing coordinate of the map projection Natural origin.
+----------------------------------------------------------------------+
```

### ProjFalseOriginLongGeoKey

```
Key ID = 3084
Type = DOUBLE
Units: GeogAngularUnit

   Gives the longitude of the False origin.
+----------------------------------------------------------------------+
```

### ProjFalseOriginLatGeoKey

```
Key ID = 3085
Type = DOUBLE
Units: GeogAngularUnit

   Gives the latitude of the False origin.
+----------------------------------------------------------------------+
```

### ProjFalseOriginEastingGeoKey

```
Key ID = 3086
Type = DOUBLE
Units: ProjLinearUnit

   Gives the easting coordinate of the false origin. This is NOT the False Easting,
   which is the easting attached to the Natural origin.
+----------------------------------------------------------------------+
```

### ProjFalseOriginNorthingGeoKey

```
Key ID = 3087
Type = DOUBLE
Units: ProjLinearUnit

   Gives the northing coordinate of the False origin. This is NOT the False Northing,
   which is the northing attached to the Natural origin.
+----------------------------------------------------------------------+
```

### ProjCenterLongGeoKey

```
Key ID = 3088
Type = DOUBLE
Units: GeogAngularUnit

   Longitude of Center of Projection. Note that this is not necessarily the origin
   of the projection.
+----------------------------------------------------------------------+
```

### ProjCenterLatGeoKey

```
Key ID = 3089
Type = DOUBLE
Units: GeogAngularUnit
```

Latitude of Center of Projection. Note that this is not necessarily the origin of the projection.

```
+-----------------------------------------------------------------------+
```

### ProjCenterEastingGeoKey

```
Key ID = 3090
Type = DOUBLE
Units: ProjLinearUnit
```

Gives the easting coordinate of the center. This is NOT the False Easting.

```
+-----------------------------------------------------------------------+
```

### ProjFalseOriginNorthingGeoKey

```
Key ID = 3091
Type = DOUBLE
Units: ProjLinearUnit
```

Gives the northing coordinate of the center. This is NOT the False Northing.

```
+-----------------------------------------------------------------------+
```

### ProjScaleAtNatOriginGeoKey

```
Key ID = 3092
Type = DOUBLE
Units: none
Alias: ProjScaleAtOriginGeoKey (Rev. 0.2)
```

Scale at Natural Origin. This is a ratio, so no units are required.

```
+-----------------------------------------------------------------------+
```

### ProjScaleAtCenterGeoKey

```
Key ID = 3093
Type = DOUBLE
Units: none
```

Scale at Center. This is a ratio, so no units are required.

```
+-----------------------------------------------------------------------+
```

### ProjAzimuthAngleGeoKey

```
Key ID = 3094
Type = DOUBLE
Units: GeogAzimuthUnit
```

Azimuth angle east of true north of the central line passing through the projection center (for elliptical (Hotine) Oblique Mercator). Note that this is the standard method of measuring azimuth, but is opposite the usual mathematical convention of positive indicating counter-clockwise.

```
+-----------------------------------------------------------------------+
```

### ProjStraightVertPoleLongGeoKey

```
Key ID = 3095
Type = DOUBLE
Units: GeogAngularUnit
```

Longitude at Straight Vertical Pole. For polar stereographic.

+----------------------------------------------------------------------+

## GeogAzimuthUnitsGeoKey

Key ID = 2060
Type = SHORT (code)
Values =  Section 6.3.1.4 Codes

This key is actually part of the "Geographic CS Parameter Keys" section, but is mentioned here as it is useful for defining units used in the azimuthal projection parameters.

+----------------------------------------------------------------------+

+-----------------------------------+

## Vertical CS Parameter Keys

+-----------------------------------+

Note: Vertical coordinate systems are not yet implemented. These sections are provided for future development, and any vertical coordinate systems in the current revision must be defined using the VerticalCitationGeoKey.

+----------------------------------------------------------------------+

## VerticalCSTypeGeoKey

Key ID = 4096
Type = SHORT (code)
Values =  Section 6.3.4.1  Codes

This key may be used to specify the vertical coordinate system.

+----------------------------------------------------------------------+

## VerticalCitationGeoKey

Key ID = 4097
Type = ASCII
Values =  text

This key may be used to document the vertical coordinate system used, and its parameters.

+----------------------------------------------------------------------+

## VerticalDatumGeoKey

Key ID = 4098
Type = SHORT (code)
Values =  Section 6.3.4.2  codes

This key may be used to specify the vertical datum for the vertical coordinate system.

+----------------------------------------------------------------------+

## VerticalUnitsGeoKey

Key ID = 4099
Type = SHORT (code)
Values =  Section 6.3.1.3  Codes

This key may be used to specify the vertical units of measurement used in the geographic coordinate system, in cases where geographic CS's need to reference the vertical coordinate. This, together with the Citation key, comprise the only fully implemented keys in this section, at present.

+--------------------------------+

## 2.7.3 Cookbook for Geocoding Data

Step 1: Determine the Coordinate system type of the raster data, based on the nature of the data: pixels derived from scanners or other optical devices represent areas, and most commonly will use the RasterPixelIsArea coordinate system. Pixel data such as digital elevation models represent points, and will probably use RasterPixelIsPoint coordinates.

Store in: GTRasterTypeGeoKey

Step 2: Determine which class of model space coordinates are most natural for this dataset:Geographic, Geocentric, or Projected Coordinate System. Usually this will be PCS.

Store in: GTModelTypeGeoKey

Step 3: This step depends on the GTModelType:

case PCS:  Determine the PCS projection system. Most of the PCS's used in standard State Plane and national grid systems are defined, so check this list first; the EPSG index in section 6.4 may be useful for this purpose.

Store in: ProjectedCSTypeGeoKey, ProjectedCSTypeGeoKey

If coded, it will not be necessary to specify the Projection datum, etc for this case, since all of those parameters are determined by the ProjectedCSTypeGeoKey code. Skip to step 4 from here.

If none of the coded PCS's match your system, then this is a user-defined PCS. Use the Projection code list to check for standard projection systems.

Store in: ProjectionGeoKey and skip to Geographic CS case.

If none of the Projection codes match your system, then this is a user-defined projection. Use the ProjCoordTransGeoKey to specify the coordinate transformation method (e.g. Transverse Mercator), and all of the associated parameters of that method. Also define the linear units used in the planar coordinate system.

Store in: ProjCoordTransGeoKey, ProjLinearUnitsGeoKey
    <and other CT related parameter keys>

Now continue on to define the Geographic CS, below.

case GEOCENTRIC:

case GEOGRAPHIC: Check the list of standard GCS's and use the corresponding code. To use a code both the Datum, Prime Meridian, and angular units must match those of the code.

Store in: GeographicTypeGeoKey and skip to Step 4.

If none of the coded GCS's match exactly, then this is a user-defined GCS. Check the list of standard datums, Prime Meridians, and angular units to define your system.

Store in: GeogGeodeticDatumGeoKey, GeogAngularUnitsGeoKey, GeogPrimeMeridianGeoKey and skip to Step 4.

If none of the datums match your system, you have a user-defined datum, which is an odd system, indeed. Use the GeogEllipsoidGeoKey to select the appropriate ellipsoid or use the GeogSemiMajorAxisGeoKey, GeogInvFlatteningGeoKey to define, and give a reference using the GeogCitationGeoKey.

Store in: GeogEllipsoidGeoKey, etc. and go to Step 4.

Step 4: Install the GeoKeys/codes into the GeoKeyDirectoryTag, and the DOUBLE and ASCII key values into the corresponding value-tags.

Step 5: Having completely defined the Raster & Model coordinate system, go to Cookbook section 2.6.2 and use the Georeferencing Tags to tie the raster image down onto the Model space.

+----------------------------------+

# 3 Examples

+----------------------------------+

Here are some examples of how GeoTIFF may be implemented at the Tag and GeoKey level, following the general "Cookbook" approach above.

+----------------------------------+

## 3.1 Common Examples

+----------------------------------+

### 3.1.1. UTM Projected Aerial Photo

We have an aerial photo which has been orthorectified and resampled to a UTM grid, zone 60, using WGS84 datum; the coordinates of the upper-left corner of the image is are given in easting/northing, as 350807.4m, 5316081.3m. The scanned map pixel scale is 100 meters/pixels (the actual dpi scanning ratio is irrelevant).

```
ModelTiepointTag       = (0, 0, 0,  350807.4, 5316081.3, 0.0)
ModelPixelScaleTag      = (100.0, 100.0, 0.0)
GeoKeyDirectoryTag:
      GTModelTypeGeoKey        =  1      (ModelTypeProjected)
      GTRasterTypeGeoKey       =  1      (RasterPixelIsArea)
```

```
        ProjectedCSTypeGeoKey    =  32660  (PCS_WGS84_UTM_zone_60N)
        PCSCitationGeoKey        =  "UTM Zone 60 N with WGS84"
```

Notes:

1) We did not need to specify the GCS lat-long, since the PCS_WGS84_UTM_zone_60N codes implies particular GCS and units already (WGS_84 and meters). The citation was added just for documentation.

2) The "GeoKeyDirectoryTag" is expressed using the "GeoKey" structure defined above. At the TIFF level the tags look like this:

```
   GeoKeyDirectoryTag=(  1,     0,     2,       4,
                      1024,     0,     1,       1,
                      1025,     0,     1,       1,
                      3072,     0,     1,       32660,
                      3073, 34737,    25,       0 )
   GeoAsciiParamsTag(34737)=("UTM Zone 60 N with WGS84|")
```

For the rest of these examples we will only show the GeoKey-level dump, with the understanding that the actual TIFF-level tag representation can be determined from the documentation.

+-----------------------------------+

### 3.1.2. Standard State Plane

We have a USGS State Plane Map of Texas, Central Zone, using NAD83, correctly oriented. The map resolution is 1000 meters/pixel, at origin. There is a grid intersection line in the image at pixel location (50,100), and corresponds to the projected coordinate system easting/northing of (949465.0, 3070309.1).

```
   ModelTiepointTag          = (  50,  100, 0, 949465.0, 3070309.1, 0)
   ModelPixelScaleTag        = (1000, 1000, 0)
   GeoKeyDirectoryTag:
         GTModelTypeGeoKey            =  1   (ModelTypeProjected)
         GTRasterTypeGeoKey           =  1   (RasterPixelIsArea)
         ProjectedCSTypeGeoKey        = 32139 (PCS_NAD83_Texas_Central)
```

Notice that in this case, since the PCS is a standard code, we do not need to define the GCS, datum, etc, since those are implied by the PCS code. Also, since this is NAD83, meters are used rather than US Survey feet (as in NAD 27).

+-----------------------------------+

### 3.1.3. Lambert Conformal Conic Aeronautical Chart

We have a 500 x 500 scanned aeronautical chart of Seattle, WA, using Lambert Conformal Conic projection, correctly oriented. The central meridian is at 120 degrees west. The map resolution is 1000 meters/pixel, at origin, and uses NAD27 datum. The standard parallels of the projection are at 41d20m N and 48d40m N. The latitude of the origin is at 45 degrees North, and occurs in the image at the raster coordinates (80,100). The origin is given a false easting and northing of 200000m, 1500000m.

```
   ModelTiepointTag          = (  80,  100, 0,  200000,  1500000,  0)
   ModelPixelScaleTag         = (1000, 1000, 0)
   GeoKeyDirectoryTag:
         GTModelTypeGeoKey               =  1     (ModelTypeProjected)
         GTRasterTypeGeoKey              =  1     (RasterPixelIsArea)
         GeographicTypeGeoKey            =  4267  (GCS_NAD27)
         ProjectedCSTypeGeoKey           =  32767 (user-defined)
         ProjectionGeoKey                =  32767 (user-defined)
         ProjLinearUnitsGeoKey           =  9001     (Linear_Meter)
         ProjCoordTransGeoKey            =  8  (CT_LambertConfConic_2SP)
              ProjStdParallel1GeoKey     =  41.333
              ProjStdParallel2GeoKey     =  48.666
              ProjCenterLongGeoKey       =-120.0
              ProjNatOriginLatGeoKey     =  45.0
              ProjFalseEastingGeoKey,    = 200000.0
              ProjFalseNorthingGeoKey,   = 1500000.0
```

Notice that the Tiepoint takes the false easting and northing into account when tying the raster point (50,100) to the projection origin.

+--------------------------------------------------------------------+

### 3.1.4. DMA ADRG Raster Graphic Map

The U.S. Defense Mapping Agency produces ARC digitized raster graphics datasets by scanning maps and geometrically resampling them into an equirectangular projection, so that they may be directly indexed with WGS84 geographic coordinates. The scale for one map is 0.2 degrees per pixel horizontally, 0.1 degrees per pixel vertically. If stored in a GeoTIFF file it contains the following information:

```
   ModelTiepointTag=(0.0, 0.0, 0.0,  -120.0,       32.0,     0.0)
   ModelPixelScale = (0.2, 0.1, 0.0)
   GeoKeyDirectoryTag:
         GTModelTypeGeoKey          =  2   (ModelTypeGeographic)
         GTRasterTypeGeoKey         =  1   (RasterPixelIsArea)
         GeographicTypeGeoKey       =  4326 (GCS_WGS_84)
```

+----------------------------------+

## 3.2 Less Common Examples

+----------------------------------+

### 3.2.1. Unrectified Aerial photo, known tiepoints, in degrees.

We have an aerial photo, and know only the WGS84 GPS location of several points in the scene: the upper left corner is 120 degrees West, 32 degrees North, the lower-left corner is at 120 degrees West, 30 degrees 20 minutes North, and the lower-right hand corner of the image is at 116 degrees 40 minutes  West, 30 degrees 20 minutes North. The  photo is not geometrically corrected, however, and the complete projection is therefore not known.

```
 ModelTiepointTag=(   0.0,    0.0, 0.0,  -120.0,       32.0,     0.0,
                      0.0, 1000.0, 0.0,  -120.0,       30.33333, 0.0,
                   1000.0, 1000.0, 0.0,  -116.6666667, 30.33333, 0.0)
   GeoKeyDirectoryTag:
         GTModelTypeGeoKey          =   1 (ModelTypeGeographic)
         GTRasterTypeGeoKey         =   1 (RasterPixelIsArea)
         GeographicTypeGeoKey       = 4326 (GCS_WGS_84)

 Remark: Since we have not specified the ModelPixelScaleTag, clients
    reading this GeoTIFF file are not permitted to infer that there
    is a simple linear relationship between the raster data and the
    geographic model coordinate space. The only points that are know
    to be exact are the ones specified in the tiepoint tag.


+----------------------------------+
```

### 3.2.2. Rotated Scanned Map

We have a scanned standard British National Grid, covering the 100km grid zone NZ. Consulting documentation for BNG we find that the southwest corner of the NZ zone has an easting,northing of 400000m, 500000m, relative to the BNG standard false origin. This scanned map has a resolution of 100 meter pixels, and was rotated 90 degrees to fit onto the scanner, so that the southwest corner is now the northwest corner. In this case we must use the ModelTransformation tag rather than the tiepoint/scale pair to map the raster data into model space:

```
   ModelTransformationTag  = (     0, 100.0,     0,   400000.0,
                               100.0,     0,     0,   500000.0,
                                   0,     0,     0,          0,
                                   0,     0,     0,          1)
   GeoKeyDirectoryTag:
         GTModelTypeGeoKey        =  1 ( ModelTypeProjected)
         GTRasterTypeGeoKey       =  1  (RasterPixelIsArea)
         ProjectedCSTypeGeoKey    =  27700 (PCS_British_National_Grid)
         PCSCitationGeoKey        =  "British National Grid, Zone NZ"
```

Remark: the matrix has 100.0 in the off-diagonals due to the 90 degree rotation; increasing I points north, and increasing J points east.

```
+----------------------------------+
```

### 3.2.3. Digital Elevation Model

The DMA stores digital elevation models using an equirectangular projection, so that it may be indexed with WGS84 geographic coordinates. Since elevation postings are point-values, the pixels should not be considered as filling areas, but as point-values at grid vertices. To accommodate the base elevation of the Angeles Crest forest, the pixel value of 0 corresponds to an elevation of 1000 meters relative to WGS84 reference ellipsoid. The upper left corner is at 120 degrees West, 32 degrees North, and has a pixel scale of 0.2 degrees/pixel longitude, 0.1 degrees/pixel latitude.

```
   ModelTiepointTag=(0.0, 0.0, 0.0,  -120.0,       32.0,    1000.0)
   ModelPixelScale = (0.2, 0.1, 1.0)
   GeoKeyDirectoryTag:
         GTModelTypeGeoKey          =  2    (ModelTypeGeographic)
         GTRasterTypeGeoKey         =  2    (RasterPixelIsPoint)
         GeographicTypeGeoKey       =  4326  (GCS_WGS_84)
         VerticalCSTypeGeoKey       =  5030  (VertCS_WGS_84_ellipsoid)
         VerticalCitationGeoKey     =  "WGS 84 Ellipsoid"
         VerticalUnitsGeoKey        =  9001     (Linear_Meter)

   Remarks:
          1) Note the "RasterPixelIsPoint" raster space, indicating that
             the DEM posting of the first pixel is at the raster point
             (0,0,0), and therefore corresponds to 120W,32N exactly.
          2) The third value of the "PixelScale" is 1.0 to indicate
             that a single pixel-value unit corresponds to 1 meter,
             and the last tiepoint value indicates that base value
             zero indicates 1000m above the reference surface.

   +----------------------------------+
```

# 4 Extended GeoTIFF

```
   +--------------------------------------------------------------------+
```

This section is for future development TBD.

Possible additional GeoKeys for Revision 2.0:

```
   PerspectHeightGeoKey   (General Vertical Nearsided Perspective)
   SOMInclinAngleGeoKey   (SOM)
   SOMAscendLongGeoKey    (SOM)
   SOMRevPeriodGeoKey     (SOM)
   SOMEndOfPathGeoKey     (SOM)  ? is this needed ?  SHORT
   SOMRatioGeoKey         (SOM)
   SOMPathNumGeoKey       (SOM)    SHORT
   SOMSatelliteNumGeoKey  (SOM)    SHORT
   OEAShapeMGeoKey        (Oblated Equal Area)
   OEAShapeNGeoKey        (Oblated Equal Area)
   OEARotationAngleGeoKey (Oblated Equal Area)
```

Other items for consideration:

o Digital Elevation Model information, such as Vertical Datums, Sounding Datums.

o Accuracy Keys for linear, circular, and spherical errors, etc.

o Source information, such as details of an original coordinate system
  and of transformations between it and the coordinate system in which
  data is being exchanged.

```
   +--------------------------------------------------------------------+
```

# 5 References

```
   +--------------------------------------------------------------------+
```

1. EPSG/POSC Projection Coding System Tables. Available via FTP to:

    ftp://mtritter.jpl.nasa.gov/pub/tiff/geotiff/tables

   or its USGS mirror site:

    ftp://ftpmcmc.cr.usgs.gov/release/geotiff/jpl-mirror/tables

APPENDIX GG

2. TIFF Revision 6.0 Specification: A PDF formatted version
is available via FTP to:

ftp://ftp.adobe.com/pub/adobe/DeveloperSupport/TechNotes/PDFfiles/TIFF6.pdf

PostScript formatted text versions available at:.

ftp://sgi.com/graphics/tiff/TIFF6.ps.Z   (compressed)
ftp://sgi.com/graphics/tiff/TIFF6.ps     (uncompressed)

3. LIBGEOTIFF -- Public Domain GeoTIFF library, available via anonymous
FTP to:

ftp://mtritter.jpl.nasa.gov/pub/tiff/geotiff/code

or its USGS mirror site:

ftp://ftpmcmc.cr.usgs.gov/release/geotiff/jpl-mirror/code

4. LIBTIFF -- Public Domain TIFF library, available via anonymous
FTP to:

ftp://sgi.com/graphics/tiff/

5. Spatial Data Transfer Standard (SDTS) of the USGS.
(Federal Information Processing Standard (FIPS) 173):

ftp://sdts.er.usgs.gov/pub/sdts/

SDTS Task Force
U.S. Geological Survey
526 National Center
Reston, VA 22092

E-mail: sdts@usgs.gov

6. Map use: reading, analysis, interpretation.
Muehrcke, Phillip C. 1986. Madison, WI: JP Publications.

7. Map projections: a working manual. Snyder, John P. 1987.
USGS Professional Paper 1395.
Washington, DC: United States Government Printing Office.

8. Notes for GIS and The Geographer's Craft at U. Texas, on the
World Wide Web (WWW) (current as of 10 April 1995):

http://wwwhost.cc.utexas.edu/ftp/pub/grg/gcraft/notes/notes.html

9. Digital Geographic Information Exchange Standard (DIGEST).
Allied Geographic Publication No 3, Edition 1.2 (AGeoP-3)
(NATO Unclassified).

10. POSC Petrotechnical Open Software Corporation Web site:

http://www.posc.org/

**APPENDIX GG**

```
+---------------------------------------------------------------------+
```

# 6 Appendices

```
+---------------------------------------------------------------------+
```

```
+-----------------------------------+
```

## 6.1 Tag ID Summary

Here are all of the TIFF tags (and their owners) that are used to store GeoTIFF information of any type. It is very unlikely that any other tags will be necessary in the future (since most additional information will be encoded as a GeoKey).

```
 ModelPixelScaleTag     = 33550 (SoftDesk)
 ModelTransformationTag = 34264 (JPL Carto Group)
 ModelTiepointTag       = 33922 (Intergraph)
 GeoKeyDirectoryTag     = 34735 (SPOT)
 GeoDoubleParamsTag     = 34736 (SPOT)
 GeoAsciiParamsTag      = 34737 (SPOT)
```

Obsoleted Implementation:

```
 IntergraphMatrixTag = 33920 (Intergraph) -- Use ModelTransformationTag.
```

```
+-----------------------------------+
```

## 6.2 Key ID Summary

```
+-----------------------------------+
```

```
+-----------------------------------+
```

### 6.2.1 GeoTIFF Configuration Keys

```
GTModelTypeGeoKey             = 1024 /* Section 6.3.1.1 Codes       */
GTRasterTypeGeoKey            = 1025 /* Section 6.3.1.2 Codes       */
GTCitationGeoKey              = 1026 /* documentation */
```

```
+-----------------------------------+
```

### 6.2.2 Geographic CS Parameter Keys

```
GeographicTypeGeoKey          = 2048 /* Section 6.3.2.1 Codes     */
GeogCitationGeoKey            = 2049 /* documentation             */
GeogGeodeticDatumGeoKey       = 2050 /* Section 6.3.2.2 Codes     */
GeogPrimeMeridianGeoKey       = 2051 /* Section 6.3.2.4 codes     */
GeogLinearUnitsGeoKey         = 2052 /* Section 6.3.1.3 Codes     */
GeogLinearUnitSizeGeoKey      = 2053 /* meters                    */
GeogAngularUnitsGeoKey        = 2054 /* Section 6.3.1.4 Codes     */
GeogAngularUnitSizeGeoKey     = 2055 /* radians                   */
GeogEllipsoidGeoKey           = 2056 /* Section 6.3.2.3 Codes     */
GeogSemiMajorAxisGeoKey       = 2057 /* GeogLinearUnits           */
GeogSemiMinorAxisGeoKey       = 2058 /* GeogLinearUnits           */
GeogInvFlatteningGeoKey       = 2059 /* ratio                     */
```

```
   GeogAzimuthUnitsGeoKey       = 2060 /* Section 6.3.1.4 Codes      */
   GeogPrimeMeridianLongGeoKey  = 2061 /* GeogAngularUnit            */

   +----------------------------------+
```

## 6.2.3 Projected CS Parameter Keys

```
   ProjectedCSTypeGeoKey          = 3072  /* Section 6.3.3.1 codes   */
   PCSCitationGeoKey              = 3073  /* documentation           */
   ProjectionGeoKey               = 3074  /* Section 6.3.3.2 codes   */
   ProjCoordTransGeoKey           = 3075  /* Section 6.3.3.3 codes   */
   ProjLinearUnitsGeoKey          = 3076  /* Section 6.3.1.3 codes   */
   ProjLinearUnitSizeGeoKey       = 3077  /* meters                  */
   ProjStdParallel1GeoKey         = 3078  /* GeogAngularUnit */
   ProjStdParallel2GeoKey         = 3079  /* GeogAngularUnit */
   ProjNatOriginLongGeoKey        = 3080  /* GeogAngularUnit */
   ProjNatOriginLatGeoKey         = 3081  /* GeogAngularUnit */
   ProjFalseEastingGeoKey         = 3082  /* ProjLinearUnits */
   ProjFalseNorthingGeoKey        = 3083  /* ProjLinearUnits */
   ProjFalseOriginLongGeoKey      = 3084  /* GeogAngularUnit */
   ProjFalseOriginLatGeoKey       = 3085  /* GeogAngularUnit */
   ProjFalseOriginEastingGeoKey   = 3086  /* ProjLinearUnits */
   ProjFalseOriginNorthingGeoKey  = 3087  /* ProjLinearUnits */
   ProjCenterLongGeoKey           = 3088  /* GeogAngularUnit */
   ProjCenterLatGeoKey            = 3089  /* GeogAngularUnit */
   ProjCenterEastingGeoKey        = 3090  /* ProjLinearUnits */
   ProjCenterNorthingGeoKey       = 3091  /* ProjLinearUnits */
   ProjScaleAtNatOriginGeoKey     = 3092  /* ratio   */
   ProjScaleAtCenterGeoKey        = 3093  /* ratio   */
   ProjAzimuthAngleGeoKey         = 3094  /* GeogAzimuthUnit */
   ProjStraightVertPoleLongGeoKey = 3095  /* GeogAngularUnit */

Aliases:

   ProjStdParallelGeoKey     = ProjStdParallel1GeoKey
   ProjOriginLongGeoKey      = ProjNatOriginLongGeoKey
   ProjOriginLatGeoKey       = ProjNatOriginLatGeoKey
   ProjScaleAtOriginGeoKey   = ProjScaleAtNatOriginGeoKey




   +----------------------------------+
```

## 6.2.4 Vertical CS Keys

```
   VerticalCSTypeGeoKey           = 4096   /* Section 6.3.4.1 codes   */
   VerticalCitationGeoKey         = 4097   /* documentation */
   VerticalDatumGeoKey            = 4098   /* Section 6.3.4.2 codes   */
   VerticalUnitsGeoKey            = 4099   /* Section 6.3.1.3 codes   */

+---------------------------------------------------------------------+
   +----------------------------------+
```

## 6.3 Key Code Summary

```
   +----------------------------------+
```

## 6.3.1 GeoTIFF General Codes

This section includes the general "Configuration" key codes, as well as general codes which are used by more than one key (e.g. units codes).

```
+---------------------------------+
```

### 6.3.1.1 Model Type Codes

```
Ranges:

   0               = undefined
   [    1,  32766] = GeoTIFF Reserved Codes
   32767           = user-defined
   [32768, 65535] = Private User Implementations

GeoTIFF defined CS Model Type Codes:

   ModelTypeProjected   = 1   /* Projection Coordinate System         */
   ModelTypeGeographic  = 2   /* Geographic latitude-longitude System */
   ModelTypeGeocentric  = 3   /* Geocentric (X,Y,Z) Coordinate System */

Notes:

   1. ModelTypeGeographic and ModelTypeProjected
      correspond to the FGDC metadata Geographic and
      Planar-Projected coordinate system types.



   +---------------------------------+
```

### 6.3.1.2 Raster Type Codes

```
Ranges:

   0              = undefined
   [    1,  1023] = Raster Type Codes (GeoTIFF Defined)
   [1024, 32766] = Reserved
   32767          = user-defined
   [32768, 65535]= Private User Implementations


Values:
   RasterPixelIsArea  = 1
   RasterPixelIsPoint = 2

Note: Use of "user-defined" or "undefined" raster codes is not recommended.

   +---------------------------------+
```

### 6.3.1.3 Linear Units Codes

There are several different kinds of units that may be used in geographically related raster data: linear units, angular units, units of time (e.g. for radar-return), CCD-voltages, etc. For this reason there will be a single, unique range for each kind of unit, broken down into the following currently defined ranges:

```
Ranges:
```

| | |
|---|---|
| 0 | = undefined |
| [ 1, 2000] | = Obsolete GeoTIFF codes |
| [2001, 8999] | = Reserved by GeoTIFF |
| [9000, 9099] | = EPSG Linear Units. |
| [9100, 9199] | = EPSG Angular Units. |
| 32767 | = user-defined unit |
| [32768, 65535] | = Private User Implementations |

Linear Unit Values (See the ESPG/POSC tables for definition):

| | |
|---|---|
| Linear_Meter = | 9001 |
| Linear_Foot = | 9002 |
| Linear_Foot_US_Survey = | 9003 |
| Linear_Foot_Modified_American = | 9004 |
| Linear_Foot_Clarke = | 9005 |
| Linear_Foot_Indian = | 9006 |
| Linear_Link = | 9007 |
| Linear_Link_Benoit = | 9008 |
| Linear_Link_Sears = | 9009 |
| Linear_Chain_Benoit = | 9010 |
| Linear_Chain_Sears = | 9011 |
| Linear_Yard_Sears = | 9012 |
| Linear_Yard_Indian = | 9013 |
| Linear_Fathom = | 9014 |
| Linear_Mile_International_Nautical = | 9015 |

+----------------------------------+

#### 6.3.1.4 Angular Units Codes

These codes shall be used for any key that requires specification of an angular unit of measurement.

Angular Units

| | |
|---|---|
| Angular_Radian = | 9101 |
| Angular_Degree = | 9102 |
| Angular_Arc_Minute = | 9103 |
| Angular_Arc_Second = | 9104 |
| Angular_Grad = | 9105 |
| Angular_Gon = | 9106 |
| Angular_DMS = | 9107 |
| Angular_DMS_Hemisphere = | 9108 |

+----------------------------------+

### 6.3.2 Geographic CS Codes

+----------------------------------+

#### 6.3.2.1 Geographic CS Type Codes

Note: A Geographic coordinate system consists of both a datum and a Prime Meridian. Some of the names are very similar, and differ only in the Prime Meridian, so be

```
   sure to use the correct one. The codes beginning with GCSE_xxx are unspecified GCS
   which use ellipsoid (xxx); it is recommended that only the codes beginning with
   GCS_ be used if possible.

Ranges:

   0 = undefined
   [    1,  1000] = Obsolete EPSG/POSC Geographic Codes
   [ 1001,  3999] = Reserved by GeoTIFF
   [ 4000, 4199]  = EPSG GCS Based on Ellipsoid only
   [ 4200, 4999]  = EPSG GCS Based on EPSG Datum
   [ 5000, 32766] = Reserved by GeoTIFF
   32767          = user-defined GCS
   [32768, 65535] = Private User Implementations

Values:

  Note: Geodetic datum using Greenwich PM have codes equal to
  the corresponding Datum code - 2000.

   GCS_Adindan =      4201
   GCS_AGD66 = 4202
   GCS_AGD84 = 4203
   GCS_Ain_el_Abd =   4204
   GCS_Afgooye =      4205
   GCS_Agadez =       4206
   GCS_Lisbon =       4207
   GCS_Aratu = 4208
   GCS_Arc_1950 =     4209
   GCS_Arc_1960 =     4210
   GCS_Batavia =      4211
   GCS_Barbados =     4212
   GCS_Beduaram =     4213
   GCS_Beijing_1954 = 4214
   GCS_Belge_1950 =   4215
   GCS_Bermuda_1957 = 4216
   GCS_Bern_1898 =    4217
   GCS_Bogota =       4218
   GCS_Bukit_Rimpah = 4219
   GCS_Camacupa =     4220
   GCS_Campo_Inchauspe =     4221
   GCS_Cape =  4222
   GCS_Carthage =     4223
   GCS_Chua =  4224
   GCS_Corrego_Alegre =      4225
   GCS_Cote_d_Ivoire =       4226
   GCS_Deir_ez_Zor =  4227
   GCS_Douala =       4228
   GCS_Egypt_1907 =   4229
   GCS_ED50 =  4230
   GCS_ED87 =  4231
   GCS_Fahud = 4232
   GCS_Gandajika_1970 =      4233
   GCS_Garoua =       4234
   GCS_Guyane_Francaise =    4235
   GCS_Hu_Tzu_Shan =  4236
   GCS_HD72 =  4237
   GCS_ID74 =  4238
   GCS_Indian_1954 =  4239
   GCS_Indian_1975 =  4240
   GCS_Jamaica_1875 = 4241
   GCS_JAD69 = 4242
   GCS_Kalianpur =    4243
   GCS_Kandawala =    4244
   GCS_Kertau =       4245
```

```
GCS_KOC =   4246
GCS_La_Canoa =     4247
GCS_PSAD56 =       4248
GCS_Lake =  4249
GCS_Leigon =       4250
GCS_Liberia_1964 = 4251
GCS_Lome =  4252
GCS_Luzon_1911 =   4253
GCS_Hito_XVIII_1963 =     4254
GCS_Herat_North =  4255
GCS_Mahe_1971 =    4256
GCS_Makassar =     4257
GCS_EUREF89 =      4258
GCS_Malongo_1987 = 4259
GCS_Manoca =       4260
GCS_Merchich =     4261
GCS_Massawa =      4262
GCS_Minna = 4263
GCS_Mhast = 4264
GCS_Monte_Mario =  4265
GCS_M_poraloko =   4266
GCS_NAD27 = 4267
GCS_NAD_Michigan = 4268
GCS_NAD83 = 4269
GCS_Nahrwan_1967 = 4270
GCS_Naparima_1972 =       4271
GCS_GD49 =  4272
GCS_NGO_1948 =     4273
GCS_Datum_73 =     4274
GCS_NTF =   4275
GCS_NSWC_9Z_2 =    4276
GCS_OSGB_1936 =    4277
GCS_OSGB70 =       4278
GCS_OS_SN80 =      4279
GCS_Padang =       4280
GCS_Palestine_1923 =      4281
GCS_Pointe_Noire = 4282
GCS_GDA94 = 4283
GCS_Pulkovo_1942 = 4284
GCS_Qatar = 4285
GCS_Qatar_1948 =   4286
GCS_Qornoq =       4287
GCS_Loma_Quintana =       4288
GCS_Amersfoort =   4289
GCS_RT38 =  4290
GCS_SAD69 = 4291
GCS_Sapper_Hill_1943 =    4292
GCS_Schwarzeck =   4293
GCS_Segora =       4294
GCS_Serindung =    4295
GCS_Sudan = 4296
GCS_Tananarive =   4297
GCS_Timbalai_1948 =       4298
GCS_TM65 =  4299
GCS_TM75 =  4300
GCS_Tokyo = 4301
GCS_Trinidad_1903 =       4302
GCS_TC_1948 =      4303
GCS_Voirol_1875 =  4304
GCS_Voirol_Unifie =       4305
GCS_Bern_1938 =    4306
GCS_Nord_Sahara_1959 =    4307
GCS_Stockholm_1938 =      4308
GCS_Yacare =       4309
```

```
   GCS_Yoff =  4310
   GCS_Zanderij =     4311
   GCS_MGI =   4312
   GCS_Belge_1972 =   4313
   GCS_DHDN =  4314
   GCS_Conakry_1905 = 4315
   GCS_WGS_72 =       4322
   GCS_WGS_72BE =     4324
   GCS_WGS_84 =       4326
   GCS_Bern_1898_Bern =      4801
   GCS_Bogota_Bogota =       4802
   GCS_Lisbon_Lisbon =       4803
   GCS_Makassar_Jakarta =    4804
   GCS_MGI_Ferro =    4805
   GCS_Monte_Mario_Rome =    4806
   GCS_NTF_Paris =    4807
   GCS_Padang_Jakarta =      4808
   GCS_Belge_1950_Brussels = 4809
   GCS_Tananarive_Paris =    4810
   GCS_Voirol_1875_Paris =   4811
   GCS_Voirol_Unifie_Paris = 4812
   GCS_Batavia_Jakarta =     4813
   GCS_ATF_Paris =    4901
   GCS_NDG_Paris =    4902

Ellipsoid-Only GCS:

   Note: the numeric code is equal to the code of the correspoding
   EPSG ellipsoid, minus 3000.

   GCSE_Airy1830 =    4001
   GCSE_AiryModified1849 =   4002
   GCSE_AustralianNationalSpheroid =4003
   GCSE_Bessel1841 =  4004
   GCSE_BesselModified =     4005
   GCSE_BesselNamibia =      4006
   GCSE_Clarke1858 =  4007
   GCSE_Clarke1866 =  4008
   GCSE_Clarke1866Michigan = 4009
   GCSE_Clarke1880_Benoit =  4010
   GCSE_Clarke1880_IGN =     4011
   GCSE_Clarke1880_RGS =     4012
   GCSE_Clarke1880_Arc =     4013
   GCSE_Clarke1880_SGA1922 = 4014
   GCSE_Everest1830_1937Adjustment =4015
   GCSE_Everest1830_1967Definition =4016
   GCSE_Everest1830_1975Definition =4017
   GCSE_Everest1830Modified =4018
   GCSE_GRS1980 =     4019
   GCSE_Helmert1906 = 4020
   GCSE_IndonesianNationalSpheroid =4021
   GCSE_International1924 =  4022
   GCSE_International1967 =  4023
   GCSE_Krassowsky1940 =     4024
   GCSE_NWL9D =       4025
   GCSE_NWL10D =      4026
   GCSE_Plessis1817 = 4027
   GCSE_Struve1860 =  4028
   GCSE_WarOffice =   4029
   GCSE_WGS84 =       4030
   GCSE_GEM10C =      4031
   GCSE_OSU86F =      4032
   GCSE_OSU91A =      4033
   GCSE_Clarke1880 =  4034
   GCSE_Sphere =      4035


   +----------------------------------+
```

### 6.3.2.2 Geodetic Datum Codes

Note: these codes do not include the Prime Meridian; if possible use the GCS codes above if the datum and Prime Meridian are on the list. Also, as with the GCS codes, the codes beginning with DatumE_xxx refer only to the specified ellipsoid (xxx); if possible use instead the named datums beginning with Datum_xxx

```
Ranges:,

   0 = undefined
   [    1,  1000] = Obsolete EPSG/POSC Datum Codes
   [ 1001,  5999] = Reserved by GeoTIFF
   [ 6000, 6199]  = EPSG Datum Based on Ellipsoid only
   [ 6200, 6999]  = EPSG Datum Based on EPSG Datum
   [ 6322, 6327]  = WGS Datum
   [ 6900, 6999]  = Archaic Datum
   [ 7000, 32766] = Reserved by GeoTIFF
   32767          = user-defined GCS
   [32768, 65535] = Private User Implementations

Values:

   Datum_Adindan =    6201
   Datum_Australian_Geodetic_Datum_1966 =   6202
   Datum_Australian_Geodetic_Datum_1984 =   6203
   Datum_Ain_el_Abd_1970 =   6204
   Datum_Afgooye =    6205
   Datum_Agadez =     6206
   Datum_Lisbon =     6207
   Datum_Aratu =      6208
   Datum_Arc_1950 =   6209
   Datum_Arc_1960 =   6210
   Datum_Batavia =    6211
   Datum_Barbados =   6212
   Datum_Beduaram =   6213
   Datum_Beijing_1954 =      6214
   Datum_Reseau_National_Belge_1950 =      6215
   Datum_Bermuda_1957 =      6216
   Datum_Bern_1898 =  6217
   Datum_Bogota =     6218
   Datum_Bukit_Rimpah =      6219
   Datum_Camacupa =   6220
   Datum_Campo_Inchauspe =   6221
   Datum_Cape =       6222
   Datum_Carthage =   6223
   Datum_Chua =       6224
   Datum_Corrego_Alegre =    6225
   Datum_Cote_d_Ivoire =     6226
   Datum_Deir_ez_Zor =       6227
   Datum_Douala =     6228
   Datum_Egypt_1907 = 6229
   Datum_European_Datum_1950 =      6230
   Datum_European_Datum_1987 =      6231
   Datum_Fahud =      6232
   Datum_Gandajika_1970 =    6233
   Datum_Garoua =     6234
   Datum_Guyane_Francaise =  6235
   Datum_Hu_Tzu_Shan =       6236
   Datum_Hungarian_Datum_1972 =     6237
```

```
Datum_Indonesian_Datum_1974 =    6238
Datum_Indian_1954 =       6239
Datum_Indian_1975 =       6240
Datum_Jamaica_1875 =      6241
Datum_Jamaica_1969 =      6242
Datum_Kalianpur =  6243
Datum_Kandawala =  6244
Datum_Kertau =     6245
Datum_Kuwait_Oil_Company =6246
Datum_La_Canoa =   6247
Datum_Provisional_S_American_Datum_1956 =      6248
Datum_Lake =       6249
Datum_Leigon =     6250
Datum_Liberia_1964 =      6251
Datum_Lome =       6252
Datum_Luzon_1911 = 6253
Datum_Hito_XVIII_1963 =   6254
Datum_Herat_North =       6255
Datum_Mahe_1971 =  6256
Datum_Makassar =   6257
Datum_European_Reference_System_1989 =  6258
Datum_Malongo_1987 =      6259
Datum_Manoca =     6260
Datum_Merchich =   6261
Datum_Massawa =    6262
Datum_Minna =      6263
Datum_Mhast =      6264
Datum_Monte_Mario =       6265
Datum_M_poraloko = 6266
Datum_North_American_Datum_1927 =6267
Datum_NAD_Michigan =      6268
Datum_North_American_Datum_1983 =6269
Datum_Nahrwan_1967 =      6270
Datum_Naparima_1972 =     6271
Datum_New_Zealand_Geodetic_Datum_1949 = 6272
Datum_NGO_1948 =   6273
Datum_Datum_73 =   6274
Datum_Nouvelle_Triangulation_Francaise =6275
Datum_NSWC_9Z_2 =  6276
Datum_OSGB_1936 =  6277
Datum_OSGB_1970_SN =      6278
Datum_OS_SN_1980 = 6279
Datum_Padang_1884 =       6280
Datum_Palestine_1923 =    6281
Datum_Pointe_Noire =      6282
Datum_Geocentric_Datum_of_Australia_1994 =     6283
Datum_Pulkovo_1942 =      6284
Datum_Qatar =      6285
Datum_Qatar_1948 = 6286
Datum_Qornoq =     6287
Datum_Loma_Quintana =     6288
Datum_Amersfoort = 6289
Datum_RT38 =       6290
Datum_South_American_Datum_1969 =6291
Datum_Sapper_Hill_1943 =  6292
Datum_Schwarzeck = 6293
Datum_Segora =     6294
Datum_Serindung =  6295
Datum_Sudan =      6296
Datum_Tananarive_1925 =   6297
Datum_Timbalai_1948 =     6298
Datum_TM65 =       6299
Datum_TM75 =       6300
Datum_Tokyo =      6301
```

```
      Datum_Trinidad_1903 =     6302
      Datum_Trucial_Coast_1948 =6303
      Datum_Voirol_1875 =       6304
      Datum_Voirol_Unifie_1960 =6305
      Datum_Bern_1938 =  6306
      Datum_Nord_Sahara_1959 =  6307
      Datum_Stockholm_1938 =    6308
      Datum_Yacare =     6309
      Datum_Yoff =       6310
      Datum_Zanderij =   6311
      Datum_Militar_Geographische_Institut =   6312
      Datum_Reseau_National_Belge_1972 =       6313
      Datum_Deutsche_Hauptdreiecksnetz =       6314
      Datum_Conakry_1905 =      6315
      Datum_WGS72 =      6322
      Datum_WGS72_Transit_Broadcast_Ephemeris =       6324
      Datum_WGS84 =      6326
      Datum_Ancienne_Triangulation_Francaise = 6901
      Datum_Nord_de_Guerre =    6902

   Ellipsoid-Only Datum:

      Note: the numeric code is equal to the corresponding ellipsoid
      code, minus 1000.

      DatumE_Airy1830 =  6001
      DatumE_AiryModified1849 = 6002
      DatumE_AustralianNationalSpheroid =      6003
      DatumE_Bessel1841 =       6004
      DatumE_BesselModified =   6005
      DatumE_BesselNamibia =    6006
      DatumE_Clarke1858 =       6007
      DatumE_Clarke1866 =       6008
      DatumE_Clarke1866Michigan =      6009
      DatumE_Clarke1880_Benoit =6010
      DatumE_Clarke1880_IGN =   6011
      DatumE_Clarke1880_RGS =   6012
      DatumE_Clarke1880_Arc =   6013
      DatumE_Clarke1880_SGA1922 =      6014
      DatumE_Everest1830_1937Adjustment =      6015
      DatumE_Everest1830_1967Definition =      6016
      DatumE_Everest1830_1975Definition =      6017
      DatumE_Everest1830Modified =     6018
      DatumE_GRS1980 =   6019
      DatumE_Helmert1906 =      6020
      DatumE_IndonesianNationalSpheroid =      6021
      DatumE_International1924 =6022
      DatumE_International1967 =6023
      DatumE_Krassowsky1960 =   6024
      DatumE_NWL9D =     6025
      DatumE_NWL10D =    6026
      DatumE_Plessis1817 =      6027
      DatumE_Struve1860 =       6028
      DatumE_WarOffice = 6029
      DatumE_WGS84 =     6030
      DatumE_GEM10C =    6031
      DatumE_OSU86F =    6032
      DatumE_OSU91A =    6033
      DatumE_Clarke1880 =       6034
      DatumE_Sphere =    6035

      +----------------------------------+
```

### 6.3.2.3 Ellipsoid Codes

```
Ranges:

   0 = undefined
   [    1, 1000]  = Obsolete EPSG/POSC Ellipsoid codes
   [1001,  6999]  = Reserved by GeoTIFF
   [7000,  7999]  = EPSG Ellipsoid codes
   [8000, 32766]  = Reserved by GeoTIFF
   32767          = user-defined
   [32768, 65535] = Private User Implementations

Values:

   Ellipse_Airy_1830 =       7001
   Ellipse_Airy_Modified_1849 =     7002
   Ellipse_Australian_National_Spheroid =   7003
   Ellipse_Bessel_1841 =     7004
   Ellipse_Bessel_Modified = 7005
   Ellipse_Bessel_Namibia =  7006
   Ellipse_Clarke_1858 =     7007
   Ellipse_Clarke_1866 =     7008
   Ellipse_Clarke_1866_Michigan =   7009
   Ellipse_Clarke_1880_Benoit =     7010
   Ellipse_Clarke_1880_IGN = 7011
   Ellipse_Clarke_1880_RGS = 7012
   Ellipse_Clarke_1880_Arc = 7013
   Ellipse_Clarke_1880_SGA_1922 =   7014
   Ellipse_Everest_1830_1937_Adjustment =   7015
   Ellipse_Everest_1830_1967_Definition =   7016
   Ellipse_Everest_1830_1975_Definition =   7017
   Ellipse_Everest_1830_Modified =  7018
   Ellipse_GRS_1980 = 7019
   Ellipse_Helmert_1906 =    7020
   Ellipse_Indonesian_National_Spheroid =   7021
   Ellipse_International_1924 =     7022
   Ellipse_International_1967 =     7023
   Ellipse_Krassowsky_1940 = 7024
   Ellipse_NWL_9D =   7025
   Ellipse_NWL_10D =  7026
   Ellipse_Plessis_1817 =    7027
   Ellipse_Struve_1860 =     7028
   Ellipse_War_Office =      7029
   Ellipse_WGS_84 =   7030
   Ellipse_GEM_10C =  7031
   Ellipse_OSU86F =   7032
   Ellipse_OSU91A =   7033
   Ellipse_Clarke_1880 =     7034
   Ellipse_Sphere =   7035




   +----------------------------------+
```

### 6.3.2.4 Prime Meridian Codes

```
Ranges:

   0 = undefined
   [    1,   100] = Obsolete EPSG/POSC Prime Meridian codes
   [  101,  7999] = Reserved by GeoTIFF
   [ 8000,  8999] = EPSG Prime Meridian Codes
   [ 9000, 32766] = Reserved by GeoTIFF
   32767          = user-defined
   [32768, 65535] = Private User Implementations

Values:

   PM_Greenwich =     8901
   PM_Lisbon = 8902
   PM_Paris =  8903
   PM_Bogota = 8904
   PM_Madrid = 8905
   PM_Rome =   8906
   PM_Bern =   8907
   PM_Jakarta =       8908
   PM_Ferro =  8909
   PM_Brussels =      8910
   PM_Stockholm =     8911



   +-----------------------------------+
```

## 6.3.3 Projected CS Codes

```
   +-----------------------------------+
```

### 6.3.3.1 Projected CS Type Codes

```
Ranges:

   [    1,   1000]  = Obsolete EPSG/POSC Projection System Codes
   [20000,  32760]  = EPSG Projection System codes
   32767            = user-defined
   [32768,  65535]  = Private User Implementations

Special Ranges:

1. For PCS utilising GeogCS with code in range 4201 through 4321
(i.e. geodetic datum code 6201 through 6319): As far as is possible
 the PCS code will be of the format gggzz where ggg is (geodetic
datum code -2000) and zz is zone.

2. For PCS utilising GeogCS with code out of range 4201 through 4321
(i.e. geodetic datum code 6201 through 6319). PCS code 20xxx where
xxx is a sequential number.

3. Other:
   WGS72 / UTM northern hemisphere: 322zz where zz is UTM zone number
   WGS72 / UTM southern hemisphere: 323zz where zz is UTM zone number
   WGS72BE / UTM northern hemisphere: 324zz where zz is UTM zone number
   WGS72BE / UTM southern hemisphere: 325zz where zz is UTM zone number
   WGS84 / UTM northern hemisphere: 326zz where zz is UTM zone number
   WGS84 / UTM southern hemisphere: 327zz where zz is UTM zone number
   US State Plane (NAD27):   267xx/320xx
   US State Plane (NAD83):   269xx/321xx


Values:

   PCS_Adindan_UTM_zone_37N =20137
   PCS_Adindan_UTM_zone_38N =20138
   PCS_AGD66_AMG_zone_48 =   20248
```

```
PCS_AGD66_AMG_zone_49 =   20249
PCS_AGD66_AMG_zone_50 =   20250
PCS_AGD66_AMG_zone_51 =   20251
PCS_AGD66_AMG_zone_52 =   20252
PCS_AGD66_AMG_zone_53 =   20253
PCS_AGD66_AMG_zone_54 =   20254
PCS_AGD66_AMG_zone_55 =   20255
PCS_AGD66_AMG_zone_56 =   20256
PCS_AGD66_AMG_zone_57 =   20257
PCS_AGD66_AMG_zone_58 =   20258
PCS_AGD84_AMG_zone_48 =   20348
PCS_AGD84_AMG_zone_49 =   20349
PCS_AGD84_AMG_zone_50 =   20350
PCS_AGD84_AMG_zone_51 =   20351
PCS_AGD84_AMG_zone_52 =   20352
PCS_AGD84_AMG_zone_53 =   20353
PCS_AGD84_AMG_zone_54 =   20354
PCS_AGD84_AMG_zone_55 =   20355
PCS_AGD84_AMG_zone_56 =   20356
PCS_AGD84_AMG_zone_57 =   20357
PCS_AGD84_AMG_zone_58 =   20358
PCS_Ain_el_Abd_UTM_zone_37N =    20437
PCS_Ain_el_Abd_UTM_zone_38N =    20438
PCS_Ain_el_Abd_UTM_zone_39N =    20439
PCS_Ain_el_Abd_Bahrain_Grid =    20499
PCS_Afgooye_UTM_zone_38N =20538
PCS_Afgooye_UTM_zone_39N =20539
PCS_Lisbon_Portugese_Grid =      20700
PCS_Aratu_UTM_zone_22S =  20822
PCS_Aratu_UTM_zone_23S =  20823
PCS_Aratu_UTM_zone_24S =  20824
PCS_Arc_1950_Lo13 =       20973
PCS_Arc_1950_Lo15 =       20975
PCS_Arc_1950_Lo17 =       20977
PCS_Arc_1950_Lo19 =       20979
PCS_Arc_1950_Lo21 =       20981
PCS_Arc_1950_Lo23 =       20983
PCS_Arc_1950_Lo25 =       20985
PCS_Arc_1950_Lo27 =       20987
PCS_Arc_1950_Lo29 =       20989
PCS_Arc_1950_Lo31 =       20991
PCS_Arc_1950_Lo33 =       20993
PCS_Arc_1950_Lo35 =       20995
PCS_Batavia_NEIEZ =       21100
PCS_Batavia_UTM_zone_48S =21148
PCS_Batavia_UTM_zone_49S =21149
PCS_Batavia_UTM_zone_50S =21150
PCS_Beijing_Gauss_zone_13 =      21413
PCS_Beijing_Gauss_zone_14 =      21414
PCS_Beijing_Gauss_zone_15 =      21415
PCS_Beijing_Gauss_zone_16 =      21416
PCS_Beijing_Gauss_zone_17 =      21417
PCS_Beijing_Gauss_zone_18 =      21418
PCS_Beijing_Gauss_zone_19 =      21419
PCS_Beijing_Gauss_zone_20 =      21420
PCS_Beijing_Gauss_zone_21 =      21421
PCS_Beijing_Gauss_zone_22 =      21422
PCS_Beijing_Gauss_zone_23 =      21423
PCS_Beijing_Gauss_13N =   21473
PCS_Beijing_Gauss_14N =   21474
PCS_Beijing_Gauss_15N =   21475
PCS_Beijing_Gauss_16N =   21476
PCS_Beijing_Gauss_17N =   21477
PCS_Beijing_Gauss_18N =   21478
```

```
PCS_Beijing_Gauss_19N =   21479
PCS_Beijing_Gauss_20N =   21480
PCS_Beijing_Gauss_21N =   21481
PCS_Beijing_Gauss_22N =   21482
PCS_Beijing_Gauss_23N =   21483
PCS_Belge_Lambert_50 =    21500
PCS_Bern_1898_Swiss_Old = 21790
PCS_Bogota_UTM_zone_17N = 21817
PCS_Bogota_UTM_zone_18N = 21818
PCS_Bogota_Colombia_3W =  21891
PCS_Bogota_Colombia_Bogota =     21892
PCS_Bogota_Colombia_3E =  21893
PCS_Bogota_Colombia_6E =  21894
PCS_Camacupa_UTM_32S =    22032
PCS_Camacupa_UTM_33S =    22033
PCS_C_Inchauspe_Argentina_1 =    22191
PCS_C_Inchauspe_Argentina_2 =    22192
PCS_C_Inchauspe_Argentina_3 =    22193
PCS_C_Inchauspe_Argentina_4 =    22194
PCS_C_Inchauspe_Argentina_5 =    22195
PCS_C_Inchauspe_Argentina_6 =    22196
PCS_C_Inchauspe_Argentina_7 =    22197
PCS_Carthage_UTM_zone_32N =      22332
PCS_Carthage_Nord_Tunisie =      22391
PCS_Carthage_Sud_Tunisie =22392
PCS_Corrego_Alegre_UTM_23S =     22523
PCS_Corrego_Alegre_UTM_24S =     22524
PCS_Douala_UTM_zone_32N = 22832
PCS_Egypt_1907_Red_Belt = 22992
PCS_Egypt_1907_Purple_Belt =     22993
PCS_Egypt_1907_Ext_Purple =      22994
PCS_ED50_UTM_zone_28N =   23028
PCS_ED50_UTM_zone_29N =   23029
PCS_ED50_UTM_zone_30N =   23030
PCS_ED50_UTM_zone_31N =   23031
PCS_ED50_UTM_zone_32N =   23032
PCS_ED50_UTM_zone_33N =   23033
PCS_ED50_UTM_zone_34N =   23034
PCS_ED50_UTM_zone_35N =   23035
PCS_ED50_UTM_zone_36N =   23036
PCS_ED50_UTM_zone_37N =   23037
PCS_ED50_UTM_zone_38N =   23038
PCS_Fahud_UTM_zone_39N =  23239
PCS_Fahud_UTM_zone_40N =  23240
PCS_Garoua_UTM_zone_33N = 23433
PCS_ID74_UTM_zone_46N =   23846
PCS_ID74_UTM_zone_47N =   23847
PCS_ID74_UTM_zone_48N =   23848
PCS_ID74_UTM_zone_49N =   23849
PCS_ID74_UTM_zone_50N =   23850
PCS_ID74_UTM_zone_51N =   23851
PCS_ID74_UTM_zone_52N =   23852
PCS_ID74_UTM_zone_53N =   23853
PCS_ID74_UTM_zone_46S =   23886
PCS_ID74_UTM_zone_47S =   23887
PCS_ID74_UTM_zone_48S =   23888
PCS_ID74_UTM_zone_49S =   23889
PCS_ID74_UTM_zone_50S =   23890
PCS_ID74_UTM_zone_51S =   23891
PCS_ID74_UTM_zone_52S =   23892
PCS_ID74_UTM_zone_53S =   23893
PCS_ID74_UTM_zone_54S =   23894
PCS_Indian_1954_UTM_47N = 23947
PCS_Indian_1954_UTM_48N = 23948
```

```
PCS_Indian_1975_UTM_47N = 24047
PCS_Indian_1975_UTM_48N = 24048
PCS_Jamaica_1875_Old_Grid =      24100
PCS_JAD69_Jamaica_Grid =  24200
PCS_Kalianpur_India_0 =   24370
PCS_Kalianpur_India_I =   24371
PCS_Kalianpur_India_IIa = 24372
PCS_Kalianpur_India_IIIa =24373
PCS_Kalianpur_India_IVa = 24374
PCS_Kalianpur_India_IIb = 24382
PCS_Kalianpur_India_IIIb =24383
PCS_Kalianpur_India_IVb = 24384
PCS_Kertau_Singapore_Grid =      24500
PCS_Kertau_UTM_zone_47N = 24547
PCS_Kertau_UTM_zone_48N = 24548
PCS_La_Canoa_UTM_zone_20N =      24720
PCS_La_Canoa_UTM_zone_21N =      24721
PCS_PSAD56_UTM_zone_18N = 24818
PCS_PSAD56_UTM_zone_19N = 24819
PCS_PSAD56_UTM_zone_20N = 24820
PCS_PSAD56_UTM_zone_21N = 24821
PCS_PSAD56_UTM_zone_17S = 24877
PCS_PSAD56_UTM_zone_18S = 24878
PCS_PSAD56_UTM_zone_19S = 24879
PCS_PSAD56_UTM_zone_20S = 24880
PCS_PSAD56_Peru_west_zone =      24891
PCS_PSAD56_Peru_central = 24892
PCS_PSAD56_Peru_east_zone =      24893
PCS_Leigon_Ghana_Grid =   25000
PCS_Lome_UTM_zone_31N =   25231
PCS_Luzon_Philippines_I = 25391
PCS_Luzon_Philippines_II =25392
PCS_Luzon_Philippines_III =      25393
PCS_Luzon_Philippines_IV =25394
PCS_Luzon_Philippines_V = 25395
PCS_Makassar_NEIEZ =      25700
PCS_Malongo_1987_UTM_32S =25932
PCS_Merchich_Nord_Maroc = 26191
PCS_Merchich_Sud_Maroc =  26192
PCS_Merchich_Sahara =     26193
PCS_Massawa_UTM_zone_37N =26237
PCS_Minna_UTM_zone_31N =  26331
PCS_Minna_UTM_zone_32N =  26332
PCS_Minna_Nigeria_West =  26391
PCS_Minna_Nigeria_Mid_Belt =     26392
PCS_Minna_Nigeria_East =  26393
PCS_Mhast_UTM_zone_32S =  26432
PCS_Monte_Mario_Italy_1 = 26591
PCS_Monte_Mario_Italy_2 = 26592
PCS_M_poraloko_UTM_32N =  26632
PCS_M_poraloko_UTM_32S =  26692
PCS_NAD27_UTM_zone_3N =   26703
PCS_NAD27_UTM_zone_4N =   26704
PCS_NAD27_UTM_zone_5N =   26705
PCS_NAD27_UTM_zone_6N =   26706
PCS_NAD27_UTM_zone_7N =   26707
PCS_NAD27_UTM_zone_8N =   26708
PCS_NAD27_UTM_zone_9N =   26709
PCS_NAD27_UTM_zone_10N =  26710
PCS_NAD27_UTM_zone_11N =  26711
PCS_NAD27_UTM_zone_12N =  26712
PCS_NAD27_UTM_zone_13N =  26713
PCS_NAD27_UTM_zone_14N =  26714
PCS_NAD27_UTM_zone_15N =  26715
```

```
PCS_NAD27_UTM_zone_16N =  26716
PCS_NAD27_UTM_zone_17N =  26717
PCS_NAD27_UTM_zone_18N =  26718
PCS_NAD27_UTM_zone_19N =  26719
PCS_NAD27_UTM_zone_20N =  26720
PCS_NAD27_UTM_zone_21N =  26721
PCS_NAD27_UTM_zone_22N =  26722
PCS_NAD27_Alabama_East =  26729
PCS_NAD27_Alabama_West =  26730
PCS_NAD27_Alaska_zone_1 = 26731
PCS_NAD27_Alaska_zone_2 = 26732
PCS_NAD27_Alaska_zone_3 = 26733
PCS_NAD27_Alaska_zone_4 = 26734
PCS_NAD27_Alaska_zone_5 = 26735
PCS_NAD27_Alaska_zone_6 = 26736
PCS_NAD27_Alaska_zone_7 = 26737
PCS_NAD27_Alaska_zone_8 = 26738
PCS_NAD27_Alaska_zone_9 = 26739
PCS_NAD27_Alaska_zone_10 =26740
PCS_NAD27_California_I =  26741
PCS_NAD27_California_II = 26742
PCS_NAD27_California_III =26743
PCS_NAD27_California_IV = 26744
PCS_NAD27_California_V =  26745
PCS_NAD27_California_VI = 26746
PCS_NAD27_California_VII =26747
PCS_NAD27_Arizona_East =  26748
PCS_NAD27_Arizona_Central =      26749
PCS_NAD27_Arizona_West =  26750
PCS_NAD27_Arkansas_North =26751
PCS_NAD27_Arkansas_South =26752
PCS_NAD27_Colorado_North =26753
PCS_NAD27_Colorado_Central =     26754
PCS_NAD27_Colorado_South =26755
PCS_NAD27_Connecticut =   26756
PCS_NAD27_Delaware =      26757
PCS_NAD27_Florida_East =  26758
PCS_NAD27_Florida_West =  26759
PCS_NAD27_Florida_North = 26760
PCS_NAD27_Hawaii_zone_1 = 26761
PCS_NAD27_Hawaii_zone_2 = 26762
PCS_NAD27_Hawaii_zone_3 = 26763
PCS_NAD27_Hawaii_zone_4 = 26764
PCS_NAD27_Hawaii_zone_5 = 26765
PCS_NAD27_Georgia_East =  26766
PCS_NAD27_Georgia_West =  26767
PCS_NAD27_Idaho_East =    26768
PCS_NAD27_Idaho_Central = 26769
PCS_NAD27_Idaho_West =    26770
PCS_NAD27_Illinois_East = 26771
PCS_NAD27_Illinois_West = 26772
PCS_NAD27_Indiana_East =  26773
PCS_NAD27_BLM_14N_feet =  26774
PCS_NAD27_Indiana_West =  26774
PCS_NAD27_BLM_15N_feet =  26775
PCS_NAD27_Iowa_North =    26775
PCS_NAD27_BLM_16N_feet =  26776
PCS_NAD27_Iowa_South =    26776
PCS_NAD27_BLM_17N_feet =  26777
PCS_NAD27_Kansas_North =  26777
PCS_NAD27_Kansas_South =  26778
PCS_NAD27_Kentucky_North =26779
PCS_NAD27_Kentucky_South =26780
PCS_NAD27_Louisiana_North =      26781
```

```
PCS_NAD27_Louisiana_South =      26782
PCS_NAD27_Maine_East =    26783
PCS_NAD27_Maine_West =    26784
PCS_NAD27_Maryland =      26785
PCS_NAD27_Massachusetts = 26786
PCS_NAD27_Massachusetts_Is =     26787
PCS_NAD27_Michigan_North =26788
PCS_NAD27_Michigan_Central =     26789
PCS_NAD27_Michigan_South =26790
PCS_NAD27_Minnesota_North =      26791
PCS_NAD27_Minnesota_Cent =26792
PCS_NAD27_Minnesota_South =      26793
PCS_NAD27_Mississippi_East =     26794
PCS_NAD27_Mississippi_West =     26795
PCS_NAD27_Missouri_East = 26796
PCS_NAD27_Missouri_Central =     26797
PCS_NAD27_Missouri_West = 26798
PCS_NAD_Michigan_Michigan_East = 26801
PCS_NAD_Michigan_Michigan_Old_Central =  26802
PCS_NAD_Michigan_Michigan_West = 26803
PCS_NAD83_UTM_zone_3N =   26903
PCS_NAD83_UTM_zone_4N =   26904
PCS_NAD83_UTM_zone_5N =   26905
PCS_NAD83_UTM_zone_6N =   26906
PCS_NAD83_UTM_zone_7N =   26907
PCS_NAD83_UTM_zone_8N =   26908
PCS_NAD83_UTM_zone_9N =   26909
PCS_NAD83_UTM_zone_10N =  26910
PCS_NAD83_UTM_zone_11N =  26911
PCS_NAD83_UTM_zone_12N =  26912
PCS_NAD83_UTM_zone_13N =  26913
PCS_NAD83_UTM_zone_14N =  26914
PCS_NAD83_UTM_zone_15N =  26915
PCS_NAD83_UTM_zone_16N =  26916
PCS_NAD83_UTM_zone_17N =  26917
PCS_NAD83_UTM_zone_18N =  26918
PCS_NAD83_UTM_zone_19N =  26919
PCS_NAD83_UTM_zone_20N =  26920
PCS_NAD83_UTM_zone_21N =  26921
PCS_NAD83_UTM_zone_22N =  26922
PCS_NAD83_UTM_zone_23N =  26923
PCS_NAD83_Alabama_East =  26929
PCS_NAD83_Alabama_West =  26930
PCS_NAD83_Alaska_zone_1 = 26931
PCS_NAD83_Alaska_zone_2 = 26932
PCS_NAD83_Alaska_zone_3 = 26933
PCS_NAD83_Alaska_zone_4 = 26934
PCS_NAD83_Alaska_zone_5 = 26935
PCS_NAD83_Alaska_zone_6 = 26936
PCS_NAD83_Alaska_zone_7 = 26937
PCS_NAD83_Alaska_zone_8 = 26938
PCS_NAD83_Alaska_zone_9 = 26939
PCS_NAD83_Alaska_zone_10 =26940
PCS_NAD83_California_1 =  26941
PCS_NAD83_California_2 =  26942
PCS_NAD83_California_3 =  26943
PCS_NAD83_California_4 =  26944
PCS_NAD83_California_5 =  26945
PCS_NAD83_California_6 =  26946
PCS_NAD83_Arizona_East =  26948
PCS_NAD83_Arizona_Central =      26949
PCS_NAD83_Arizona_West =  26950
PCS_NAD83_Arkansas_North =26951
PCS_NAD83_Arkansas_South =26952
```

```
PCS_NAD83_Colorado_North =26953
PCS_NAD83_Colorado_Central =     26954
PCS_NAD83_Colorado_South =26955
PCS_NAD83_Connecticut =   26956
PCS_NAD83_Delaware =      26957
PCS_NAD83_Florida_East =  26958
PCS_NAD83_Florida_West =  26959
PCS_NAD83_Florida_North = 26960
PCS_NAD83_Hawaii_zone_1 = 26961
PCS_NAD83_Hawaii_zone_2 = 26962
PCS_NAD83_Hawaii_zone_3 = 26963
PCS_NAD83_Hawaii_zone_4 = 26964
PCS_NAD83_Hawaii_zone_5 = 26965
PCS_NAD83_Georgia_East =  26966
PCS_NAD83_Georgia_West =  26967
PCS_NAD83_Idaho_East =    26968
PCS_NAD83_Idaho_Central = 26969
PCS_NAD83_Idaho_West =    26970
PCS_NAD83_Illinois_East = 26971
PCS_NAD83_Illinois_West = 26972
PCS_NAD83_Indiana_East =  26973
PCS_NAD83_Indiana_West =  26974
PCS_NAD83_Iowa_North =    26975
PCS_NAD83_Iowa_South =    26976
PCS_NAD83_Kansas_North =  26977
PCS_NAD83_Kansas_South =  26978
PCS_NAD83_Kentucky_North =26979
PCS_NAD83_Kentucky_South =26980
PCS_NAD83_Louisiana_North =      26981
PCS_NAD83_Louisiana_South =      26982
PCS_NAD83_Maine_East =    26983
PCS_NAD83_Maine_West =    26984
PCS_NAD83_Maryland =      26985
PCS_NAD83_Massachusetts = 26986
PCS_NAD83_Massachusetts_Is =     26987
PCS_NAD83_Michigan_North =26988
PCS_NAD83_Michigan_Central =     26989
PCS_NAD83_Michigan_South =26990
PCS_NAD83_Minnesota_North =      26991
PCS_NAD83_Minnesota_Cent =26992
PCS_NAD83_Minnesota_South =      26993
PCS_NAD83_Mississippi_East =     26994
PCS_NAD83_Mississippi_West =     26995
PCS_NAD83_Missouri_East = 26996
PCS_NAD83_Missouri_Central =     26997
PCS_NAD83_Missouri_West = 26998
PCS_Nahrwan_1967_UTM_38N =27038
PCS_Nahrwan_1967_UTM_39N =27039
PCS_Nahrwan_1967_UTM_40N =27040
PCS_Naparima_UTM_20N =    27120
PCS_GD49_NZ_Map_Grid =    27200
PCS_GD49_North_Island_Grid =     27291
PCS_GD49_South_Island_Grid =     27292
PCS_Datum_73_UTM_zone_29N =      27429
PCS_ATF_Nord_de_Guerre =  27500
PCS_NTF_France_I = 27581
PCS_NTF_France_II =       27582
PCS_NTF_France_III =      27583
PCS_NTF_Nord_France =     27591
PCS_NTF_Centre_France =   27592
PCS_NTF_Sud_France =      27593
PCS_British_National_Grid =      27700
PCS_Point_Noire_UTM_32S = 28232
PCS_GDA94_MGA_zone_48 =   28348
```

```
PCS_GDA94_MGA_zone_49 =   28349
PCS_GDA94_MGA_zone_50 =   28350
PCS_GDA94_MGA_zone_51 =   28351
PCS_GDA94_MGA_zone_52 =   28352
PCS_GDA94_MGA_zone_53 =   28353
PCS_GDA94_MGA_zone_54 =   28354
PCS_GDA94_MGA_zone_55 =   28355
PCS_GDA94_MGA_zone_56 =   28356
PCS_GDA94_MGA_zone_57 =   28357
PCS_GDA94_MGA_zone_58 =   28358
PCS_Pulkovo_Gauss_zone_4 =28404
PCS_Pulkovo_Gauss_zone_5 =28405
PCS_Pulkovo_Gauss_zone_6 =28406
PCS_Pulkovo_Gauss_zone_7 =28407
PCS_Pulkovo_Gauss_zone_8 =28408
PCS_Pulkovo_Gauss_zone_9 =28409
PCS_Pulkovo_Gauss_zone_10 =      28410
PCS_Pulkovo_Gauss_zone_11 =      28411
PCS_Pulkovo_Gauss_zone_12 =      28412
PCS_Pulkovo_Gauss_zone_13 =      28413
PCS_Pulkovo_Gauss_zone_14 =      28414
PCS_Pulkovo_Gauss_zone_15 =      28415
PCS_Pulkovo_Gauss_zone_16 =      28416
PCS_Pulkovo_Gauss_zone_17 =      28417
PCS_Pulkovo_Gauss_zone_18 =      28418
PCS_Pulkovo_Gauss_zone_19 =      28419
PCS_Pulkovo_Gauss_zone_20 =      28420
PCS_Pulkovo_Gauss_zone_21 =      28421
PCS_Pulkovo_Gauss_zone_22 =      28422
PCS_Pulkovo_Gauss_zone_23 =      28423
PCS_Pulkovo_Gauss_zone_24 =      28424
PCS_Pulkovo_Gauss_zone_25 =      28425
PCS_Pulkovo_Gauss_zone_26 =      28426
PCS_Pulkovo_Gauss_zone_27 =      28427
PCS_Pulkovo_Gauss_zone_28 =      28428
PCS_Pulkovo_Gauss_zone_29 =      28429
PCS_Pulkovo_Gauss_zone_30 =      28430
PCS_Pulkovo_Gauss_zone_31 =      28431
PCS_Pulkovo_Gauss_zone_32 =      28432
PCS_Pulkovo_Gauss_4N =    28464
PCS_Pulkovo_Gauss_5N =    28465
PCS_Pulkovo_Gauss_6N =    28466
PCS_Pulkovo_Gauss_7N =    28467
PCS_Pulkovo_Gauss_8N =    28468
PCS_Pulkovo_Gauss_9N =    28469
PCS_Pulkovo_Gauss_10N =   28470
PCS_Pulkovo_Gauss_11N =   28471
PCS_Pulkovo_Gauss_12N =   28472
PCS_Pulkovo_Gauss_13N =   28473
PCS_Pulkovo_Gauss_14N =   28474
PCS_Pulkovo_Gauss_15N =   28475
PCS_Pulkovo_Gauss_16N =   28476
PCS_Pulkovo_Gauss_17N =   28477
PCS_Pulkovo_Gauss_18N =   28478
PCS_Pulkovo_Gauss_19N =   28479
PCS_Pulkovo_Gauss_20N =   28480
PCS_Pulkovo_Gauss_21N =   28481
PCS_Pulkovo_Gauss_22N =   28482
PCS_Pulkovo_Gauss_23N =   28483
PCS_Pulkovo_Gauss_24N =   28484
PCS_Pulkovo_Gauss_25N =   28485
PCS_Pulkovo_Gauss_26N =   28486
PCS_Pulkovo_Gauss_27N =   28487
PCS_Pulkovo_Gauss_28N =   28488
```

```
PCS_Pulkovo_Gauss_29N =   28489
PCS_Pulkovo_Gauss_30N =   28490
PCS_Pulkovo_Gauss_31N =   28491
PCS_Pulkovo_Gauss_32N =   28492
PCS_Qatar_National_Grid = 28600
PCS_RD_Netherlands_Old =  28991
PCS_RD_Netherlands_New =  28992
PCS_SAD69_UTM_zone_18N =  29118
PCS_SAD69_UTM_zone_19N =  29119
PCS_SAD69_UTM_zone_20N =  29120
PCS_SAD69_UTM_zone_21N =  29121
PCS_SAD69_UTM_zone_22N =  29122
PCS_SAD69_UTM_zone_17S =  29177
PCS_SAD69_UTM_zone_18S =  29178
PCS_SAD69_UTM_zone_19S =  29179
PCS_SAD69_UTM_zone_20S =  29180
PCS_SAD69_UTM_zone_21S =  29181
PCS_SAD69_UTM_zone_22S =  29182
PCS_SAD69_UTM_zone_23S =  29183
PCS_SAD69_UTM_zone_24S =  29184
PCS_SAD69_UTM_zone_25S =  29185
PCS_Sapper_Hill_UTM_20S = 29220
PCS_Sapper_Hill_UTM_21S = 29221
PCS_Schwarzeck_UTM_33S =  29333
PCS_Sudan_UTM_zone_35N =  29635
PCS_Sudan_UTM_zone_36N =  29636
PCS_Tananarive_Laborde =  29700
PCS_Tananarive_UTM_38S =  29738
PCS_Tananarive_UTM_39S =  29739
PCS_Timbalai_1948_Borneo =29800
PCS_Timbalai_1948_UTM_49N =      29849
PCS_Timbalai_1948_UTM_50N =      29850
PCS_TM65_Irish_Nat_Grid = 29900
PCS_Trinidad_1903_Trinidad =     30200
PCS_TC_1948_UTM_zone_39N =30339
PCS_TC_1948_UTM_zone_40N =30340
PCS_Voirol_N_Algerie_ancien =    30491
PCS_Voirol_S_Algerie_ancien =    30492
PCS_Voirol_Unifie_N_Algerie =    30591
PCS_Voirol_Unifie_S_Algerie =    30592
PCS_Bern_1938_Swiss_New = 30600
PCS_Nord_Sahara_UTM_29N = 30729
PCS_Nord_Sahara_UTM_30N = 30730
PCS_Nord_Sahara_UTM_31N = 30731
PCS_Nord_Sahara_UTM_32N = 30732
PCS_Yoff_UTM_zone_28N =   31028
PCS_Zanderij_UTM_zone_21N =      31121
PCS_MGI_Austria_West =    31291
PCS_MGI_Austria_Central = 31292
PCS_MGI_Austria_East =    31293
PCS_Belge_Lambert_72 =    31300
PCS_DHDN_Germany_zone_1 = 31491
PCS_DHDN_Germany_zone_2 = 31492
PCS_DHDN_Germany_zone_3 = 31493
PCS_DHDN_Germany_zone_4 = 31494
PCS_DHDN_Germany_zone_5 = 31495
PCS_NAD27_Montana_North = 32001
PCS_NAD27_Montana_Central =      32002
PCS_NAD27_Montana_South = 32003
PCS_NAD27_Nebraska_North =32005
PCS_NAD27_Nebraska_South =32006
PCS_NAD27_Nevada_East =   32007
PCS_NAD27_Nevada_Central =32008
PCS_NAD27_Nevada_West =   32009
```

```
PCS_NAD27_New_Hampshire = 32010
PCS_NAD27_New_Jersey =    32011
PCS_NAD27_New_Mexico_East =      32012
PCS_NAD27_New_Mexico_Cent =      32013
PCS_NAD27_New_Mexico_West =      32014
PCS_NAD27_New_York_East = 32015
PCS_NAD27_New_York_Central =     32016
PCS_NAD27_New_York_West = 32017
PCS_NAD27_New_York_Long_Is =     32018
PCS_NAD27_North_Carolina =32019
PCS_NAD27_North_Dakota_N =32020
PCS_NAD27_North_Dakota_S =32021
PCS_NAD27_Ohio_North =    32022
PCS_NAD27_Ohio_South =    32023
PCS_NAD27_Oklahoma_North =32024
PCS_NAD27_Oklahoma_South =32025
PCS_NAD27_Oregon_North =  32026
PCS_NAD27_Oregon_South =  32027
PCS_NAD27_Pennsylvania_N =32028
PCS_NAD27_Pennsylvania_S =32029
PCS_NAD27_Rhode_Island =  32030
PCS_NAD27_South_Carolina_N =     32031
PCS_NAD27_South_Carolina_S =     32033
PCS_NAD27_South_Dakota_N =32034
PCS_NAD27_South_Dakota_S =32035
PCS_NAD27_Tennessee =     32036
PCS_NAD27_Texas_North =   32037
PCS_NAD27_Texas_North_Cen =      32038
PCS_NAD27_Texas_Central = 32039
PCS_NAD27_Texas_South_Cen =      32040
PCS_NAD27_Texas_South =   32041
PCS_NAD27_Utah_North =    32042
PCS_NAD27_Utah_Central =  32043
PCS_NAD27_Utah_South =    32044
PCS_NAD27_Vermont =       32045
PCS_NAD27_Virginia_North =32046
PCS_NAD27_Virginia_South =32047
PCS_NAD27_Washington_North =     32048
PCS_NAD27_Washington_South =     32049
PCS_NAD27_West_Virginia_N =      32050
PCS_NAD27_West_Virginia_S =      32051
PCS_NAD27_Wisconsin_North =      32052
PCS_NAD27_Wisconsin_Cen = 32053
PCS_NAD27_Wisconsin_South =      32054
PCS_NAD27_Wyoming_East =  32055
PCS_NAD27_Wyoming_E_Cen = 32056
PCS_NAD27_Wyoming_W_Cen = 32057
PCS_NAD27_Wyoming_West =  32058
PCS_NAD27_Puerto_Rico =   32059
PCS_NAD27_St_Croix =      32060
PCS_NAD83_Montana =       32100
PCS_NAD83_Nebraska =      32104
PCS_NAD83_Nevada_East =   32107
PCS_NAD83_Nevada_Central =32108
PCS_NAD83_Nevada_West =   32109
PCS_NAD83_New_Hampshire = 32110
PCS_NAD83_New_Jersey =    32111
PCS_NAD83_New_Mexico_East =      32112
PCS_NAD83_New_Mexico_Cent =      32113
PCS_NAD83_New_Mexico_West =      32114
PCS_NAD83_New_York_East = 32115
PCS_NAD83_New_York_Central =     32116
PCS_NAD83_New_York_West = 32117
PCS_NAD83_New_York_Long_Is =     32118
```

```
PCS_NAD83_North_Carolina =32119
PCS_NAD83_North_Dakota_N =32120
PCS_NAD83_North_Dakota_S =32121
PCS_NAD83_Ohio_North =    32122
PCS_NAD83_Ohio_South =    32123
PCS_NAD83_Oklahoma_North =32124
PCS_NAD83_Oklahoma_South =32125
PCS_NAD83_Oregon_North =  32126
PCS_NAD83_Oregon_South =  32127
PCS_NAD83_Pennsylvania_N =32128
PCS_NAD83_Pennsylvania_S =32129
PCS_NAD83_Rhode_Island =  32130
PCS_NAD83_South_Carolina =32133
PCS_NAD83_South_Dakota_N =32134
PCS_NAD83_South_Dakota_S =32135
PCS_NAD83_Tennessee =     32136
PCS_NAD83_Texas_North =   32137
PCS_NAD83_Texas_North_Cen =      32138
PCS_NAD83_Texas_Central = 32139
PCS_NAD83_Texas_South_Cen =      32140
PCS_NAD83_Texas_South =   32141
PCS_NAD83_Utah_North =    32142
PCS_NAD83_Utah_Central =  32143
PCS_NAD83_Utah_South =    32144
PCS_NAD83_Vermont =       32145
PCS_NAD83_Virginia_North =32146
PCS_NAD83_Virginia_South =32147
PCS_NAD83_Washington_North =     32148
PCS_NAD83_Washington_South =     32149
PCS_NAD83_West_Virginia_N =      32150
PCS_NAD83_West_Virginia_S =      32151
PCS_NAD83_Wisconsin_North =      32152
PCS_NAD83_Wisconsin_Cen = 32153
PCS_NAD83_Wisconsin_South =      32154
PCS_NAD83_Wyoming_East =  32155
PCS_NAD83_Wyoming_E_Cen = 32156
PCS_NAD83_Wyoming_W_Cen = 32157
PCS_NAD83_Wyoming_West =  32158
PCS_NAD83_Puerto_Rico_Virgin_Is =32161
PCS_WGS72_UTM_zone_1N =   32201
PCS_WGS72_UTM_zone_2N =   32202
PCS_WGS72_UTM_zone_3N =   32203
PCS_WGS72_UTM_zone_4N =   32204
PCS_WGS72_UTM_zone_5N =   32205
PCS_WGS72_UTM_zone_6N =   32206
PCS_WGS72_UTM_zone_7N =   32207
PCS_WGS72_UTM_zone_8N =   32208
PCS_WGS72_UTM_zone_9N =   32209
PCS_WGS72_UTM_zone_10N =  32210
PCS_WGS72_UTM_zone_11N =  32211
PCS_WGS72_UTM_zone_12N =  32212
PCS_WGS72_UTM_zone_13N =  32213
PCS_WGS72_UTM_zone_14N =  32214
PCS_WGS72_UTM_zone_15N =  32215
PCS_WGS72_UTM_zone_16N =  32216
PCS_WGS72_UTM_zone_17N =  32217
PCS_WGS72_UTM_zone_18N =  32218
PCS_WGS72_UTM_zone_19N =  32219
PCS_WGS72_UTM_zone_20N =  32220
PCS_WGS72_UTM_zone_21N =  32221
PCS_WGS72_UTM_zone_22N =  32222
PCS_WGS72_UTM_zone_23N =  32223
PCS_WGS72_UTM_zone_24N =  32224
PCS_WGS72_UTM_zone_25N =  32225
```

```
PCS_WGS72_UTM_zone_26N =  32226
PCS_WGS72_UTM_zone_27N =  32227
PCS_WGS72_UTM_zone_28N =  32228
PCS_WGS72_UTM_zone_29N =  32229
PCS_WGS72_UTM_zone_30N =  32230
PCS_WGS72_UTM_zone_31N =  32231
PCS_WGS72_UTM_zone_32N =  32232
PCS_WGS72_UTM_zone_33N =  32233
PCS_WGS72_UTM_zone_34N =  32234
PCS_WGS72_UTM_zone_35N =  32235
PCS_WGS72_UTM_zone_36N =  32236
PCS_WGS72_UTM_zone_37N =  32237
PCS_WGS72_UTM_zone_38N =  32238
PCS_WGS72_UTM_zone_39N =  32239
PCS_WGS72_UTM_zone_40N =  32240
PCS_WGS72_UTM_zone_41N =  32241
PCS_WGS72_UTM_zone_42N =  32242
PCS_WGS72_UTM_zone_43N =  32243
PCS_WGS72_UTM_zone_44N =  32244
PCS_WGS72_UTM_zone_45N =  32245
PCS_WGS72_UTM_zone_46N =  32246
PCS_WGS72_UTM_zone_47N =  32247
PCS_WGS72_UTM_zone_48N =  32248
PCS_WGS72_UTM_zone_49N =  32249
PCS_WGS72_UTM_zone_50N =  32250
PCS_WGS72_UTM_zone_51N =  32251
PCS_WGS72_UTM_zone_52N =  32252
PCS_WGS72_UTM_zone_53N =  32253
PCS_WGS72_UTM_zone_54N =  32254
PCS_WGS72_UTM_zone_55N =  32255
PCS_WGS72_UTM_zone_56N =  32256
PCS_WGS72_UTM_zone_57N =  32257
PCS_WGS72_UTM_zone_58N =  32258
PCS_WGS72_UTM_zone_59N =  32259
PCS_WGS72_UTM_zone_60N =  32260
PCS_WGS72_UTM_zone_1S =   32301
PCS_WGS72_UTM_zone_2S =   32302
PCS_WGS72_UTM_zone_3S =   32303
PCS_WGS72_UTM_zone_4S =   32304
PCS_WGS72_UTM_zone_5S =   32305
PCS_WGS72_UTM_zone_6S =   32306
PCS_WGS72_UTM_zone_7S =   32307
PCS_WGS72_UTM_zone_8S =   32308
PCS_WGS72_UTM_zone_9S =   32309
PCS_WGS72_UTM_zone_10S =  32310
PCS_WGS72_UTM_zone_11S =  32311
PCS_WGS72_UTM_zone_12S =  32312
PCS_WGS72_UTM_zone_13S =  32313
PCS_WGS72_UTM_zone_14S =  32314
PCS_WGS72_UTM_zone_15S =  32315
PCS_WGS72_UTM_zone_16S =  32316
PCS_WGS72_UTM_zone_17S =  32317
PCS_WGS72_UTM_zone_18S =  32318
PCS_WGS72_UTM_zone_19S =  32319
PCS_WGS72_UTM_zone_20S =  32320
PCS_WGS72_UTM_zone_21S =  32321
PCS_WGS72_UTM_zone_22S =  32322
PCS_WGS72_UTM_zone_23S =  32323
PCS_WGS72_UTM_zone_24S =  32324
PCS_WGS72_UTM_zone_25S =  32325
PCS_WGS72_UTM_zone_26S =  32326
PCS_WGS72_UTM_zone_27S =  32327
PCS_WGS72_UTM_zone_28S =  32328
PCS_WGS72_UTM_zone_29S =  32329
```

```
PCS_WGS72_UTM_zone_30S =  32330
PCS_WGS72_UTM_zone_31S =  32331
PCS_WGS72_UTM_zone_32S =  32332
PCS_WGS72_UTM_zone_33S =  32333
PCS_WGS72_UTM_zone_34S =  32334
PCS_WGS72_UTM_zone_35S =  32335
PCS_WGS72_UTM_zone_36S =  32336
PCS_WGS72_UTM_zone_37S =  32337
PCS_WGS72_UTM_zone_38S =  32338
PCS_WGS72_UTM_zone_39S =  32339
PCS_WGS72_UTM_zone_40S =  32340
PCS_WGS72_UTM_zone_41S =  32341
PCS_WGS72_UTM_zone_42S =  32342
PCS_WGS72_UTM_zone_43S =  32343
PCS_WGS72_UTM_zone_44S =  32344
PCS_WGS72_UTM_zone_45S =  32345
PCS_WGS72_UTM_zone_46S =  32346
PCS_WGS72_UTM_zone_47S =  32347
PCS_WGS72_UTM_zone_48S =  32348
PCS_WGS72_UTM_zone_49S =  32349
PCS_WGS72_UTM_zone_50S =  32350
PCS_WGS72_UTM_zone_51S =  32351
PCS_WGS72_UTM_zone_52S =  32352
PCS_WGS72_UTM_zone_53S =  32353
PCS_WGS72_UTM_zone_54S =  32354
PCS_WGS72_UTM_zone_55S =  32355
PCS_WGS72_UTM_zone_56S =  32356
PCS_WGS72_UTM_zone_57S =  32357
PCS_WGS72_UTM_zone_58S =  32358
PCS_WGS72_UTM_zone_59S =  32359
PCS_WGS72_UTM_zone_60S =  32360
PCS_WGS72BE_UTM_zone_1N = 32401
PCS_WGS72BE_UTM_zone_2N = 32402
PCS_WGS72BE_UTM_zone_3N = 32403
PCS_WGS72BE_UTM_zone_4N = 32404
PCS_WGS72BE_UTM_zone_5N = 32405
PCS_WGS72BE_UTM_zone_6N = 32406
PCS_WGS72BE_UTM_zone_7N = 32407
PCS_WGS72BE_UTM_zone_8N = 32408
PCS_WGS72BE_UTM_zone_9N = 32409
PCS_WGS72BE_UTM_zone_10N =32410
PCS_WGS72BE_UTM_zone_11N =32411
PCS_WGS72BE_UTM_zone_12N =32412
PCS_WGS72BE_UTM_zone_13N =32413
PCS_WGS72BE_UTM_zone_14N =32414
PCS_WGS72BE_UTM_zone_15N =32415
PCS_WGS72BE_UTM_zone_16N =32416
PCS_WGS72BE_UTM_zone_17N =32417
PCS_WGS72BE_UTM_zone_18N =32418
PCS_WGS72BE_UTM_zone_19N =32419
PCS_WGS72BE_UTM_zone_20N =32420
PCS_WGS72BE_UTM_zone_21N =32421
PCS_WGS72BE_UTM_zone_22N =32422
PCS_WGS72BE_UTM_zone_23N =32423
PCS_WGS72BE_UTM_zone_24N =32424
PCS_WGS72BE_UTM_zone_25N =32425
PCS_WGS72BE_UTM_zone_26N =32426
PCS_WGS72BE_UTM_zone_27N =32427
PCS_WGS72BE_UTM_zone_28N =32428
PCS_WGS72BE_UTM_zone_29N =32429
PCS_WGS72BE_UTM_zone_30N =32430
PCS_WGS72BE_UTM_zone_31N =32431
PCS_WGS72BE_UTM_zone_32N =32432
PCS_WGS72BE_UTM_zone_33N =32433
```

```
PCS_WGS72BE_UTM_zone_34N =32434
PCS_WGS72BE_UTM_zone_35N =32435
PCS_WGS72BE_UTM_zone_36N =32436
PCS_WGS72BE_UTM_zone_37N =32437
PCS_WGS72BE_UTM_zone_38N =32438
PCS_WGS72BE_UTM_zone_39N =32439
PCS_WGS72BE_UTM_zone_40N =32440
PCS_WGS72BE_UTM_zone_41N =32441
PCS_WGS72BE_UTM_zone_42N =32442
PCS_WGS72BE_UTM_zone_43N =32443
PCS_WGS72BE_UTM_zone_44N =32444
PCS_WGS72BE_UTM_zone_45N =32445
PCS_WGS72BE_UTM_zone_46N =32446
PCS_WGS72BE_UTM_zone_47N =32447
PCS_WGS72BE_UTM_zone_48N =32448
PCS_WGS72BE_UTM_zone_49N =32449
PCS_WGS72BE_UTM_zone_50N =32450
PCS_WGS72BE_UTM_zone_51N =32451
PCS_WGS72BE_UTM_zone_52N =32452
PCS_WGS72BE_UTM_zone_53N =32453
PCS_WGS72BE_UTM_zone_54N =32454
PCS_WGS72BE_UTM_zone_55N =32455
PCS_WGS72BE_UTM_zone_56N =32456
PCS_WGS72BE_UTM_zone_57N =32457
PCS_WGS72BE_UTM_zone_58N =32458
PCS_WGS72BE_UTM_zone_59N =32459
PCS_WGS72BE_UTM_zone_60N =32460
PCS_WGS72BE_UTM_zone_1S = 32501
PCS_WGS72BE_UTM_zone_2S = 32502
PCS_WGS72BE_UTM_zone_3S = 32503
PCS_WGS72BE_UTM_zone_4S = 32504
PCS_WGS72BE_UTM_zone_5S = 32505
PCS_WGS72BE_UTM_zone_6S = 32506
PCS_WGS72BE_UTM_zone_7S = 32507
PCS_WGS72BE_UTM_zone_8S = 32508
PCS_WGS72BE_UTM_zone_9S = 32509
PCS_WGS72BE_UTM_zone_10S =32510
PCS_WGS72BE_UTM_zone_11S =32511
PCS_WGS72BE_UTM_zone_12S =32512
PCS_WGS72BE_UTM_zone_13S =32513
PCS_WGS72BE_UTM_zone_14S =32514
PCS_WGS72BE_UTM_zone_15S =32515
PCS_WGS72BE_UTM_zone_16S =32516
PCS_WGS72BE_UTM_zone_17S =32517
PCS_WGS72BE_UTM_zone_18S =32518
PCS_WGS72BE_UTM_zone_19S =32519
PCS_WGS72BE_UTM_zone_20S =32520
PCS_WGS72BE_UTM_zone_21S =32521
PCS_WGS72BE_UTM_zone_22S =32522
PCS_WGS72BE_UTM_zone_23S =32523
PCS_WGS72BE_UTM_zone_24S =32524
PCS_WGS72BE_UTM_zone_25S =32525
PCS_WGS72BE_UTM_zone_26S =32526
PCS_WGS72BE_UTM_zone_27S =32527
PCS_WGS72BE_UTM_zone_28S =32528
PCS_WGS72BE_UTM_zone_29S =32529
PCS_WGS72BE_UTM_zone_30S =32530
PCS_WGS72BE_UTM_zone_31S =32531
PCS_WGS72BE_UTM_zone_32S =32532
PCS_WGS72BE_UTM_zone_33S =32533
PCS_WGS72BE_UTM_zone_34S =32534
PCS_WGS72BE_UTM_zone_35S =32535
PCS_WGS72BE_UTM_zone_36S =32536
PCS_WGS72BE_UTM_zone_37S =32537
```

```
PCS_WGS72BE_UTM_zone_38S =32538
PCS_WGS72BE_UTM_zone_39S =32539
PCS_WGS72BE_UTM_zone_40S =32540
PCS_WGS72BE_UTM_zone_41S =32541
PCS_WGS72BE_UTM_zone_42S =32542
PCS_WGS72BE_UTM_zone_43S =32543
PCS_WGS72BE_UTM_zone_44S =32544
PCS_WGS72BE_UTM_zone_45S =32545
PCS_WGS72BE_UTM_zone_46S =32546
PCS_WGS72BE_UTM_zone_47S =32547
PCS_WGS72BE_UTM_zone_48S =32548
PCS_WGS72BE_UTM_zone_49S =32549
PCS_WGS72BE_UTM_zone_50S =32550
PCS_WGS72BE_UTM_zone_51S =32551
PCS_WGS72BE_UTM_zone_52S =32552
PCS_WGS72BE_UTM_zone_53S =32553
PCS_WGS72BE_UTM_zone_54S =32554
PCS_WGS72BE_UTM_zone_55S =32555
PCS_WGS72BE_UTM_zone_56S =32556
PCS_WGS72BE_UTM_zone_57S =32557
PCS_WGS72BE_UTM_zone_58S =32558
PCS_WGS72BE_UTM_zone_59S =32559
PCS_WGS72BE_UTM_zone_60S =32560
PCS_WGS84_UTM_zone_1N =   32601
PCS_WGS84_UTM_zone_2N =   32602
PCS_WGS84_UTM_zone_3N =   32603
PCS_WGS84_UTM_zone_4N =   32604
PCS_WGS84_UTM_zone_5N =   32605
PCS_WGS84_UTM_zone_6N =   32606
PCS_WGS84_UTM_zone_7N =   32607
PCS_WGS84_UTM_zone_8N =   32608
PCS_WGS84_UTM_zone_9N =   32609
PCS_WGS84_UTM_zone_10N =  32610
PCS_WGS84_UTM_zone_11N =  32611
PCS_WGS84_UTM_zone_12N =  32612
PCS_WGS84_UTM_zone_13N =  32613
PCS_WGS84_UTM_zone_14N =  32614
PCS_WGS84_UTM_zone_15N =  32615
PCS_WGS84_UTM_zone_16N =  32616
PCS_WGS84_UTM_zone_17N =  32617
PCS_WGS84_UTM_zone_18N =  32618
PCS_WGS84_UTM_zone_19N =  32619
PCS_WGS84_UTM_zone_20N =  32620
PCS_WGS84_UTM_zone_21N =  32621
PCS_WGS84_UTM_zone_22N =  32622
PCS_WGS84_UTM_zone_23N =  32623
PCS_WGS84_UTM_zone_24N =  32624
PCS_WGS84_UTM_zone_25N =  32625
PCS_WGS84_UTM_zone_26N =  32626
PCS_WGS84_UTM_zone_27N =  32627
PCS_WGS84_UTM_zone_28N =  32628
PCS_WGS84_UTM_zone_29N =  32629
PCS_WGS84_UTM_zone_30N =  32630
PCS_WGS84_UTM_zone_31N =  32631
PCS_WGS84_UTM_zone_32N =  32632
PCS_WGS84_UTM_zone_33N =  32633
PCS_WGS84_UTM_zone_34N =  32634
PCS_WGS84_UTM_zone_35N =  32635
PCS_WGS84_UTM_zone_36N =  32636
PCS_WGS84_UTM_zone_37N =  32637
PCS_WGS84_UTM_zone_38N =  32638
PCS_WGS84_UTM_zone_39N =  32639
PCS_WGS84_UTM_zone_40N =  32640
PCS_WGS84_UTM_zone_41N =  32641
```

```
PCS_WGS84_UTM_zone_42N =  32642
PCS_WGS84_UTM_zone_43N =  32643
PCS_WGS84_UTM_zone_44N =  32644
PCS_WGS84_UTM_zone_45N =  32645
PCS_WGS84_UTM_zone_46N =  32646
PCS_WGS84_UTM_zone_47N =  32647
PCS_WGS84_UTM_zone_48N =  32648
PCS_WGS84_UTM_zone_49N =  32649
PCS_WGS84_UTM_zone_50N =  32650
PCS_WGS84_UTM_zone_51N =  32651
PCS_WGS84_UTM_zone_52N =  32652
PCS_WGS84_UTM_zone_53N =  32653
PCS_WGS84_UTM_zone_54N =  32654
PCS_WGS84_UTM_zone_55N =  32655
PCS_WGS84_UTM_zone_56N =  32656
PCS_WGS84_UTM_zone_57N =  32657
PCS_WGS84_UTM_zone_58N =  32658
PCS_WGS84_UTM_zone_59N =  32659
PCS_WGS84_UTM_zone_60N =  32660
PCS_WGS84_UTM_zone_1S =   32701
PCS_WGS84_UTM_zone_2S =   32702
PCS_WGS84_UTM_zone_3S =   32703
PCS_WGS84_UTM_zone_4S =   32704
PCS_WGS84_UTM_zone_5S =   32705
PCS_WGS84_UTM_zone_6S =   32706
PCS_WGS84_UTM_zone_7S =   32707
PCS_WGS84_UTM_zone_8S =   32708
PCS_WGS84_UTM_zone_9S =   32709
PCS_WGS84_UTM_zone_10S =  32710
PCS_WGS84_UTM_zone_11S =  32711
PCS_WGS84_UTM_zone_12S =  32712
PCS_WGS84_UTM_zone_13S =  32713
PCS_WGS84_UTM_zone_14S =  32714
PCS_WGS84_UTM_zone_15S =  32715
PCS_WGS84_UTM_zone_16S =  32716
PCS_WGS84_UTM_zone_17S =  32717
PCS_WGS84_UTM_zone_18S =  32718
PCS_WGS84_UTM_zone_19S =  32719
PCS_WGS84_UTM_zone_20S =  32720
PCS_WGS84_UTM_zone_21S =  32721
PCS_WGS84_UTM_zone_22S =  32722
PCS_WGS84_UTM_zone_23S =  32723
PCS_WGS84_UTM_zone_24S =  32724
PCS_WGS84_UTM_zone_25S =  32725
PCS_WGS84_UTM_zone_26S =  32726
PCS_WGS84_UTM_zone_27S =  32727
PCS_WGS84_UTM_zone_28S =  32728
PCS_WGS84_UTM_zone_29S =  32729
PCS_WGS84_UTM_zone_30S =  32730
PCS_WGS84_UTM_zone_31S =  32731
PCS_WGS84_UTM_zone_32S =  32732
PCS_WGS84_UTM_zone_33S =  32733
PCS_WGS84_UTM_zone_34S =  32734
PCS_WGS84_UTM_zone_35S =  32735
PCS_WGS84_UTM_zone_36S =  32736
PCS_WGS84_UTM_zone_37S =  32737
PCS_WGS84_UTM_zone_38S =  32738
PCS_WGS84_UTM_zone_39S =  32739
PCS_WGS84_UTM_zone_40S =  32740
PCS_WGS84_UTM_zone_41S =  32741
PCS_WGS84_UTM_zone_42S =  32742
PCS_WGS84_UTM_zone_43S =  32743
PCS_WGS84_UTM_zone_44S =  32744
PCS_WGS84_UTM_zone_45S =  32745
```

```
   PCS_WGS84_UTM_zone_46S =  32746
   PCS_WGS84_UTM_zone_47S =  32747
   PCS_WGS84_UTM_zone_48S =  32748
   PCS_WGS84_UTM_zone_49S =  32749
   PCS_WGS84_UTM_zone_50S =  32750
   PCS_WGS84_UTM_zone_51S =  32751
   PCS_WGS84_UTM_zone_52S =  32752
   PCS_WGS84_UTM_zone_53S =  32753
   PCS_WGS84_UTM_zone_54S =  32754
   PCS_WGS84_UTM_zone_55S =  32755
   PCS_WGS84_UTM_zone_56S =  32756
   PCS_WGS84_UTM_zone_57S =  32757
   PCS_WGS84_UTM_zone_58S =  32758
   PCS_WGS84_UTM_zone_59S =  32759
   PCS_WGS84_UTM_zone_60S =  32760

   +----------------------------------+
```

### 6.3.3.2 Projection Codes

Note: Projections do not include GCS or PCS definitions. If possible, use the PCS code for standard projected coordinate systems, and use this code only if nonstandard datums are required.

```
Ranges:

   0 = undefined
   [    1,  9999] = Obsolete EPSG/POSC Projection codes
   [10000, 19999] = EPSG/POSC Projection codes
   32767          = user-defined
   [32768, 65535] = Private User Implementations

Special Ranges:

  US State Plane Format:    1sszz
          where ss is USC&GS State code
          zz is USC&GS zone code for NAD27 zones
          zz is (USC&GS zone code + 30) for NAD83 zones

  Larger zoned systems (16000-17999)
   UTM (North) Format:  160zz
   UTM (South) Format:  161zz
   zoned Universal Gauss-Kruger     Format:  162zz
   Universal Gauss-Kruger (unzoned) Format:  163zz
   Australian Map Grid       Format:  174zz
   Southern African STM      Format:  175zz

  Smaller zoned systems:     Format:  18ssz
            where ss is sequential system number
            z is zone code

  Single zone projections    Format:   199ss
            where ss is sequential system number

Values:

   Proj_Alabama_CS27_East =  10101
   Proj_Alabama_CS27_West =  10102
   Proj_Alabama_CS83_East =  10131
   Proj_Alabama_CS83_West =  10132
   Proj_Arizona_Coordinate_System_east =    10201
   Proj_Arizona_Coordinate_System_Central = 10202
   Proj_Arizona_Coordinate_System_west =    10203
```

```
Proj_Arizona_CS83_east =  10231
Proj_Arizona_CS83_Central =      10232
Proj_Arizona_CS83_west =  10233
Proj_Arkansas_CS27_North =10301
Proj_Arkansas_CS27_South =10302
Proj_Arkansas_CS83_North =10331
Proj_Arkansas_CS83_South =10332
Proj_California_CS27_I =  10401
Proj_California_CS27_II = 10402
Proj_California_CS27_III =10403
Proj_California_CS27_IV = 10404
Proj_California_CS27_V =  10405
Proj_California_CS27_VI = 10406
Proj_California_CS27_VII =10407
Proj_California_CS83_1 =  10431
Proj_California_CS83_2 =  10432
Proj_California_CS83_3 =  10433
Proj_California_CS83_4 =  10434
Proj_California_CS83_5 =  10435
Proj_California_CS83_6 =  10436
Proj_Colorado_CS27_North =10501
Proj_Colorado_CS27_Central =     10502
Proj_Colorado_CS27_South =10503
Proj_Colorado_CS83_North =10531
Proj_Colorado_CS83_Central =     10532
Proj_Colorado_CS83_South =10533
Proj_Connecticut_CS27 =   10600
Proj_Connecticut_CS83 =   10630
Proj_Delaware_CS27 =      10700
Proj_Delaware_CS83 =      10730
Proj_Florida_CS27_East =  10901
Proj_Florida_CS27_West =  10902
Proj_Florida_CS27_North = 10903
Proj_Florida_CS83_East =  10931
Proj_Florida_CS83_West =  10932
Proj_Florida_CS83_North = 10933
Proj_Georgia_CS27_East =  11001
Proj_Georgia_CS27_West =  11002
Proj_Georgia_CS83_East =  11031
Proj_Georgia_CS83_West =  11032
Proj_Idaho_CS27_East =    11101
Proj_Idaho_CS27_Central = 11102
Proj_Idaho_CS27_West =    11103
Proj_Idaho_CS83_East =    11131
Proj_Idaho_CS83_Central = 11132
Proj_Idaho_CS83_West =    11133
Proj_Illinois_CS27_East = 11201
Proj_Illinois_CS27_West = 11202
Proj_Illinois_CS83_East = 11231
Proj_Illinois_CS83_West = 11232
Proj_Indiana_CS27_East =  11301
Proj_Indiana_CS27_West =  11302
Proj_Indiana_CS83_East =  11331
Proj_Indiana_CS83_West =  11332
Proj_Iowa_CS27_North =    11401
Proj_Iowa_CS27_South =    11402
Proj_Iowa_CS83_North =    11431
Proj_Iowa_CS83_South =    11432
Proj_Kansas_CS27_North =  11501
Proj_Kansas_CS27_South =  11502
Proj_Kansas_CS83_North =  11531
Proj_Kansas_CS83_South =  11532
Proj_Kentucky_CS27_North =11601
Proj_Kentucky_CS27_South =11602
```

```
Proj_Kentucky_CS83_North =11631
Proj_Kentucky_CS83_South =11632
Proj_Louisiana_CS27_North =      11701
Proj_Louisiana_CS27_South =      11702
Proj_Louisiana_CS83_North =      11731
Proj_Louisiana_CS83_South =      11732
Proj_Maine_CS27_East =    11801
Proj_Maine_CS27_West =    11802
Proj_Maine_CS83_East =    11831
Proj_Maine_CS83_West =    11832
Proj_Maryland_CS27 =      11900
Proj_Maryland_CS83 =      11930
Proj_Massachusetts_CS27_Mainland =       12001
Proj_Massachusetts_CS27_Island = 12002
Proj_Massachusetts_CS83_Mainland =       12031
Proj_Massachusetts_CS83_Island = 12032
Proj_Michigan_State_Plane_East = 12101
Proj_Michigan_State_Plane_Old_Central =  12102
Proj_Michigan_State_Plane_West = 12103
Proj_Michigan_CS27_North =12111
Proj_Michigan_CS27_Central =     12112
Proj_Michigan_CS27_South =12113
Proj_Michigan_CS83_North =12141
Proj_Michigan_CS83_Central =     12142
Proj_Michigan_CS83_South =12143
Proj_Minnesota_CS27_North =      12201
Proj_Minnesota_CS27_Central =    12202
Proj_Minnesota_CS27_South =      12203
Proj_Minnesota_CS83_North =      12231
Proj_Minnesota_CS83_Central =    12232
Proj_Minnesota_CS83_South =      12233
Proj_Mississippi_CS27_East =     12301
Proj_Mississippi_CS27_West =     12302
Proj_Mississippi_CS83_East =     12331
Proj_Mississippi_CS83_West =     12332
Proj_Missouri_CS27_East = 12401
Proj_Missouri_CS27_Central =     12402
Proj_Missouri_CS27_West = 12403
Proj_Missouri_CS83_East = 12431
Proj_Missouri_CS83_Central =     12432
Proj_Missouri_CS83_West = 12433
Proj_Montana_CS27_North = 12501
Proj_Montana_CS27_Central =      12502
Proj_Montana_CS27_South = 12503
Proj_Montana_CS83 =       12530
Proj_Nebraska_CS27_North =12601
Proj_Nebraska_CS27_South =12602
Proj_Nebraska_CS83 =      12630
Proj_Nevada_CS27_East =   12701
Proj_Nevada_CS27_Central =12702
Proj_Nevada_CS27_West =   12703
Proj_Nevada_CS83_East =   12731
Proj_Nevada_CS83_Central =12732
Proj_Nevada_CS83_West =   12733
Proj_New_Hampshire_CS27 = 12800
Proj_New_Hampshire_CS83 = 12830
Proj_New_Jersey_CS27 =    12900
Proj_New_Jersey_CS83 =    12930
Proj_New_Mexico_CS27_East =      13001
Proj_New_Mexico_CS27_Central =   13002
Proj_New_Mexico_CS27_West =      13003
Proj_New_Mexico_CS83_East =      13031
Proj_New_Mexico_CS83_Central =   13032
Proj_New_Mexico_CS83_West =      13033
```

```
Proj_New_York_CS27_East = 13101
Proj_New_York_CS27_Central =     13102
Proj_New_York_CS27_West = 13103
Proj_New_York_CS27_Long_Island = 13104
Proj_New_York_CS83_East = 13131
Proj_New_York_CS83_Central =     13132
Proj_New_York_CS83_West = 13133
Proj_New_York_CS83_Long_Island = 13134
Proj_North_Carolina_CS27 =13200
Proj_North_Carolina_CS83 =13230
Proj_North_Dakota_CS27_North =   13301
Proj_North_Dakota_CS27_South =   13302
Proj_North_Dakota_CS83_North =   13331
Proj_North_Dakota_CS83_South =   13332
Proj_Ohio_CS27_North =    13401
Proj_Ohio_CS27_South =    13402
Proj_Ohio_CS83_North =    13431
Proj_Ohio_CS83_South =    13432
Proj_Oklahoma_CS27_North =13501
Proj_Oklahoma_CS27_South =13502
Proj_Oklahoma_CS83_North =13531
Proj_Oklahoma_CS83_South =13532
Proj_Oregon_CS27_North =  13601
Proj_Oregon_CS27_South =  13602
Proj_Oregon_CS83_North =  13631
Proj_Oregon_CS83_South =  13632
Proj_Pennsylvania_CS27_North =   13701
Proj_Pennsylvania_CS27_South =   13702
Proj_Pennsylvania_CS83_North =   13731
Proj_Pennsylvania_CS83_South =   13732
Proj_Rhode_Island_CS27 =  13800
Proj_Rhode_Island_CS83 =  13830
Proj_South_Carolina_CS27_North = 13901
Proj_South_Carolina_CS27_South = 13902
Proj_South_Carolina_CS83 =13930
Proj_South_Dakota_CS27_North =   14001
Proj_South_Dakota_CS27_South =   14002
Proj_South_Dakota_CS83_North =   14031
Proj_South_Dakota_CS83_South =   14032
Proj_Tennessee_CS27 =     14100
Proj_Tennessee_CS83 =     14130
Proj_Texas_CS27_North =   14201
Proj_Texas_CS27_North_Central =  14202
Proj_Texas_CS27_Central = 14203
Proj_Texas_CS27_South_Central =  14204
Proj_Texas_CS27_South =   14205
Proj_Texas_CS83_North =   14231
Proj_Texas_CS83_North_Central =  14232
Proj_Texas_CS83_Central = 14233
Proj_Texas_CS83_South_Central =  14234
Proj_Texas_CS83_South =   14235
Proj_Utah_CS27_North =    14301
Proj_Utah_CS27_Central =  14302
Proj_Utah_CS27_South =    14303
Proj_Utah_CS83_North =    14331
Proj_Utah_CS83_Central =  14332
Proj_Utah_CS83_South =    14333
Proj_Vermont_CS27 =       14400
Proj_Vermont_CS83 =       14430
Proj_Virginia_CS27_North =14501
Proj_Virginia_CS27_South =14502
Proj_Virginia_CS83_North =14531
Proj_Virginia_CS83_South =14532
Proj_Washington_CS27_North =     14601
```

```
Proj_Washington_CS27_South =     14602
Proj_Washington_CS83_North =     14631
Proj_Washington_CS83_South =     14632
Proj_West_Virginia_CS27_North =  14701
Proj_West_Virginia_CS27_South =  14702
Proj_West_Virginia_CS83_North =  14731
Proj_West_Virginia_CS83_South =  14732
Proj_Wisconsin_CS27_North =      14801
Proj_Wisconsin_CS27_Central =    14802
Proj_Wisconsin_CS27_South =      14803
Proj_Wisconsin_CS83_North =      14831
Proj_Wisconsin_CS83_Central =    14832
Proj_Wisconsin_CS83_South =      14833
Proj_Wyoming_CS27_East =  14901
Proj_Wyoming_CS27_East_Central = 14902
Proj_Wyoming_CS27_West_Central = 14903
Proj_Wyoming_CS27_West =  14904
Proj_Wyoming_CS83_East =  14931
Proj_Wyoming_CS83_East_Central = 14932
Proj_Wyoming_CS83_West_Central = 14933
Proj_Wyoming_CS83_West =  14934
Proj_Alaska_CS27_1 =      15001
Proj_Alaska_CS27_2 =      15002
Proj_Alaska_CS27_3 =      15003
Proj_Alaska_CS27_4 =      15004
Proj_Alaska_CS27_5 =      15005
Proj_Alaska_CS27_6 =      15006
Proj_Alaska_CS27_7 =      15007
Proj_Alaska_CS27_8 =      15008
Proj_Alaska_CS27_9 =      15009
Proj_Alaska_CS27_10 =     15010
Proj_Alaska_CS83_1 =      15031
Proj_Alaska_CS83_2 =      15032
Proj_Alaska_CS83_3 =      15033
Proj_Alaska_CS83_4 =      15034
Proj_Alaska_CS83_5 =      15035
Proj_Alaska_CS83_6 =      15036
Proj_Alaska_CS83_7 =      15037
Proj_Alaska_CS83_8 =      15038
Proj_Alaska_CS83_9 =      15039
Proj_Alaska_CS83_10 =     15040
Proj_Hawaii_CS27_1 =      15101
Proj_Hawaii_CS27_2 =      15102
Proj_Hawaii_CS27_3 =      15103
Proj_Hawaii_CS27_4 =      15104
Proj_Hawaii_CS27_5 =      15105
Proj_Hawaii_CS83_1 =      15131
Proj_Hawaii_CS83_2 =      15132
Proj_Hawaii_CS83_3 =      15133
Proj_Hawaii_CS83_4 =      15134
Proj_Hawaii_CS83_5 =      15135
Proj_Puerto_Rico_CS27 =   15201
Proj_St_Croix =    15202
Proj_Puerto_Rico_Virgin_Is =     15230
Proj_BLM_14N_feet =       15914
Proj_BLM_15N_feet =       15915
Proj_BLM_16N_feet =       15916
Proj_BLM_17N_feet =       15917
Proj_Map_Grid_of_Australia_48 =  17348
Proj_Map_Grid_of_Australia_49 =  17349
Proj_Map_Grid_of_Australia_50 =  17350
Proj_Map_Grid_of_Australia_51 =  17351
Proj_Map_Grid_of_Australia_52 =  17352
Proj_Map_Grid_of_Australia_53 =  17353
```

```
   Proj_Map_Grid_of_Australia_54 =  17354
   Proj_Map_Grid_of_Australia_55 =  17355
   Proj_Map_Grid_of_Australia_56 =  17356
   Proj_Map_Grid_of_Australia_57 =  17357
   Proj_Map_Grid_of_Australia_58 =  17358
   Proj_Australian_Map_Grid_48 =    17448
   Proj_Australian_Map_Grid_49 =    17449
   Proj_Australian_Map_Grid_50 =    17450
   Proj_Australian_Map_Grid_51 =    17451
   Proj_Australian_Map_Grid_52 =    17452
   Proj_Australian_Map_Grid_53 =    17453
   Proj_Australian_Map_Grid_54 =    17454
   Proj_Australian_Map_Grid_55 =    17455
   Proj_Australian_Map_Grid_56 =    17456
   Proj_Australian_Map_Grid_57 =    17457
   Proj_Australian_Map_Grid_58 =    17458
   Proj_Argentina_1 = 18031
   Proj_Argentina_2 = 18032
   Proj_Argentina_3 = 18033
   Proj_Argentina_4 = 18034
   Proj_Argentina_5 = 18035
   Proj_Argentina_6 = 18036
   Proj_Argentina_7 = 18037
   Proj_Colombia_3W = 18051
   Proj_Colombia_Bogota =    18052
   Proj_Colombia_3E = 18053
   Proj_Colombia_6E = 18054
   Proj_Egypt_Red_Belt =     18072
   Proj_Egypt_Purple_Belt =  18073
   Proj_Extended_Purple_Belt =      18074
   Proj_New_Zealand_North_Island_Nat_Grid = 18141
   Proj_New_Zealand_South_Island_Nat_Grid = 18142
   Proj_Bahrain_Grid =       19900
   Proj_Netherlands_E_Indies_Equatorial =   19905
   Proj_RSO_Borneo =  19912


   +-----------------------------------+
```

### 6.3.3.3 Coordinate Transformation Codes

```
Ranges:

   0 = undefined
   [    1, 16383] = GeoTIFF Coordinate Transformation codes
   [16384, 32766] = Reserved by GeoTIFF
   32767          = user-defined
   [32768, 65535] = Private User Implementations

Values:

   CT_TransverseMercator =   1
   CT_TransvMercator_Modified_Alaska = 2
   CT_ObliqueMercator =      3
   CT_ObliqueMercator_Laborde =     4
   CT_ObliqueMercator_Rosenmund =   5
   CT_ObliqueMercator_Spherical =   6
   CT_Mercator =      7
   CT_LambertConfConic_2SP = 8
   CT_LambertConfConic_Helmert =    9
   CT_LambertAzimEqualArea = 10
   CT_AlbersEqualArea =      11
   CT_AzimuthalEquidistant = 12
   CT_EquidistantConic =     13
   CT_Stereographic = 14
   CT_PolarStereographic =   15
   CT_ObliqueStereographic = 16
   CT_Equirectangular =      17
   CT_CassiniSoldner =       18
   CT_Gnomonic =      19
   CT_MillerCylindrical =    20
   CT_Orthographic =  21
   CT_Polyconic =     22
   CT_Robinson =      23
   CT_Sinusoidal =    24
   CT_VanDerGrinten = 25
   CT_NewZealandMapGrid =    26
   CT_TransvMercator_SouthOriented= 27

Aliases:

   CT_AlaskaConformal =             CT_TransvMercator_Modified_Alaska
   CT_TransvEquidistCylindrical =   CT_CassiniSoldner
   CT_ObliqueMercator_Hotine =      CT_ObliqueMercator
   CT_SwissObliqueCylindrical =     CT_ObliqueMercator_Rosenmund
   CT_GaussBoaga =                  CT_TransverseMercator
   CT_GaussKruger =                 CT_TransverseMercator
   CT_LambertConfConic =            CT_LambertConfConic_2SP
   CT_LambertConfConic_Helmert =    CT_LambertConfConic_1SP
   CT_SouthOrientedGaussConformal = CT_TransvMercator_SouthOriented

   +----------------------------------+
```

## 6.3.4 Vertical CS Codes

```
   +----------------------------------+
```

### 6.3.4.1 Vertical CS Type Codes

```
Ranges:

   0               = undefined
   [    1,   4999] = Reserved
   [ 5000,   5099] = EPSG Ellipsoid Vertical CS Codes
   [ 5100,   5199] = EPSG Orthometric Vertical CS Codes
   [ 5200,   5999] = Reserved EPSG
   [ 6000,  32766] = Reserved
   32767           = user-defined
   [32768, 65535]  = Private User Implementations

Values:

   VertCS_Airy_1830_ellipsoid =     5001
   VertCS_Airy_Modified_1849_ellipsoid =    5002
   VertCS_ANS_ellipsoid =    5003
   VertCS_Bessel_1841_ellipsoid =   5004
   VertCS_Bessel_Modified_ellipsoid =       5005
   VertCS_Bessel_Namibia_ellipsoid =5006
   VertCS_Clarke_1858_ellipsoid =   5007
   VertCS_Clarke_1866_ellipsoid =   5008
   VertCS_Clarke_1880_Benoit_ellipsoid =    5010
   VertCS_Clarke_1880_IGN_ellipsoid =       5011
   VertCS_Clarke_1880_RGS_ellipsoid =       5012
   VertCS_Clarke_1880_Arc_ellipsoid =       5013
   VertCS_Clarke_1880_SGA_1922_ellipsoid =  5014
   VertCS_Everest_1830_1937_Adjustment_ellipsoid = 5015
   VertCS_Everest_1830_1967_Definition_ellipsoid = 5016
   VertCS_Everest_1830_1975_Definition_ellipsoid = 5017
   VertCS_Everest_1830_Modified_ellipsoid = 5018
   VertCS_GRS_1980_ellipsoid =      5019
   VertCS_Helmert_1906_ellipsoid =  5020
   VertCS_INS_ellipsoid =    5021
   VertCS_International_1924_ellipsoid =    5022
   VertCS_International_1967_ellipsoid =    5023
   VertCS_Krassowsky_1940_ellipsoid =       5024
   VertCS_NWL_9D_ellipsoid = 5025
   VertCS_NWL_10D_ellipsoid =5026
   VertCS_Plessis_1817_ellipsoid =  5027
   VertCS_Struve_1860_ellipsoid =   5028
   VertCS_War_Office_ellipsoid =    5029
   VertCS_WGS_84_ellipsoid = 5030
   VertCS_GEM_10C_ellipsoid =5031
   VertCS_OSU86F_ellipsoid = 5032
   VertCS_OSU91A_ellipsoid = 5033

  Orthometric Vertical CS;

   VertCS_Newlyn =    5101
   VertCS_North_American_Vertical_Datum_1929 =    5102
   VertCS_North_American_Vertical_Datum_1988 =    5103
   VertCS_Yellow_Sea_1956 =  5104
   VertCS_Baltic_Sea =       5105
   VertCS_Caspian_Sea =      5106


   +----------------------------------+
```

### 6.3.4.2 Vertical CS Datum Codes

```
Ranges:

   0                = undefined
   [    1,  16383] = Vertical Datum Codes
   [16384,  32766] = Reserved
   32767            = user-defined
   [32768, 65535]  = Private User Implementations
```

No vertical datum codes are currently defined, other than those implied by the corrsponding Vertical CS code.

```
   +--------------------------------------------------------------------+

   +----------------------------------+
```

### 6.4 EPSG Geodesy Parameter Index

```
   +----------------------------------+
```

Here is a summary of the index ranges for the various coding systems used by EPSG in their tables. A copy of this index may be acquired at the FTP sites mentioned in the references in section 5. The "value" table entries below describe how values from one table are related to codes from another table.

```
Summary
--------
```

| Entity | digit | Range |
|---|---|---|
| Prime Meridian | 8 | 8000 thru 8999 |
| Ellipsoid | 7 | 7000 thru 7999 |
| Geodetic Datum | 6 | 6000 thru 6999 |
| Vertical datum | 5 | 5000 thru 5999 |
| Geographic Coordinate System | 4 | 4000 thru 4999 |
| Projected Coordinate Systems | 2 or 3 | 20000 thru 32760 |
| Map Projection | 1 | 10000 - 19999 |

Geodetic Datum Codes

| Datum Type | Value | Range | Currently Defined |
|---|---|---|---|
| Unspecified Geodetic Datum | [EC-1000] | 6000 thru 6099 | 6001 thru 6035 |
| Geodetic Datum | | 6100 thru 6321 | 6200 thru 6315 |
| WGS 72; WGS 72BE and WGS84 | | 6322 thru 6327 | 6322 thru 6327 |
| Geodetic Datum (ancient) | | 6900 thru 6999 | 6901 thru 6902 |

Note for Values: EC = corresponding Ellipsoid Code.

Vertical Datum Codes

| Datum Type | Value | Range | Currently Defined |
|---|---|---|---|
| Ellipsoidal | [EC-1000] | 5000 thru 5099 | 5001 thru 5035 |
| Orthometric | | 5100 thru 5899 | 5101 thru 5106 |

Note for Values: EC = corresponding Ellipsoid Code.

Geographic Coordinate System Codes

| GCS Type | Value | Range | Currently Defined |
|---|---|---|---|
| Unknown geodetic datum | [GDC-2000] | 4000 thru 4099 | 4001 thru 4045 |
| Known datum (Greenwich) | [GDC-2000] | 4100 thru 4321 | 4200 thru 4315 |
| WGS 72; WGS 72BE and WGS84 | | 4322 thru 4327 | 4322 thru 4327 |
| Known datum (not Greenwich) | | 4800 thru 4899 | 4801 thru 4812 |
| Known datum (ancient) | [GDC-2000] | 4900 thru 4999 | 4901 thru 4902 |

Note for Values: GDC = corresponding Geodetic Datum Code

Map Projection System Codes

```
   US State Plane  ( 10000-15999 )
      Format:     1sszz
         where ss is USC&GS State code  01 thru 59
         zz is (USC&GS zone code)      for NAD27 zones
         zz is (USC&GS zone code + 30) for NAD83 zones
```

Larger zoned systems ( 16000-17999 )

| System | Format | zz Range | |
|---|---|---|---|
| UTM (North) | 160zz | 01 | 60 |
| UTM (South) | 161zz | 01 | 60 |
| zoned Universal Gauss-Kruger | 162zz | 04 | 32 |

```
      Universal Gauss-Kruger (unzoned)  163zz   04   3
      Australian Map Grid               174zz   48   58
      Southern African STM              175zz   13   35


   Smaller zoned systems  ( 18000-18999 )
       Format:  18ssz
           where ss is sequential system number  01   18
           z is zone code

   Single zone projections ( 19900-19999 )
        Format:   199ss
           where ss is sequential system number  00   25

Projected Coordinate Systems
----------------------------

For PCS utilising GeogCS with code in range 4201 through 4321
(i.e. geodetic datum code 6201 through 6319):

   As far as is possible the PCS code will be of the format
   gggzz where ggg is (geodetic datum code -6000) and zz is zone.

For PCS utilising GeogCS with code out of range 4201 through 4321
(i.e.geodetic datum code 6201 through 6319):
    PCS code 20xxx where xxx is a sequential number

WGS72 / UTM North     322zz where zz is UTM zone number   32201   32260
WGS72 / UTM South     323zz where zz is UTM zone number   32301   32360
WGS72BE / UTM North   324zz where zz is UTM zone number   32401   32460
WGS72BE / UTM South   325zz where zz is UTM zone number   32501   32560
WGS84 / UTM North     326zz where zz is UTM zone number   32601   32660
WGS84 / UTM South     327zz where zz is UTM zone number   32701   32760
US State Plane (NAD27)   267xx or 320xx where xx is a sequential number
US State Plane (NAD83)   269xx or 321xx where xx is a sequential number



   +-----------------------------------------------------------------------+
```

# 7 Glossary

```
      +-----------------------------------------------------------------------+
```

ASCII: [American Standard Code for Information Interchange] The predominant character set encoding of present-day computers.

Cell: A rectangular area in Raster space, in which a single pixel value is filled.

Code: In GeoTIFF, a code is a value assigned to a GeoKey, and has one of 65536 possible values.

Coordinate System: A systematic way of assigning real (x,y,z..) coordinates to a surface or volume. In Geodetics the surface is an ellipsoid used to model the earth.

Datum: a mathematical approximation to all or part of the earth's surface. Defining a datum requires the definition of an ellipsoid, its location and orientation, as well as the area for which the datum is valid.

Device Space A coordinate space referencing scanner, printers and display devices.

DOUBLE: 8-byte IEEE double precision floating point.

Ellipsoid: A mathematically defined quadratic surface used to model the earth.

EPSG: European Petroleum Survey Group.

Flattening: For an ellipsoid with major and minor axis lengths (a,b), the flattening is defined by:,

$$f = (a - b)/a$$

For the earth, the value of f is approximately 1/298.3

Geocoding: An image is geocoded if a precise algorithm for determining the earth-location of each point in the image is defined.

Geographic Coordinate System: A Geographic CS consists of a well-defined ellipsoidal datum, a Prime Meridian, and an angular unit, allowing the assignment of a Latitude-Longitude (and optionally, geodetic height) vector to a location on earth.

GeoKey In GeoTIFF, a GeoKey is equivalent in function to a TIFF tag, but uses a different storage mechanism.

APPENDIX GG

Georeferencing:
An image is georeferenced if the location of its pixels in some model space is defined, but the transformation tying model space to the earth is not known.

GeoTIFF:
A standard for storing georeference and geocoding information in a TIFF 6.0 compliant raster file.

Grid
A coordinate mesh upon which pixels are placed

IEEE
Institute of Electrical and Electronics Engineers, Inc.

IFD:
In TIFF format, an Image File Directory, containing all the TIFF tags for one image in the file (there may be more than one).

Meridian:
Arc of constant longitude, passing through the poles.

Model Space
A flat geometrical space used to model a portion of the earth.

Parallel:
Lines of constant latitude, parallel to the equator.

Pixel:
A dimensionless point-measurement, stored in a raster file.

POSC:
Petrotechnical Open Software Corporation.

Prime Meridian:
An arbitrarily chosen meridian, used as reference for all others, and defined as 0 degrees longitude.

Projection
A projection in GeoTIFF consists of a linear (X,Y) coordinate system, and a coordinate transformation method (such as Transverse Mercator) to tie this system to an unspecified Geographic CS..

| | |
|---|---|
| Projected Coordinate System | The result of the application of a projection transformation of a Geographic coordinate system |
| Raster Space: | A continuous planar space in which pixel values are visually realized. |
| RATIONAL: | In TIFF format, a RATIONAL value is a fractional value represented by the ratio of two unsigned 4-byte integers. |
| SDTS | The USGS Spatial Data Transmission Standard. |
| Tag: | In TIFF format, a tag is packet of numerical or ASCII values, which have a numerical "Tag" ID indicating their information content. |
| TIFF: | Acronym for Tagged Image File Format; a platform-independent, extensive specification for storing raster data and ancillary information in a single file. |
| USGS | US Geological Survey |

+-----------------------------------------------------------------+,

# END OF SPECIFICATION

+-----------------------------------------------------------------+

APPENDIX HH

# TIFF™

## Revision 6.0

Final — June 3, 1992

**Adobe Developers Association**

Adobe Systems Incorporated
1585 Charleston Road
P.O. Box 7900
Mountain View, CA 94039-7900

E-Mail: devsup-person@adobe.com

**A copy of this specification can be found in**

http://www.adobe.com/Support/TechNotes.html

and

ftp://ftp.adobe.com/pub/adobe/DeveloperSupport/TechNotes/PDFfiles

APPENDIX HH

## Copyright

## Licenses and Trademarks

PostScript is a trademark of Adobe Systems Incorporated. All instances of the name PostScript in the text are references to the PostScript language as defined by Adobe Systems Incorporated unless otherwise stated. The name PostScript also is used as a product trademark for Adobe Systems' implementation of the PostScript language interpreter.

Any references to a "PostScript printer," a "PostScript file," or a "PostScript driver" refer to printers, files, and driver programs (respectively) which are written in or support the PostScript language. The sentences in this specification that use "PostScript language" as an adjective phrase are so constructed to reinforce that the name refers to the standard language definition as set forth by Adobe Systems Incorporated.

PostScript, the PostScript logo, Display PostScript, Adobe, the Adobe logo, Adobe Illustrator, Aldus, PageMaker, TIFF, OPI, TrapWise, Tran-Script, Carta, and Sonata are trademarks of Adobe Systems Incorporated or its subsidiaries, and may be registered in some jurisdictions.

Apple, LaserWriter, and Macintosh are registered trademarks and Finder and System 7 are trademarks of Apple, Computer, Inc. Microsoft and MS-DOS are registered trademarks and Windows is a trademark of Microsoft Corporation. UNIX is a registered trademark of UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc. All other trademarks are the property of their respective owners.

## Production Notes

This document was created electronically using Adobe PageMaker® 6.0.

APPENDIX HH

# Contents

APPENDIX HH

# Introduction

## About this Specification

This document describes TIFF, a tag-based file format for storing and interchanging raster images.

### History

The first version of the TIFF specification was published by Aldus Corporation in the fall of 1986, after a series of meetings with various scanner manufacturers and software developers. It did not have a revision number but should have been labeled Revision 3.0 since there were two major earlier draft releases.

Revision 4.0 contained mostly minor enhancements and was released in April 1987. Revision 5.0, released in October 1988, added support for palette color images and LZW compression.

### Scope

TIFF describes image data that typically comes from scanners, frame grabbers, and paint- and photo-retouching programs.

TIFF is not a printer language or page description language. The purpose of TIFF is to describe and store raster image data.

A primary goal of TIFF is to provide a rich environment within which applications can exchange image data. This richness is required to take advantage of the varying capabilities of scanners and other imaging devices.

Though TIFF is a rich format, it can easily be used for simple scanners and applications as well because the number of required fields is small.

TIFF will be enhanced on a continuing basis as new imaging needs arise. A high priority has been given to structuring TIFF so that future enhancements can be added without causing unnecessary hardship to developers.

APPENDIX HH

## Features

- TIFF is capable of describing bilevel, grayscale, palette-color, and full-color image data in several color spaces.
- TIFF includes a number of compression schemes that allow developers to choose the best space or time tradeoff for their applications.
- TIFF is not tied to specific scanners, printers, or computer display hardware.
- TIFF is portable. It does not favor particular operating systems, file systems, compilers, or processors.
- TIFF is designed to be extensible—to evolve gracefully as new needs arise.
- TIFF allows the inclusion of an unlimited amount of private or special-purpose information.

APPENDIX HH

# Revision Notes

## Minor changes to TIFF 6.0, March 1995

*Updated contact information and TIFF administration policies, since Aldus Corporation merged with Adobe Systems Incorporated on September 1, 1994.*

*The technical content and pagination are unchanged from the original June 3, 1992 release.*

## TIFF 5.0 to TIFF 6.0

This revision replaces TIFF Revision 5.0.

In the main body of the document, paragraphs that contain new or substantially-changed information are shown in italics.

## New Features in Revision 6.0

Major enhancements to TIFF 6.0 are described in Part 2. They include:

- CMYK image definition
- A revised RGB Colorimetry section.
- YCbCr image definition
- CIE L*a*b* image definition
- Tiled image definition
- JPEG compression

## Clarifications

- The LZW compression section more clearly explains when to switch the coding bit length.
- The interaction between Compression=2 (CCITT Huffman) and PhotometricInterpretation was clarified.
- The data organization of uncompressed data (Compression=1) when BitsPerSample is greater than 8 was clarified. See the Compression field description.
- The discussion of CCITT Group 3 and Group 4 bilevel image encodings was clarified and expanded, and Group3Options and Group4Options fields were renamed T4Options and T6Options. See Section 11.

APPENDIX HH

## Organizational Changes

- To make the organization more consistent and expandable, appendices were transformed into numbered sections.
- The document was divided into two parts—Baseline and Extensions—to help developers make better and more consistent implementation choices. Part 1, the Baseline section, describes those features that all general-purpose TIFF readers should support. Part 2, the Extensions section, describes a number of features that can be used by special or advanced applications.
- An index and table of contents were added.

## Changes in Requirements

- To illustrate a Baseline TIFF file earlier in the document, the material from Appendix G ("TIFF Classes") in Revision 5 was integrated into the main body of the specification . As part of this integration, the TIFF Classes terminology was replaced by the more monolithic Baseline TIFF terminology. The intent was to further encourage all mainstream TIFF readers to support the Baseline TIFF requirements for bilevel, grayscale, RGB, and palette-color images.
- Due to licensing issues, LZW compression support was moved out of the "Part 1: Baseline TIFF" and into "Part 2: Extensions."
- Baseline TIFF requirements for bit depths in palette-color images were weakened a bit.

## Changes in Terminology

In previous versions of the specification, the term "tag" reffered both to the identifying number of a TIFF field and to the entire field. In this version, the term "tag" refers only to the identifying number. The term "field" refers to the entire field, including the value.

## Compatibility

Every attempt has been made to add functionality in such a way as to minimize compatibility problems with files and software that were based on earlier versions of the TIFF specification. The goal is that TIFF files should never become obsolete and that TIFF software should not have to be revised more frequently than absolutely necessary. In particular, Baseline TIFF 6.0 files will generally be readable even by older applications that assume TIFF 5.0 or an earlier version of the specification.

However, TIFF 6.0 files that use one of the major new extensions, such as a new compression scheme or color space, will not be successfully read by older software. In such cases, the older applications must gracefully give up and refuse to import the image, providing the user with a reasonably informative message.

**APPENDIX HH**

# TIFF Administration

## *Information and Support*

The most recent version of the TIFF specification is available in PDF format on the Adobe WWW and ftp servers See the cover page of the specification for the required addresses.

Because of the widespread use of TIFF for in many environments, Adobe is unable to provide a general consulting service for TIFF implementors. TIFF developers are encouraged to study sample TIFF files, read TIFF documentation thoroughly, and work with developers of other products that are important to you.

If your TIFF question specifically concerns compatibility with an Adobe Systems product, please contact Adobe Developer Support at devsup-person@adobe.com.

Most companies that use TIFF can answer questions about support for TIFF in their products. Contact the appropriate product manager or developer support service group.

## *Private Fields and Values*

An organization might wish to store information meaningful to only that organization in a TIFF file. Tags numbered 32768 or higher, sometimes called private tags, are reserved for that purpose.

Upon request, the TIFF administrator (send email to devsup-person@adobe.com) will allocate and register one or more private tags for an organization, to avoid possible conflicts with other organizations. You do not need to tell the TIFF administrator what you plan to use them for, but giving us this information may help other developers to avoid some duplication of effort. We will likely make the tag database public at some point.

Private enumerated values can be accommodated in a similar fashion. For example, you may wish to experiment with a new compression scheme within TIFF. Enumeration constants numbered 32768 or higher are reserved for private usage. Upon request, the administrator will allocate and register one or more enumerated values for a particular field (Compression, in our example), to avoid possible conflicts.

Tags and values allocated in the private number range are not prohibited from being included in a future revision of this specification. Several such instances exist in the current TIFF specification.

Do not choose your own tag numbers. Doing so could cause serious compatibility problems in the future. However, if there is little or no chance that your TIFF files will escape your private environment, please consider using TIFF tags in the “reusable” 65000-65535 range. You do not need to contact Adobe when using numbers in this range.

If you need more than 10 tags, we suggest that you reserve a single private tag, define it as a LONG TIFF data type, and use its value as a pointer (offset) to a private IFD or other data structure of your choosing. Within that IFD, you can use whatever tags you want, since no one else will know that it is an IFD unless you tell them.

## Submitting a Proposal

Any person or group that wants to propose a change or addition to the TIFF specification should prepare a proposal that includes the following information:

- Name of the person or group making the request, and your affiliation.
- The reason for the request.
- A list of changes exactly as you propose that they appear in the specification. Use inserts, callouts, or other obvious editorial techniques to indicate areas of change, and number each change.
- Discussion of the potential impact on the installed base.
- A list of contacts outside your company that support your position. Include their affiliation.

Please send your proposal to devsup-person@adobe.com.

## The TIFF Advisory Committee

The TIFF Advisory Committee is a working group of TIFF experts from a number of hardware and software manufacturers. It was formed in the spring of 1991 to provide a forum for debating and refining proposals for the 6.0 release of the TIFF specification.

If you are a TIFF expert and think you have the time and interest to work on this committee, contact devsup-person@adobe.com for further information. For the TIFF 6.0 release, the group met every two or three months, usually on the west coast of the U.S. Accessibility via Internet email is a requirement for membership, since that has proven to be an invaluable means for getting work done between meetings.

## Other TIFF Extensions

~~The Aldus TIFF sections on CompuServe and AppleLink~~ (new location is under construction; check the Adobe WWW home page (http://www.adobe.com) for future developements) will contain proposed TIFF extensions from other companies that are not approved by Adobe as part of Baseline TIFF.

These proposals typically represent specialized uses of TIFF that do not fall within the domain of publishing or general graphics or picture interchange. Generally, these features will not be widely supported. If you do write files that incorporate these extensions, be sure to either not call them TIFF files or mark them in some way so that they will not be confused with mainstream TIFF files.

If you have such a document, send it to devsup-person@adobe.com. All submissions must be PDF documents or simple text. Be sure to include contact information—at least an email address.

# Part 1: Baseline TIFF

The TIFF specification is divided into two parts. Part 1 describes *Baseline TIFF*. Baseline TIFF is the core of TIFF, the essentials that all mainstream TIFF developers should support in their products.

APPENDIX HH

# Section 1: Notation

## Decimal and Hexadecimal

Unless otherwise noted, all numeric values in this document are expressed in decimal. (".H" is appended to hexidecimal values.)

## Compliance

*Is* and *shall* indicate mandatory requirements. All compliant writers and readers must meet the specification.

*Should* indicates a recommendation.

*May* indicates an option.

*Features designated 'not recommended for general data interchange' are considered extensions to Baseline TIFF. Files that use such features shall be designated "Extended TIFF 6.0" files, and the particular extensions used should be documented. A Baseline TIFF 6.0 reader is not required to support any extensions.*

APPENDIX HH

# Section 2: TIFF Structure

TIFF is an image file format. In this document, a *file* is defined to be a sequence of 8-bit bytes, where the bytes are numbered from 0 to N. The largest possible TIFF file is 2**32 bytes in length.

A TIFF file begins with an 8-byte *image file header* that points to an *image file directory* (*IFD*). An image file directory contains information about the image, as well as pointers to the actual image data.

The following paragraphs describe the image file header and IFD in more detail.

See Figure 1.

## *Image File Header*

A TIFF file begins with an 8-byte image file header, containing the following information:

Bytes 0-1: The byte order used within the file. Legal values are:

"II" (4949.H)

"MM" (4D4D.H)

In the "II" format, byte order is always from the least significant byte to the most significant byte, for both 16-bit and 32-bit integers This is called *little-endian* byte order. In the "MM" format, byte order is always from most significant to least significant, for both 16-bit and 32-bit integers. This is called *big-endian* byte order.

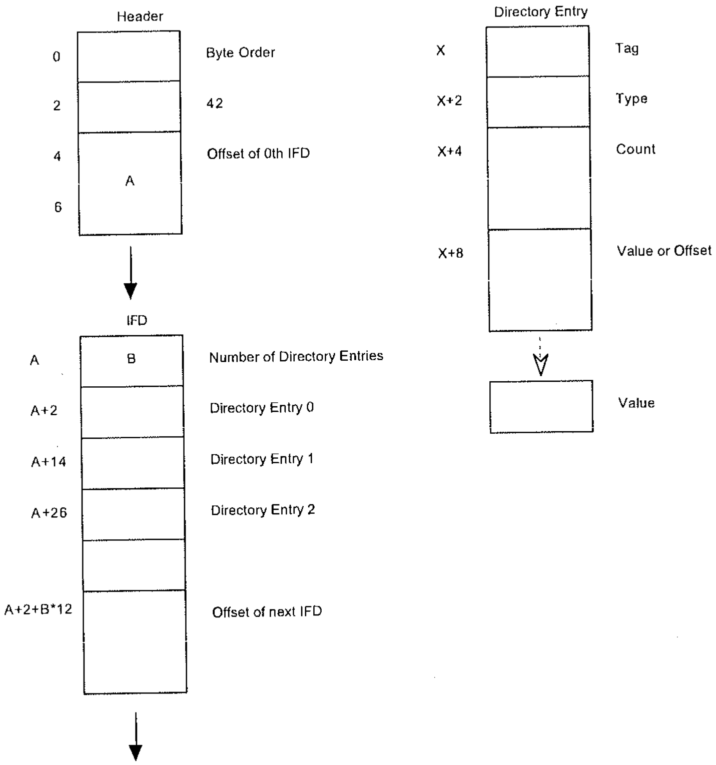
Bytes 2-3 An arbitrary but carefully chosen number (42) that further identifies the file as a TIFF file.

The byte order depends on the value of Bytes 0-1.

Bytes 4-7 The offset (in bytes) of the first IFD. The directory may be at any location in the file after the header but *must begin on a word boundary*. In particular, an Image File Directory may follow the image data it describes. Readers must follow the pointers wherever they may lead.

The term *byte offset* is always used in this document to refer to a location with respect to the beginning of the TIFF file. The first byte of the file has an offset of 0.

Figure 1



## Image File Directory

An *Image File Directory* (*IFD*) consists of a 2-byte count of the number of directory entries (i.e., the number of fields), followed by a sequence of 12-byte field entries, followed by a 4-byte offset of the next IFD (or 0 if none). (Do not forget to write the 4 bytes of 0 after the last IFD.)

There must be at least 1 IFD in a TIFF file and each IFD must have at least one entry.

See Figure 1.

### IFD Entry

Each 12-byte IFD entry has the following format:

Bytes 0-1 The Tag that identifies the field.

Bytes 2-3 The field Type.

Bytes 4-7 The number of values, *Count* of the indicated Type.

Bytes 8-11 The Value Offset, the file offset (in bytes) of the Value for the field. The Value is expected to begin on a word boundary; the corresponding Value Offset will thus be an even number. This file offset may point anywhere in the file, even after the image data.

### IFD Terminology

A *TIFF field* is a logical entity consisting of TIFF tag and its value. This logical concept is implemented as an *IFD Entry*, plus the actual value if it doesn't fit into the value/offset part, the last 4 bytes of the IFD Entry. The terms *TIFF field* and *IFD entry* are interchangeable in most contexts.

### Sort Order

The entries in an IFD must be sorted in ascending order by Tag. Note that this is not the order in which the fields are described in this document. The Values to which directory entries point need not be in any particular order in the file.

### Value/Offset

To save time and space the Value Offset contains the Value instead of pointing to the Value if and only if the Value fits into 4 bytes. If the Value is shorter than 4 bytes, it is left-justified within the 4-byte Value Offset, i.e., stored in the lower-numbered bytes. Whether the Value fits within 4 bytes is determined by the Type and Count of the field.

### Count

Count—called *Length* in previous versions of the specification—is the number of values. Note that Count is not the total number of bytes. For example, a single 16-bit word (SHORT) has a Count of 1; not 2.

### Types

The field types and their sizes are:

| | |
|---|---|
| 1 = BYTE | 8-bit unsigned integer. |
| 2 = ASCII | 8-bit byte that contains a 7-bit ASCII code; the last byte must be NUL (binary zero). |
| 3 = SHORT | 16-bit (2-byte) unsigned integer. |
| 4 = LONG | 32-bit (4-byte) unsigned integer. |
| 5 = RATIONAL | Two LONGs: the first represents the numerator of a fraction; the second, the denominator. |

The value of the Count part of an ASCII field entry includes the NUL. If padding is necessary, the Count does not include the pad byte. Note that there is no initial "count byte" as in Pascal-style strings.

*Any ASCII field can contain multiple strings, each terminated with a NUL. A single string is preferred whenever possible. The Count for multi-string fields is the number of bytes in all the strings in that field plus their terminating NUL bytes. Only one NUL is allowed between strings, so that the strings following the first string will often begin on an odd byte.*

The reader must check the type to verify that it contains an expected value. TIFF currently allows more than 1 valid type for some fields. For example, ImageWidth and ImageLength are usually specified as having type SHORT. But images with more than 64K rows or columns must use the LONG field type.

*TIFF readers should accept BYTE, SHORT, or LONG values for any unsigned integer field. This allows a single procedure to retrieve any integer value, makes reading more robust, and saves disk space in some situations.*

*In TIFF 6.0, some new field types have been defined:*

| | |
|---|---|
| 6 = SBYTE | An 8-bit signed (twos-complement) integer. |
| 7 = UNDEFINED | An 8-bit byte that may contain anything, depending on the definition of the field. |
| 8 = SSHORT | A 16-bit (2-byte) signed (twos-complement) integer. |
| 9 = SLONG | A 32-bit (4-byte) signed (twos-complement) integer. |
| 10 = SRATIONAL | Two SLONG's: the first represents the numerator of a fraction, the second the denominator. |
| 11 = FLOAT | Single precision (4-byte) IEEE format. |
| 12 = DOUBLE | Double precision (8-byte) IEEE format. |

*These new field types are also governed by the byte order (II or MM) in the TIFF header.*

***Warning: It is possible that other TIFF field types will be added in the future. Readers should skip over fields containing an unexpected field type.***

### *Fields are arrays*

*Each TIFF field has an associated Count. This means that all fields are actually one-dimensional arrays, even though most fields contain only a single value.*

*For example, to store a complicated data structure in a single private field, use the UNDEFINED field type and set the Count to the number of bytes required to hold the data structure.*

## *Multiple Images per TIFF File*

There may be more than one IFD in a TIFF file. Each IFD defines a *subfile*. One potential use of subfiles is to describe related images, such as the pages of a facsimile transmission. A Baseline TIFF reader is not required to read any IFDs beyond the first one.

APPENDIX HH

# Section 3: Bilevel Images

Now that the overall TIFF structure has been described, we can move on to filling the structure with actual fields (tags and values) that describe raster image data.

To make all of this clearer, the discussion will be organized according to the four Baseline TIFF image types: bilevel, grayscale, palette-color, and full-color images. This section describes bilevel images.

Fields required to describe bilevel images are introduced and described briefly here. Full descriptions of each field can be found in Section 8.

## Color

A bilevel image contains two colors—black and white. TIFF allows an application to write out bilevel data in either a white-is-zero or black-is-zero format. The field that records this information is called PhotometricInterpretation.

### PhotometricInterpretation

Tag = 262 (106.H)

Type = SHORT

Values:

0 = WhiteIsZero. For bilevel and grayscale images: 0 is imaged as white. The maximum value is imaged as black. This is the normal value for Compression=2.

1 = BlackIsZero. For bilevel and grayscale images: 0 is imaged as black. The maximum value is imaged as white. If this value is specified for Compression=2, the image should display and print reversed.

## Compression

Data can be stored either compressed or uncompressed.

### Compression

Tag = 259 (103.H)

Type = SHORT

Values:

1 = No compression, but pack data into bytes as tightly as possible, leaving no unused bits (except at the end of a row). The component values are stored as an array of type BYTE. Each scan line (row) is padded to the next BYTE boundary.

2 = CCITT Group 3 1-Dimensional Modified Huffman run length encoding. See

Section 10 for a description of Modified Huffman Compression.

32773 = PackBits compression, a simple byte-oriented run length scheme. See the PackBits section for details.

Data compression applies only to raster image data. All other TIFF fields are unaffected.

*Baseline TIFF readers must handle all three compression schemes.*

## Rows and Columns

An image is organized as a rectangular array of pixels. The dimensions of this array are stored in the following fields:

### ImageLength

Tag = 257 (101.H)

Type = SHORT or LONG

The number of rows (sometimes described as *scanlines*) in the image.

### ImageWidth

Tag = 256 (100.H)

Type = SHORT or LONG

The number of columns in the image, i.e., the number of pixels per scanline.

## Physical Dimensions

Applications often want to know the size of the picture represented by an image. This information can be calculated from ImageWidth and ImageLength given the following resolution data:

### ResolutionUnit

Tag = 296 (128.H)

Type = SHORT

Values:

1 = No absolute unit of measurement. Used for images that may have a non-square aspect ratio but no meaningful absolute dimensions.

2 = Inch.

3 = Centimeter.

Default = 2 (inch).

### XResolution

Tag = 282 (11A.H)

Type = RATIONAL

The number of pixels per ResolutionUnit in the ImageWidth (typically, horizontal - see Orientation) direction.

### YResolution

Tag = 283 (11B.H)

Type = RATIONAL

The number of pixels per ResolutionUnit in the ImageLength (typically, vertical) direction.

## Location of the Data

Compressed or uncompressed image data can be stored almost anywhere in a TIFF file. TIFF also supports breaking an image into separate strips for increased editing flexibility and efficient I/O buffering. The location and size of each strip is given by the following fields:

### RowsPerStrip

Tag = 278 (116.H)

Type = SHORT or LONG

The number of rows in each strip (except possibly the last strip.)

For example, if ImageLength is 24, and RowsPerStrip is 10, then there are 3 strips, with 10 rows in the first strip, 10 rows in the second strip, and 4 rows in the third strip. (The data in the last strip is not padded with 6 extra rows of dummy data.)

### StripOffsets

Tag = 273 (111.H)

Type = SHORT or LONG

For each strip, the byte offset of that strip.

### StripByteCounts

Tag = 279 (117.H)

Type = SHORT or LONG

For each strip, the number of bytes in that strip *after any compression.*

APPENDIX HH

Putting it all together (along with a couple of less-important fields that are discussed later), a sample bilevel image file might contain the following fields:

## *A Sample Bilevel TIFF File*

| **Offset (hex)** | **Description** | **Value (numeric values are expressed in hexadecimal notation)** | | | |
|---|---|---|---|---|---|
| ***Header:*** | | | | | |
| 0000 | Byte Order | 4D4D | | | |
| 0002 | 42 | 002A | | | |
| 0004 | 1st IFD offset | 00000014 | | | |
| ***IFD:*** | | | | | |
| 0014 | Number of Directory Entries | 000C | | | |
| 0016 | NewSubfileType | 00FE | 0004 | 00000001 | 00000000 |
| 0022 | ImageWidth | 0100 | 0004 | 00000001 | 000007D0 |
| 002E | ImageLength | 0101 | 0004 | 00000001 | 00000BB8 |
| 003A | Compression | 0103 | 0003 | 00000001 | 8005 0000 |
| 0046 | PhotometricInterpretation | 0106 | 0003 | 00000001 | 0001 0000 |
| 0052 | StripOffsets | 0111 | 0004 | 000000BC | 000000B6 |
| 005E | RowsPerStrip | 0116 | 0004 | 00000001 | 00000010 |
| 006A | StripByteCounts | 0117 | 0003 | 000000BC | 000003A6 |
| 0076 | XResolution | 011A | 0005 | 00000001 | 00000696 |
| 0082 | YResolution | 011B | 0005 | 00000001 | 0000069E |
| 008E | Software | 0131 | 0002 | 0000000E | 000006A6 |
| 009A | DateTime | 0132 | 0002 | 00000014 | 000006B6 |
| 00A6 | Next IFD offset | 00000000 | | | |
| ***Values longer than 4 bytes:*** | | | | | |
| 00B6 | StripOffsets | Offset0, Offset1, ... Offset187 | | | |
| 03A6 | StripByteCounts | Count0, Count1, ... Count187 | | | |
| 0696 | XResolution | 0000012C 00000001 | | | |
| 069E | YResolution | 0000012C 00000001 | | | |
| 06A6 | Software | "PageMaker 4.0" | | | |
| 06B6 | DateTime | "1988:02:18 13:59:59" | | | |
| ***Image Data:*** | | | | | |
| 00000700 | | Compressed data for strip 10 | | | |
| xxxxxxxx | | Compressed data for strip 179 | | | |
| xxxxxxxx | | Compressed data for strip 53 | | | |
| xxxxxxxx | | Compressed data for strip 160 | | | |
| . | | | | | |
| . | | | | | |

***End of example***

## Comments on the Bilevel Image Example

- The IFD in this example starts at 14h. It could have started anywhere in the file providing the offset was an even number greater than or equal to 8 (since the TIFF header is always the first 8 bytes of a TIFF file).
- With 16 rows per strip, there are 188 strips in all.
- The example uses a number of optional fields such as DateTime. TIFF readers must safely skip over these fields if they do not understand or do not wish to use the information. Baseline TIFF readers must not require that such fields be present.
- To make a point, this example has highly-fragmented image data. The strips of the image are not in sequential order. The point of this example is to illustrate that strip offsets must not be ignored. Never assume that strip N+1 follows strip N on disk. It is not required that the image data follow the IFD information.

## Required Fields for Bilevel Images

Here is a list of required fields for Baseline TIFF bilevel images. The fields are listed in numerical order, as they would appear in the IFD. Note that the previous example omits some of these fields. This is permitted because the fields that were omitted each have a default and the default is appropriate for this file.

| TagName | Decimal | Hex | Type | Value |
|---|---|---|---|---|
| ImageWidth | 256 | 100 | SHORT or LONG | |
| ImageLength | 257 | 101 | SHORT or LONG | |
| Compression | 259 | 103 | SHORT | 1, 2 or 32773 |
| PhotometricInterpretation | 262 | 106 | SHORT | 0 or 1 |
| StripOffsets | 273 | 111 | SHORT or LONG | |
| RowsPerStrip | 278 | 116 | SHORT or LONG | |
| StripByteCounts | 279 | 117 | LONG or SHORT | |
| XResolution | 282 | 11A | RATIONAL | |
| YResolution | 283 | 11B | RATIONAL | |
| ResolutionUnit | 296 | 128 | SHORT | 1, 2 or 3 |

Baseline TIFF bilevel images were called TIFF Class B images in earlier versions of the TIFF specification.

APPENDIX HH

# Section 4: Grayscale Images

Grayscale images are a generalization of bilevel images. Bilevel images can store only black and white image data, but grayscale images can also store shades of gray.

To describe such images, you must add or change the following fields. The other required fields are the same as those required for bilevel images.

## Differences from Bilevel Images

**Compression = 1** ***or 32773 (PackBits).*** In Baseline TIFF, grayscale images can either be stored as uncompressed data or compressed with the PackBits algorithm.

Caution: PackBits is often ineffective on continuous tone images, including many grayscale images. In such cases, it is better to leave the image uncompressed.

### BitsPerSample

Tag = 258 (102.H)

Type = SHORT

The number of bits per component.

Allowable values for Baseline TIFF grayscale images are **4** and **8**, allowing either 16 or 256 distinct shades of gray.

## Required Fields for Grayscale Images

These are the required fields for grayscale images (in numerical order):

| TagName | Decimal | Hex | Type | Value |
|---|---|---|---|---|
| ImageWidth | 256 | 100 | SHORT or LONG | |
| ImageLength | 257 | 101 | SHORT or LONG | |
| BitsPerSample | 258 | 102 | SHORT | 4 or 8 |
| Compression | 259 | 103 | SHORT | 1 or 32773 |
| PhotometricInterpretation | 262 | 106 | SHORT | 0 or 1 |
| StripOffsets | 273 | 111 | SHORT or LONG | |
| RowsPerStrip | 278 | 116 | SHORT or LONG | |
| StripByteCounts | 279 | 117 | LONG or SHORT | |
| XResolution | 282 | 11A | RATIONAL | |
| YResolution | 283 | 11B | RATIONAL | |
| ResolutionUnit | 296 | 128 | SHORT | 1 or 2 or 3 |

Baseline TIFF grayscale images were called TIFF Class G images in earlier versions of the TIFF specification.

# Section 5: Palette-color Images

Palette-color images are similar to grayscale images. They still have one component per pixel, but the component value is used as an index into a full RGB-lookup table. To describe such images, you need to add or change the following fields. The other required fields are the same as those for grayscale images.

## *Differences from Grayscale Images*

**PhotometricInterpretation = 3 (Palette Color).**

### *ColorMap*

Tag = 320 (140.H)

Type = SHORT

N = 3 * (2**BitsPerSample)

This field defines a Red-Green-Blue color map (often called a lookup table) for palette color images. In a palette-color image, a pixel value is used to index into an RGB-lookup table. For example, a palette-color pixel having a value of 0 would be displayed according to the 0th Red, Green, Blue triplet.

In a TIFF ColorMap, all the Red values come first, followed by the Green values, then the Blue values. In the ColorMap, black is represented by 0,0,0 and white is represented by 65535, 65535, 65535.

## *Required Fields for Palette Color Images*

These are the required fields for palette-color images (in numerical order):

| TagName | Decimal | Hex | Type | Value |
|---|---|---|---|---|
| ImageWidth | 256 | 100 | SHORT or LONG | |
| ImageLength | 257 | 101 | SHORT or LONG | |
| BitsPerSample | 258 | 102 | SHORT | 4 or 8 |
| Compression | 259 | 103 | SHORT | 1 or 32773 |
| PhotometricInterpretation | 262 | 106 | SHORT | 3 |
| StripOffsets | 273 | 111 | SHORT or LONG | |
| RowsPerStrip | 278 | 116 | SHORT or LONG | |
| StripByteCounts | 279 | 117 | LONG or SHORT | |
| XResolution | 282 | 11A | RATIONAL | |
| YResolution | 283 | 11B | RATIONAL | |
| ResolutionUnit | 296 | 128 | SHORT | 1 or 2 or 3 |
| ColorMap | 320 | 140 | SHORT | |

Baseline TIFF palette-color images were called TIFF Class P images in earlier versions of the TIFF specification.

# Section 6: RGB Full Color Images

In an RGB image, each pixel is made up of three components: red, green, and blue. There is no ColorMap.

To describe an RGB image, you need to add or change the following fields and values. The other required fields are the same as those required for palette-color images.

## Differences from Palette Color Images

**BitsPerSample = 8,8,8**. Each component is 8 bits deep in a Baseline TIFF RGB image.

**PhotometricInterpretation = 2 (RGB).**

There is no **ColorMap**.

### SamplesPerPixel

Tag = 277 (115.H)

Type = SHORT

The number of components per pixel. This number is 3 for RGB images, unless extra samples are present. See the ExtraSamples field for further information.

## Required Fields for RGB Images

These are the required fields for RGB images (in numerical order):

| TagName | Decimal | Hex | Type | Value |
|---|---|---|---|---|
| ImageWidth | 256 | 100 | SHORT or LONG | |
| ImageLength | 257 | 101 | SHORT or LONG | |
| BitsPerSample | 258 | 102 | SHORT | 8,8,8 |
| Compression | 259 | 103 | SHORT | 1 or 32773 |
| PhotometricInterpretation | 262 | 106 | SHORT | 2 |
| StripOffsets | 273 | 111 | SHORT or LONG | |
| SamplesPerPixel | 277 | 115 | SHORT | 3 or more |
| RowsPerStrip | 278 | 116 | SHORT or LONG | |
| StripByteCounts | 279 | 117 | LONG or SHORT | |
| XResolution | 282 | 11A | RATIONAL | |
| YResolution | 283 | 11B | RATIONAL | |
| ResolutionUnit | 296 | 128 | SHORT | 1, 2 or 3 |

The BitsPerSample values listed above apply only to the main image data. If ExtraSamples are present, the appropriate BitsPerSample values for those samples must also be included.

Baseline TIFF RGB images were called TIFF Class R images in earlier versions of the TIFF specification.

# Section 7: Additional Baseline TIFF Requirements

This section describes characteristics required of all Baseline TIFF files.

## *General Requirements*

**Options.** Where there are options, TIFF writers can use whichever they want. Baseline TIFF readers must be able to handle all of them.

**Defaults.** TIFF writers may, but are not required to, write out a field that has a default value, if the default value is the one desired. TIFF readers must be prepared to handle either situation.

**Other fields.** TIFF readers must be prepared to encounter fields other than those required in TIFF files. TIFF writers are allowed to write optional fields such as Make, Model, and DateTime, and TIFF readers may use such fields if they exist. TIFF readers must not, however, refuse to read the file if such optional fields do not exist. *TIFF readers must also be prepared to encounter and ignore private fields not described in the TIFF specification.*

**'MM' and 'II' byte order.** TIFF readers must be able to handle both byte orders. TIFF writers can do whichever is most convenient or efficient.

**Multiple subfiles.** TIFF readers must be prepared for multiple images (subfiles) per TIFF file, although they are not required to do anything with images after the first one. TIFF writers are required to write a long word of 0 after the last IFD (to signal that this is the last IFD), as described earlier in this specification.

If multiple subfiles are written, the first one must be the full-resolution image. Subsequent images, such as reduced-resolution images, may be in any order in the TIFF file. If a reader wants to use such images, it must scan the corresponding IFD's before deciding how to proceed.

**TIFF Editors.** Editors—applications that modify TIFF files—have a few additional requirements:

- TIFF editors must be especially careful about subfiles. If a TIFF editor edits a full-resolution subfile, but does not update an accompanying reduced-resolution subfile, a reader that uses the reduced-resolution subfile for screen display will display the wrong thing. So TIFF editors must either create a new reduced-resolution subfile when they alter a full-resolution subfile or they must delete any subfiles that they aren't prepared to deal with.
- A similar situation arises with the fields in an IFD. It is unnecessary—and possibly dangerous—for an editor to copy fields it does not understand because the editor might alter the file in a way that is incompatible with the unknown fields.

***No Duplicate Pointers.*** *No data should be referenced from more than one place. TIFF readers and editors are under no obligation to detect this condition and handle it properly. This would not be a problem if TIFF files were read-only enti-*

APPENDIX HH

*ties, but they are not. This warning covers both TIFF field value offsets and fields that are defined as offsets, such as StripOffsets.*

***Point to real data.*** *All strip offsets must reference valid locations. (It is not legal to use an offset of 0 to mean something special.)*

***Beware of extra components.*** *Some TIFF files may have more components per pixel than you think. A Baseline TIFF reader must skip over them gracefully, using the values of the SamplesPerPixel and BitsPerSample fields. For example, it is possible that the data will have a PhotometricInterpretation of RGB but have 4 SamplesPerPixel. See ExtraSamples for further details.*

***Beware of new field types.*** *Be prepared to handle unexpected field types such as floating-point data. A Baseline TIFF reader must skip over such fields gracefully. Do not expect that BYTE, ASCII, SHORT, LONG, and RATIONAL will always be a complete list of field types.*

***Beware of new pixel types.*** *Some TIFF files may have pixel data that consists of something other than unsigned integers. If the SampleFormat field is present and the value is not 1, a Baseline TIFF reader that cannot handle the SampleFormat value must terminate the import process gracefully.*

## Notes on Required Fields

**ImageWidth, ImageLength.** Both “SHORT” and “LONG” TIFF field types are allowed and must be handled properly by readers. TIFF writers can use either type. TIFF readers are not required to read arbitrarily large files however. Some readers will give up if the entire image cannot fit into available memory. (In such cases the reader should inform the user about the problem.) Others will probably not be able to handle ImageWidth greater than 65535.

**RowsPerStrip.** SHORT or LONG. Readers must be able to handle any value between 1 and 2**32-1. However, some readers may try to read an entire strip into memory at one time. If the entire image is one strip, the application may run out of memory. Recommendation: Set RowsPerStrip such that the size of each strip is about 8K bytes. Do this even for uncompressed data because it is easy for a writer and makes things simpler for readers. Note that extremely wide high-resolution images may have rows larger than 8K bytes; in this case, RowsPerStrip should be 1, and the strip will be larger than 8K.

**StripOffsets.** SHORT or LONG.

**StripByteCounts.** SHORT or LONG.

**XResolution, YResolution.** RATIONAL. Note that the X and Y resolutions may be unequal. A TIFF reader must be able to handle this case. Typically, TIFF pixel-editors do not care about the resolution, but applications (such as page layout programs) do care.

**ResolutionUnit.** SHORT. TIFF readers must be prepared to handle all three values for ResolutionUnit.

# Section 8: Baseline Field Reference Guide

This section contains detailed information about all the Baseline fields defined in this version of TIFF. A *Baseline field* is any field commonly found in a Baseline TIFF file, whether required or not.

For convenience, fields that were defined in earlier versions of the TIFF specification but are no longer generally recommended have also been included in this section.

New fields that are associated with optional features are not listed in this section. See Part 2 for descriptions of these new fields. There is a complete list of all fields described in this specification in Appendix A, and there are entries for all TIFF fields in the index.

More fields may be added in future versions. Whenever possible they will be added in a way that allows old TIFF readers to read newer TIFF files.

The documentation for each field contains:

- the name of the field
- the Tag number
- the field Type
- the required Number of Values (N); i.e., the Count
- comments describing the field
- the default, if any

If the field does not exist, readers must assume the default value for the field.

Most of the fields described in this part of the document are not required or are required only for particular types of TIFF files. See the preceding sections for lists of required fields.

Before defining the fields, you must understand these basic concepts: A Baseline TIFF *image* is defined to be a two-dimensional array of *pixels*, each of which consists of one or more color *components*. Monochromatic data has one color component per pixel, while RGB color data has three color components per pixel.

## The Fields

### Artist

Person who created the image.

Tag = 315 (13B.H)

Type = ASCII

Note: some older TIFF files used this tag for storing Copyright information.

## BitsPerSample

Number of bits per component.

Tag = 258 (102.H)

Type = SHORT

N = SamplesPerPixel

Note that this field allows a different number of bits per component for each component corresponding to a pixel. For example, RGB color data could use a different number of bits per component for each of the three color planes. Most RGB files will have the same number of BitsPerSample for each component. Even in this case, the writer must write all three values.

Default = 1. See also SamplesPerPixel.

## CellLength

The length of the dithering or halftoning matrix used to create a dithered or halftoned bilevel file.

Tag = 265 (109.H)

Type = SHORT

N = 1

This field should only be present if Threshholding = 2

No default. See also Threshholding.

## CellWidth

The width of the dithering or halftoning matrix used to create a dithered or halftoned bilevel file.Tag = 264 (108.H)

Type = SHORT

N = 1

No default. See also Threshholding.

## ColorMap

A color map for palette color images.

Tag = 320 (140.H)

Type = SHORT

N = 3 * (2**BitsPerSample)

This field defines a Red-Green-Blue color map (often called a lookup table) for palette-color images. In a palette-color image, a pixel value is used to index into an RGB lookup table. For example, a palette-color pixel having a value of 0 would be displayed according to the 0th Red, Green, Blue triplet.

In a TIFF ColorMap, all the Red values come first, followed by the Green values, then the Blue values. The number of values for each color is 2**BitsPerSample. Therefore, the ColorMap field for an 8-bit palette-color image would have 3 * 256 values.

The width of each value is 16 bits, as implied by the type of SHORT. 0 represents the minimum intensity, and 65535 represents the maximum intensity. Black is represented by 0,0,0, and white by 65535, 65535, 65535.

See also PhotometricInterpretation—palette color.

No default. ColorMap must be included in all palette-color images.

## *Compression*

Compression scheme used on the image data.

Tag = 259 (103.H)

Type = SHORT

N = 1

1 = No compression, but pack data into bytes as tightly as possible leaving no unused bits except at the end of a row.

| <u>If</u> | <u>Then the sample values are stored as an array of type:</u> |
|---|---|
| BitsPerSample = 16 for all samples | SHORT |
| BitsPerSample = 32 for all samples | LONG |
| Otherwise | BYTE |

*Each row is padded to the next BYTE/SHORT/LONG boundary, consistent with the preceding BitsPerSample rule.*

If the image data is stored as an array of SHORTs or LONGs, the byte ordering must be consistent with that specified in bytes 0 and 1 of the TIFF file header. Therefore, little-endian format files will have the least significant bytes preceding the most significant bytes, while big-endian format files will have the opposite order.

If the number of bits per component is not a power of 2, and you are willing to give up some space for better performance, use the next higher power of 2. For example, if your data can be represented in 6 bits, set BitsPerSample to 8 instead of 6, and then convert the range of the values from [0,63] to [0,255].

Rows must begin on byte boundaries. *(SHORT boundaries if the data is stored as SHORTs, LONG boundaries if the data is stored as LONGs).*

Some graphics systems require image data rows to be word-aligned or double-word-aligned, and padded to word-boundaries or double-word boundaries. Uncompressed TIFF rows will need to be copied into word-aligned or double-word-aligned row buffers before being passed to the graphics routines in these environments.

2 = CCITT Group 3 1-Dimensional Modified Huffman run-length encoding. See Section 10. BitsPerSample must be 1, since this type of compression is defined only for bilevel images.

32773 = PackBits compression, a simple byte-oriented run-length scheme. See Section 9 for details.

Data compression applies only to the image data, pointed to by StripOffsets.

Default = 1.

## Copyright

Copyright notice.

Tag = 33432 (8298.H)

Type = ASCII

Copyright notice of the person or organization that claims the copyright to the image. The complete copyright statement should be listed in this field including any dates and statements of claims. For example, "Copyright, John Smith, 19xx. All rights reserved."

## DateTime

Date and time of image creation.

Tag = 306 (132.H)

Type = ASCII

N = 20

The format is: "YYYY:MM:DD HH:MM:SS", with hours like those on a 24-hour clock, and one space character between the date and the time. The length of the string, including the terminating NUL, is 20 bytes.

## ExtraSamples

Description of extra components.

Tag = 338 (152.H)

Type = SHORT

N = m

Specifies that each pixel has *m* extra components whose interpretation is defined by one of the values listed below. When this field is used, the SamplesPerPixel field has a value greater than the PhotometricInterpretation field suggests.

For example, full-color RGB data normally has SamplesPerPixel=3. If SamplesPerPixel is greater than 3, then the ExtraSamples field describes the meaning of the extra samples. If SamplesPerPixel is, say, 5 then ExtraSamples will contain 2 values, one for each extra sample.

ExtraSamples is typically used to include non-color information, such as opacity, in an image. The possible values for each item in the field's value are:

0 = Unspecified data

1 = Associated alpha data (with pre-multiplied color)

2 = Unassociated alpha data

Associated alpha data is opacity information; it is fully described in Section 21. Unassociated alpha data is transparency information that logically exists independent of an image; it is commonly called a soft matte. Note that including both unassociated and associated alpha is undefined because associated alpha specifies that color components are pre-multiplied by the alpha component, while unassociated alpha specifies the opposite.

By convention, extra components that are present must be stored as the "last components" in each pixel. For example, if SamplesPerPixel is 4 and there is 1 extra component, then it is located in the last component location (SamplesPerPixel-1) in each pixel.

Components designated as "extra" are just like other components in a pixel. In particular, the size of such components is defined by the value of the BitsPerSample field.

With the introduction of this field, TIFF readers must not assume a particular SamplesPerPixel value based on the value of the PhotometricInterpretation field. For example, if the file is an RGB file, SamplesPerPixel may be greater than 3.

The default is no extra samples. This field must be present if there are extra samples.

See also SamplesPerPixel, AssociatedAlpha.

## *FillOrder*

The logical order of bits within a byte.

Tag = 266 (10A.H)

Type = SHORT

N = 1

1 = pixels are arranged within a byte such that pixels with lower column values are stored in the higher-order bits of the byte.

1-bit uncompressed data example: Pixel 0 of a row is stored in the high-order bit of byte 0, pixel 1 is stored in the next-highest bit, ..., pixel 7 is stored in the low-order bit of byte 0, pixel 8 is stored in the high-order bit of byte 1, and so on.

CCITT 1-bit compressed data example: The high-order bit of the first compression code is stored in the high-order bit of byte 0, the next-highest bit of the first compression code is stored in the next-highest bit of byte 0, and so on.

2 = pixels are arranged within a byte such that pixels with lower column values are stored in the lower-order bits of the byte.

We recommend that FillOrder=2 be used only in special-purpose applications. It is easy and inexpensive for writers to reverse bit order by using a 256-byte lookup table. *FillOrder = 2 should be used only when BitsPerSample = 1 and the data is either uncompressed or compressed using CCITT 1D or 2D compression, to avoid potentially ambigous situations.*

Support for FillOrder=2 is not required in a Baseline TIFF compliant reader

Default is FillOrder = 1.

## FreeByteCounts

For each string of contiguous unused bytes in a TIFF file, the number of bytes in the string.

Tag = 289 (121.H)

Type = LONG

Not recommended for general interchange.

See also FreeOffsets.

## FreeOffsets

For each string of contiguous unused bytes in a TIFF file, the byte offset of the string.

Tag = 288 (120.H)

Type = LONG

Not recommended for general interchange.

See also FreeByteCounts.

## GrayResponseCurve

For grayscale data, the optical density of each possible pixel value.

Tag = 291 (123.H)

Type = SHORT

N = 2**BitsPerSample

The 0th value of GrayResponseCurve corresponds to the optical density of a pixel having a value of 0, and so on.

This field may provide useful information for sophisticated applications, but it is currently ignored by most TIFF readers.

See also GrayResponseUnit, PhotometricInterpretation.

## GrayResponseUnit

The precision of the information contained in the GrayResponseCurve.

Tag = 290 (122.H)

Type = SHORT

N = 1

Because optical density is specified in terms of fractional numbers, this field is necessary to interpret the stored integer information. For example, if GrayScaleResponseUnits is set to 4 (ten-thousandths of a unit), and a GrayScaleResponseCurve number for gray level 4 is 3455, then the resulting actual value is 0.3455.

Optical densitometers typically measure densities within the range of 0.0 to 2.0.

1 = Number represents tenths of a unit.

2 = Number represents hundredths of a unit.

3 = Number represents thousandths of a unit.

4 = Number represents ten-thousandths of a unit.

5 = Number represents hundred-thousandths of a unit.

Modifies GrayResponseCurve.

See also GrayResponseCurve.

For historical reasons, the default is 2. However, for greater accuracy, 3 is recommended.

### HostComputer

The computer and/or operating system in use at the time of image creation.

Tag = 316 (13C.H)

Type = ASCII

See also Make, Model, Software.

### ImageDescription

A string that describes the subject of the image.

Tag = 270 (10E.H)

Type = ASCII

For example, a user may wish to attach a comment such as "1988 company picnic" to an image.

### ImageLength

The number of rows of pixels in the image.

Tag = 257 (101.H)

Type = SHORT or LONG

N = 1

No default. See also ImageWidth.

### ImageWidth

The number of columns in the image, i.e., the number of pixels per row.

Tag = 256 (100.H)

Type = SHORT or LONG

N = 1

No default. See also ImageLength.

## Make

The scanner manufacturer.

Tag = 271 (10F.H)

Type = ASCII

Manufacturer of the scanner, video digitizer, or other type of equipment used to generate the image. Synthetic images should not include this field.

See also Model, Software.

## MaxSampleValue

The maximum component value used.

Tag = 281 (119.H)

Type = SHORT

N = SamplesPerPixel

This field is not to be used to affect the visual appearance of an image when it is displayed or printed. Nor should this field affect the interpretation of any other field; it is used only for statistical purposes.

Default is 2**(BitsPerSample) - 1.

## MinSampleValue

The minimum component value used.

Tag = 280 (118.H)

Type = SHORT

N = SamplesPerPixel

See also MaxSampleValue.

Default is 0.

## Model

The scanner model name or number.

Tag = 272 (110.H)

Type = ASCII

The model name or number of the scanner, video digitizer, or other type of equipment used to generate the image.

See also Make, Software.

## NewSubfileType

A general indication of the kind of data contained in this subfile.

Tag = 254 (FE.H)

Type = LONG

N = 1

Replaces the old SubfileType field, due to limitations in the definition of that field.

NewSubfileType is mainly useful when there are multiple subfiles in a single TIFF file.

This field is made up of a set of 32 flag bits. Unused bits are expected to be 0. Bit 0 is the low-order bit.

Currently defined values are:

Bit 0 is 1 if the image is a reduced-resolution version of another image in this TIFF file; else the bit is 0.

Bit 1 is 1 if the image is a single page of a multi-page image (see the PageNumber field description); else the bit is 0.

Bit 2 is 1 if the image defines a transparency mask for another image in this TIFF file. The PhotometricInterpretation value must be 4, designating a transparency mask.

These values are defined as bit flags because they are independent of each other.

Default is 0.

## Orientation

The orientation of the image with respect to the rows and columns.

Tag = 274 (112.H)

Type = SHORT

N = 1

1 = The 0th row represents the visual top of the image, and the 0th column represents the visual left-hand side.

2 = The 0th row represents the visual top of the image, and the 0th column represents the visual right-hand side.

3 = The 0th row represents the visual bottom of the image, and the 0th column represents the visual right-hand side.

4 = The 0th row represents the visual bottom of the image, and the 0th column represents the visual left-hand side.

5 = The 0th row represents the visual left-hand side of the image, and the 0th column represents the visual top.

6 = The 0th row represents the visual right-hand side of the image, and the 0th column represents the visual top.

7 = The 0th row represents the visual right-hand side of the image, and the 0th column represents the visual bottom.

8 = The 0th row represents the visual left-hand side of the image, and the 0th column represents the visual bottom.

Default is 1.

*Support for orientations other than 1 is not a Baseline TIFF requirement.*

## PhotometricInterpretation

The color space of the image data.

Tag = 262 (106.H)

Type = SHORT

N = 1

0 = WhiteIsZero. For bilevel and grayscale images: 0 is imaged as white. 2**BitsPerSample-1 is imaged as black. This is the normal value for Compression=2.

1 = BlackIsZero. For bilevel and grayscale images: 0 is imaged as black. 2**BitsPerSample-1 is imaged as white. If this value is specified for Compression=2, the image should display and print reversed.

2 = RGB. In the RGB model, a color is described as a combination of the three primary colors of light (red, green, and blue) in particular concentrations. For each of the three components, 0 represents minimum intensity, and 2**BitsPerSample - 1 represents maximum intensity. Thus an RGB value of (0,0,0) represents black, and (255,255,255) represents white, assuming 8-bit components. For PlanarConfiguration = 1, the components are stored in the indicated order: first Red, then Green, then Blue. For PlanarConfiguration = 2, the StripOffsets for the component planes are stored in the indicated order: first the Red component plane StripOffsets, then the Green plane StripOffsets, then the Blue plane StripOffsets.

3= Palette color. In this model, a color is described with a single component. The value of the component is used as an index into the red, green and blue curves in the ColorMap field to retrieve an RGB triplet that defines the color. When PhotometricInterpretation=3 is used, ColorMap must be present and SamplesPerPixel must be 1.

4 = Transparency Mask.

This means that the image is used to define an irregularly shaped region of another image in the same TIFF file. SamplesPerPixel and BitsPerSample must be 1. PackBits compression is recommended. The 1-bits define the interior of the region; the 0-bits define the exterior of the region.

A reader application can use the mask to determine which parts of the image to display. Main image pixels that correspond to 1-bits in the transparency mask are imaged to the screen or printer, but main image pixels that correspond to 0-bits in the mask are not displayed or printed.

*The image mask is typically at a higher resolution than the main image, if the main image is grayscale or color so that the edges* can be sharp.

There is no default for PhotometricInterpretation, *and it is required.* Do not rely on applications defaulting to what you want.

APPENDIX HH

## PlanarConfiguration

How the components of each pixel are stored.

Tag = 284 (11C.H)

Type = SHORT

N = 1

1 = *Chunky* format. The component values for each pixel are stored contiguously. The order of the components within the pixel is specified by PhotometricInterpretation. For example, for RGB data, the data is stored as RGBRGBRGB…

2 = *Planar* format. The components are stored in separate "component planes." The values in StripOffsets and StripByteCounts are then arranged as a 2-dimensional array, with SamplesPerPixel rows and StripsPerImage columns. (All of the columns for row 0 are stored first, followed by the columns of row 1, and so on.) PhotometricInterpretation describes the type of data stored in each component plane. For example, RGB data is stored with the Red components in one component plane, the Green in another, and the Blue in another.

*PlanarConfiguration=2 is not currently in widespread use and it is not recommended for general interchange. It is used as an extension and Baseline TIFF readers are not required to support it.*

If SamplesPerPixel is 1, PlanarConfiguration is irrelevant, and need not be included.

If a row interleave effect is desired, a writer might write out the data as PlanarConfiguration=2—separate sample planes—but break up the planes into multiple strips (one row per strip, perhaps) and interleave the strips.

Default is 1. See also BitsPerSample, SamplesPerPixel.

## ResolutionUnit

The unit of measurement for XResolution and YResolution.

Tag = 296 (128.H)

Type = SHORT

N = 1

To be used with XResolution and YResolution.

1 = No absolute unit of measurement. Used for images that may have a non-square aspect ratio, but no meaningful absolute dimensions.

The drawback of ResolutionUnit=1 is that different applications will import the image at different sizes. Even if the decision is arbitrary, it might be better to use dots per inch or dots per centimeter, and to pick XResolution and YResolution so that the aspect ratio is correct and the maximum dimension of the image is about four inches (the "four" is arbitrary.)

2 = Inch.

3 = Centimeter.

Default is 2.

## RowsPerStrip

The number of rows per strip.

Tag = 278 (116.H)

Type = SHORT or LONG

N = 1

TIFF image data is organized into strips for faster random access and efficient I/O buffering.

RowsPerStrip and ImageLength together tell us the number of strips in the entire image. The equation is:

**StripsPerImage** = floor ((ImageLength + RowsPerStrip - 1) / RowsPerStrip).

StripsPerImage is *not* a field. It is merely a value that a TIFF reader will want to compute because it specifies the number of StripOffsets and StripByteCounts for the image.

Note that either SHORT or LONG values can be used to specify RowsPerStrip. SHORT values may be used for small TIFF files. It should be noted, however, that earlier TIFF specification revisions required LONG values and that some software may not accept SHORT values.

The default is 2**32 - 1, which is effectively infinity. That is, the entire image is one strip.

Use of a single strip is not recommended. Choose RowsPerStrip such that each strip is about 8K bytes, even if the data is not compressed, since it makes buffering simpler for readers. The "8K" value is fairly arbitrary, but seems to work well.

See also ImageLength, StripOffsets, StripByteCounts, TileWidth, TileLength, TileOffsets, TileByteCounts.

## SamplesPerPixel

The number of components per pixel.

Tag = 277 (115.H)

Type = SHORT

N = 1

SamplesPerPixel is *usually* 1 for bilevel, grayscale, and palette-color images. SamplesPerPixel is *usually* 3 for RGB images.

Default = 1. See also BitsPerSample, PhotometricInterpretation, *ExtraSamples.*

## Software

Name and version number of the software package(s) used to create the image.

Tag = 305 (131.H)

Type = ASCII

See also Make, Model.

## StripByteCounts

For each strip, the number of bytes in the strip after compression.

Tag = 279 (117.H)

Type = SHORT or LONG

N = StripsPerImage for PlanarConfiguration equal to 1.

= SamplesPerPixel * StripsPerImage for PlanarConfiguration equal to 2

*This tag is required for Baseline TIFF files.*

No default.

See also StripOffsets, RowsPerStrip, TileOffsets, TileByteCounts.

## StripOffsets

For each strip, the byte offset of that strip.

Tag = 273 (111.H)

Type = SHORT or LONG

N = StripsPerImage for PlanarConfiguration equal to 1.

= SamplesPerPixel * StripsPerImage for PlanarConfiguration equal to 2

The offset is specified with respect to the beginning of the TIFF file. Note that this implies that each strip has a location independent of the locations of other strips. This feature may be useful for editing applications. This required field is the only way for a reader to find the image data. *(Unless TileOffsets is used; see TileOffsets.)*

Note that either SHORT or LONG values may be used to specify the strip offsets. SHORT values may be used for small TIFF files. It should be noted, however, that earlier TIFF specifications required LONG strip offsets and that some software may not accept SHORT values.

*For maximum compatibility with operating systems such as MS-DOS and Windows, the StripOffsets array should be less than or equal to 64K bytes in length, and the strips themselves, in both compressed and uncompressed forms, should not be larger than 64K bytes.*

No default. See also StripByteCounts, RowsPerStrip, TileOffsets, TileByteCounts.

## SubfileType

A general indication of the kind of data contained in this subfile.

Tag = 255 (FF.H)

Type = SHORT

N = 1

Currently defined values are:

1 = full-resolution image data

2 = reduced-resolution image data

3 = a single page of a multi-page image (see the PageNumber field description).

Note that several image types may be found in a single TIFF file, with each subfile described by its own IFD.

No default.

This field is deprecated. The NewSubfileType field should be used instead.

### *Threshholding*

For black and white TIFF files that represent shades of gray, the technique used to convert from gray to black and white pixels.

Tag = 263 (107.H)

Type = SHORT

N = 1

1 = No dithering or halftoning has been applied to the image data.

2 = An ordered dither or halftone technique has been applied to the image data.

3 = A randomized process such as error diffusion has been applied to the image data.

Default is Threshholding = 1. See also CellWidth, CellLength.

### *XResolution*

The number of pixels per ResolutionUnit in the ImageWidth direction.

Tag = 282 (11A.H)

Type = RATIONAL

N = 1

It is not mandatory that the image be actually displayed or printed at the size implied by this parameter. It is up to the application to use this information as it wishes.

No default. See also YResolution, ResolutionUnit.

### *YResolution*

The number of pixels per ResolutionUnit in the ImageLength direction.

Tag = 283 (11B.H)

Type = RATIONAL

N = 1

No default. See also XResolution, ResolutionUnit.

APPENDIX HH

# Section 9: PackBits Compression

This section describes TIFF compression type 32773, a simple byte-oriented run-length scheme.

## Description

In choosing a simple byte-oriented run-length compression scheme, we arbitrarily chose the Apple Macintosh PackBits scheme. It has a good worst case behavior (at most 1 extra byte for every 128 input bytes). For Macintosh users, the toolbox utilities PackBits and UnPackBits will do the work for you, but it is easy to implement your own routines.

A pseudo code fragment to unpack might look like this:

```
Loop until you get the number of unpacked bytes you are expecting:
    Read the next source byte into n.
    If n is between 0 and 127 inclusive, copy the next n+1 bytes literally.
    Else if n is between -127 and -1 inclusive, copy the next byte -n+1
    times.
    Else if n is -128, noop.
Endloop
```

In the inverse routine, it is best to encode a 2-byte repeat run as a replicate run except when preceded and followed by a literal run. In that case, it is best to merge the three runs into one literal run. Always encode 3-byte repeats as replicate runs.

That is the essence of the algorithm. Here are some additional rules:

- Pack each row separately. Do not compress across row boundaries.
- The number of uncompressed bytes per row is defined to be (ImageWidth + 7) / 8. If the uncompressed bitmap is required to have an even number of bytes per row, decompress into word-aligned buffers.
- If a run is larger than 128 bytes, encode the remainder of the run as one or more additional replicate runs.

When PackBits data is decompressed, the result should be interpreted as per compression type 1 (no compression).

APPENDIX HH

# Section 10: Modified Huffman Compression

This section describes TIFF compression scheme 2, a method for compressing bilevel data based on the CCITT Group 3 1D facsimile compression scheme.

## References

- "Standardization of Group 3 facsimile apparatus for document transmission," Recommendation T.4, Volume VII, Fascicle VII.3, Terminal Equipment and Protocols for Telematic Services, The International Telegraph and Telephone Consultative Committee (CCITT), Geneva, 1985, pages 16 through 31.
- "Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus," Recommendation T.6, Volume VII, Fascicle VII.3, Terminal Equipment and Protocols for Telematic Services, The International Telegraph and Telephone Consultative Committee (CCITT), Geneva, 1985, pages 40 through 48.

We do not believe that these documents are necessary in order to implement Compression=2. We have included (verbatim in most places) all the pertinent information in this section. However, if you wish to order the documents, you can write to ANSI, Attention: Sales, 1430 Broadway, New York, N.Y., 10018. Ask for the publication listed above—it contains both Recommendation T.4 and T.6.

## Relationship to the CCITT Specifications

The CCITT Group 3 and Group 4 specifications describe communications protocols for a particular class of devices. They are not by themselves sufficient to describe a disk data format. Fortunately, however, the CCITT coding schemes can be readily adapted to this different environment. The following is one such adaptation. Most of the language is copied directly from the CCITT specifications.

*See Section 11 for additional CCITT compression options.*

## Coding Scheme

A line (row) of data is composed of a series of variable length code words. Each code word represents a run length of all white or all black. (Actually, more than one code word may be required to code a given run, in a manner described below.) White runs and black runs alternate.

To ensure that the receiver (decompressor) maintains color synchronization, all data lines begin with a white run-length code word set. If the actual scan line begins with a black run, a white run-length of zero is sent (written). Black or white run-lengths are defined by the code words in Tables 1 and 2. The code words are of two types: Terminating code words and Make-up code words. Each run-length is represented by zero or more Make-up code words followed by exactly one Terminating code word.

Run lengths in the range of 0 to 63 pels (pixels) are encoded with their appropriate Terminating code word. Note that there is a different list of code words for black and white run-lengths.

Run lengths in the range of 64 to 2623 (2560+63) pels are encoded first by the Make-up code word representing the run-length that is nearest to, not longer than, that required. This is then followed by the Terminating code word representing the difference between the required run-length and the run-length represented by the Make-up code.

Run lengths in the range of lengths longer than or equal to 2624 pels are coded first by the Make-up code of 2560. If the remaining part of the run (after the first Make-up code of 2560) is 2560 pels or greater, additional Make-up code(s) of 2560 are issued until the remaining part of the run becomes less than 2560 pels. Then the remaining part of the run is encoded by Terminating code or by Make-up code plus Terminating code, according to the range mentioned above.

It is considered an unrecoverable error if the sum of the run-lengths for a line does not equal the value of the ImageWidth field.

New rows always begin on the next available byte boundary.

No EOL code words are used. No fill bits are used, except for the ignored bits at the end of the last byte of a row. RTC is not used.

*An encoded CCITT string is self-photometric, defined in terms of white and black runs. Yet TIFF defines a tag called PhotometricInterpretation that also purports to define what is white and what is black. Somewhat arbitrarily, we adopt the following convention:*

*The "normal" PhotometricInterpretation for bilevel CCITT compressed data is WhiteIsZero. In this case, the CCITT "white" runs are to be interpretated as white, and the CCITT "black" runs are to be interpreted as black. However, if the PhotometricInterpretation is BlackIsZero, the TIFF reader must reverse the meaning of white and black when displaying and printing the image.*

APPENDIX HH

*Table 1/T.4 Terminating codes*

| White run length | Code word | Black run length | Code word |
|---|---|---|---|
| 0 | 00110101 | 0 | 0000110111 |
| 1 | 000111 | 1 | 010 |
| 2 | 0111 | 2 | 11 |
| 3 | 1000 | 3 | 10 |
| 4 | 1011 | 4 | 011 |
| 5 | 1100 | 5 | 0011 |
| 6 | 1110 | 6 | 0010 |
| 7 | 1111 | 7 | 00011 |
| 8 | 10011 | 8 | 000101 |
| 9 | 10100 | 9 | 000100 |
| 10 | 00111 | 10 | 0000100 |
| 11 | 01000 | 11 | 0000101 |
| 12 | 001000 | 12 | 0000111 |
| 13 | 000011 | 13 | 00000100 |
| 14 | 110100 | 14 | 00000111 |
| 15 | 110101 | 15 | 000011000 |
| 16 | 101010 | 16 | 0000010111 |
| 17 | 101011 | 17 | 0000011000 |
| 18 | 0100111 | 18 | 0000001000 |
| 19 | 0001100 | 19 | 00001100111 |
| 20 | 0001000 | 20 | 00001101000 |
| 21 | 0010111 | 21 | 00001101100 |
| 22 | 0000011 | 22 | 00000110111 |
| 23 | 0000100 | 23 | 00000101000 |
| 24 | 0101000 | 24 | 00000010111 |
| 25 | 0101011 | 25 | 00000011000 |
| 26 | 0010011 | 26 | 000011001010 |
| 27 | 0100100 | 27 | 000011001011 |
| 28 | 0011000 | 28 | 000011001100 |
| 29 | 00000010 | 29 | 000011001101 |
| 30 | 00000011 | 30 | 000001101000 |
| 31 | 00011010 | 31 | 000001101001 |
| 32 | 00011011 | 32 | 000001101010 |
| 33 | 00010010 | 33 | 000001101011 |
| 34 | 00010011 | 34 | 000011010010 |
| 35 | 00010100 | 35 | 000011010011 |
| 36 | 00010101 | 36 | 000011010100 |
| 37 | 00010110 | 37 | 000011010101 |
| 38 | 00010111 | 38 | 000011010110 |
| 39 | 00101000 | 39 | 000011010111 |
| 40 | 00101001 | 40 | 000001101100 |
| 41 | 00101010 | 41 | 000001101101 |
| 42 | 00101011 | 42 | 000011011010 |
| 43 | 00101100 | 43 | 000011011011 |
| 44 | 00101101 | 44 | 000001010100 |
| 45 | 00000100 | 45 | 000001010101 |
| 46 | 00000101 | 46 | 000001010110 |
| 47 | 00001010 | 47 | 000001010111 |
| 48 | 00001011 | 48 | 000001100100 |
| 49 | 01010010 | 49 | 000001100101 |
| 50 | 01010011 | 50 | 000001010010 |
| 51 | 01010100 | 51 | 000001010011 |

| White<br>run<br>length | Code<br>word | Black<br>run<br>length | Code<br>word |
|---|---|---|---|
| 52 | 01010101 | 52 | 000000100100 |
| 53 | 00100100 | 53 | 000000110111 |
| 54 | 00100101 | 54 | 000000111000 |
| 55 | 01011000 | 55 | 000000100111 |
| 56 | 01011001 | 56 | 000000101000 |
| 57 | 01011010 | 57 | 000001011000 |
| 58 | 01011011 | 58 | 000001011001 |
| 59 | 01001010 | 59 | 000000101011 |
| 60 | 01001011 | 60 | 000000101100 |
| 61 | 00110010 | 61 | 000001011010 |
| 62 | 00110011 | 62 | 000001100110 |
| 63 | 00110100 | 63 | 000001100111 |

*Table 2/T.4 Make-up codes*

| White<br>run<br>length | Code<br>word | Black<br>run<br>length | Code<br>word |
|---|---|---|---|
| 64 | 11011 | 64 | 0000001111 |
| 128 | 10010 | 128 | 000011001000 |
| 192 | 010111 | 192 | 000011001001 |
| 256 | 0110111 | 256 | 000001011011 |
| 320 | 00110110 | 320 | 000000110011 |
| 384 | 00110111 | 384 | 000000110100 |
| 448 | 01100100 | 448 | 000000110101 |
| 512 | 01100101 | 512 | 0000001101100 |
| 576 | 01101000 | 576 | 0000001101101 |
| 640 | 01100111 | 640 | 0000001001010 |
| 704 | 011001100 | 704 | 0000001001011 |
| 768 | 011001101 | 768 | 0000001001100 |
| 832 | 011010010 | 832 | 0000001001101 |
| 896 | 011010011 | 896 | 0000001110010 |
| 960 | 011010100 | 960 | 0000001110011 |
| 1024 | 011010101 | 1024 | 0000001110100 |
| 1088 | 011010110 | 1088 | 0000001110101 |
| 1152 | 011010111 | 1152 | 0000001110110 |
| 1216 | 011011000 | 1216 | 0000001110111 |
| 1280 | 011011001 | 1280 | 0000001010010 |
| 1344 | 011011010 | 1344 | 0000001010011 |
| 1408 | 011011011 | 1408 | 0000001010100 |
| 1472 | 010011000 | 1472 | 0000001010101 |
| 1536 | 010011001 | 1536 | 0000001011010 |
| 1600 | 010011010 | 1600 | 0000001011011 |
| 1664 | 011000 | 1664 | 0000001100100 |
| 1728 | 010011011 | 1728 | 0000001100101 |
| EOL | 000000000001 | EOL | 00000000000 |

APPENDIX HH

### *Additional make-up codes*

| White and Black run length | Make-up code word |
|---|---|
| 1792 | 00000001000 |
| 1856 | 00000001100 |
| 1920 | 00000001101 |
| 1984 | 000000010010 |
| 2048 | 000000010011 |
| 2112 | 000000010100 |
| 2176 | 000000010101 |
| 2240 | 000000010110 |
| 2304 | 000000010111 |
| 2368 | 000000011100 |
| 2432 | 000000011101 |
| 2496 | 000000011110 |
| 2560 | 000000011111 |

APPENDIX HH

# Part 2: TIFF Extensions

Part 2 contains extensions to Baseline TIFF. TIFF Extensions are TIFF features that may not be supported by all TIFF readers. TIFF creators who use these features will have to work closely with TIFF readers in their part of the industry to ensure successful interchange.

The features described in this part were either contained in earlier versions of the specification, or have been approved by the TIFF Advisory Committee.

APPENDIX HH

# Section 11: CCITT Bilevel Encodings

The following fields are used when storing binary pixel arrays using one of the encodings adopted for raster-graphic interchange in numerous CCITT and ISO (International Organization for Standards) recommendations and standards. These encodings are often spoken of as "Group III compression" and "Group IV compression" because their application in facsimile transmission is the most widely known.

For the specialized use of these encodings in storing facsimile-transmission images, further guidelines can be obtained from the TIFF Class F document, available on-line in the same locations as this specification. This document is administered by another organization; paper copies are not available from Adobe.

## Compression

Tag = 259 (103.H)

Type = SHORT

N = 1

3 = T4-encoding: CCITT T.4 bi-level encoding as specified in section 4, Coding, of CCITT Recommendation T.4: "Standardization of Group 3 Facsimile apparatus for document transmission." International Telephone and Telegraph Consultative Committee (CCITT, Geneva: 1988).

See the T4Options field for T4-encoding options such as 1D vs 2D coding.

4 = T6-encoding: CCITT T.6 bi-level encoding as specified in section 2 of CCITT Recommendation T.6: "Facsimile coding schemes and coding control functions for Group 4 facsimile apparatus." International Telephone and Telegraph Consultative Committee (CCITT, Geneva: 1988).

See the T6Options field for T6-encoding options such as escape into uncompressed mode to avoid negative-compression cases.

## Application in Image Interchange

CCITT Recommendations T.4 and T.6 are specified in terms of the serial bit-by-bit creation and processing of a variable-length binary string that encodes bi-level (black and white) pixels of a rectangular image array. Generally, the encoding schemes are described in terms of bit-serial communication procedures and the end-to-end coordination that is required to gain reliable delivery over inherently unreliable data links. The Group 4 procedures, with their T6-encoding, represent a significant simplification because it is assumed that a reliable communication medium is employed, whether ISDN or X.25 or some other trustworthy transport vehicle. Because image-storage systems and computers achieve data integrity and communication reliability in other ways, the T6-encoding tends to be prefered for imaging applications. When computer storage and retrieval and interchange of facsimile material are of interest, the T4-encodings provide a better match to the current generation of Group 3 facsimile products and their defenses against data corruption as the result of transmission defects.

Whichever form of encoding is preferable for a given application, there are a number of adjustments that need to be made to account for the capture of the CCITT binary-encoding strings as part of electronically-stored material and digital-image interchange.

*PhotometricInterpretation.* An encoded CCITT string is self-photometric, defined in terms of white and black runs. Yet TIFF defines a tag called PhotometricInterpretation that also purports to define what is white and what is black. Somewhat arbitrarily, we adopt the following convention:

The "normal" PhotometricInterpretation for bilevel CCITT compressed data is WhiteIsZero. In this case, the CCITT "white" runs are to be interpretated as white, and the CCITT "black" runs are to be interpreted as black. However, if the PhotometricInterpretation is BlackIsZero, the TIFF reader must reverse the meaning of white and black when displaying and printing the image.

*FillOrder.* When CCITT encodings are used directly over a typical serial communication link, the order of the bits in the encoded string is the sequential order of the string, bit-by-bit, from beginning to end. This poses the following question: In which order should consecutive blocks of eight bits be assembled into octets (standard data bytes) for use within a computer system? The answer differs depending on whether we are concerned about preserving the serial-transmission sequence or preserving only the format of byte-organized sequences in memory and in stored files.

From the perspective of electronic interchange, as long as a receiver's reassembly of bits into bytes properly mirrors the way in which the bytes were disassembled by the transmitter, no one cares which order is seen on the transmission link because each multiple of 8 bits is transparently transmitted.

Common practice is to record arbitrary binary strings into storage sequences such that the first sequential bit of the string is found in the high-order bit of the first octet of the stored byte sequence. This is the standard case specified by TIFF FillOrder = 1, used in most bitmap interchange and the only case required in Baseline TIFF. This is also the approach used for the octets of standard 8-bit character data, with little attention paid to the fact that the most common forms of data communication transmit and reassemble individual 8-bit frames with the low-order-bit first!

For bit-serial transmission to a distant unit whose approach to assembling bits into bytes is unknown and supposed to be irrelevant, it is necessary to satisfy the expected sequencing of bits over the transmission link. This is the normal case for communication between facsimile units and also for computers and modems emulating standard Group 3 facsimile units. In this case, if the CCITT encoding is captured directly off of the link via standard communication adapters, TIFF FillOrder = 2 will usually apply to that stored data form.

Consequently, different TIFF FillOrder cases may arise when CCITT encodings are obtained by synthesis within a computer (including Group 4 transmission, which is treated more like computer data) instead of by capture from a Group 3 facsimile unit.

Because this is such a subtle situation, with surprisingly disruptive consequences for FillOrder mismatches, the following practice is urged whenever CCITT bilevel encodings are used:

a. TIFF FillOrder (tag 266) should always be explicitly specified.

b. FillOrder = 1 should be employed wherever possible in persistent material that is intended for interchange. This is the only reliable case for widespread interchange among computer systems, and it is important to explicitly confirm the honoring of standard assumptions.

c. FillOrder = 2 should occur only in highly-localized and preferably-transient material, as in a facsimile server supporting group 3 facsimile equipment. The tag should be present as a safeguard against the CCITT encoding "leaking" into an unsuspecting application, allowing readers to detect and warn against the occurence.

There are interchange situations where fill order is not distinguished, as when filtering the CCITT encoding into a PostScript level 2 image operation. In this case, as in most other cases of computer-based information interchange, FillOrder=1 is assumed, and any padding to a multiple of 8 bits is accomplished by adding a sufficient number of 0-bits to the end of the sequence.

*Strips and Tiles.* When CCITT bi-level encoding is employed, interaction with stripping (Section 3) and tiling (Section 15) is as follows:

a. Decompose the image into segments—individual pixel arrays representing the desired strip or tile configuration. The CCITT encoding procedures are applied most flexibly if the segments each have a multiple of 4 lines.

b. Individually encode each segment according to the specified CCITT bi-level encoding, as if each segment is a separate raster-graphic image.

The reason for this general rule is that CCITT bi-level encodings are generally progressive. That is, the initial line of pixels is encoded, and then subsequent lines, according to a variety of options, are encoded in terms of changes that need to be made to the preceding (unencoded) line. For strips and tiles to be individually usable, they must each start as fresh, independent encodings.

*Miscellaneous features.* There are provisions in CCITT encoding that are mostly meaningful during facsimile-transmission procedures. There is generally no significant application when storing images in TIFF or other data interchange formats, although TIFF applications should be tolerant and flexible in this regard. These features tend to have significance only when facilitating transfer between facsimile and non-facsimile applications of the encoded raster-graphic images. Further considerations for fill sequences, end-of-line flags, return-to-control (end-of-block) sequences and byte padding are introduced in discussion of the individual encoding options.

## T4Options

Tag = 292 (124.H)

Type = LONG

N = 1

*See Compression=3.* This field is made up of a set of 32 flag bits. Unused bits must be set to 0. Bit 0 is the low-order bit.

Bit 0 is 1 for 2-dimensional coding (otherwise 1-dimensional is assumed). For 2-D coding, if more than one strip is specified, each strip must begin with a 1-

dimensionally coded line. That is, RowsPerStrip should be a multiple of "Parameter K," as documented in the CCITT specification.

Bit 1 is 1 if uncompressed mode is used.

Bit 2 is 1 if fill bits have been added as necessary before EOL codes such that EOL always ends on a byte boundary, thus ensuring an EOL-sequence of 1 byte preceded by a zero nibble: xxxx-0000 0000-0001.

Default is 0, for basic 1-dimensional coding. See also Compression.

## *T6Options*

Tag = 293 (125.H)

Type = LONG

N = 1

See *Compression = 4.* This field is made up of a set of 32 flag bits. Unused bits must be set to 0. Bit 0 is the low-order bit. The default value is 0 (all bits 0).

bit 0 is unused and always 0.

bit 1 is 1 if uncompressed mode is allowed in the encoding.

In earlier versions of TIFF, this tag was named Group4Options. The significance has not changed and the present definition is compatible. The name of the tag has been changed to be consistent with the nomenclature of other T.6-encoding applications.

Readers should honor this option tag, and only this option tag, whenever T.6-Encoding is specified for Compression.

For T.6-Encoding, each segment (strip or tile) is encoded as if it were a separate image. The encoded string from each segment starts a fresh byte.

There are no one-dimensional line encodings in T.6-Encoding. Instead, even the first row of the segment's pixel array is encoded two-dimensionally by always assuming an invisible preceding row of all-white pixels. The 2-dimensional procedure for encoding the body of individual rows is the same as that used for 2-dimensional T.4-encoding and is described fully in the CCITT specifications.

The beginning of the encoding for each row of a strip or tile is conducted as if there is an imaginary preceding (0-width) white pixel, that is as if a fresh run of white pixels has just commenced. The completion of each line is encoded as if there are imaginary pixels beyond the end of the current line, and of the preceding line, in effect, of colors chosen such that the line is exactly completable by a code word, making the imaginary next pixel a changing element that's not actually used.

The encodings of successive lines follow contiguously in the binary T.6-Encoding stream with no special initiation or separation codewords. There are no provisions for fill codes or explicit end-of-line indicators. The encoding of the last line of the pixel array is followed immediately, in place of any additional line encodings, by a 24-bit End-of-Facsimile Block (EOFB).

000000000001000000000001.B.

The EOFB sequence is immediately followed by enough 0-bit padding to fit the entire stream into a sequence of 8-bit bytes.

*General Application.* Because of the single uniform encoding procedure, without disruptions by end-of-line codes and shifts into one-dimensional encodings, T.6-encoding is very popular for compression of bi-level images in document imaging systems. T.6-encoding trades off redundancy for minimum encoded size, relying on the underlying storage and transmission systems for reliable retention and communication of the encoded stream.

TIFF readers will operate most smoothly by always ignoring bits beyond the EOFB. Some writers may produce additional bytes of pad bits beyond the byte containing the final bit of the EOFB. Robust readers will not be disturbed by this prospect.

It is not possible to correctly decode a T.6-Encoding without knowledge of the exact number of pixels in each line of the pixel array. ImageWidth (or TileWidth, if used) must be stated exactly and accurately. If an image or segment is overscanned, producing extraneous pixels at the beginning or ending of lines, these pixels must be counted. Any cropping must be accomplished by other means. It is not possible to recover from a pixel-count deviation, even when one is detected. Failure of any row to be completed as expected is cause for abandoning further decoding of the entire segment. There is no requirement that ImageWidth be a multiple of eight, of course, and readers must be prepared to pad the final octet bytes of decoded bitmap rows with additional bits.

If a TIFF reader encounters EOFB before the expected number of lines has been extracted, it is appropriate to assume that the missing rows consist entirely of white pixels. Cautious readers might produce an unobtrusive warning if such an EOFB is followed by anything other than pad bits.

Readers that successfully decode the RowsPerStrip (or TileLength or residual ImageLength) number of lines are not required to verify that an EOFB follows. That is, it is generally appropriate to stop decoding when the expected lines are decoded or the EOFB is detected, whichever occurs first. Whether error indications or warnings are also appropriate depends upon the application and whether more precise troubleshooting of encoding deviations is important.

TIFF writers should always encode the full, prescribed number of rows, with a proper EOFB immediately following in the encoding. Padding should be by the least number of 0-bits needed for the T.6-encoding to exactly occupy a multiple of 8 bits. Only 0-bits should be used for padding, and StripByteCount (or TileByteCount) should not extend to any bytes not containing properly-formed T.6-encoding. In addition, even though not required by T.6-encoding rules, successful interchange with a large variety of readers and applications will be enhanced if writers can arrange for the number of pixels per line and the number of lines per strip to be multiples of eight.

*Uncompressed Mode.* Although T.6-encodings of simple bi-level images result in data compressions of 10:1 and better, some pixel-array patterns have T.6-encodings that require more bits than their simple bi-level bitmaps. When such cases are detected by encoding procedures, there is an optional extension for shifting to a form of uncompressed coding within the T.6-encoding string.

Uncompressed mode is not well-specified and many applications discourage its usage, prefering alternatives such as different compressions on a segment-by-segment (strip or tile) basis, or by simply leaving the image uncompressed in its

APPENDIX HH

entirety. The main complication for readers is in properly restoring T.6-encoding after the uncompressed sequence is laid down in the current row.

Readers that have no provision for uncompressed mode will generally reject any case in which the flag is set. Readers that are able to process uncompressed-mode content within T.6-encoding strings can safely ignore this flag and simply process any uncompressed-mode occurences correctly.

Writers that are unable to guarantee the absence of uncompressed-mode material in any of the T.6-encoded segments must set the flag. The flag should be cleared (or defaulted) only when absence of uncompressed-mode material is assured. Writers that are able to inhibit the generation of uncompressed-mode extensions are encouraged to do so in order to maximize the acceptability of their T.6-encoding strings in interchange situations.

Because uncompressed-mode is not commonly used, the following description is best taken as suggestive of the general machinery. Interpolation of fine details can easily vary between implementations.

Uncompressed mode is signalled by the occurence of the 10-bit extension code string

    0000001111.B

outside of any run-length make-up code or extension. Original unencoded image information follows. In this unencoded information, a 0-bit evidently signifies a white pixel, a 1-bit signifies a black pixel, and the TIFF PhotometricInterpretation will influence how these bits are mapped into any final uncompressed bitmap for use. The only modification made to the unencoded information is insertion of a 1-bit after every block of five consecutive 0-bits from the original image information. This is a transparency device that allows longer sequencences of 0-bits to be reserved for control conditions, especially ending the uncompressed-mode sequence. When it is time to return to compressed mode, the 8-bit exit sequence

    0000001t.B

is appended to the material. The 0-bits of the exit sequence are not considered in applying the 1-bit insertion rule; up to four information 0-bits can legally precede the exit sequence. The trailing bit, 't,' specifies the color (via 0 or 1) that is understood in the next run of compressed-mode encoding. This lets a color other than white be assumed for the 0-width pixel on the left of the edge between the last uncompressed pixel and the resumed 2-dimensional scan.

Writers should confine uncompressed-mode sequences to the interiors of individual rows, never attempting to "wrap" from one row to the next. Readers must operate properly when the only encoding for a single row consists of an uncompressed-mode escape, a complete row of (proper 1-inserted) uncompressed information, and the extension exit. Technically, the exit pixel, 't,' should probably then be the opposite color of the last true pixel of the row, but readers should be generous in this case.

In handling these complex encodings, the encounter of material from a defective source or a corrupted file is particularly unsettling and mysterious. Robust readers will do well to defend against falling off the end of the world; e.g., unexpected EOFB sequences should be handled, and attempted access to data bytes that are not within the bounds of the present segment (or the TIFF file itself) should be avoided.

Microsoft et al. Exhibit 1005

# Section 12: Document Storage and Retrieval

These fields may be useful for document storage and retrieval applications. They will very likely be ignored by other applications.

## DocumentName

The name of the document from which this image was scanned.

Tag = 269 (10D.H)

Type = ASCII

See also PageName.

## PageName

The name of the page from which this image was scanned.

Tag = 285 (11D.H)

Type = ASCII

See also DocumentName.

No default.

## PageNumber

The page number of the page from which this image was scanned.

Tag = 297 (129.H)

Type = SHORT

N = 2

This field is used to specify page numbers of a multiple page (e.g. facsimile) document. PageNumber[0] is the page number; PageNumber[1] is the total number of pages in the document. If PageNumber[1] is 0, the total number of pages in the document is not available.

Pages need not appear in numerical order.

The first page is numbered 0 (zero).

No default.

## XPosition

X position of the image.

Tag = 286 (11E.H)

Type = RATIONAL

N = 1

The X offset in ResolutionUnits of the left side of the image, with respect to the left side of the page.

No default. See also YPosition.

## YPosition

Y position of the image.

Tag = 287 (11F.H)

Type = RATIONAL

N = 1

The Y offset in ResolutionUnits of the top of the image, with respect to the top of the page. In the TIFF coordinate scheme, the positive Y direction is down, so that YPosition is always positive.

No default. See also XPosition.

APPENDIX HH

# Section 13: LZW Compression

This section describes TIFF compression scheme 5, an adaptive compression scheme for raster images.

## Restrictions

When LZW compression was added to the TIFF specification, in Revision 5.0, it was thought to be public domain. This is, apparently, not the case.

The following paragraph has been approved by the Unisys Corporation:

"The LZW compression method is said to be the subject of United States patent number 4,558,302 and corresponding foreign patents owned by the Unisys Corporation. Software and hardware developers may be required to license this patent in order to develop and market products using the TIFF LZW compression option. Unisys has agreed that developers may obtain such a license on reasonable, nondiscriminatory terms and conditions. Further information can be obtained from: Welch Licensing Department, Office of the General Counsel, M/S C1SW19, Unisys Corporation, Blue Bell, Pennsylvania, 19424."

Reportedly, there are also other companies with patents that may affect LZW implementors.

## *Reference*

Terry A. Welch, "A Technique for High Performance Data Compression", IEEE Computer, vol. 17 no. 6 (June 1984). Describes the basic Lempel-Ziv & Welch (LZW) algorithm in very general terms. The author's goal is to describe a hardware-based compressor that could be built into a disk controller or database engine and used on all types of data. There is no specific discussion of raster images. This section gives sufficient information so that the article is not required reading.

## *Characteristics*

LZW compression has the following characteristics:

- LZW works for images of various bit depths.
- LZW has a reasonable worst-case behavior.
- LZW handles a wide variety of repetitive patterns well.
- LZW is reasonably fast for both compression and decompression.
- LZW does not require floating point software or hardware.

APPENDIX HH

- LZW is lossless. All information is preserved. But if noise or information is removed from an image, perhaps by smoothing or zeroing some low-order bitplanes, LZW compresses images to a smaller size. Thus, 5-bit, 6-bit, or 7-bit data masquerading as 8-bit data compresses better than true 8-bit data. Smooth images also compress better than noisy images, and simple images compress better than complex images.
- LZW works quite well on bilevel images, too. On our test images, it almost always beat PackBits and generally tied CCITT 1D (Modified Huffman) compression. LZW also handles halftoned data better than most bilevel compression schemes.

## The Algorithm

Each strip is compressed independently. We strongly recommend that RowsPerStrip be chosen such that each strip contains about 8K bytes before compression. We want to keep the strips small enough so that the compressed and uncompressed versions of the strip can be kept entirely in memory, even on small machines, but are large enough to maintain nearly optimal compression ratios.

The LZW algorithm is based on a translation table, or string table, that maps strings of input characters into codes. The TIFF implementation uses variable-length codes, with a maximum code length of 12 bits. This string table is different for every strip and does not need to be reatained for the decompressor. The trick is to make the decompressor automatically build the same table as is built when the data is compressed. We use a C-like pseudocode to describe the coding scheme:

```
InitializeStringTable();
WriteCode(ClearCode);
Ω = the empty string;
for each character in the strip {
        K = GetNextCharacter();
        if Ω+K is in the string table {
                Ω = Ω+K; /* string concatenation */
        } else {
                WriteCode (CodeFromString(Ω));
                AddTableEntry(Ω+K);
                Ω = K;
        }
} /* end of for loop */
WriteCode (CodeFromString(Ω));
WriteCode (EndOfInformation);
```

That's it. The scheme is simple, although it is challenging to implement efficiently. But we need a few explanations before we go on to decompression.

The "characters" that make up the LZW strings are bytes containing TIFF uncompressed (Compression=1) image data, in our implementation. For example, if BitsPerSample is 4, each 8-bit LZW character will contain two 4-bit pixels. If BitsPerSample is 16, each 16-bit pixel will span two 8-bit LZW characters.

It is also possible to implement a version of LZW in which the LZW character depth equals BitsPerSample, as described in Draft 2 of Revision 5.0. But there is a major problem with this approach. If BitsPerSample is greater than 11, we can not use 12-bit-maximum codes and the resulting LZW table is unacceptably large. Fortunately, due to the adaptive nature of LZW, we do not pay a significant compression ratio penalty for combining several pixels into one byte before compressing. For example, our 4-bit sample images compressed about 3 percent worse, and our 1-bit images compressed about 5 percent better. And it is easier to write an LZW compressor that always uses the same character depth than it is to write one that handles varying depths.

We can now describe some of the routine and variable references in our pseudocode:

InitializeStringTable() initializes the string table to contain all possible single-character strings. There are 256 of them, numbered 0 through 255, since our characters are bytes.

WriteCode() writes a code to the output stream. The first code written is a ClearCode, which is defined to be code #256.

Ω is our "prefix string."

GetNextCharacter() retrieves the next character value from the input stream. This will be a number between 0 and 255 because our characters are bytes.

The "+" signs indicate string concatenation.

AddTableEntry() adds a table entry. (InitializeStringTable() has already put 256 entries in our table. Each entry consists of a single-character string, and its associated code value, which, in our application, is identical to the character itself. That is, the 0th entry in our table consists of the string <0>, with a corresponding code value of <0>, the 1st entry in the table consists of the string <1>, with a corresponding code value of <1> and the 255th entry in our table consists of the string <255>, with a corresponding code value of <255>.) So, the first entry that added to our string table will be at position 256, right? Well, not quite, because we reserve code #256 for a special "Clear" code. We also reserve code #257 for a special "EndOfInformation" code that we write out at the end of the strip. So the first multiple-character entry added to the string table will be at position 258.

For example, suppose we have input data that looks like this:

Pixel 0:<7>

Pixel 1:<7>

Pixel 2:<7>

Pixel 3:<8>

Pixel 4:<8>

Pixel 5:<7>

Pixel 6:<7>

Pixel 7:<6>

Pixel 8:<6>

First, we read Pixel 0 into K. ΩK is then simply <7>, because Ω is an empty string at this point. Is the string <7> already in the string table? Of course, because all single character strings were put in the table by InitializeStringTable(). So set Ω equal to <7>, and then go to the top of the loop.

Read Pixel 1 into K. Does ΩK (<7><7>) exist in the string table? No, so we write the code associated with Ω to output (write <7> to output) and add ΩK (<7><7>) to the table as entry 258. Store K (<7>) into Ω. Note that although we have added the string consisting of Pixel 0 and Pixel 1 to the table, we "re-use" Pixel 1 as the beginning of the next string.

Back at the top of the loop, we read Pixel 2 into K. Does ΩK (<7><7>) exist in the string table? Yes, the entry we just added, entry 258, contains exactly <7><7>. So we add K to the end of Ω so that Ω is now <7><7>.

Back at the top of the loop, we read Pixel 3 into K. Does ΩK (<7><7><8>) exist in the string table? No, so we write the code associated with Ω (<258>) to output and then add ΩK to the table as entry 259. Store K (<8>) into Ω.

Back at the top of the loop, we read Pixel 4 into K. Does ΩK (<8><8>) exist in the string table? No, so we write the code associated with Ω (<8>) to output and then add ΩK to the table as entry 260. Store K (<8>) into Ω.

Continuing, we get the following results:

| After reading: | We write to output: | And add table entry: |
|---|---|---|
| Pixel 0 | | |
| Pixel 1 | <7> | 258: <7><7> |
| Pixel 2 | | |
| Pixel 3 | <258> | 259: <7><7><8> |
| Pixel 4 | <8> | 260:<8><8> |
| Pixel 5 | <8> | 261: <8><7> |
| Pixel 6 | | |
| Pixel 7 | <258> | 262: <7><7><6> |
| Pixel 8 | <6> | 263: <6><6> |

WriteCode() also requires some explanation. In our example, the output code stream, <7><258><8><8><258><6> should be written using as few bits as possible. When we are just starting out, we can use 9-bit codes, since our new string table entries are greater than 255 but less than 512. *After adding table entry 511, switch to 10-bit codes (i.e., entry 512 should be a 10-bit code.) Likewise, switch to 11-bit codes after table entry 1023, and 12-bit codes after table entry 2047.* We will arbitrarily limit ourselves to 12-bit codes, so that our table can have at most 4096 entries. The table should not be any larger.

*Whenever you add a code to the output stream, it "counts" toward the decision about bumping the code bit length. This is important when writing the last code word before an EOI code or ClearCode, to avoid code length errors.*

What happens if we run out of room in our string table? This is where the ClearCode comes in. As soon as we use entry 4094, we write out a (12-bit) ClearCode. (If we wait any longer to write the ClearCode, the decompressor might try to interpret the ClearCode as a 13-bit code.) At this point, the compressor reinitializes the string table and then writes out 9-bit codes again.

Note that whenever you write a code and add a table entry, Ω is not left empty. It contains exactly one character. Be careful not to lose it when you write an end-of-table ClearCode. You can either write it out as a 12-bit code before writing the ClearCode, in which case you need to do it right after adding table entry 4093, or

APPENDIX HH

you can write it as a 9-bit code after the ClearCode . Decompression gives the same result in either case.

To make things a little simpler for the decompressor, we will require that each strip begins with a ClearCode and ends with an EndOfInformation code. Every LZW-compressed strip must begin on a byte boundary. It need not begin on a word boundary. LZW compression codes are stored into bytes in high-to-low-order fashion, i.e., FillOrder is assumed to be 1. The compressed codes are written as bytes (not words) so that the compressed data will be identical whether it is an 'II' or 'MM' file.

Note that the LZW string table is a continuously updated history of the strings that have been encountered in the data. Thus, it reflects the characteristics of the data, providing a high degree of adaptability.

## LZW Decoding

The procedure for decompression is a little more complicated:

```
    while ((Code = GetNextCode()) != EoiCode) {
            if (Code == ClearCode) {
                    InitializeTable();
                    Code = GetNextCode();
                    if (Code == EoiCode)
                            break;
                    WriteString(StringFromCode(Code));
                    OldCode = Code;
            } /* end of ClearCode case */
            else {
                    if (IsInTable(Code)) {
                            WriteString(StringFromCode(Code));
                            AddStringToTable(StringFromCode(OldCode
)+FirstChar(StringFromCode(Code)));
                            OldCode = Code;
                    } else {
                            OutString = StringFromCode(OldCode) +
FirstChar(StringFromCode(OldCode));
                            WriteString(OutString);
                            AddStringToTable(OutString);
                            OldCode = Code;
                    }
            } /* end of not-ClearCode case */
    } /* end of while loop */
```

The function GetNextCode() retrieves the next code from the LZW-coded data. It must keep track of bit boundaries. It knows that the first code that it gets will be a 9-bit code. We add a table entry each time we get a code. So, GetNextCode() must switch over to 10-bit codes as soon as string *#510* is stored into the table. *Similarly, the switch is made to 11-bit codes after #1022 and to 12-bit codes after #2046.*

Microsoft et al. Exhibit 1005

The function StringFromCode() gets the string associated with a particular code from the string table.

The function AddStringToTable() adds a string to the string table. The "+" sign joining the two parts of the argument to AddStringToTable indicates string concatenation.

StringFromCode() looks up the string associated with a given code.

WriteString() adds a string to the output stream.

## When SamplesPerPixel Is Greater Than 1

So far, we have described the compression scheme as if SamplesPerPixel were always 1, as is the case with palette-color and grayscale images. But what do we do with RGB image data?

Tests on our sample images indicate that the LZW compression ratio is nearly identical whether PlanarConfiguration=1 or PlanarConfiguration=2, for RGB images. So, use whichever configuration you prefer and simply compress the bytes in the strip.

Note: Compression ratios on our test RGB images were disappointingly low: between 1.1 to 1 and 1.5 to 1, depending on the image. Vendors are urged to do what they can to remove as much noise as possible from their images. Preliminary tests indicate that significantly better compression ratios are possible with less-noisy images. Even something as simple as zeroing-out one or two least-significant bitplanes can be effective, producing little or no perceptible image degradation.

## Implementation

The exact structure of the string table and the method used to determine if a string is already in the table are probably the most significant design decisions in the implementation of a LZW compressor and decompressor. Hashing has been suggested as a useful technique for the compressor. We have chosen a tree-based approach, with good results. The decompressor is more straightforward and faster because no search is involved—strings can be accessed directly by code value.

## LZW Extensions

Some images compress better using LZW coding if they are first subjected to a process wherein each pixel value is replaced by the difference between the pixel and the preceding pixel. See the following Section.

APPENDIX HH

## Acknowledgments

See the first page of this section for the LZW reference.

The use of ClearCode as a technique for handling overflow was borrowed from the compression scheme used by the Graphics Interchange Format (GIF), a small-color-paint-image-file format used by CompuServe that also uses an adaptation of the LZW technique.

APPENDIX HH

# Section 14: Differencing Predictor

This section defines a Predictor that greatly improves compression ratios for some images.

### Predictor

Tag = 317 (13D.H)

Type = SHORT

N = 1

A predictor is a mathematical operator that is applied to the image data before an encoding scheme is applied. Currently this field is used only with LZW (Compression=5) encoding because LZW is probably the only TIFF encoding scheme that benefits significantly from a predictor step. See Section 13.

The possible values are:

1 = No prediction scheme used before coding.

2 = Horizontal differencing.

Default is 1.

## The algorithm

Make use of the fact that many continuous-tone images rarely vary much in pixel value from one pixel to the next. In such images, if we replace the pixel values by differences between consecutive pixels, many of the differences should be 0, plus or minus 1, and so on. This reduces the apparent information content and allows LZW to encode the data more compactly.

Assuming 8-bit grayscale pixels for the moment, a basic C implementation might look something like this:

```
char      image[ ][ ];
int       row, col;

/* take horizontal differences:
 */
for (row = 0; row < nrows; row++)
          for (col = ncols - 1; col >= 1; col--)
                    image[row][col] -= image[row][col-1];
```

If we don't have 8-bit components, we need to work a little harder to make better use of the architecture of most CPUs. Suppose we have 4-bit components packed two per byte in the normal TIFF uncompressed (i.e., Compression=1) fashion. To find differences, we want to first expand each 4-bit component into an 8-bit byte, so that we have one component per byte, low-order justified. We then perform the horizontal differencing illustrated in the example above. Once the differencing has been completed, we then repack the 4-bit differences two to a byte, in the normal TIFF uncompressed fashion.

If the components are greater than 8 bits deep, expanding the components into 16-bit words instead of 8-bit bytes seems like the best way to perform the subtraction on most computers.

Note that we have not lost any data up to this point, nor will we lose any data later on. It might seem at first that our differencing might turn 8-bit components into 9-bit differences, 4-bit components into 5-bit differences, and so on. But it turns out that we can completely ignore the "overflow" bits caused by subtracting a larger number from a smaller number and still reverse the process without error. Normal two's complement arithmetic does just what we want. Try an example by hand if you need more convincing.

Up to this point we have implicitly assumed that we are compressing bilevel or grayscale images. An additional consideration arises in the case of color images.

If PlanarConfiguration is 2, there is no problem. Differencing works the same as it does for grayscale data.

If PlanarConfiguration is 1, however, things get a little trickier. If we didn't do anything special, we would subtract red component values from green component values, green component values from blue component values, and blue component values from red component values. This would not give the LZW coding stage much redundancy to work with. So, we will do our horizontal differences with an offset of SamplesPerPixel (3, in the RGB case). In other words, we will subtract red from red, green from green, and blue from blue. The LZW coding stage is identical to the SamplesPerPixel=1 case. We require that BitsPerSample be the same for all 3 components.

## Results and Guidelines

LZW without differencing works well for 1-bit images, 4-bit grayscale images, and many palette-color images. But natural 24-bit color images and some 8-bit grayscale images do much better with differencing.

Although the combination of LZW coding with horizontal differencing does not result in any loss of data, it may be worthwhile in some situations to give up some information by removing as much noise as possible from the image data before doing the differencing, especially with 8-bit components. The simplest way to get rid of noise is to mask off one or two low-order bits of each 8-bit component. On our 24-bit test images, LZW with horizontal differencing yielded an average compression ratio of 1.4 to 1. When the low-order bit was masked from each component, the compression ratio climbed to 1.8 to 1; the compression ratio was 2.4 to 1 when masking two bits, and 3.4 to 1 when masking three bits. Of course, the more you mask, the more you risk losing useful information along with the noise. We encourage you to experiment to find the best compromise for your device. For some applications, it may be useful to let the user make the final decision.

Incidentally, we tried taking both horizontal and vertical differences, but the extra complexity of two-dimensional differencing did not appear to pay off for most of our test images. About one third of the images compressed slightly better with two-dimensional differencing, about one third compressed slightly worse, and the rest were about the same.

# Section 15: Tiled Images

## Introduction

### Motivation

This section describes how to organize images into tiles instead of strips.

For low-resolution to medium-resolution images, the standard TIFF method of breaking the image into strips is adequate. However high-resolution images can be accessed more efficiently—and compression tends to work better—if the image is broken into roughly square tiles instead of horizontally-wide but vertically-narrow strips.

### Relationship to existing fields

**When the tiling fields described below are used, they replace the StripOffsets, StripByteCounts, and RowsPerStrip fields.** Use of tiles will therefore cause older TIFF readers to give up because they will have no way of knowing where the image data is or how it is organized. **Do not** use both strip-oriented and tile-oriented fields in the same TIFF file.

### Padding

Tile size is defined by TileWidth and TileLength. All tiles in an image are the same size; that is, they have the same pixel dimensions.

Boundary tiles are padded to the tile boundaries. For example, if TileWidth is 64 and ImageWidth is 129, then the image is 3 tiles wide and 63 pixels of padding must be added to fill the rightmost column of tiles. The same holds for TileLength and ImageLength. It doesn't matter what value is used for padding, because good TIFF readers display only the pixels defined by ImageWidth and ImageLength and ignore any padded pixels. Some compression schemes work best if the padding is accomplished by replicating the last column and last row instead of padding with 0's.

The price for padding the image out to tile boundaries is that some space is wasted. But compression usually shrinks the padded areas to almost nothing. Even if data is not compressed, remember that tiling is intended for large images. Large images have lots of comparatively small tiles, so that the percentage of wasted space will be very small, generally on the order of a few percent or less.

The advantages of padding an image to the tile boundaries are that implementations can be simpler and faster and that it is more compatible with tile-oriented compression schemes such as JPEG. See Section 22.

Tiles are compressed individually, just as strips are compressed. *That is, each row of data in a tile is treated as a separate "scanline" when compressing.* Compres-

APPENDIX HH

sion includes any padded areas of the rightmost and bottom tiles so that all the tiles in an image are the same size when uncompressed.

All of the following fields are required for tiled images:

## Fields

### TileWidth

Tag = 322 (142.H)

Type = SHORT or LONG

N = 1

The tile width in pixels. This is the number of columns in each tile.

Assuming integer arithmetic, three computed values that are useful in the following field descriptions are:

TilesAcross = (ImageWidth + TileWidth - 1) / TileWidth

TilesDown = (ImageLength + TileLength - 1) / TileLength

TilesPerImage = TilesAcross * TilesDown

These computed values are not TIFF fields; they are simply values determined by the ImageWidth, TileWidth, ImageLength, and TileLength fields.

TileWidth and ImageWidth together determine the number of tiles that span the width of the image (TilesAcross). TileLength and ImageLength together determine the number of tiles that span the length of the image (TilesDown).

We recommend choosing TileWidth and TileLength such that the resulting tiles are about 4K to 32K bytes before compression. This seems to be a reasonable value for most applications and compression schemes.

TileWidth must be a multiple of 16. This restriction improves performance in some graphics environments and enhances compatibility with compression schemes such as JPEG.

Tiles need not be square.

Note that ImageWidth can be less than TileWidth, although this means that the tiles are too large or that you are using tiling on really small images, neither of which is recommended. The same observation holds for ImageLength and TileLength.

No default. See also TileLength, TileOffsets, TileByteCounts.

### TileLength

Tag = 323 (143.H)

Type = SHORT or LONG

N = 1

The tile length (height) in pixels. This is the number of rows in each tile.

TileLength must be a multiple of 16 for compatibility with compression schemes such as JPEG.

Replaces RowsPerStrip in tiled TIFF files.

No default. See also TileWidth, TileOffsets, TileByteCounts.

## TileOffsets

Tag = 324 (144.H)

Type = LONG

N = TilesPerImage for PlanarConfiguration = 1

= SamplesPerPixel * TilesPerImage for PlanarConfiguration = 2

For each tile, the byte offset of that tile, as compressed and stored on disk. The offset is specified with respect to the beginning of the TIFF file. Note that this implies that each tile has a location independent of the locations of other tiles.

Offsets are ordered left-to-right and top-to-bottom. For PlanarConfiguration = 2, the offsets for the first component plane are stored first, followed by all the offsets for the second component plane, and so on.

No default. See also TileWidth, TileLength, TileByteCounts.

## TileByteCounts

Tag = 325 (145.H)

Type = SHORT or LONG

N = TilesPerImage for PlanarConfiguration = 1

= SamplesPerPixel * TilesPerImage for PlanarConfiguration = 2

For each tile, the number of (compressed) bytes in that tile.

See TileOffsets for a description of how the byte counts are ordered.

No default. See also TileWidth, TileLength, TileOffsets.

# Section 16: CMYK Images

## Motivation

This section describes how to store separated (usually CMYK) image data in a TIFF file.

In a separated image, each pixel consists of N components. Each component represents the amount of a particular ink that is to be used to represent the image at that location, typically using a halftoning technique.

For example, in a CMYK image, each pixel consists of 4 components. Each component represents the amount of cyan, magenta, yellow, or black process ink that is to be used to represent the image at that location.

The fields described in this section can be used for more than simple 4-color process (CMYK) printing. They can also be used for describing an image made up of more than 4 inks, such an image made up of a cyan, magenta, yellow, red, green, blue, and black inks. Such an image is sometimes called a high-fidelity image and has the advantage of slightly extending the printed color gamut.

Since separated images are quite device-specific and are restricted to color prepress use, they should not be used for general image data interchange. Separated images are to be used only for prepress applications in which the imagesetter, paper, ink, and printing press characteristics are known by the creator of the separated image.

Note: there is no single method of converting RGB data to CMYK data and back. In a perfect world, something close to cyan = 255-red, magenta = 255-green, and yellow = 255-blue should work; but characteristics of printing inks and printing presses, economics, and the fact that the meaning of RGB itself depends on other parameters combine to spoil this simplicity.

## Requirements

In addition to satisfying the normal Baseline TIFF requirements, a separated TIFF file must have the following characteristics:

- **SamplesPerPixel = N.** SHORT. The number of inks. (For example, N=4 for CMYK, because we have one component each for cyan, magenta, yellow, and black.)
- **BitsPerSample = 8,8,8,8 (for CMYK).** SHORT. For now, only 8-bit components are recommended. The value "8" is repeated SamplesPerPixel times.
- **PhotometricInterpretation = 5 (Separated - usually CMYK).** SHORT. The components represent the desired percent dot coverage of each ink, where the larger component values represent a higher percentage of ink dot coverage and smaller values represent less coverage.

## Fields

In addition, there are some new fields, all of which are optional.

### InkSet

Tag = 332 (14C.H)

Type = SHORT

N = 1

The set of inks used in a separated (PhotometricInterpretation=5) image.

1 = CMYK. The order of the components is cyan, magenta, yellow, black. Usually, a value of 0 represents 0% ink coverage and a value of 255 represents 100% ink coverage for that component, but see DotRange below. The InkNames field should not exist when InkSet=1.

2 = not CMYK. See the InkNames field for a description of the inks to be used.

Default is 1 (CMYK).

### NumberOfInks

Tag = 334 (14E.H)

Type = SHORT

N = 1

The number of inks. Usually equal to SamplesPerPixel, unless there are extra samples.

See also ExtraSamples.

Default is 4.

### InkNames

Tag = 333 (14D.H)

Type = ASCII

N = total number of characters in all the ink name strings, including the NULs.

The name of each ink used in a separated (PhotometricInterpretation=5) image, written as a list of concatenated, NUL-terminated ASCII strings. The number of strings must be equal to NumberOfInks.

The samples are in the same order as the ink names.

See also InkSet, NumberOfInks.

No default.

### DotRange

Tag = 336 (150.H)

Type = BYTE or SHORT

N = 2, or 2*SamplesPerPixel

The component values that correspond to a 0% dot and 100% dot. DotRange[0] corresponds to a 0% dot, and DotRange[1] corresponds to a 100% dot.

If a DotRange pair is included for each component, the values for each component are stored together, so that the pair for Cyan would be first, followed by the pair for Magenta, and so on. *Use of multiple dot ranges is, however, strongly discouraged in the interests of simplicity and compatibility with ANSI IT8 standards.*

A number of prepress systems like to keep some "headroom" and "footroom" on both ends of the range. What to do with components that are less than the 0% aim point or greater than the 100% aim point is not specified and is application-dependent.

It is strongly recommended that a CMYK TIFF writer not attempt to use this field to reverse the sense of the pixel values so that smaller values mean more ink instead of less ink. That is, DotRange[0] should be less than DotRange[1].

DotRange[0] and DotRange[1] must be within the range [0, (2**BitsPerSample) - 1].

Default: a component value of 0 corresponds to a 0% dot, and a component value of 255 (assuming 8-bit pixels) corresponds to a 100% dot. That is, DotRange[0] = 0 and DotRange[1] = (2**BitsPerSample) - 1.

### TargetPrinter

Tag = 337 (151.H)

Type = ASCII

N = any

A description of the printing environment for which this separation is intended.

## History

This Section has been expanded from earlier drafts, with the addition of the **InkSet, InkNames, NumberOfInks, DotRange**, and **TargetPrinter**, but is backward-compatible with earlier draft versions.

Possible future enhancements: definition of the characterization information so that the CMYK data can be retargeted to a different printing environment and so that display on a CRT or proofing device can more accurately represent the color. ANSI IT8 is working on such a proposal.

APPENDIX HH

# Section 17: HalftoneHints

This section describes a scheme for properly placing highlights and shadows in halftoned images.

## Introduction

The single most easily recognized failing of continuous tone images is the incorrect placement of highlight and shadow. It is critical that a halftone process be capable of printing the lightest areas of the image as the smallest halftone spot capable of the output device, at the specified printer resolution and screen ruling. Specular highlights (small ultra-white areas) as well as the shadow areas should be printable as paper only.

Consistency in highlight and shadow placement allows the user to obtain predictable results on a wide variety of halftone output devices. Proper implementation of theHalftoneHints field will provide a significant step toward device independent imaging, such that low cost printers may to be used as effective proofing devices for images which will later be halftoned on a high-resolution imagesetter.

## The HalftoneHints Field

### HalftoneHints

Tag = 321 (141.H)

Type = SHORT

N = 2

The purpose of the HalftoneHints field is to convey to the halftone function the range of gray levels within a colorimetrically-specified image that should retain tonal detail. The field contains two values of sixteen bits each and, therefore, is contained wholly within the field itself; no offset is required. The first word specifies the highlight gray level which should be halftoned at the lightest printable tint of the final output device. The second word specifies the shadow gray level which should be halftoned at the darkest printable tint of the final output device. Portions of the image which are whiter than the highlight gray level will quickly, if not immediately, fade to specular highlights. There is no default value specified, since the highlight and shadow gray levels are a function of the subject matter of a particular image.

Appropriate values may be derived algorithmically or may be specified by the user, either directly or indirectly.

The HalftoneHints field, as defined here, defines an achromatic function. It can be used just as effectively with color images as with monochrome images. When used with opponent color spaces such as CIE L*a*b* or YCbCr, it refers to the achromatic component only; L* in the case of CIELab, and Y in the case of

YCbCr. When used with tri-stimulus spaces such as RGB, it suggests to retain tonal detail for all colors with an NTSC gray component within the bounds of the R=G=B=Highlight to R=G=B=Shadow range.

## Comments for TIFF Writers

TIFF writers are encouraged to include the HalftoneHints field in all color or grayscale images where BitsPerSample >1. Although no default value is specified, prior to the introduction of this field it has been common practice to implicitly specify the highlight and shadow gray levels as 1 and 2**BitsperSample-2 and manipulate the image data to this definition. There are some disadvantages to this technique, and it is not feasible for a fixed gamut colorimetric image type. Appropriate values may be derived algorithmically or may be specified by the user directly or indirectly. Automatic algorithms exist for analyzing the histogram of the achromatic intensity of an image and defining the minimum and maximum values as the highlight and shadow settings such that tonal detail is retained throughout the image. This kind of algorithm may try to impose a highlight or shadow where none really exists in the image, which may require user controls to override the automatic setting.

It should be noted that the choice of the highlight and shadow values is somewhat output dependent. For instance, in situations where the dynamic range of the output medium is very limited (as in newsprint and, to a lesser degree, laser output), it may be desirable for the user to clip some of the lightest or darkest tones to avoid the reduced contrast resulting from compressing the tone of the entire image. Different settings might be chosen for 150-line halftone printed on coated stock. Keep in mind that these values may be adjusted later (which might not be possible unless the image is stored as a colorimetric, fixed, full-gamut image), and that more sophisticated page-layout applications may be capable of presenting a user interface to consider these decisions at a point where the halftone process is well understood.

It should be noted that although CCDs are linear intensity detectors, TIFF writers may choose to manipulate the image to store gamma-compensated data. Gamma-compensated data is more efficient at encoding an image than is linear intensity data because it requires fewer BitsPerPixel to eliminate banding in the darker tones. It also has the advantage of being closer to the tone response of the display or printer and is, therefore, less likely to produce poor results from applications that are not rigorous about their treatment of images. Be aware that the PhotometricInterpretation value of 0 or 1 (grayscale) implies linear data because no gamma is specified. The PhotometricInterpretation value of 2 (RGB data) specifies the NTSC gamma of 2.2 as a default. If data is written as something other than the default, then a GrayResponseCurve field or a TransferFunction field must be present to define the deviation. For grayscale data, be sure that the densities in the GrayResponseCurve are consistent with the PhotometricInterpretation field and the HalftoneHints field.

# Comments for TIFF Readers

TIFF readers that send a grayscale image to a halftone output device, whether it is a binary laser printer or a PostScript imagesetter should make an effort to maintain the highlight and shadow placement. This requires two steps. First, determine the highlight and shadow gray level of a particular image. Second, communicate that information to the halftone engine.

To determine the highlight and shadow gray levels, begin by looking for a HalftoneHints field. If it exists, it takes precedence. The first word represents the gray level of the highlight and the second word represents the gray level of the shadow. If the image is a colorimetric image (i.e. it has a GrayResponseCurve field or a TransferFunction field) but does not contain a HalftoneHints field, then the gamut mapping techniques described earlier should be used to determine the highlight and shadow values. If neither of these conditions are true, then the file should be treated as if a HalftoneHints field had indicated a highlight at gray level 1 and a shadow at gray level 2**BitsPerPixel-2 (or vice-versa depending on the PhotometricInterpretation field). Once the highlight and shadow gray levels have been determined, the next step is to communicate this information to the halftone module. The halftone module may exist within the same application as the TIFF reader, it may exist within a separate printer driver, or it may exist within the Raster Image Processor (RIP) of the printer itself. Whether the halftone process is a simple dither pattern or a general purpose spot function, it has some gray level at which the lightest printable tint will be rendered. The HalftoneHint concept is best implemented in an environment where this lightest printable tint is easily and consistently specified.

There are several ways in which an application can communicate the highlight and shadow to the halftone function. Some environments may allow the application to pass the highlight and shadow to the halftone module explicitly along with the image. This is the best approach, but many environments do not yet provide this capability. Other environments may provide fixed gray levels at which the highlight and shadow will be rendered. For these cases, the application should build a tone map that matches the highlight and shadow specified in the image to the highlight and shadow gray level of the halftone module. This approach requires more work by the application software, but will provide excellent results. Some environments will not have any consistent concept of highlight and shadow at all. In these environments, the best an application can do is characterize each of the supported printers and save the observed highlight and shadow gray levels. The application can then use these values to achieve the desired results, providing the environment doesn't change.

Once the highlight and shadow areas are selected, care should be taken to appropriately map intermediate gray levels to those expected by the halftone engine, which may or may not be linear Reflectance. Note that although CCDs are linear intensity detectors and many TIFF files are stored as linear intensity, most output devices require significant tone compensation (sometimes called gamma correction) to correctly display or print linear data. Be aware that the PhotometricInterpretation value of 0, 1 implies linear data because no gamma is specified. The PhotometricInterpretation value of 2 (RGB data) specifies the NTSC gamma of 2.2 as a default. If a GrayResponseCurve field or a TransferFunction field is present, it may define something other than the default.

# Some Background on the Halftone Process

To obtain the best results when printing a continuous-tone raster image, it is seldom desirable to simply reproduce the tones of the original on the printed page. Most often there is some gamut mapping required. Often this is because the tonal range of the original extends beyond the tonal range of the output medium. In some cases, the tone range of the original is within the gamut of the output medium, but it may be more pleasing to expand the tone of the image to fill the range of the output. Given that the tone of the original is to be adjusted, there is a whole range of possibilities for the level of sophistication that may be undertaken by a software application.

Printing monochrome output is far less sophisticated than printing color output. For monochrome output the first priority is to control the placement of the highlight and the shadow. Ideally, a quality halftone will have sufficient levels of gray so that a standard observer cannot distinguish the interface between any two adjacent levels of gray. In practice, however, there is often a significant step between the tone of the paper and the tone of the lightest printable tint. Although usually less severe, the problem is similar between solid ink and the darkest printable tint. Since the dynamic range between the lightest printable tint and the darkest printable tint is usually less than one would like, it is common to maximize the tone of the image within these bounds. Not all images will have a highlight (an area of the image which is desirable to print as light as possible while still retaining tonal detail). If one exists, it should be carefully controlled to print at the lightest printable tint of the output medium. Similarly, the darkest areas of the image to retain tonal detail should be printed as the darkest printable tint of the output medium. Tones lighter or darker than these may be clipped at the limits of the paper and ink. Satisfactory results may be obtained in monochrome work by doing nothing more than a perceptually-linear mapping of the image between these rigorously controlled endpoints. This level of sophistication is sufficient for many mid-range applications, although the results often appear flatter (i.e. lower in contrast) than desired.

The next step is to increase contrast slightly in the tonal range of the image that contains the most important subject matter. To perform this step well requires considerably more information about the image and about the press. To know where to add contrast, the algorithm must have access to first the keyness of the image; the tone range which the user considers most important. To know how much contrast to add, the algorithm must have access to the absolute tone of the original and the dynamic range of the output device so that it may calculate the amount of tone compression to which the image is actually subjected.

Most images are called normal key. The important subject areas of a normal key image are in the midtones. These images do well when a so-called "sympathetic curve" is applied, which increases the contrast in midtones slightly at the expense of contrast in the lighter and darker tones. White china on a white tablecloth is an example of a high key image. High key images benefit from higher contrast in lighter tones, with less contrast needed in the midtones and darker tones. Low key images have important subject matter in the darker tones and benefit from increasing the contrast in the darker tones. Specifying the keyness of an image might be attempted by automatic techniques, but it will likely fail without user input. For example, a photo of a bride in a white wedding dress it may be a high key image if you are selling wedding dresses, but may be a normal key image if you are the parents of the bride and are more interested in her smile.

Sophisticated color reproduction employs all of these principles, and then applies them in three dimensions. The mapping of the highlight and shadow becomes only one small, albeit critical, portion of the total issue of mapping colors that are too saturated for the output medium. Here again, automatic techniques may be employed as a first pass, with the user becoming involved in the clip or compress mapping decision. The HalftoneHints field is still useful in communicating which portions of the intensity of the image must be retained and which may be clipped. Again, a sophisticated application may override these settings if later user input is received.

APPENDIX HH

# Section 18: Associated Alpha Handling

This section describes a scheme for handling images with alpha data.

## Introduction

A common technique in computer graphics is to assemble an image from one or more elements that are rendered separately. When elements are combined using compositing techniques, matte or coverage information must be present for each pixel to create a properly anti-aliased accumulation of the full image [Porter84]. This matting information is an example of additional per-pixel data that must be maintained with an image. This section describes how to use the ExtraSamples field to store the requisite matting information, commonly called the associated alpha or just alpha. This scheme enables efficient manipulation of image data during compositing operations.

Images with matting information are stored in their natural format but with an additional component per pixel. The ExtraSample field is included with the image to indicate that an extra component of each pixel contains associated alpha data. In addition, when associated alpha data are included with RGB data, the RGB components must be stored premultiplied by the associated alpha component and component values in the range [0,2**BitsPerSample-1] are implicitly mapped onto the [0,1] interval. That is, for each pixel (r,g,b) and opacity A, where r, g, b, and A are in the range [0,1], (A*r,A*g,A*b,A) must be stored in the file. If A is zero, then the color components should be interpreted as zero. Storing data in this pre-multiplied format, allows compositing operations to be implemented most efficiently. In addition, storing pre-multiplied data makes it possible to specify colors with components outside the normal [0,1] interval. The latter is useful for defining certain operations that effect only the luminescence [Porter84].

## Fields

### ExtraSamples

Tag = 338 (152.H)

Type = SHORT

N = 1

This field must have a value of 1 (associated alpha data with pre-multiplied color components). The associated alpha data stored in component SamplesPerPixel-1 of each pixel contains the opacity of that pixel, and the color information is pre-multiplied by alpha.

APPENDIX HH

## Comments

Associated alpha data is just another component added to each pixel. Thus, for example, its size is defined by the value of the BitsPerSample field.

Note that since data is stored with RGB components already multiplied by alpha, naive applications that want to display an RGBA image on a display can do so simply by displaying the RGB component values. This works because it is effectively the same as merging the image with a black background. That is, to merge one image with another, the color of resultant pixels are calculated as:

$$C_r = C_{over} * A_{over} + C_{under} * (1-A_{over})$$

Since the "under image" is a black background, this equation reduces to

$$C_r = C_{over} * A_{over}$$

which is exactly the pre-multiplied color; i.e. what is stored in the image.

On the other hand, to print an RGBA image, one must composite the image over a suitable background page color. For a white background, this is easily done by adding 1 - A to each color component. For an arbitrary background color $C_{back}$, the *printed color* of each pixel is

$$C_{print} = C_{image} + C_{back} * (1-A_{image})$$

(since $C_{image}$ is pre-multiplied).

Since the ExtraSamples field is independent of other fields, this scheme permits alpha information to be stored in whatever organization is appropriate. In particular, components can be stored packed (PlanarConfiguration=1); this is important for good I/O performance and for good memory access performance on machines that are sensitive to data locality. However, if this scheme is used, TIFF readers must not derive the SamplesPerPixel from the value of the PhotometricInterpretation field (e.g., if RGB, then SamplesPerPixel is 3).

In addition to being independent of data storage-related fields, the field is also independent of the PhotometricInterpretation field. This means, for example, that it is easy to use this field to specify grayscale data and associated matte information. Note that a palette-color image with associated alpha will not have the colormap indices pre-multiplied; rather, the RGB colormap values will be pre-multiplied.

## Unassociated Alpha and Transparency Masks

Some image manipulation applications support notions of transparency masks and soft-edge masks. The associated alpha information described in this section is different from this *unassociated alpha* information in many ways, most importantly:

- Associated alpha describes opacity or coverage at each pixel, while clipping-related alpha information describes a boolean relationship. That is, associated alpha can specify fractional coverage at a pixel, while masks specify either 0 or 100 percent coverage.
- Once defined, associated alpha is not intended to be removed or edited, except as a result of compositing the image; it is an integral part of an image.

Unassociated alpha, on the other hand, is designed as an ancillary piece of information.

## References

[Porter84] "Compositing Digital Images". Thomas Porter, Tom Duff; Lucasfilm Ltd. ACM SIGGRAPH Proceedings Volume 18, Number 3. July, 1984.

# Section 19: Data Sample Format

This section describes a scheme for specifying data sample type information.

TIFF implicitly types all data samples as unsigned integer values. Certain applications, however, require the ability to store image-related data in other formats such as floating point. This section presents a scheme for describing a variety of data sample formats.

## Fields

### SampleFormat

Tag = 339 (153.H)

Type = SHORT

N = SamplesPerPixel

This field specifies how to interpret each data sample in a pixel. Possible values are:

1 = unsigned integer data

2 = two's complement signed integer data

3 = IEEE floating point data [IEEE]

4 = undefined data format

Note that the SampleFormat field does not specify the size of data samples; this is still done by the BitsPerSample field.

A field value of "undefined" is a statement by the writer that it did not know how to interpret the data samples; for example, if it were copying an existing image. A reader would typically treat an image with "undefined" data as if the field were not present (i.e. as unsigned integer data).

Default is 1, unsigned integer data.

### SMinSampleValue

Tag = 340 (154.H)

Type = the field type that best matches the sample data

N = SamplesPerPixel

This field specifies the minimum sample value. Note that a value should be given for each data sample. That is, if the image has 3 SamplesPerPixel, 3 values must be specified.

The default for SMinSampleValue and SMaxSampleValue is the full range of the data type.

### SMaxSampleValue

Tag = 341 (155.H)

Type = the field type that best matches the sample data

N = SamplesPerPixel

This new field specifies the maximum sample value.

## Comments

The SampleFormat field allows more general imaging (such as image processing) applications to employ TIFF as a valid file format.

SMinSampleValue and SMaxSampleValue become more meaningful when image data is typed. The presence of these fields makes it possible for readers to assume that data samples are bound to the range [SMinSampleValue, SMaxSampleValue] without scanning the image data.

## References

[IEEE] "IEEE Standard 754 for Binary Floating-point Arithmetic".

APPENDIX HH

# Section 20: RGB Image Colorimetry

Without additional information, RGB data is device-specific; that is, without an absolute color meaning. This section describes a scheme for describing and characterizing RGB image data.

## Introduction

Color printers, displays, and scanners continue to improve in quality and availability while they drop in price. Now the problem is to display color images so that they appear to be identical on different hardware.

The key to reproducing the same color on different devices is to use the CIE 1931 XYZ color-matching functions, the international standard for color comparison. Using CIE XYZ, an image's colorimetry information can fully describe its color interpretation. The approach taken here is essentially calibrated RGB. It implies a transformation from the RGB color space of the pixels to CIE 1931 XYZ.

The appearance of a color depends not only on its absolute tristimulus values, but also on the conditions under which it is viewed, including the nature of the surround and the adaptation state of the viewer. Colors having the same absolute tristimulus values appear the same in identical viewing conditions. The more complex issue of color appearance under different viewing conditions is addressed by [4]. The colorimetry information presented here plays an important role in color appearance under different viewing conditions.

Assuming identical viewing conditions, an application using the tags described below can display an image on different hardware and achieve colorimetrically identical results. The process of using this colorimetry information for displaying an image is straightforward on a color monitor but it is more complex for color printers. Also, the results will be limited by the color gamut and other characteristics of the display or printing device.

The following fields describe the image colorimetry information of a TIFF image:

| | |
|---|---|
| *WhitePoint* | chromaticity of the white point of the image |
| *PrimaryChromaticities* | chromaticities of the primaries of the image |
| *TransferFunction* | transfer function for the pixel data |
| *TransferRange* | extends the range of the transfer function |
| *ReferenceBlackWhite* | pixel component headroom and footroom parameters |

The TransferFunction, TransferRange, and ReferenceBlackWhite fields have defaults based on industry standards. An image has a colorimetric interpretation if and only if both the WhitePoint and PrimaryChromaticities fields are present. An image without these colorimetry fields will be displayed in an application and hardware dependent manner.

Note: In the following definitions, BitsPerSample is used as if it were a single number when in fact it is an array of SamplesPerPixel numbers. The elements of this array may not always be equal, for example: 5/6/5 16-bit pixels. BitsPerSample should be interpreted as the BitsPerSample value associated with a particular component. In the case of unequal BitsPerSample values, the definitions below can be extended in a straightforward manner.

This section has the following differences with Appendix H in TIFF 5.0:

- removed the use of image colorimetry defaults
- renamed the ColorResponseCurves field as TransferFunction
- optionally allowed a single TransferFunction table to describe all three channels
- described the use of the TransferFunction field for YCbCr, Palette, WhiteIsZero and BlackIsZero PhotometricInterpretation types
- added the TransferRange tag to expand the range of the TransferFunction below black and above white
- added the ReferenceBlackWhite field
- addressed the issue of color appearance

## *Colorimetry Field Definitions*

### *WhitePoint*

Tag = 318 (13E.H)

Type = RATIONAL

N = 2

The chromaticity of the white point of the image. This is the chromaticity when each of the primaries has its ReferenceWhite value. The value is described using the 1931 CIE xy chromaticity diagram and only the chromaticity is specified. This value can correspond to the chromaticity of the alignment white of a monitor, the filter set and light source combination of a scanner or the imaging model of a rendering package. The ordering is white[x], white[y].

For example, the CIE Standard Illuminant D65 used by CCIR Recommendation 709 and Kodak PhotoYCC is:

3127/10000,3290/10000

No default.

### *PrimaryChromaticities*

Tag =319 (13F.H)

Type = RATIONAL

N = 6

The chromaticities of the primaries of the image. This is the chromaticity for each of the primaries when it has its ReferenceWhite value and the other primaries have their ReferenceBlack values. These values are described using the 1931 CIE xy chromaticity diagram and only the chromaticities are specified. These values can correspond to the chromaticities of the phosphors of a monitor, the filter set and light source combination of a scanner or the imaging model of a rendering package. The ordering is red[x], red[y], green[x], green[y], blue[x], and blue[y].

For example the CCIR Recommendation 709 primaries are:

640/1000,330/1000,

300/1000, 600/1000,

150/1000, 60/1000

No default.

## *TransferFunction*

Tag =301 (12D.H)

Type = SHORT

N = {1 or 3} * (1 << BitsPerSample)

Describes a transfer function for the image in tabular style. Pixel components can be gamma-compensated, companded, non-uniformly quantized, or coded in some other way. The TransferFunction maps the pixel components from a non-linear BitsPerSample (e.g. 8-bit) form into a 16-bit linear form without a perceptible loss of accuracy.

If N = 1 << BitsPerSample, the transfer function is the same for each channel and all channels share a single table. Of course, this assumes that each channel has the same BitsPerSample value.

If N = 3 * (1 << BitsPerSample), there are three tables, and the ordering is the same as it is for pixel components of the PhotometricInterpretation field. These tables are separate and not interleaved. For example, with RGB images all red entries come first, followed by all green entries, followed by all blue entries.

The length of each component table is 1 << BitsPerSample. The width of each entry is 16 bits as implied by the type SHORT. Normally the value 0 represents the minimum intensity and 65535 represents the maximum intensity and the values [0, 0, 0] represent black and [65535,65535, 65535] represent white. If the TransferRange tag is present then it is used to determine the minimum and maximum values, and a scaling normalization.

The TransferFunction can be applied to images with a PhotometricInterpretation value of RGB, Palette, YCbCr, WhiteIsZero, and BlackIsZero. The TransferFunction is not used with other PhotometricInterpretation types.

For RGB PhotometricInterpretation, ReferenceBlackWhite expands the coding range, TransferRange expands the range of the TransferFunction, and the TransferFunction tables decompand the RGB value. The WhitePoint and PrimaryChromaticities further describe the RGB colorimetry.

For Palette color PhotometricInterpretation, the Colormap maps the pixel into three 16-bit values that when scaled to BitsPerSample-bits serve as indices into the TransferFunction tables which decompand the RGB value. The WhitePoint and PrimaryChromaticities further describe the underlying RGB colorimetry.

A Palette value can be scaled into a TransferFunction index by:

```
index= (value * ((1 << BitsPerSample) - 1)) / 65535;
```

A TransferFunction index can be scaled into a Palette color value by:

```
value= (index * 65535L) / ((1 << BitsPerSample) - 1);
```

Be careful if you intend to create Palette images with a TransferFunction. If the Colormap tag is directly converted from a hardware colormap, it may have a device gamma already incorporated into the DAC values.

For YCbCr PhotometricInterpretation, ReferenceBlackWhite expands the coding range, the YCbCrCoefficients describe the decoding matrix to transform YCbCr into RGB, TransferRange expands the range of the TransferFunction, and the TransferFunction tables decompand the RGB value. The WhitePoint and PrimaryChromaticities fields provide further description of the underlying RGB colorimetry.

After coding range expansion by ReferenceBlackWhite and TransferFunction expansion by TransferRange, RGB values may be outside the domain of the TransferFunction. Also, the display device matrix can transform RGB values into display device RGB values outside the domain of the device. These values are handled in an application-dependent manner.

For RGB images with non-default ReferenceBlackWhite coding range expansion and for YCbCr images, the resolution of the TransferFunction may be insufficient. For example, after the YCbCr transformation matrix, the decoded RGB values must be rounded to index into the TransferFunction tables. Applications needing the extra accuracy should interpolate between the elements of the TransferFunction tables. Linear interpolation is recommended.

For WhiteIsZero and BlackIsZero PhotometricInterpretation, the TransferFunction decompands the grayscale pixel value to a linear 16-bit form. Note that a TransferFunction value of 0 represents black and 65535 represents white regardless of whether a grayscale image is WhiteIsZero or BlackIsZero. For example, the zeroth element of a WhiteIsZero TransferFunction table will likely be 65535. This extension of the TransferFunction field for grayscale images is intended to replace the GrayResponseCurve field.

The TransferFunction does not describe a transfer characteristic outside of the range for ReferenceBlackWhite.

Default is a single table corresponding to the NTSC standard gamma value of 2.2. This table is used for each channel. It can be generated by:

```
NValues = 1 << BitsPerSample;
for (TF[0]= 0, i = 1; i < NValues; i++)
        TF[i]= floor(pow(i / (NValues - 1.0), 2.2) * 65535 + 0.5);
```

APPENDIX HH

## TransferRange

Tag = 342 (156.H)

Type = SHORT

N = 6

Expands the range of the TransferFunction. The first value within a pair is associated with TransferBlack and the second is associated with TransferWhite. The ordering of pairs is the same as for pixel components of the PhotometricInterpretation type. By default, theTransferFunction is defined over a range from a minimum intensity, 0 or nominal black, to a maximum intensity,(1 << BitsPerSample) - 1 or nominal white. Kodak PhotoYCC uses an extended range TransferFunction in order to describe highlights, saturated colors and shadow detail beyond this range. The TransferRange expands the TransferFunction to support these values. It is defined only for RGB and YCbCr PhotometricInterpretations.

After ReferenceBlackWhite and/or YCbCr decoding has taken place, an RGB value can be represented as a real number. It is then rounded to create an index into the TransferFunctiontable. In the absence of a TransferRange tag, or if the tag has the default values, the rounded value is an index and the normalized intensity value is:

```
index = (int) (value + (value < 0.0? -0.5 : 0.5));
intensity = TF[index] / 65535;
```

If the TransferRange tag is present and has non-default values, it provides an offset to be used with the rounded index. It also describes a scaling. The normalized intensity value is:

```
index = (int) (value + (value < 0.0? -0.5 : 0.5));
intensity = (TF[index + TransferRange[Black]] -
        TF[TransferRange[Black]])
        / (TF[TransferRange[White]] - TF[TransferRange[Black]]);
```

An application can write a TransferFunction with a non-defaultTransferRange as follows:

```
black_offset = scale_factor * Transfer(-TransferRange[Black]ar /
        (TransferRange[White] - TransferRange[Black]));
for (i = 0; i < (1 << BitsPerSample); i++)
        TF[i] = floor(0.5 - black_offset + scale_factor
                * Transfer((i - TransferRange[Black])
                / (TransferRange[White] - TransferRange[Black])));
```

The TIFF writer chooses scale_factor such that the TransferFunction fits into a 16-bit unsigned short, and chooses the TransferRange so that the most important part of the TransferFunction fits into the table.

Default is [0, NV, 0, NV, 0, NV] where NV = (1 <<BitsPerSample) - 1.

## ReferenceBlackWhite

Tag =532 (214.H)

Type = RATIONAL

N = 6

Specifies a pair of headroom and footroom image data values (codes) for each pixel component. The first component code within a pair is associated with ReferenceBlack, and the second is associated with ReferenceWhite. The ordering of pairs is the same as those for pixel components of the PhotometricInterpretation type. ReferenceBlackWhite can be applied to images with a PhotometricInterpretation value of RGB or YCbCr. ReferenceBlackWhite is not used with other PhotometricInterpretation values.

Computer graphics commonly places black and white at the extremities of the binary representation of image data; for example, black at code 0 and white at code 255. In other disciplines, such as printing, film, and video, there are practical reasons to provide footroom codes below ReferenceBlack and headroom codes above ReferenceWhite.

In film applications, they correspond to the densities Dmax and Dmin. In video applications, ReferenceBlack corresponds to 7.5 IRE and 0 IRE in systems with and without setup respectively, and ReferenceWhite corresponds to 100 IRE units.

Using YCbCr (See Section 21) and the CCIR Recommendation 601.1 video standard as an example, code 16 represents ReferenceBlack, and code 235 represents ReferenceWhite for the luminance component (Y). For the chrominance components, Cb and Cr, code 128 represents ReferenceBlack, and code 240 represents ReferenceWhite. With Cb and Cr, the ReferenceWhite value is used to code reference blue and reference red respectively.

The full range component value is converted from the code by:

```
FullRangeValue = (code - ReferenceBlack) * CodingRange
        / (ReferenceWhite - ReferenceBlack);
```

The code is converted from the full-range component value by:

```
code = (FullRangeValue * (ReferenceWhite - ReferenceBlack)
        / CodingRange) + ReferenceBlack;
```

For RGB images and the Y component of YCbCr images, CodingRange is defined as:

```
CodingRange = 2 ** BitsPerSample - 1;
```

For the Cb and Cr components of YCbCr images, CodingRange is defined as:

```
CodingRange = 127;
```

For RGB images, in the default special case of no headroom or footroom, this conversion can be skipped because the scaling multiplier equals 1.0 and the value equals the code.

For YCbCr images, in the case of no headroom or footroom, the conversion for Y can be skipped because the value equals the code. For Cb and Cr, ReferenceBlack must still be subtracted from the code. In the general case, the scaling multiplication for the Cb and Cr component codes can be factored into the YCbCr transform matrix.

Useful ReferenceBlackWhite values for YCbCr images are:

[0/1, 255/1,128/1, 255/1, 128/1, 255/1]

no headroom/footroom

[15/1, 235/1, 128/1, 240/1, 128/1, 240/1]

CCIR Recommendation 601.1 headroom/footroom

APPENDIX HH

Useful ReferenceBlackWhite values for BitsPerSample = 8,8,8 Class R images are:

[0/1, 255/1,0/1, 255/1, 0/1, 255/1]

no headroom/footroom

[16/1, 235/1, 16/1, 235/1, 16/1, 235/1]

CCIR Recommendation 601.1 headroom/footroom

Default is [0/,NV/1, 0/1, NV/1, 0/1, NV/1] where NV = 2 ** BitsPerSample - 1.

## References

[1] *The Reproduction of Colour in Photography, Printing and Television,* R. W. G. Hunt, Fountain Press, Tolworth, England,1987.

[2] *Principles of Color Technology*, Billmeyer and Saltzman, Wiley-Interscience, New York, 1981.

[3] *Colorimetric Properties of Video Displays*, William Cowan, University of Waterloo, Waterloo, Canada, 1989.

[4] *TIFF Color Appearance Guidelines*, Dave Farber, Eastman Kodak Company, Rochester, New York.

# Section 21: $YC_bC_r$ Images

## Introduction

Digitizers of video sources that create RGB data are becoming more capable and less expensive. The RGB color space is adequate for this purpose. However, for both digital video and image compression applications a color difference color space is needed. The television industry depends on $YC_bC_r$ for digital video. For image compression, subsampling the chrominance components allows for greater compression. TIFF $YC_bC_r$ (which we shall call *Class Y*) supports these images and applications.

Class Y is based on CCIR Recommendation 601-1, "Encoding Parameters of Digital Television for Studios." Class Y also has parameters that allow the description of related standards such as CCIR Recommendation 709 and technological variations such as component-sample positioning.

$YC_bC_r$ is a distinct PhotometricInterpretation type. RGB pixels are converted to and from $YC_bC_r$ for storage and display.

Class Y defines the following fields:

| | |
|---|---|
| $YC_bC_r$Coefficients | transformation from RGB to $YC_bC_r$ |
| $YC_bC_r$SubSampling | subsampling of the chrominance components |
| $YC_bC_r$Positioning | positioning of chrominance component samples relative to the luminance samples |

In addition, ReferenceBlackWhite, which specifies coding range expansion, is required by Class Y. See Section 20.

Class Y $YC_bC_r$ images have three components: Y, the luminance component, and $C_b$ and $C_r$, two chrominance components. Class Y uses the international standard notation $YC_bC_r$ for color-difference component coding. This is often incorrectly called YUV, which properly applies only to composite coding.

The transformations between $YC_bC_r$ and RGB are linear transformations of uninterpreted RGB sample data, typically gamma-corrected values. The $YC_bC_r$Coefficients field describes the parameters of this transformation.

Another feature of Class Y comes from subsampling the chrominance components. A Class Y image can be compressed by reducing the spatial resolution of chrominance components. This takes advantage of the relative insensitivity of the human visual system to chrominance detail. The $YC_bC_r$SubSampling field describes the degree of subsampling which has taken place.

When a Class Y image is subsampled, each $C_b$ and $C_r$ sample is associated with a group of luminance samples. The $YC_bC_r$Positioning field describes the position of the chrominance component samples relative to the group of luminance samples: centered or cosited.

Class Y requires use of the ReferenceBlackWhite field. This field expands the coding range by describing the reference black and white values for the different components that allow headroom and footroom for digital video images. Since the

APPENDIX HH

default for ReferenceBlackWhite is inappropriate for Class Y, it must be used explicitly.

At first, it might seem that the information conveyed by Class Y and the RGB Colorimetry section is redundant. However, decoding $YC_bC_r$ to RGB primaries requires the $YC_bC_r$ fields, and interpretation of the resulting RGB primaries requires the colorimetry and transfer function information. See the RGB Colorimetry section for details.

## Extensions to Existing Fields

Class Y images use a distinct PhotometricInterpretation Field value:

### PhotometricInterpretation

Tag = 262 (106.H)

Type = SHORT

N = 1

This Field indicates the color space of the image. The new value is:

6 = $YC_bC_r$

A value of 6 indicates that the image data is in the $YC_bC_r$ color space. TIFF uses the international standard notation $YC_bC_r$ for color-difference sample coding. Y is the luminance component. $C_b$ and $C_r$ are the two chrominance components. RGB pixels are converted to and from $YC_bC_r$ form for storage and display.

## Fields Defined in Class Y

### $YC_bC_r$Coefficients

Tag = 529 (211.H)

Type = RATIONAL

N = 3

The transformation from RGB to $YC_bC_r$ image data. The transformation is specified as three rational values that represent the coefficients used to compute luminance, Y.

The three rational coefficient values, *LumaRed*, *LumaGreen* and *LumaBlue*, are the proportions of red, green, and blue respectively in luminance, Y.

Y, $C_b$, and $C_r$ may be computed from RGB using the luminance coefficients specified by this field as follows:

$$Y = ( \mathit{LumaRed} * R + \mathit{LumaGreen} * G + \mathit{LumaBlue} * B )$$

$$C_b = ( B - Y ) / ( 2 - 2 * \mathit{LumaBlue} )$$

$C_r = (R - Y) / (2 - 2 * \mathit{LumaRed})$

R, G, and B may be computed from $YC_bC_r$ as follows:

$R = C_r * (2 - 2 * \mathit{LumaRed}) + Y$

$G = (Y - \mathit{LumaBlue} * B - \mathit{LumaRed} * R) / \mathit{LumaGreen}$

$B = C_b * (2 - 2 * \mathit{LumaBlue}) + Y$

In disciplines such as printing, film, and video, there are practical reasons to provide footroom codes below the ReferenceBlack code and headroom codes above ReferenceWhite code. In such cases the values of the transformation matrix used to convert from $YC_bC_r$ to RGB must be multiplied by a scale factor to produce full-range RGB values. These scale factors depend on the reference ranges specified by the ReferenceBlackWhite field. See the ReferenceBlackWhite and TransferFunction fields for more details.

The values coded by this field will typically reflect the transformation specified by a standard for $YC_bC_r$ encoding. The following table contains examples of commonly used values.

| Standard | *LumaRed* | *LumaGreen* | *LumaBlue* |
|---|---|---|---|
| CCIR Recommendation 601-1 | 299 / 1000 | 587 / 1000 | 114 / 1000 |
| CCIR Recommendation 709 | 2125 / 10000 | 7154 / 10000 | 721 / 10000 |

The default values for this field are those defined by CCIR Recommendation 601-1: 299/1000, 587/1000 and 114/1000, for *LumaRed*, *LumaGreen* and *LumaBlue*, respectively.

## $YC_bC_r$SubSampling

Tag = 530 (212.H)

Type = SHORT

N = 2

Specifies the subsampling factors used for the chrominance components of a $YC_bC_r$ image. The two fields of this field, *$YC_bC_r$SubsampleHoriz* and *$YC_bC_r$SubsampleVert*, specify the horizontal and vertical subsampling factors respectively.

The two fields of this field are defined as follows:

Short 0: *$YC_bC_r$SubsampleHoriz*:

1 = ImageWidth of this chroma image is equal to the ImageWidth of the associated luma image.

2 = ImageWidth of this chroma image is halfthe ImageWidth of the associated luma image.

4 = ImageWidth of this chroma image is one-quarter the ImageWidth of the associated luma image.

Short 1: *$YC_bC_r$SubsampleVert*:

1 = ImageLength (height) of this chroma image is equal to the ImageLength of the associated luma image.

2 = ImageLength (height) of this chroma image is half the ImageLength of the associated luma image.

4 = ImageLength (height) of this chroma image is one-quarter the ImageLength of the associated luma image.

Both $C_b$ and $C_r$ have the same subsampling ratio. Also, *YC$_b$C$_r$SubsampleVert* shall always be less than or equal to *YC$_b$C$_r$SubsampleHoriz.*

ImageWidth and ImageLength are constrained to be integer multiples of *YC$_b$C$_r$SubsampleHoriz* and *YC$_b$C$_r$SubsampleVert* respectively. TileWidth and TileLength have the same constraints. RowsPerStrip must be an integer multiple of *YC$_b$C$_r$SubsampleVert.*

The default values of this field are [ 2, 2 ].

## YC$_b$C$_r$Positioning

Tag = 531 (213.H)

Type = SHORT

N = 1

Specifies the positioning of subsampled chrominance components relative to luminance samples.

Specification of the spatial positioning of pixel samples relative to the other samples is necessary for proper image post processing and accurate image presentation. In Class Y files, the position of the subsampled chrominance components are defined with respect to the luminance component. Because components must be sampled orthogonally (along rows and columns), the spatial position of the samples in a given subsampled component may be determined by specifying the horizontal and vertical offsets of the first sample (i.e. the sample in the upper-left corner) with respect to the luminance component. The horizontal and vertical offsets of the first chrominance sample are denoted Xoffset[0,0] and Yoffset[0,0] respectively. Xoffset[0,0] and Yoffset[0,0] are defined in terms of the number of samples in the luminance component.

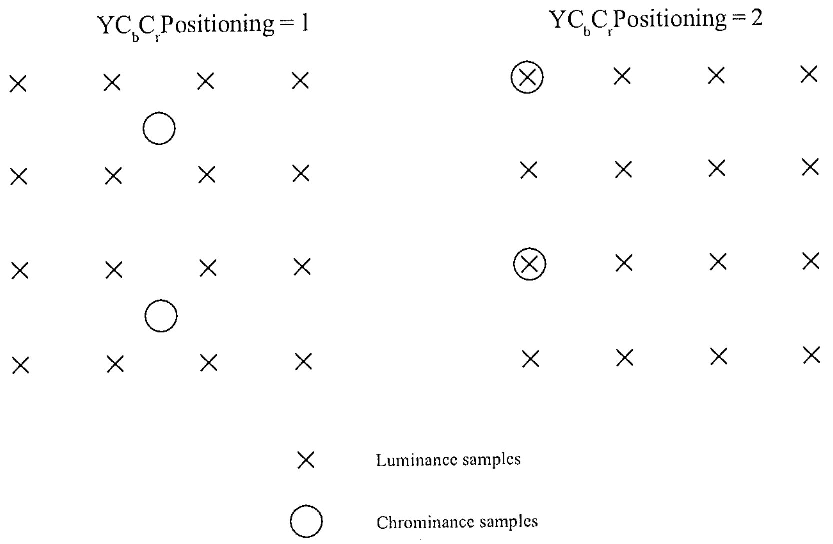
The values for this field are defined as follows:

| Tag value | YC$_b$C$_r$ Positioning | X and Y offsets of first chrominance sample |
|---|---|---|
| 1 | centered | Xoffset[0,0] = *ChromaSubsampleHoriz* / 2 - 0.5<br>Yoffset[0,0] = *ChromaSubsampleVert* / 2 - 0.5 |
| 2 | cosited | Xoffset[0,0] = 0<br>Yoffset[0,0] = 0 |

Field value 1 (centered) must be specified for compatibility with industry standards such as PostScript Level 2 and QuickTime. Field value 2 (cosited) must be specified for compatibility with most digital video standards, such as CCIR Recommendation 601-1.

As an example, for *ChromaSubsampleHoriz* = 4 and *ChromaSubsampleVert* = 2, the centers of the samples are positioned as illustrated below:

APPENDIX HH



$YC_bC_r$Positioning = 1

$YC_bC_r$Positioning = 2

Proper subsampling of the chrominance components incorporates an anti-aliasing filter that reduces the spectral bandwidth of the full-resolution samples. The type of filter used for subsampling determines the value of the $YC_bC_r$Positioning field.

For $YC_bC_r$Positioning = 1 (centered), subsampling of the chrominance components can easily be accomplished using a symmetrical digital filter with an even number of taps (coefficients). A commonly used filter for 2:1 subsampling utilizes two taps (1/2,1/2).

For $YC_bC_r$Positioning = 2 (cosited), subsampling of the chrominance components can easily be accomplished using a symmetrical digital filter with an odd number of taps. A commonly used filter for 2:1 subsampling utilizes three taps (1/4,1/2,1/4).

The default value of this field is 1.

## Ordering of Component Samples

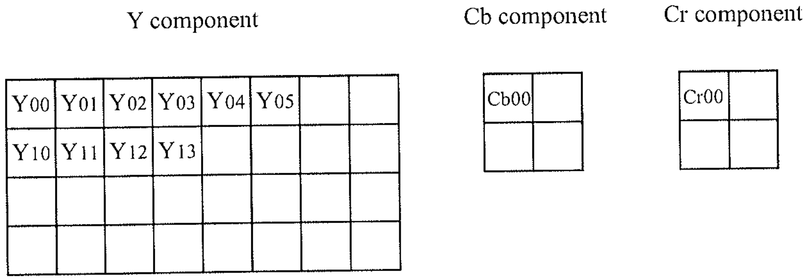
This section defines the ordering convention used for Y, $C_b$, and $C_r$ component samples when the PlanarConfiguration field value = 1 (interleaving). For PlanarConfiguration = 2, component samples are stored as 3 separate planes, and the ordering is the same as that used for other PhotometricInterpretation field values.

For PlanarConfiguration = 1, the component sample order is based on the subsampling factors, *ChromaSubsampleHoriz* and *ChromaSubsampleVert,* defined by the $YC_bC_r$SubSampling field. The image data within a TIFF file is comprised of one or more "data units", where a data unit is defined to be a sequence of samples:

- one or more Y samples
- a $C_b$ sample
- a $C_r$ sample

The Y samples within a data unit are specified as a two-dimensional array having *ChromaSubsampleVert* rows of *ChromaSubsampleHoriz* samples.

Expanding on the example in the previous section, consider a $YC_bC_r$ image having *ChromaSubsampleHoriz* = 4 and *ChromaSubsampleVert* = 2:



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Y00 | Y01 | Y02 | Y03 | Y04 | Y05 | | |
| Y10 | Y11 | Y12 | Y13 | | | | |
| | | | | | | | |
| | | | | | | | |

| | |
|---|---|
| Cb00 | |
| | |

| | |
|---|---|
| Cr00 | |
| | |

For PlanarConfiguration = 1, the sample order is:

$$Y_{00}, Y_{01}, Y_{02}, Y_{03}, Y_{10}, Y_{11}, Y_{12}, Y_{13}, Cb_{00}, Cr_{00}, Y_{04}, Y_{05} \ldots$$

## Minimum Requirements for YCbCr Images

In addition to satisfying the general Baseline TIFF requirements, a YCbCr file must have the following characteristics:

- SamplesPerPixel = 3. SHORT. Three components representing Y, Cb and Cr.
- BitsPerSample = 8,8,8. SHORT.
- Compression = none (1), LZW (5) or JPEG (6). SHORT.
- PhotometricInterpretation = $YC_bC_r$ (6). SHORT.
- ReferenceBlackWhite = 6 RATIONALS. Specify the reference values for black and white.

If the conversion from RGB is not according to CCIR Recommendation 601-1, code $YC_bC_r$Coefficients.

APPENDIX HH

# Section 22: JPEG Compression

## Introduction

Image compression reduces the storage requirements of pictorial data. In addition, it reduces the time required for access to, communication with, and display of images. To address the standardization of compression techniques an international standards group was formed: the Joint Photographic Experts Group (JPEG). JPEG has as its objective to create a joint ISO/CCITT standard for continuous tone image compression (color and grayscale).

JPEG decided that because of the broad scope of the standard, no one algorithmic procedure was able to satisfy the requirements of all applications. It was decided to specify different algorithmic processes, where each process is targeted to satisfy the requirements of a class of applications. Thus, the JPEG standard became a "toolkit" whereby the particular algorithmic "tools" are selected according to the needs of the application environment.

The algorithmic processes fall into two classes: lossy and lossless. Those based on the Discrete Cosine Transform (DCT) are lossy and typically provide for substantial compression without significant degradation of the reconstructed image with respect to the source image.

The simplest DCT-based coding process is the baseline process. It provides a capability that is sufficient for most applications. There are additional DCT-based processes that extend the baseline process to a broader range of applications.

The second class of coding processes is targeted for those applications requiring lossless compression. The lossless processes are not DCT-based and are utilized independently of any of the DCT-based processes.

This Section describes the JPEG baseline, the JPEG lossless processes, and the extensions to TIFF defined to support JPEG compression.

## JPEG Baseline Process

The baseline process is a DCT-based algorithm that compresses images having 8 bits per component. The baseline process operates only in sequential mode. In sequential mode, the image is processed from left to right and top to bottom in a single pass by compressing the first row of data, followed by the second row, and continuing until the end of image is reached. Sequential operation has minimal buffering requirements and thus permits inexpensive implementations.

The JPEG baseline process is an algorithm which inherently introduces error into the reconstructed image and cannot be utilized for lossless compression. The algorithm accepts as input only those images having 8 bits per component. Images with fewer than 8 bits per component may be compressed using the baseline process algorithm by left justifying each input component within a byte before compression.

APPENDIX HH

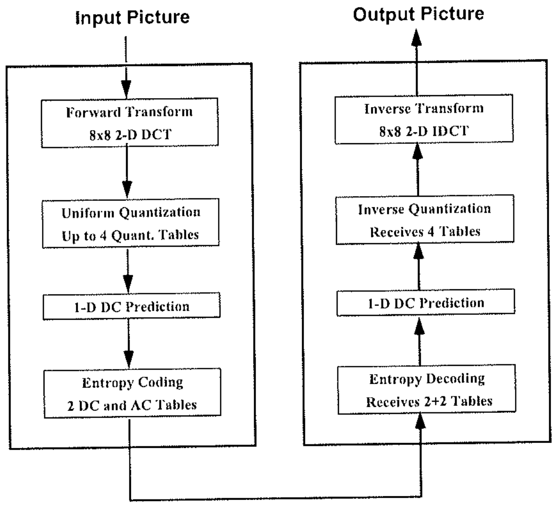


Figure 1. Baseline Process Encoder and Decoder

A functional block diagram of the Baseline encoding and decoding processes is contained in Figure 1. Encoder operation consists of dividing each component of the input image into 8x8 blocks, performing the two-dimensional DCT on each block, quantizing each DCT coefficient uniformly, subtracting the quantized DC coefficient from the corresponding term in the previous block, and then entropy coding the quantized coefficients using variable length codes (VLCs). Decoding is performed by inverting each of the encoder operations in the reverse order.

## The DCT

Before performing the foward DCT, input pixels are level-shifted so that they range from -128 to +127. Blocks of 8x8 pixels are transformed with the two-dimensional 8x8 DCT:

$$F(u,v) = \frac{1}{4} C(u)C(v) \Sigma\Sigma f(x,y) \cos \frac{\pi(2x+1)u}{16} \cos\frac{\pi(2y+1)v}{16}$$

and blocks are inverse transformed by the decoder with the Inverse DCT:

$$f(x,y) = \frac{1}{4} \Sigma \Sigma\, C(u)C(v)\, F(u,v) \cos\frac{\pi(2x+1)u}{16} \cos\frac{\pi(2y+1)v}{16}$$

with $u, v, x, y = 0, 1, 2, \ldots 7$

where $x, y$ = spatial coordinates in the pel domain
$u, v$ = coordinates in the transform domain
$C(u), C(v)$ = 1 / sqrt(2) for $u, v = 0$
1 otherwise

Although the exact method for computation of the DCT and IDCT is not subject to standardization and will not be specified by JPEG, it is probable that JPEG will adopt DCT-conformance specifications that designate the accuracy to which the DCT must be computed. The DCT-conformance specifications will assure that any two JPEG implementations will produce visually-similar reconstructed images.

## Quantization

The coefficients of the DCT are quantized to reduce their magnitude and increase the number of zero-value coefficients. The DCT coefficients are independently quantized by uniform quantizers. A uniform quantizer divides the real number line into steps of equal size, as shown in Figure 2. The quantization step-size applied to each coefficient is determined from the contents of a 64-element quantization table.

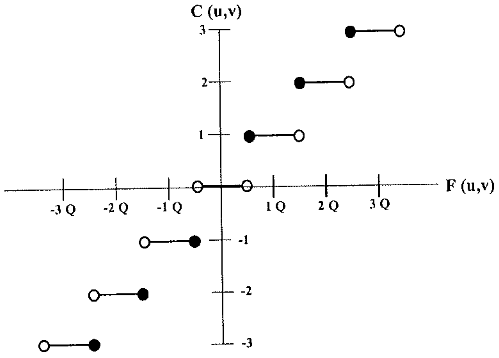


Figure 2. Uniform Quantization

The baseline process provides for up to 4 different quantization tables to be defined and assigned to separate interleaved components within a single scan of the input image. Although the values of each quantization table should ideally be determined through rigorous subjective testing which estimates the human psycho-visual thresholds for each DCT coefficient and for each color component of the input image, JPEG has developed quantization tables which work well for CCIR 601 resolution images and has published these in the informational section of the proposed standard.

## DC Prediction

The DCT coefficient located in the upper-left corner of the transformed block represents the average spatial intensity of the block and is referred to as the "DC coefficient". After the DCT coefficients are quantized, but before they are entropy coded, DC prediction is performed. DC prediction simply means that the DC term of the previous block is subtracted from the DC term of the current block prior to encoding.

APPENDIX HH

## Zig-Zag Scan

Prior to entropy coding, the DCT coefficients are ordered into a one-dimensional sequence according to a "zig-zag" scan. The DC coefficient is coded first, followed by AC coefficient coding, proceeding in the order illustrated in Figure 3.

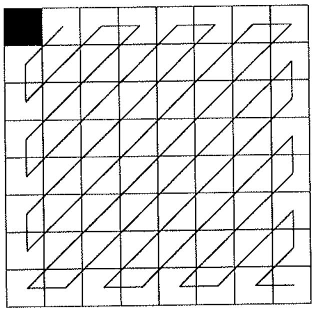


Figure 3. Zig-Zag Scan of DCT Coefficients

## Entropy Coding

The quantized DCT coefficients are further compressed using entropy coding. The baseline process performs entropy coding using variable length codes (VLCs) and variable length integers (VLIs).

VLCs, commonly known as Huffman codes, compress data symbols by creating shorter codes to represent frequently-occurring symbols and longer codes for occasionally-occurring symbols. One reason for using VLCs is that they are easily implemented by means of lookup tables.

Separate code tables are provided for the coding of DC and AC coefficients. The following paragraphs describe the respective coding methods used for coding DC and AC coefficients.

## DC Coefficient Coding

DC prediction produces a "differential DC coefficient" that is typically small in magnitude due to the high correlation of neighboring DC coefficients. Each differential DC coefficient is encoded by a VLC which represents the number of significant bits in the DC term followed by a VLI representing the value itself. The VLC is coded by first determining the number of significant bits, SSSS, in the differential DC coefficient through the following table:

| SSSS | Differential DC Value |
|---|---|
| 0 | 0 |
| 1 | -1, 1 |
| 2 | -3,-2, 2,3 |
| 3 | -7..-4, 4..7 |
| 4 | -15..-8, 8..15 |
| 5 | -31..-16, 16..31 |

APPENDIX HH

| | |
|---|---|
| 6 | -63..-32, 32..63 |
| 7 | -127..-64, 64..127 |
| 8 | -255..-128, 128..255 |
| 9 | -511..-256, 256..511 |
| 10 | -1023..-512, 512..1023 |
| 11 | -2047..-1024, 1024..2047 |
| 12 | -4095..-2048, 2048..4095 |

SSSS is then coded from the selected DC VLC table. The VLC is followed by a VLI having SSSS bits that represents the value of the differential DC coefficient itself. If the coefficient is positive, the VLI is simply the low-order bits of the coefficient. If the coefficient is negative, then the VLI is the low-order bits of the coefficient-1.

## *AC Coefficient Coding*

In a similar fashion, AC coefficients are coded with alternating VLC and VLI codes. The VLC table, however, is a two-dimensional table that is indexed by a composite 8-bit value. The lower 4 bits of the 8-bit value, i.e. the column index, is the number of significant bits, SSSS, of a non-zero AC coefficient. SSSS is computed through the same table as that used for coding the DC coefficient. The higher-order 4 bits, the row index, is the number of zero coefficients, NNNN, that precede the non-zero AC coefficient. The first column of the two-dimensional coding table contains codes that represent control functions. Figure 4 illustrates the general structure of the AC coding table.

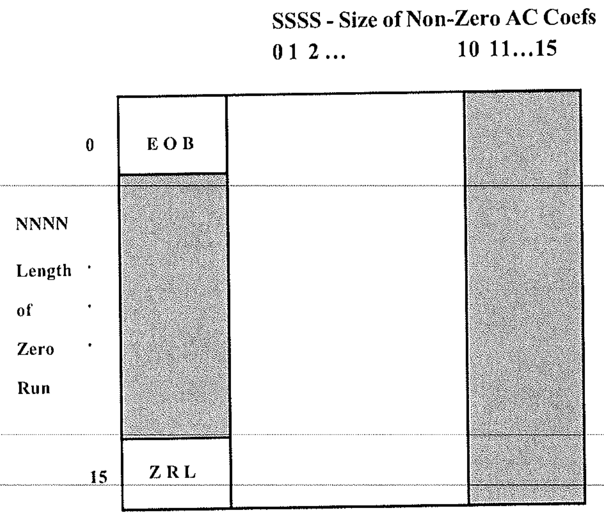


Figure 4. 2-D Run-Size Value Array for AC Coefs
The shaded portions are undefined in the baseline process

The flow chart in Figure 5 specifies the AC coefficient coding procedure. AC coefficients are coded by traversing the block in the zig-zag sequence and count-

Microsoft et al. Exhibit 1005

ing the number of zero coefficients until a non-zero AC coefficient is encountered. If the count of consecutive zero coefficients exceeds 15, then a ZRL code is coded and the zero run-length count is reset. When a non-zero AC coefficient is found, the number of significant bits in the non-zero coefficient, SSSS, is combined with the zero run-length that precedes the coefficient, NNNN, to form an index into the two-dimensional VLC table. The selected VLC is then coded. The VLC is followed by a VLI that represents the value of the AC coefficient. This process is repeated until the end of the block is reached. If the last AC coefficient is zero, then an End of Block (EOB) VLC is encoded.

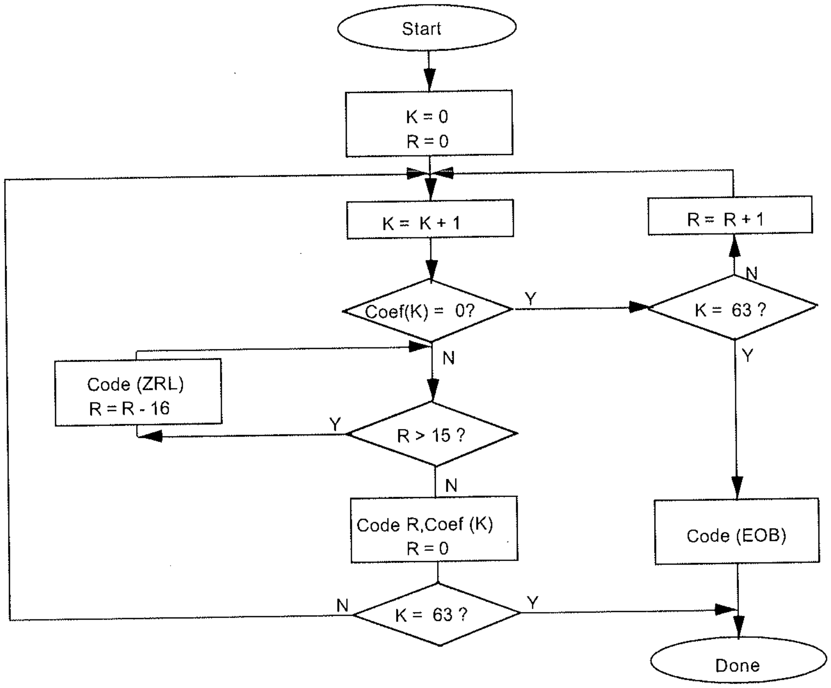


Figure 5. Encoding Procedure for AC Coefs

## JPEG Lossless Processes

The JPEG lossless coding processes utilize a spatial-prediction algorithm based upon a two-dimensional Differential Pulse Code Modulation (DPCM) technique. They are compatible with a wider range of input pixel precision than the DCT-based algorithms (2 to 16 bits per component). Although the primary motivation for specifying a spatial algorithm is to provide a method for lossless compression, JPEG allows for quantization of the input data, resulting in lossy compression and higher compression rates.

Although JPEG provides for use of either the Huffman or Arithmetic entropy-coding models by the processes for lossless coding, only the Huffman coding model is supported by this version of TIFF. The following is a brief overview of the lossless process with Huffman coding.

## Control Structure

Much of the control structure developed for the sequential DCT procedures is also used for sequential lossless coding. Either interleaved or non-interleaved data ordering may be used.

## Coding Model

The coding model developed for coding the DC coefficients of the DCT is extended to allow a number of one-dimensional and two-dimensional predictors for the lossless coding function. Each component uses an independent predictor.

## Prediction

Figure 6 shows the relationship between the neighboring values used for prediction and the sample being coded.

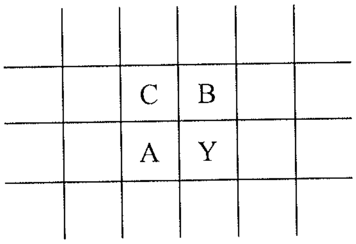


Figure 6. Relationship between sample and prediction samples

Y is the sample to be coded and A, B, and C are the samples immediately to the left, immediately above, and diagonally to the left and above.

The allowed predictors are listed in the following table.

| Selection-value | Prediction |
|---|---|
| 0 | no prediction (differential coding) |
| 1 | A |
| 2 | B |
| 3 | C |
| 4 | A+B-C |
| 5 | A+((B-C)/2) |
| 6 | B+((A-C)/2) |
| 7 | (A+B)/2 |

Selection-value 0 shall only be used for differential coding in the hierarchical mode. Selections 1, 2 and 3 are one-dimensional predictors and selections 4, 5, 6, and 7 are two dimensional predictors. The divide by 2 in the prediction equations is done by a arithmetic-right-shift of the integer values.

APPENDIX HH

The difference between the prediction value and the input is calculated modulo 2**16. Therefore, the prediction can also be treated as a modulo 2**16 value. In the decoder the difference is decoded and added, modulo 2**16, to the prediction.

### Huffman Coding of the Prediction Error

The Huffman coding procedures defined for coding the DC coefficients are used to code the modulo 2**16 differences. The table for DC coding is extended to 17 entries that allows for coding of the modulo 2**16 differences.

### Point Transformation Prior to Lossless Coding

For the lossless processes only, the input image data may optionally be scaled (quantized) prior to coding by specifying a nonzero value in the point transformation parameter. Point transformation is defined to be division by a power of 2.

If the point transformation field is nonzero for a component, a point transformation of the input is performed prior to the lossless coding. The input is divided by 2**Pt, where Pt is the value of the point transform signaling field. The output of the decoder is rescaled to the input range by multiplying by 2**Pt. Note that the scaling of input and output can be performed by arithmetic shifts.

## Overview of the JPEG Extension to TIFF

In extending the TIFF definition to include JPEG compressed data, it is necessary to note the following:

- JPEG is effective only on continuous-tone color spaces:

| | | |
|---|---|---|
| Grayscale | (Photometric Interpretation = 1) | |
| RGB | (Photometric Interpretation = 2) | |
| CMYK | (Photometric Interpretation = 5) | (See the CMYK Images section.) |
| $YC_bC_r$ | (Photometric Interpretation = 6) | (See the YCbCr images section.) |

- Color conversion to $YC_bC_r$ is often used as part of the compression process because the chrominance components can be subsampled and compressed to a greater degree without significant visual loss of quality. Fields are defined to describe how this conversion has taken place and the degree of subsampling employed (see the YCbCr Images section).

- New fields have been defined to specify the JPEG parameters used for compression and to allow quantization tables and Huffman code tables to be incorporated into the TIFF file.

APPENDIX HH

- TIFF is compatible with compressed image data that conforms to the syntax of the JPEG interchange format for compressed image data. Fields are defined that may be utilized to facilitate conversion from TIFF to interchange format.

- The PlanarConfiguration Field is used to specify whether or not the compressed data is interleaved as defined by JPEG. For any of the JPEG DCT-based processes, the interleaved data units are coded 8x8 blocks rather than component samples.

- Although JPEG codes consecutive image blocks in a single contiguous bitstream, it is extremely useful to employ the concept of tiles in an image. The TIFF Tiles section defines some new fields for tiles. These fields should be stored in place of the older fields for strips. The concept of tiling an image in both dimensions is important because JPEG hardware may be limited in the size of each block that is handled.

- Note that the nomenclature used in the TIFF specification is different from the JPEG Draft International Standardittee Draft (ISO DIS 10918-1) in some respects. The following terms should be equated when reading this Section:

| TIFF name | JPEG DIS name |
|---|---|
| ImageWidth | Number of Pixels |
| ImageLength | Number of Lines |
| SamplesPerPixel | Number of Components |
| JPEGQTable | Quantization Table |
| JPEGDCTable | Huffman Table for DC coefficients |
| JPEGACTable | Huffman Table for AC coefficients |

## *Strips and Tiles*

The JPEG extension to TIFF has been designed to be consistent with the existing TIFF strip and tile structures and to allow quick conversion to and from the stream-oriented compressed image format defined by JPEG.

Compressed images conforming to the syntax of the JPEG interchange format can be converted to TIFF simply by defining a single strip or tile for the entire image and then concatenating the TIFF image description fields to the JPEG compressed image data. The strip or tile offset field points directly to the start of the entropy coded data (not to a JPEG marker).

Multiple strips or tiles are supported in JPEG compressed images using restart markers. Restart markers, inserted periodically into the compressed image data, delineate image segments known as restart intervals. At the start of each restart interval, the coding state is reset to default values, allowing every restart interval to be decoded independently of previously decoded data. TIFF strip and tile offsets shall always point to the start of a restart interval. Equivalently, each strip or tile contains an integral number of restart intervals. Restart markers need not be present in a TIFF file; they are implicitly coded at the start of every strip or tile.

To maximize interchangeability of TIFF files with other formats, a restriction is placed on tile height for files containing JPEG-compressed image data conforming to the JPEG interchange format syntax. The restriction, imposed only when the tile width is shorter than the image width and when the JPEGInterchangeFormat Field is present and non-zero, states that the tile height must be equal to the height of one JPEG Minimum Coded Unit (MCU). This restriction ensures that TIFF files may be converted to JPEG interchange format without undergoing decompression.

## Extensions to Existing Fields

### Compression

Tag = 259 (103.H)

Type = SHORT

N = 1

This Field indicates the type of compression used. The new value is:

6 = JPEG

## JPEG Fields

### JPEGProc

Tag = 512 (200.H)

Type = SHORT

N = 1

This Field indicates the JPEG process used to produce the compressed data. The values for this field are defined to be consistent with the numbering convention used in ISO DIS 10918-2. Two values are defined at this time.

1= Baseline sequential process

14= Lossless process with Huffman coding

When the lossless process with Huffman coding is selected by this Field, the Huffman tables used to encode the image are specified by the JPEGDCTables field, and the JPEGACTables field is not used.

Values indicating JPEG processes other than those specified above will be defined in the future.

Not all of the fields described in this section are relevant to the JPEG process selected by this Field. The following table specifies the fields that are applicable to each value defined by this Field.

| Tag Name | JPEGProc =1 | JPEGProc =14 |
|---|---|---|
| JPEGInterchangeFormat | X | X |
| JPEGInterchangeFormatLength | X | X |
| JPEGRestart Interval | X | X |
| JPEGLosslessPredictors | | X |
| JPEGPointTransforms | | X |
| JPEGQTables | X | |
| JPEGDCTables | X | X |
| JPEGACTables | X | |

This Field is mandatory whenever the Compression Field is JPEG (no default).

## *JPEGInterchangeFormat*

Tag = 513 (201.H)

Type = LONG

N = 1

This Field indicates whether a JPEG interchange format bitstream is present in the TIFF file. If a JPEG interchange format bitstream is present, then this Field points to the Start of Image (SOI) marker code.

If this Field is zero or not present, a JPEG interchange format bitstream is not present.

## *JPEGInterchangeFormatLength*

Tag = 514 (202.H)

Type = LONG

N = 1

This Field indicates the length in bytes of the JPEG interchange format bitstream. This Field is useful for extracting the JPEG interchange format bitstream without parsing the bitstream.

This Field is relevant only if the JPEGInterchangeFormat Field is present and is non-zero.

## *JPEGRestartInterval*

Tag = 515 (203.H)

Type = SHORT

N = 1

This Field indicates the length of the restart interval used in the compressed image data. The restart interval is defined as the number of Minimum Coded Units (MCUs) between restart markers.

Restart intervals are used in JPEG compressed images to provide support for multiple strips or tiles. At the start of each restart interval, the coding state is reset to default values, allowing every restart interval to be decoded independently of previously decoded data. TIFF strip and tile offsets shall always point to the start of a restart interval. Equivalently, each strip or tile contains an integral number of restart intervals. Restart markers need not be present in a TIFF file; they are implicitly coded at the start of every strip or tile.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more information about the restart interval and restart markers.

If this Field is zero or is not present, the compressed data does not contain restart markers.

## *JPEGLosslessPredictors*

Tag = 517 (205.H)

Type = SHORT

N = SamplesPerPixel

This Field points to a list of lossless predictor-selection values, one per component.
The allowed predictors are listed in the following table.

| Selection-value | Prediction |
|---|---|
| 1 | A |
| 2 | B |
| 3 | C |
| 4 | A+B-C |
| 5 | A+((B-C)/2) |
| 6 | B+((A-C)/2) |
| 7 | (A+B)/2 |

A, B, and C are the samples immediately to the left, immediately above, and diagonally to the left and above the sample to be coded, respectively.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details.

This Field is mandatory whenever the JPEGProc Field specifies one of the lossless processes (no default).

## *JPEGPointTransforms*

Tag = 518 (206.H)

Type = SHORT

N = SamplesPerPixel

APPENDIX HH

This Field points to a list of point transform values, one per component. This Field is relevant only for lossless processes.

If the point transformation value is nonzero for a component, a point transformation of the input is performed prior to the lossless coding. The input is divided by 2**Pt, where Pt is the point transform value. The output of the decoder is rescaled to the input range by multiplying by 2**Pt. Note that the scaling of input and output can be performed by arithmetic shifts.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details. The default value of this Field is 0 for each component (no scaling).

## JPEGQTables

Tag = 519 (207.H)

Type = LONG

N = SamplesPerPixel

This Field points to a list of offsets to the quantization tables, one per component. Each table consists of 64 BYTES (one for each DCT coefficient in the 8x8 block). The quantization tables are stored in zigzag order.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details.

It is strongly recommended that, within the TIFF file, each component be assigned separate tables. This Field is mandatory whenever the JPEGProc Field specifies a DCT-based process (no default).

## JPEGDCTables

Tag = 520 (208.H)

Type = LONG

N = SamplesPerPixel

This Field points to a list of offsets to the DC Huffman tables or the lossless Huffman tables, one per component.

The format of each table is as follows:

16 BYTES of "BITS", indicating the number of codes of lengths 1 to 16;

Up to 17 BYTES of "VALUES", indicating the values associated with those codes, in order of length.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details.

It is strongly recommended that, within the TIFF file, each component be assigned separate tables. This Field is mandatory for all JPEG processes (no default).

## JPEGACTables

Tag = 521 (209.H)

Type = LONG

N = SamplesPerPixel

This Field points to a list of offsets to the Huffman AC tables, one per component. The format of each table is as follows:

> 16 BYTES of "BITS", indicating the number of codes of lengths 1 to 16;
>
> Up to 256 BYTES of "VALUES", indicating the values associated with those codes, in order of length.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details.

It is strongly recommended that, within the TIFF file, each component be assigned separate tables. This Field is mandatory whenever the JPEGProc Field specifies a DCT-based process (no default).

APPENDIX HH

## Minimum Requirements for TIFF with JPEG Compression

The table on the following page shows the minimum requirements of a TIFF file that uses tiling and contains JPEG data compressed with the Baseline process.

| | |
|---|---|
| Tag = NewSubFileType (254)<br>Type = Long<br>Length = 1<br>Value = 0 | Single image |
| Tag = ImageWidth (256)<br>Type = Long<br>Length = 1<br>Value = ? | |
| Tag = ImageLength (257)<br>Type = Long<br>Length = 1<br>Value = ? | |
| Tag = BitsPerSample (258)<br>Type = Short<br>Length = SamplesPerPixel<br>Value = ? | 8 : Monochrome<br>8,8,8 : RGB<br>8,8,8 : YCbCr<br>8,8,8,8 : CMYK |
| Tag = Compression (259)<br>Type = Long<br>Length = 1<br>Value = 6 | 6 : JPEG compression |
| Tag = PhotometricInterpretation (262)<br>Type = Short<br>Length = 1<br>Value = ? | 0,1 : Monochrome<br>2 : RGB<br>5 : CMYK<br>6 : YCbCr |
| Tag = SamplesPerPixel (277)<br>Type = Short<br>Length = 1<br>Value = ? | 1 : Monochrome<br>3 : RGB<br>3 : YCbCr<br>4 : CMYK |
| Tag = XResolution (282)<br>Type = Rational<br>Length = 1<br>Value = ? | |
| Tag = YResolution (283)<br>Type = Rational<br>Length = 1<br>Value = ? | |
| Tag = PlanarConfiguration (284)<br>Type = Short<br>Length = 1<br>Value = ? | 1 : Block Interleaved<br>2 : Not interleaved |
| Tag = ResolutionUnit (296)<br>Type = Short<br>Length = 1<br>Value = ? | |
| Tag = TileWidth (322)<br>Type = Short<br>Length = 1<br>Value = ? | Multiple of 8 |
| Tag = TileLength (323)<br>Type = Short<br>Length = 1<br>Value = ? | Multiple of 8 |
| Tag = TileOffsets (324)<br>Type = Long<br>Length = Number of tiles<br>Value = ? | |
| Tag = TileByteCounts (325)<br>Type = Long<br>Length = Number of tiles<br>Value = ? | |
| Tag = JPEGProc (512)<br>Type = Short<br>Length = 1<br>Value = ? | 1 : Baseline process |
| Tag = JPEGQTables (519)<br>Type = Long<br>Length = SamplesPerPixel<br>Value = ? | Offsets to tables |
| Tag = JPEGDCTables (520)<br>Type = Long<br>Length = SamplesPerPixel<br>Value = ? | Offsets to tables |
| Tag = JPEGACTables (521)<br>Type = Long<br>Length = SamplesPerPixel<br>Value = ? | Offsets to tables |

## References

[1] Wallace, G., "Overview of the JPEG Still Picture Compression Algorithm", Electronic Imaging East '90.

[2] ISO/IEC DIS 10918-1, "Digital Compression and Coding of Continuous-tone Still Images", Sept. 1991.

APPENDIX HH

# Section 23: CIE L*a*b* Images

## What is CIE L*a*b*?

CIE La*b* is a color space that is colorimetric, has separate lightness and chroma channels, and is approximately perceptually uniform. It has excellent applicability for device-independent manipulation of continuous tone images. These attributes make it an excellent choice for many image editing functions.

1976 CIEL*a*b* is represented as a Euclidean space with the following three quantities plotted along axes at right angles: *L** representing lightness, *a** representing the red/green axis, and *b** representing the yellow/blue axis. The formulas for 1976 CIE *L*a*b** follow:

$$L^{*}=116(Y/Y_{n})^{1/3}-16 \qquad \text{for } Y/Y_{n} > 0.008856$$

$$L^{*}=903.3(Y/Y_{n}) \qquad \text{for } Y/Y_{n} <= 0.008856 \qquad \text{*see note below.}$$

$$a^{*}=500\left[(X/X_{n})^{1/3}-(Y/Y_{n})^{1/3}\right]$$

$$b^{*}=200\left[(Y/Y_{n})^{1/3}-(Z/Z_{n})^{1/3}\right].$$

where $X_n$,$Y_n$, and $Z_n$ are the CIE *X, Y*, and *Z* tristimulus values of an *appropriate* reference white. Also, if any of the ratios $X/X_n$, $Y/Y_n$, or $Z/Z_n$ is equal to or less than 0.008856, it is replaced in the formulas with

$$7.787F + 16/116,$$

where *F* is $X/X_n$, $Y/Y_n$, or $Z/Z_n$, as appropriate (note: these low-light conditions are of no relevance for most document-imaging applications). Tiff is defined such that each quantity be encoded with 8 bits. This provides 256 levels of *L** lightness; 256 levels (+/- 127) of *a**, and 256 levels (+/- 127) of *b**. Dividing the 0-100 range of *L** into 256 levels provides lightness steps that are less than half the size of a "just noticeable difference". This eliminates banding, even under conditions of substantial tonal manipulation. Limiting the theoretically unbounded *a** and *b** ranges to +/- 127 allows encoding in 8 bits without eliminating any but the most saturated self-luminous colors. It is anticipated that the rare specialized applications requiring support of these extreme cases would be unlikely to use CIELAB anyway. All object colors, in fact all colors within the theoretical MacAdam limits, fall within the +/- 127 *a*/b** range.

APPENDIX HH

## The TIFF CIELAB Fields

### PhotometricInterpretation

Tag = 262 (106.H)

Type = SHORT

N = 1

8 = 1976 CIE *L*a*b**

### Usage of other Fields.

BitsPerSample: 8

SamplesPerPixel - ExtraSamples: 3 for *L*a*b**, 1 implies *L** only, for monochrome data.

Compression: same as other multi-bit formats. JPEG compression applies.

PlanarConfiguration: both chunky and planar data could be supported.

WhitePoint: does not apply

PrimaryChromaticicities: does not apply.

TransferFunction: does not apply

Alpha Channel information will follow the lead of other data types.

The reference white for this data type is the *perfect reflecting diffuser* (100% diffuse reflectance at all visible wavelengths). The *L** range is from 0 (perfect absorbing black) to 100 (perfect reflecting diffuse white). The *a** and *b** ranges will be represented as signed 8 bit values having the range -127 to +127.

## Converting between RGB and CIELAB, a Caveat

The above CIELAB formulae are derived from CIE *XYZ*. Converting from CIELAB to *RGB* requires an additional set of formulae for converting between *RGB* and *XYZ*. For standard NTSC primaries these are:

| | | | | |
|---|---|---|---|---|
| 0.60700.17400.2000 | | R | | X |
| 0.29900.58700.1140 | * | G | = | Y |
| 0.00000.06601.1110 | | B | | Z |

Generally, D65 illumination is used and a perfect reflecting diffuser is used for the reference white.

Since CIELAB is not a directly displayable format, some conversion to RGB will be required. While look-up table accelerated CIELAB to RGB conversion is certainly possible and fast, TIFF writers may choose to include a low resolution RGB subfile as an integral part of TIFF CIELAB.

APPENDIX HH

## Color Difference Measurements in CIELAB

The differences between two colors in $L^*$, $a^*$, and $b^*$ are denoted by D$L^*$, D$a^*$, and D$b^*$, respectively, with the total (3-dimensional) color difference represented as:

$$\Delta E^*_{\mathrm{ab}} = \left[(\Delta E^*)^2+(\Delta a^*)^2+(\Delta b^*)^2\right]^{1/2}.$$

This color difference can also be expressed in terms of $L^*$, $C^*$, and a measure of hue. In this case, $h_{ab}$ is *not* used because it is an angular measure and cannot be combined with $L^*$ and $C^*$ directly. A linear-distance form of hue is used instead:

*CIE* 1976 *a,b hue-difference,* $\Delta H^*_{ab}$

$$\Delta H^*_{ab} = \left[(\Delta E^*)^2-(\Delta L^*)^2-(\Delta C^*)^2\right]^{1/2}.$$

where D$C^*$ is the chroma difference between the two colors. The total color difference expression using this hue-difference is:

$$\Delta E^*_{\mathrm{ab}} = \left[(\Delta L^*)^2+(\Delta H^*)^2+(\Delta b^*)^2\right]^{1/2}.$$

It is important to remember that color difference is 3-dimensional: much more can be learned from a DL*a*b* triplet than from a single DE value. The DL*C*H* form is often the most useful since it gives the error information in a form that has more familiar perception correlates. Caution is in order, however, when using DH* for large hue differences since it is a straight-line approximation of a curved hue distance.

## The Merits of CIELAB

### Colorimetric.

First and foremost, CIELAB is colorimetric. It is traceable to the internationally-recognized standard CIE 1931 Standard Observer. This insures that it encodes color in a manner that is accurately modeled after the human vision system. Colors seen as matching are encoded identically, and colors seen as not matching are encoded differently. CIELAB provides an unambiguous definition of color without the necessity of additional information such as with RGB (primary chromaticities, white point, and gamma curves).

### Device Independent.

Unlike RGB spaces which associate closely with physical phosphor colors, CIELAB contains no device association. CIELAB is not tailored for one device or device type at the expense of all others.

## Full Color Gamut.

Any one image or imaging device usually encounters a very limited subset of the entire range of humanly-perceptible color. Collectively, however, these images and devices span a much larger gamut of color. A truly versatile exchange color space should encompass all of these colors, ideally providing support for all visible color. RGB, PhotoYCC, YCbCr, and other display spaces suffer from gamut limitations that exclude significant regions of easily printable colors. CIELAB is defined for all visible color.

## Efficiency

A good exchange space will maximize accuracy of translations between itself and other spaces. It will represent colors compactly for a given accuracy. These attributes are provided through visual uniformity. One of the greatest disadvantages of the classic CIE system (and RGB systems as well) is that colors within it are not equally spaced visually. Encoding full-color images in a linear-intensity space, such as the typical RGB space or, especially, the XYZ space, requires a very large range (greater than 8-bits/primary) to eliminate banding artifacts. Adopting a *non-*linear RGB space improves the efficiency but not nearly to the extent as with a perceptually uniform space where these problems are nearly eliminated. A uniform space is also more efficiently compressed (see below).

## Public Domain / Single Standard

CIELAB maintains no preferential attachments to any private organization. Its existence as a single standard leaves no room for ambiguity. Since 1976, CIELAB has continually gained popularity as a widely-accepted and heavily-used standard.

## Luminance/Chrominance Separation.

The advantages for image size compression made possible by having a separate lightness or luminance channel are immense. Many such spaces exist. The degree to which the luminance information is fully-isolated into a single channel is an important consideration. Recent studies (Kasson and Plouffe of IBM) support CIELAB as a leading candidate placing it above CIELUV, YIQ, YUV, YCC, and XYZ.

Other advantages support a separate lightness or luminance channel. Tone and contrast editing and detail enhancement are most easily accomplished with such a channel. Conversion to a black and white representation is also easiest with this type of space.

When the chrominance channels are encoded as opponents as with CIELAB, there are other compression, image manipulation, and white point handling advantages.

### Compressibility (Data).

Opponent spaces such as CIELAB are inherently more compressible than tristimulus spaces such as RGB. The chroma content of an image can be compressed to a greater extent, without objectionable loss, than can the lightness content. The opponent arrangement of CIELAB allows for spatial subsampling and efficent compression using JPEG.

### Compressibility (Gamut).

Adjusting the color range of an image to match the capabilities of the intended output device is a critical function within computational color reproduction. Luminance/chrominance separation, especially when provided in a polar form, is desirable for facilitating gamut compression. Accurate gamut compression in a tri-linear color space is difficult.

CIELAB has a polar form (*metric hue angle,* and *metric chroma*, described below) that serves compression needs fairly well. Because CIELAB is not perfectly uniform, problems can arise when compressing along constant hue lines. Noticeable hue errors are sometimes introduced. This problem is no less severe with other contending color spaces.

This polar form also provides advantages for local color editing of images. The polar form is not proposed as part of the TIFF addition.

## Getting the Most from CIELAB

### Image Editors

The advantages of image editing within a perceptually uniform polar color space are tremendous. A detailed description of these advantages is beyond the scope of this section. As previously mentioned, many common tonal manipulation tasks are most efficiently performed when only a single channel is affected. Edge enhancement, contrast adjustment, and general tone-curve manipulation all ideally affect only the lightness component of an image.

A perceptual polar space works excellently for specifying a color range for masking purposes. For example, a red shirt can be quickly changed to a green shirt without drawing an outline mask. The operation can be performed with a loosely, quickly-drawn mask region combined with a hue (and perhaps chroma) range that encompasses the shirt's colors. The hue component of the shirt can then be adjusted, leaving the lightness and chroma detail in place.

Color cast adjustment is easily realized by shifting either or both of the chroma channels over the entire image or blending them over the region of interest.

### Converting from CIELAB to a device specific space

For fast conversion to an RGB display, CIELAB can be decoded using 3x3 matrixing followed by gamma correction. The computational complexity required for accurate CRT display is the same with CIELAB as with extended luminance-chrominance spaces.

Converting CIELAB for accurate printing on CMYK devices requires computational complexity no greater than with *accurate* conversion from any other colorimetric space. Gamut compression becomes one of the more significant tasks for any such conversion.

APPENDIX HH

# Part 3: Appendices

Part 3 contains additional information that is not part of the TIFF specification, but may be of use to developers.

APPENDIX HH

# Appendix A: TIFF Tags Sorted by Number

| TagName | Decimal | Hex | Type | Number of values |
|---|---|---|---|---|
| NewSubfileType | 254 | FE | LONG | 1 |
| SubfileType | 255 | FF | SHORT | 1 |
| ImageWidth | 256 | 100 | SHORT or LONG | 1 |
| ImageLength | 257 | 101 | SHORT or LONG | 1 |
| BitsPerSample | 258 | 102 | SHORT | SamplesPerPixel |
| Compression | 259 | 103 | SHORT | 1 |
| Uncompressed | 1 | | | |
| CCITT 1D | 2 | | | |
| Group 3 Fax | 3 | | | |
| Group 4 Fax | 4 | | | |
| LZW | 5 | | | |
| JPEG | 6 | | | |
| PackBits | 32773 | | | |
| PhotometricInterpretation | 262 | 106 | SHORT | 1 |
| WhiteIsZero | 0 | | | |
| BlackIsZero | 1 | | | |
| RGB | 2 | | | |
| RGB Palette | 3 | | | |
| Transparency mask | 4 | | | |
| CMYK | 5 | | | |
| YCbCr | 6 | | | |
| CIELab | 8 | | | |
| Threshholding | 263 | 107 | SHORT | 1 |
| CellWidth | 264 | 108 | SHORT | 1 |
| CellLength | 265 | 109 | SHORT | 1 |
| FillOrder | 266 | 10A | SHORT | 1 |
| DocumentName | 269 | 10D | ASCII | |
| ImageDescription | 270 | 10E | ASCII | |
| Make | 271 | 10F | ASCII | |
| Model | 272 | 110 | ASCII | |
| StripOffsets | 273 | 111 | SHORT or LONG | StripsPerImage |
| Orientation | 274 | 112 | SHORT | 1 |
| SamplesPerPixel | 277 | 115 | SHORT | 1 |
| RowsPerStrip | 278 | 116 | SHORT or LONG | 1 |
| StripByteCounts | 279 | 117 | LONG or SHORT | StripsPerImage |
| MinSampleValue | 280 | 118 | SHORT | SamplesPerPixel |
| MaxSampleValue | 281 | 119 | SHORT | SamplesPerPixel |
| XResolution | 282 | 11A | RATIONAL | 1 |
| YResolution | 283 | 11B | RATIONAL | 1 |
| PlanarConfiguration | 284 | 11C | SHORT | 1 |
| PageName | 285 | 11D | ASCII | |
| XPosition | 286 | 11E | RATIONAL | |
| YPosition | 287 | 11F | RATIONAL | |
| FreeOffsets | 288 | 120 | LONG | |
| FreeByteCounts | 289 | 121 | LONG | |
| GrayResponseUnit | 290 | 122 | SHORT | 1 |

APPENDIX HH

| | | | | |
|---|---|---|---|---|
| GrayResponseCurve | 291 | 123 | SHORT | 2**BitsPerSample |
| T4Options | 292 | 124 | LONG | 1 |
| T6Options | 293 | 125 | LONG | 1 |
| ResolutionUnit | 296 | 128 | SHORT | 1 |
| PageNumber | 297 | 129 | SHORT | 2 |
| TransferFunction | 301 | 12D | SHORT | {1 or SamplesPerPixel}* 2** BitsPerSample |
| Software | 305 | 131 | ASCII | |
| DateTime | 306 | 132 | ASCII | 20 |
| Artist | 315 | 13B | ASCII | |
| HostComputer | 316 | 13C | ASCII | |
| Predictor | 317 | 13D | SHORT | 1 |
| WhitePoint | 318 | 13E | RATIONAL | 2 |
| PrimaryChromaticities | 319 | 13F | RATIONAL | 6 |
| ColorMap | 320 | 140 | SHORT | 3 * (2**BitsPerSample) |
| HalftoneHints | 321 | 141 | SHORT | 2 |
| TileWidth | 322 | 142 | SHORT or LONG | 1 |
| TileLength | 323 | 143 | SHORT or LONG | 1 |
| TileOffsets | 324 | 144 | LONG | TilesPerImage |
| TileByteCounts | 325 | 145 | SHORT or LONG | TilesPerImage |
| InkSet | 332 | 14C | SHORT | 1 |
| InkNames | 333 | 14D | ASCII | total number of charac ters in all ink name strings, including zeros |
| NumberOfInks | 334 | 14E | SHORT | 1 |
| DotRange | 336 | 150 | BYTE or SHORT | 2, or 2* NumberOfInks |
| TargetPrinter | 337 | 151 | ASCII | any |
| ExtraSamples | 338 | 152 | BYTE | number of extra components per pixel |
| SampleFormat | 339 | 153 | SHORT | SamplesPerPixel |
| SMinSampleValue | 340 | 154 | Any | SamplesPerPixel |
| SMaxSampleValue | 341 | 155 | Any | SamplesPerPixel |
| TransferRange | 342 | 156 | SHORT | 6 |
| JPEGProc | 512 | 200 | SHORT | 1 |
| JPEGInterchangeFormat | 513 | 201 | LONG | 1 |
| JPEGInterchangeFormatLngth | 514 | 202 | LONG | 1 |
| JPEGRestartInterval | 515 | 203 | SHORT | 1 |
| JPEGLosslessPredictors | 517 | 205 | SHORT | SamplesPerPixel |
| JPEGPointTransforms | 518 | 206 | SHORT | SamplesPerPixel |
| JPEGQTables | 519 | 207 | LONG | SamplesPerPixel |
| JPEGDCTables | 520 | 208 | LONG | SamplesPerPixel |
| JPEGACTables | 521 | 209 | LONG | SamplesPerPixel |
| YCbCrCoefficients | 529 | 211 | RATIONAL | 3 |
| YCbCrSubSampling | 530 | 212 | SHORT | 2 |
| YCbCrPositioning | 531 | 213 | SHORT | 1 |
| ReferenceBlackWhite | 532 | 214 | LONG | 2*SamplesPerPixel |
| Copyright | 33432 | 8298 | ASCII | Any |

APPENDIX HH

# Appendix B: Operating System Considerations

## *Extensions and Filetypes*

The recommended MS-DOS, UNIX, and OS/2 file extension for TIFF files is ".TIF".

On an Apple Macintosh computer, the recommended Filetype is "TIFF". It is a good idea to also name TIFF files with a ".TIF" extension so that they can easily imported if transferred to a different operating system.

APPENDIX HH

# Index

APPENDIX HH

APPENDIX II

September 11, 1996

# *FlashPix Format Specification*

**Version 1.0**

The information in this document is believed to be accurate as of the date of publication. However, *Kodak* will not be liable for any damages, including indirect or consequential, from use of this document.

**The *FlashPix*$^{TM}$ format is defined in a specification and a test suite, developed and published by *Eastman Kodak Company* in collaboration with *Hewlett-Packard Company, Live Picture Inc.* and *Microsoft Corporation*. Only products that meet the specification and pass the test suite may use the *FlashPix* file format name.**

For change requests, please send e-mail to “format_change_request@kodak.com”

A public listserver has been established for interested parties to share information and ideas regarding the *FlashPix* format. To subscribe to this listserver do the following:

1. Compose an e-mail message to “maillist@pixel.kodak.com”

2. The note should not contain a subject line and should have only one line in the body as follows:

subscribe FORMAT firstname lastname (i.e. subscribe FORMAT John Doe)

3. Whatever address you send this note from will be the address all listserver messages are sent to (i.e. john_doe@kodak.com)

4. A confirmation message will be sent back to you containing instructions on how to communicate with the listserver.

# Contents

## List of figures

## List of tables

APPENDIX II

September 11, 1996

# SECTION 1 *Introduction*

## 1.1 Purpose

This document is the technical specification that defines the image file format for *FlashPix* images. The effort to define the *FlashPix* format is a cooperative endeavor that includes the Eastman Kodak Company, Microsoft Corporation, the Hewlett-Packard Company, and Live Picture, Inc. The FlashPix format builds on the best features of existing formats (Kodak Image Pac, Live Picture IVUE, Hewlett-Packard JPEG, TIFF, TIFF/EP, and so on), and combines these features with an object orientation and other powerful new capabilities.The *FlashPix* format enables the industry to deliver desktop computer solutions that make it easy, enjoyable, and commonplace to use digital color photographic images in offices and homes.

## 1.2 Specification Organization

This document is divided into several sections, each isolating one aspect of the *FlashPix* specification. The sections are as follows:

- *Section 1: Introduction* defines structured storage.
- *Section 2: Image Data Representation* describes the resolution hierarchy and the organization of image data into tiles.
- *Section 3: The FlashPix Image Object* describes the format, in terms of Structured Storage, of the *FlashPix* image object.

- *Section 4: Image Data Format* describes the format of the individual streams used to store the actual image data.
- *Section 5: Color Space Specifications* defines the PhotoYCC and NIF RGB color spaces.
- *Section 6: Image Info Property Set* describes the non-image information in a *FlashPix* image.
- *Section 7: FlashPix Image View Object* defines the *FlashPix* image view object. This object allows a default view (including orientation, crop and color adjustment) to be specified for a *FlashPix* image without modifying pixel values.
- *Appendix A: Structured Storage* defines the binary format of structured storage files and relevant data structures.

## 1.3 Conventions

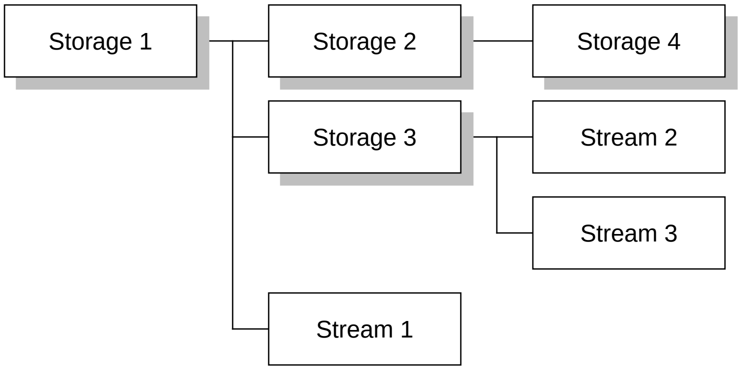
The following conventions are used in this document.

- All numbers starting with a “0x” are in hexadecimal.
- Special characters in strings are indicated using the octal format, where any three digit number preceded by a “\” represents the ASCII value of the character in octal. For example, a newline character would be represented by “**\012**.”
- Spaces in strings are explicitly indicated using their octal representation, “**\040**.”
- Stream and storage names are specified in standard C language “printf” syntax.

## 1.4 Structured Storage

The *FlashPix* format is based on a compound object storage model called structured storage. A file in structured storage format contains two types of objects: storages and streams. Storages are analogous to directories in a file system; streams are analogous to files. A storage may contain both zero or more additional storages and zero or more streams. The streams and storages in a *FlashPix* file are individually addressable. Figure 1.1 shows the convention used in this document to illustrate storages and streams.

**FIGURE 1.1** **Conventions for storages and streams in illustrations**



The entire structured storage file appears in the host file system as one file. In this example, storage 1 represents the root storage. It is the highest level storage of the file and is the entity that is visible in the host file system. It contains two storages (2 and 3) and one stream (1). Storage 2 contains one empty storage (4). Storage 3 contains two streams (2 and 3).

All entities in a *FlashPix* file have a class ID that identifies the type of the object. Class IDs are defined as globally unique identifiers (GUID's). They are represented as 128 bit numbers that are considered impossible to duplicate. GUID's are generated using the algorithm specified for the generation of universal unique identifiers for remote procedure calls[17].

## 1.4.1 Property Sets

Structured storage defines property sets as a stream for storing tagged data. The *FlashPix* format uses this mechanism extensively for storing data other than actual pixel values.

As defined, property sets are very flexible. All property sets must be in Windows Format. Windows format is indicated in the property set header by setting the wByteOrder field to 0xFFFE and the wFormat field to 0x0. Furthermore, the codepage must be written into the requisite property (PID = 1) in each and every FlashPix property set as described below. A binary specification of property sets is include in this specification in Section A.2.

With the sole exception of the OLE standard Summary Information Property Set each and every *FlashPix* property set must be in the Unicode (1200) codepage, and all strings in that set must be stored as wide 16-bit characters (LPWSTR). Due to its origin and use in non-*FlashPix* applications, the Summary Information Property set has different conditions than all other property sets. This property set must be in the Western European ANSI (1252) code page, and all strings in it must be stored as 8-bit characters (LPSTR). For legacy reasons, all *FlashPix* property set readers must be able to handle a 1200 codepage in the Summary Information Property set as if it was 1252. (Note - this last restriction will not introduce any character shifts since the 8-bit subsection of the Unicode codepage is exactly the 1252 codepage.)

The properties defined for each property set are listed in the property set definition. All property ID codes not explicitly listed for the property set are reserved for registered extensions. Where valid property values are listed, those not explicitly listed are reserved for registered extensions.

## 1.4.2 Summary Information Property Set

| | |
|---|---|
| Stream name: | **\005**SummaryInformation |
| Class ID: | F29F85E0-4FF9-1068-AB91-08002B27B3D9 |
| Format ID: | F29F85E0-4FF9-1068-AB91-08002B27B3D9 |

Structured storage defines one property set that can be found in every *FlashPix* object to provide a basic level of information about the object. This summary information property set is used in *FlashPix* image objects and *FlashPix* image view objects. This property set is in general use in OLE. It must be in the Western European ANSI (1252) codepage. Because of this restriction, this property set is not truly internationalizable in version 1.0 of the *FlashPix* specification.

The property set must also have at least one section that has a format ID the same as the class ID. The properties of the summary information property set are listed in Table 1.1.

**Title property (optional)**
This property is available for the application to record the object title.

**Subject property (optional)**
This property is available for the application to record the subject of the object.

**Author property (optional)**
This property is available for the application to record the author of the object.

**Keywords property (optional)**
This property is available for the application to record keywords about the object.

**Comments property (optional)**
This property is available for the application to record comments about the object.

**Template property (optional)**
This property is not used with *FlashPix* objects.

**Last saved by property (optional)**
This property is available for the application to record the name of the user who last saved the object.

**Revision number property (optional)**
This property is available for the application to record the number of times the object has been saved.

**TABLE 1.1** **Valid properties of the summary information property set**

| Property Name | ID Code | Type |
|---|---|---|
| Title | 0x00000002 | VT_LPSTR |
| Subject | 0x00000003 | VT_LPSTR |
| Author | 0x00000004 | VT_LPSTR |
| Keywords | 0x00000005 | VT_LPSTR |
| Comments | 0x00000006 | VT_LPSTR |
| Template | 0x00000007 | VT_LPSTR |
| Last saved by | 0x00000008 | VT_LPSTR |
| Revision number | 0x00000009 | VT_LPSTR |
| Total editing time | 0x0000000A | VT_FILETIME |
| Last printed | 0x0000000B | VT_FILETIME |
| Create time/date | 0x0000000C | VT_FILETIME |
| Last saved time/date | 0x0000000D | VT_FILETIME |
| Number of pages | 0x0000000E | VT_I4 |
| Number of words | 0x0000000F | VT_I4 |
| Number of characters | 0x00000010 | VT_I4 |
| Thumbnail | 0x00000011 | VT_CF |
| Name of creating application | 0x00000012 | VT_LPSTR |
| Security | 0x00000013 | VT_I4 |

**Total editing time property (optional)**

This property is available for the application to record the duration of an object editing session.

**Last printed property (optional)**

This property is available for the application to record when the object was last printed.

**Create date/time property (optional)**

This property is available for the application to record the creation date and time for the object. This value should not be updated after it is initially written.

**Last saved date/time property (optional)**

This property is available for the application to record the date and time that the object is saved. It is strongly recommended that this property be used in *FlashPix* objects.

**Number of pages property (optional)**

This property is not used in *FlashPix* objects.

**Number of words property (optional)**

This property is not used in *FlashPix* objects.

#### Number of characters property (optional)

This property is not used in *FlashPix* objects.

#### Thumbnail property (required in some situations)

This property is available for the application to record a small bitmap representation of the *FlashPix* object. For the *FlashPix* image view object, the thumbnail property is optional for the summary information property set if this *FlashPix* image view points to a non-hierarchical source image object written in an embedded capture environment. The thumbnail is otherwise required for *FlashPix* image view objects. The thumbnail is optional for the source and result *FlashPix* image objects.

- Thumbnail data should reflect image contents within the thumbnail format limits and must be oriented the same way as the object it is contained in. Refer to Sections 3.1.2 and 7.1 for additional information about how the thumbnail property is used.
- The thumbnail image is stored in CF_DIB format which is a simple rectangular array of pixels with a small header as defined in [23].
- For single channel images (including opacity only images), treat them as monochrome without an opacity channel for purposes of the thumbnail.
- For multicolored images, all pixels are stored in 24 bit (bi.BitCount = 24) BGR format (in the NIF RGB color space). For single channel images, all pixels are stored in 8 bit (bi.BitCount = 8) format.
- Palettized color representations are not allowed for 24 bit DIB's. However, for single channel thumbnails, a palette entry must be provided which serves as the 8 to 24 bit identity lookup table. It is highly suggested that this palette be a pure grayscale ramp of exactly 256 RGBQUAD elements (e.g. biClrUsed = 0) running from black to white. The palette should consist of a sequence of 256 32-bit RGBQUAD structures [x,x,x,0] for all x running from 0 to 255. Note that DIB palettes require the fourth (reserved) channel to be identically zero as defined in [23].
- The thumbnail image data is stored uncompressed.
- For images with an opacity channel in addition to image data channels, the thumbnail should be stored as if it had been composited on a fully opaque white background.
- The larger of the thumbnail stored height and width must be 96 pixels. The image should be resized to this dimension instead of padding a smaller image. It is not required to pad the smaller dimension to 96 pixels.

#### Name of creating application property (optional)

This property is available to the application to record the name of the application that created the object. It is strongly recommended that this property be used in *FlashPix* objects.

#### Security property (optional)

This property is not used in *FlashPix* objects.

## 1.4.3 File identification

A *FlashPix* file must be an image view object and its object type class ID must be stored in the root storage header. Many object based systems (e.g. OLE) will use the class ID found in the header of the root storage as a key for launching an application. In this way an application can be designated to handle all files of this object type by default, regardless of their creator.

The *FlashPix* image view object and *FlashPix* image object storages are required to have a CompObj type stream. Their object type class ID is required to be stored in the clipboard format field of that stream as well as in the header of the storage. The clipboard format field is what should be used to determine the class ID of these objects.

In non-OLE environments, *FlashPix* format files are identified by other means; including file name extensions and file types.

Macintosh systems use a file type designation to identify file content [24]. The file type is a 4-character code stored with each file. For example, the code 'TEXT' indicates that the file contains ASCII text. The Macintosh also associates a "file creator" with each file. This is also a 4-character code. It identifies the application that created the file.

These two codes are stored in the Finder Desktop Database [24]. When a file is double-clicked, the Finder uses the file's creator ID to determine its associated application. On the Macintosh, *FlashPix* files have been registered with a file type of "FPix" (Hex) 46506978.

On platforms (e.g., UNIX) that do not support GUIDs or file types as a means of associating files and applications/components, file extensions should be used. These are specified by the user as a period (".") followed by up to 3 letters. For example, an extension of (".FPX") can be used to indicate that the file contains a *FlashPix* file.

## 1.4.4 OS-level file treatment in Windows or with OLE

It is recommended that in Windows or OLE-enabled environments, core *FlashPix* files be managed independent of their creating application and that the user receives some control over which *FlashPix* reader becomes the server for *FlashPix* files.

The former can be accomplished by writing and saving core *FlashPix* files using the class ID of the *FlashPix* object type in the header portion of the root storage. Such files should also use a ‘.FPX’ extension. Cases where it is more appropriate to use the creating application’s class ID as the root storage class ID include: images with significant use of an application’s extensions, user approval via explicit prompting or preference setting, or for images used in a closed system.

In OLE-enabled environments, core *FlashPix* reader applications should be able to register as the server application for the *FlashPix* image view object class ID. If all core *FlashPix* readers did this, the last installed application would become the default server application. Due to the confusion this can cause the user, it is strongly recommended that the installation procedure for a core *FlashPix* reader application offer it as the default *FlashPix* application. The installation would not insert the new application as the server of the *FlashPix* image view object class ID unless the user agreed. Whenever possible, the installation procedure would also register the application as serving the .FPX extension.

## 1.4.5 *FlashPix* Streams

*FlashPix* class IDs are used to identify the type of an entity. Unfortunately, standard structured storage streams do not have a class ID. To deal with this problem a *FlashPix* stream differs from an OLE stream in that the first 28 bytes of the actual stream contain header information, part of which is the class ID of the stream. These bytes are not counted when determining offsets into the stream nor when determining the stream length. For example, the first byte of data in a *FlashPix* stream is stored in byte 28 of the actual stream (the first byte is byte 0). These 28 bytes are in the format of a property set header, as defined in Section A.2.1.1.

Note that property sets, on the other hand, do have a class ID field, and thus do not need modification. Property sets in the *FlashPix* format are not stored as *FlashPix* streams, but as standard streams. However, the header of a *FlashPix* stream is the same as the header of a property set.

## 1.4.6 String and Character Representation

There are numerous fields in the *FlashPix* format that contain character strings. These may be broken down into two general classes: structured storage related names such as stream and storage pathnames, and descriptive strings stored in property sets such as image title, film type and keywords.

To promote the ability to transfer *FlashPix* images internationally, nearly all strings in the *FlashPix* format are stored in the Unicode format. However, to promote the ability to transfer *FlashPix* files among different operating systems, all structured storage related names are required to be in the 7-bit ASCII compatible part of Unicode. Descriptive strings are allowed a much wider range, since they can be ignored outside the language they were generated in.

### 1.4.6.1 Storage and Stream Names

The *FlashPix* file is built on top of structured storage files, which in turn are built on top of the host file system. Hence, it must follow the conventions placed on stream and storage names, and the file system names.

Each structured storage file has a single root storage. The name of the file root storage may be any valid storage name. However, it is recommended that the file name in the host file system be used.

Names of *FlashPix* streams contained within storage objects are managed by the implementation of the particular storage object in question. Names are stored case-preserving, but are compared case-insensitive. As a result, all *FlashPix* writers which define storage and stream names must choose names which will work in either situation. *FlashPix* streams and storage names may be up to 31 characters in length.

Although storage and stream names are actually stored as 16-bit Unicode characters, the characters must be within the lower 7-bit ASCII range with the following additional restrictions:

- The names "." and ".." are reserved for future use.
- The four characters "\", "/", ":", or "!" are not allowed.

Restricting such string names to 7-bit ASCII greatly promotes interoperability across different platforms without significantly impacting internationalization, as these stream and storage names are rarely exposed to the user.

In addition, the name space in a storage is partitioned into different areas of ownership. Different pieces of code have the right to create elements in each area of the name space:

- Storage and stream names beginning with characters "**\001**" through "**\004**" are reserved for OLE.
- Property set names must begin with the "**\005**" character.
- Storage and stream names beginning with character "**\006**" through "**\037**" are reserved for OLE or for future use.
- Any other character may be used to begin a storage or stream name.

### 1.4.6.2 Property Set Code Page and Strings

The other major area string names kept in the *FlashPix* format are in property sets. Property sets are defined with a "code page" specifying the type of characters allowed, and numerous strings. All strings in the property set share the same code page.

All property sets in the *FlashPix* format must belong to the Unicode code page, as specified in Section 1.4.1. The code page ID must be stored in property ID 0x00000001 in all *FlashPix* property sets. These property sets are defined in the rest of this document. Hence it is an error to not specify the code page, or to specify it with any other value than 0x04B0, in any property set in a *FlashPix* file, as described in Section A.2.2.2. However, applications may write application specific property sets in a *FlashPix* file. Only the stream names for the private extensions are not required to be in the Unicode code page, although this is strongly recommended. Note that the count at the start of VT_LPWSTR is to be interpreted as a character count, not a byte count. The count includes the null character at the end of the string.

Descriptive strings such as names, camera types, and subjects are not required to be within the 7-bit ASCII subset of Unicode. They are more often displayed and manipulated by the user, and it is a requirement that the storage of the full character set is supported.

However, it is not required that a non-internationalized application be able to display and manipulate characters outside the ASCII and local character set. It is perfectly acceptable for a German *FlashPix* reader/writer, for instance, to display question marks or similar symbols if passed Kanji characters. However, it is required that all Unicode characters be preserved on copying. If that string was not modified by the German user, it must remain readable under a Japanese version of the application.

String properties may be stored as either 8-bit strings (VT_LPSTR) or 16-bit strings (VT_LPWSTR). Note because of Unicode compatibility, the upper 128 elements in an 8-bit string fall into the Western-European 8-bit character range in the Unicode code page[20]. Hence, strings in those languages may be adequately stored as 8-bit strings. Characters from other languages (Greek, Russian, Japanese, for example) must be stored in 16-bit strings.

All *FlashPix* property set readers must be able to read either 8 or 16 bit (Unicode) strings for any property. This imposes a requirement that all property set readers be able to convert 8 or 16 bit strings into the desired internal format for the application. It is advised that 16-bit strings be utilized whenever possible.

## 1.5 Format Compliance

The *FlashPix* image format has a core definition and defined extensions. *FlashPix* files, reader software, and writer software must be compliant with the core definition and may optionally support one or more extensions.

The core *FlashPix* format definition specifies the required and optional data elements and allowed data values that compose *FlashPix* files and default actions of *FlashPix* reader software:

- Core *FlashPix* files must contain all required core *FlashPix* data elements and any of the core *FlashPix* optional data elements using only those values enumerated in the core *FlashPix* definition.
- Core *FlashPix* reader software must read all valid core *FlashPix* file permutations and take all default actions defined in the core *FlashPix* specification.
- Core *FlashPix* writer software must write at least one valid core *FlashPix* file permutation.

Extensions to the core *FlashPix* format may be defined to add features that are not supported in the core *FlashPix* definition. Each extension must be defined in a way which does not prevent core *FlashPix* reader software from productively interpreting *FlashPix* files with the extension present. Core *FlashPix* reader software that doesn’t support a particular extension will ignore its added data elements and will resort to default actions when non-core values are present in core data elements.

Extensions to the *FlashPix* format are characterized by the feature capability being added and defined by the required and optional data elements and values associated with the feature. An extended *FlashPix* file meets the definition of core *FlashPix* files, but has additional data elements for the extensions present in the file. Extended *FlashPix* reader software is core *FlashPix* reader software with additional capability to productively interpret one or more defined *FlashPix* extensions. Extended *FlashPix* writer software is core *FlashPix* writer software with additional capability to create the data elements associated with one or more defined *FlashPix* extensions.

*FlashPix* extensions may either be registered or private. Registered extensions are collaboratively defined via the the *FlashPix* format Advisory Council and published publicly as separate documents from the core *FlashPix* format specification. Private extensions are defined by any interested party and shared at their discretion.

## 1.6 *FlashPix* File Overview

A core *FlashPix* file is composed of a *FlashPix* image view object which contains scriptable image transforms, a source *FlashPix* image object, and, optionally, a result *FlashPix* image object containing the result of applying the transforms to the source *FlashPix* image object. The *FlashPix* image object is defined in *Section 3: The FlashPix Image Object* and the remainder of the *FlashPix* image view object is defined in *Section 7: FlashPix Image View Object*. Those storages and streams in italics are optional or optional under specific circumstances.

Figure 1.2 is an overview of a *FlashPix* file, a *FlashPix* image view object. Figure 1.3 describes the content of each *FlashPix* image object in the *FlashPix* image view object and Figure 1.4 describes the content of each resolution storage in the *FlashPix* image objects.

**FIGURE 1.2** ***FlashPix*** **image view object**

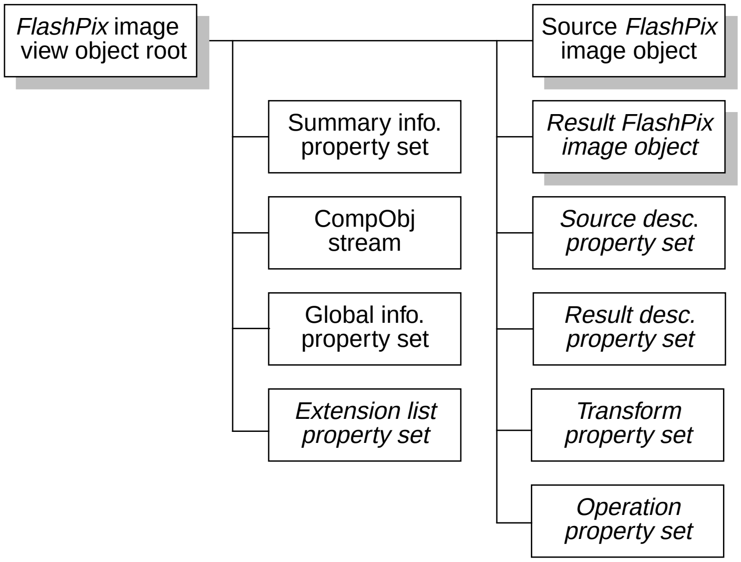

*FlashPix* image view object root
Summary info. property set
CompObj stream
Global info. property set
*Extension list property set*
Source *FlashPix* image object
*Result FlashPix image object*
*Source desc. property set*
*Result desc. property set*
*Transform property set*
*Operation property set*

**FIGURE 1.3** ***FlashPix*** **image object**

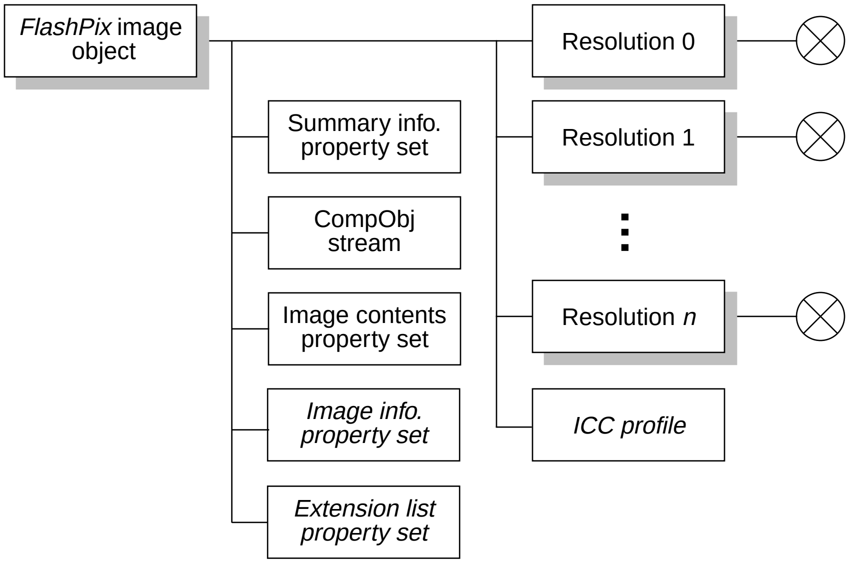
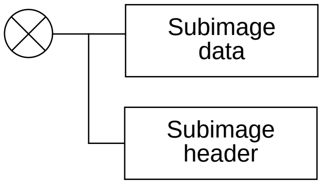

*FlashPix* image object
Summary info. property set
CompObj stream
Image contents property set
*Image info. property set*
*Extension list property set*
Resolution 0
Resolution 1
Resolution *n*
*ICC profile*

FIGURE 1.4 Contents of a resolution storage



## 1.6.1 Extension management

Both the *FlashPix* image view object and the *FlashPix* image objects may be independently extended. The extensions associated with each object are recorded in that object's extension list property set. Extensions to the *FlashPix* image view object may be located anywhere within the *FlashPix* file, even within a *FlashPix* image object. It is, however, not recommended that a *FlashPix* image view object extension place all of its data elements within a *FlashPix* image object. Extensions to a *FlashPix* image object must be entirely within the *FlashPix* image object.

*FlashPix* reader software must be able to interpret the extension list property sets and may also be an extended reader with the capability to interpret specific extensions which may be present in extended objects.

A particular *FlashPix* extension may not be valid if the core *FlashPix* data elements are edited. Each extension must have its persistence defined by the authoring application. The extension is marked as either: valid independent of any edits, invalid upon edits, or potentially invalid upon edits.

When opening a *FlashPix* file, the reader software should determine if there are any extensions present in the file which it does not support and which will become invalid or potentially invalid upon editing. If there are such extensions in the file, the user should be informed that the file is extended and that edits to the file may cause the loss of some of the extensions. For each extension that the reader software supports and which is marked as potentially invalid upon edits, the modification date of the extended object (in its summary info property set) must be compared to the extension modification date. If the extended object's modification date is more recent than that of the extension, the reader must determine if the extension is still valid. If it is not, it must either be updated or all of its data elements listed in the extension list property set must be deleted.

# SECTION 2 Image Data Representation

## 2.1 Coordinate systems

*FlashPix* files store image data in a hierarchy of resolutions from the highest available for an image down to the lowest defined in the format. Image editing operations need to be supported within an individual resolution, but must also be applied to all other resolutions.

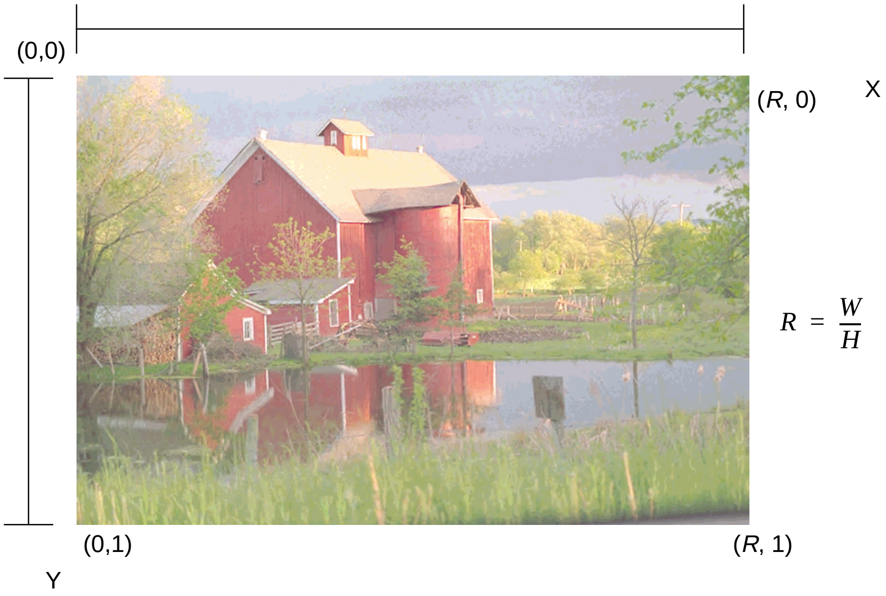
Therefore, two different coordinate systems are defined: resolution-independent and resolution-dependent.

### 2.1.1 Resolution-Independent Coordinates

In some situations, the image must be described by a coordinate system independent of the pixel.

Figure 2.1 shows a resolution-independent coordinate system. The image is described in a Cartesian system, with the X-axis horizontal and pointing to the right, the Y-axis vertical and pointing downward, and the origin at the upper left corner. The scale is such that the height of the image is normalized to 1.0. To keep the scale of the X-axis and the Y-axis the same, the image width is its aspect ratio (width/height). Thus, a square part of any image has equal width and height in this coordinate system.

**FIGURE 2.1** **Resolution-independent coordinates**

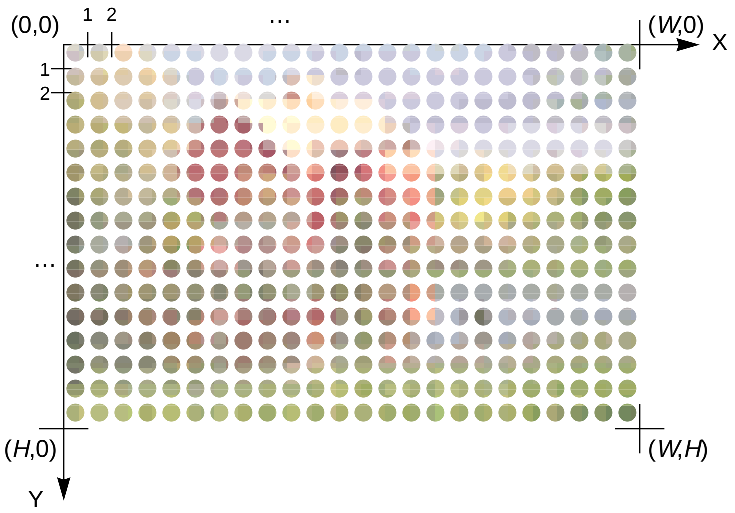


## 2.1.2 Resolution-Dependent Coordinates

At a given resolution, the normalized coordinate system described above must be converted to a set of discrete pixels. Then the continuous resolution-dependent coordinate system in Figure 2.2 is used. This is simply a scaled version of the previous coordinates. The values $(x, y)$ in this coordinate system are still real (floating point) numbers.

To define the actual pixels of the image, an integer grid is overlaid on the coordinate system. The discrete pixel referred to by $(i, j)$, where $i$ and $j$ are integers, is centered at location $(i+0.5, j+0.5)$. The half-unit shift makes the conversion between discrete and continuous descriptions simple. The point $(x, y)$ falls in the unit square labelled $(\lfloor x \rfloor, \lfloor y \rfloor)$ and containing the pixel at $(\lfloor x \rfloor+0.5, \lfloor y \rfloor+0.5)$. No rounding is required.

FIGURE 2.2 Resolution-dependent coordinates



## 2.2 Multiple resolutions

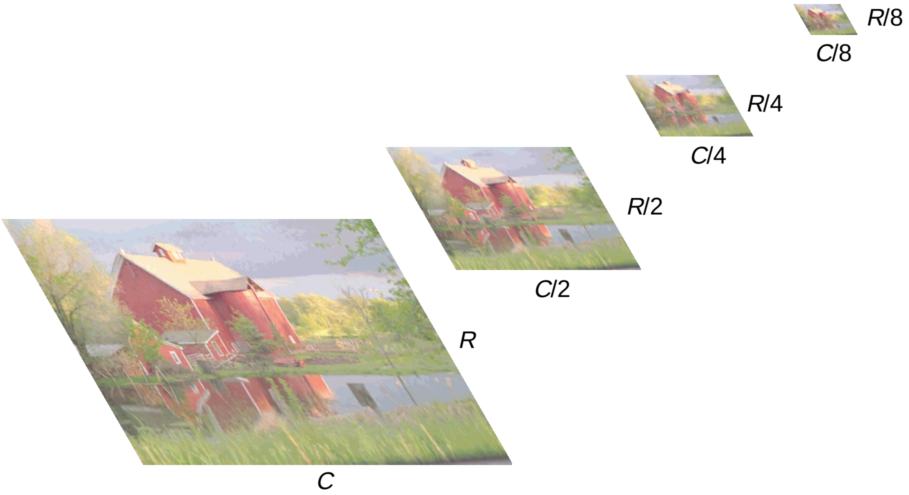
A *FlashPix* file must contain either a single resolution or the entire multi-resolution hierarchy. Each resolution in the full hierarchy is separated from the next higher resolution version by a spatial factor of 2× in both the *x* and *y* directions.

The series of resolutions continues until both the width and height of the smallest resolution are less than or equal to the width and height of a tile, 64 pixels. In the Figure 2.3 example, the tile width is smaller than *R*/4 but no smaller than *R*/8.

In Figure 2.3, the full resolution image is *R* rows × *C* columns. The actual spatial resolution (in pixels per inch, for example) is irrelevant, since neither the desired output size nor the output resolution is known. Each successively smaller resolution has half the number of rows and columns as the previous resolution. In this example, the second resolution is *R*/2 rows × *C*/2 columns.

Note that the *FlashPix* format uses centered subsampling. The pixels in resolution *i*-1 fall between the pixels in resolution *i*.

**FIGURE 2.3** **Sample resolution hierarchy**



## 2.2.1 Resolution sizes

The size of a decimated image is determined from equation 2.1, where $(w_0,h_0)$ is the width and height of the larger resolution and $(w_1,h_1)$ are the width and height of the smaller resolution:

$$(w_1, h_1) = \left(\left\lfloor \frac{(w_0+1)}{2} \right\rfloor, \left\lfloor \frac{(h_0+1)}{2} \right\rfloor\right) \tag{2.1}$$

Note that this rounding affects the size of the image in resolution independent coordinates. The height of the largest resolution image is defined to be 1.0. Using the rounding method in Equation 2.1, the height of one resolution given the height of the next largest resolution can be determined as follows, where $h_0$ is the height of the larger resolution in resolution independent coordinates, $p_0$ is the height of the larger resolution in pixels, $h_1$ is the height of the next smaller resolution in resolution independent coordinates, and $p_1$ is the height of the next smaller resolution in pixels, as defined by Equation 2.1:

$$h_1 = \frac{2p_1}{p_0} \times h_0 \tag{2.2}$$

Failing to make this correction to the height and width of the image (in resolution independent coordinates) when dealing with resolutions other than the largest resolution may cause slight errors in the alignment of the multiple resolutions of the image.

### 2.2.2 Non-Hierarchical *FlashPix* Images

This specification also defines a non-hierarchical version of *FlashPix* images. In some cases, such as in the case of a digital camera, the system has neither the computing power to generate the full hierarchy nor the space to store the additional resolutions. In these cases, only the full-resolution image is stored. However, when this image can be used in an interactive manner, the full hierarchy must be constructed. This may happen in an acquire module accessing the device, in a separate converter program, or when an interactive core reader application opens the image and finds that it's hierarchy does not exist. A result image object, as defined in Section 7.1, also must be hierarchical when used in an interactive environment.

Non-hierarchical *FlashPix* images differ from fully hierarchical *FlashPix* images in that there is only one resolution. This distinction is specified in Section 3.1.5.1.

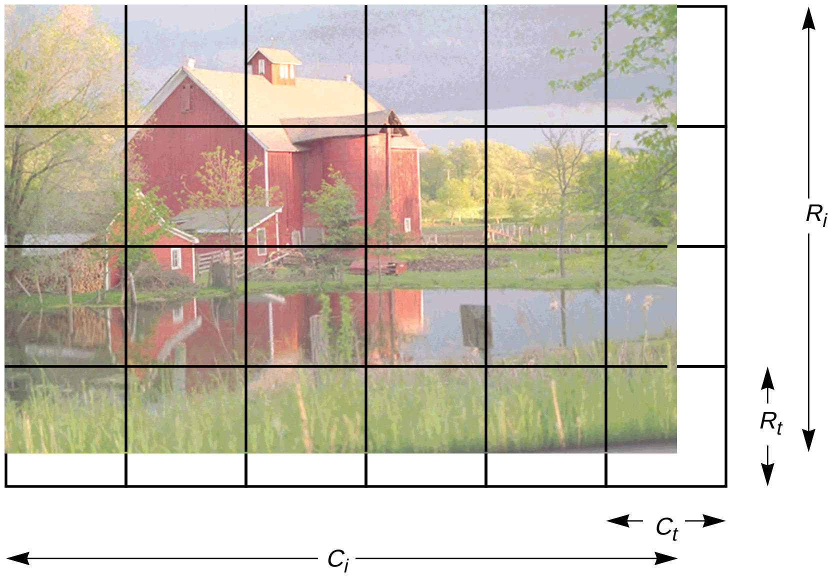
## 2.3 Tiling

In addition to providing the image at several resolutions, each resolution image is organized into tiles to provide more efficient access to any portion of the image.

### 2.3.1 Breaking an Image into Tiles

Figure 2.4 shows an image divided into tiles.

**FIGURE 2.4** **A tiled image**



The example image is $R_i$ rows × $C_i$ columns, organized into $R_t$ row × $C_t$ column tiles. For any image, the number of tiles per row ($N_R$) and the number of tiles per column ($N_C$) are:

$$N_R = \left\lceil \frac{R_i}{R_t} \right\rceil \qquad N_C = \left\lceil \frac{C_i}{C_t} \right\rceil \tag{2.3}$$

For the image shown in Figure 2.4, $N_R = 4$ and $N_C = 6$.

However, it is unlikely that the image size will be a multiple of the tile size. As illustrated in Figure 2.4, the tiles on the right and bottom of the image will only partially contain valid image data. Since only full tiles can be stored in a *FlashPix* image, incomplete tiles must be padded to the full tile width and height. Tiles should be padded with values extruded from the image itself. For example, pixels to the right of the image should be padded with the value of the rightmost pixel in each row. Pixels specifying opacity data should be padded just as if they were image data, as the actual image size is specified by the actual width and height, not by the opacity data.

To access a particular pixel, the tile containing that pixel must first be located. To calculate the tile, described by the coordinate pair ($T_C$, $T_R$), where the upper left tile is (0, 0) and the lower right tile is ($N_C$ - 1, $N_R$ - 1), containing the pixel ($c$+0.5, $r$+0.5), use the following formulas:

$$T_R = \left\lfloor \frac{r}{R_T} \right\rfloor \qquad T_C = \left\lfloor \frac{c}{C_T} \right\rfloor \tag{2.4}$$

The tiles are numbered sequentially, starting at the upper left tile and proceeding from left to right and top to bottom. The number of the tile ($T_C$, $T_R$), $T$, is given by the following formula:

$$T = T_R \times N_C + T_C \tag{2.5}$$

Once the correct tile has been located, the location ($c'$, $r'$) of the unit square containing the desired pixel, with respect to the tile, can be determined:

$$r' = r \% R_T \qquad c' = c \% C_T \tag{2.6}$$

where ‘%’ represents the modulus operator.

# SECTION 3 *The FlashPix Image Object*

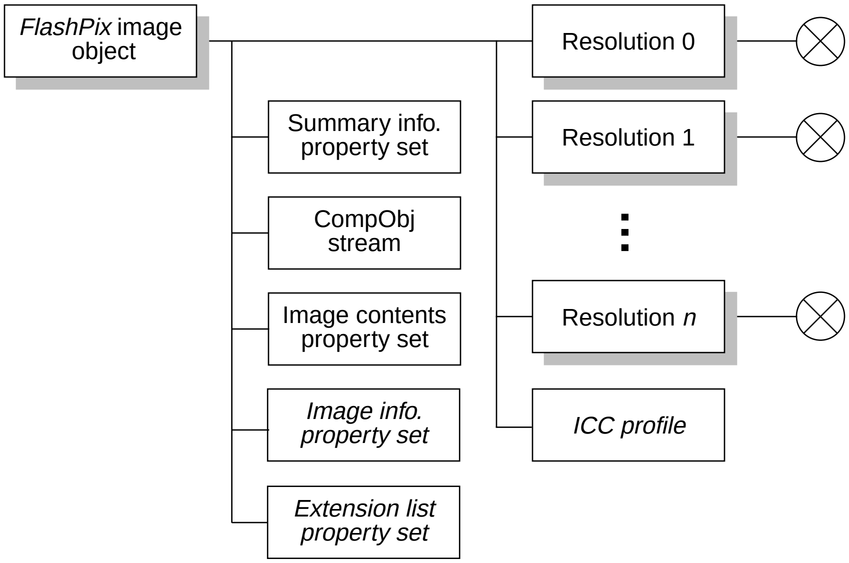
## 3.1 *FlashPix* Image Object Structure

The *FlashPix* image object is a storage whose contents are together treated as an image in the *FlashPix* format. Figure 3.1 shows the storages and streams in a *FlashPix* image object.

| | |
|---|---|
| Storage name: | Data\**040**Object\**040**Store\**040**%06d |
| Class ID: | 56616000-C154-11CE-8553-00AA00A1F95B |

The single numeric parameter in the storage name represents the index of the image as described in Section 7.1.2. This class ID is to be used for all *FlashPix* image objects whether or not they contain any extensions.

**FIGURE 3.1** ***FlashPix* image object storages and streams**

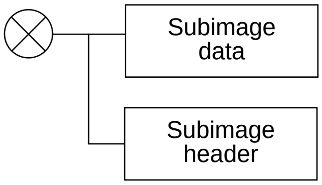

*FlashPix* image object
Summary info. property set
CompObj stream
Image contents property set
*Image info. property set*
*Extension list property set*
Resolution 0
Resolution 1
Resolution *n*
*ICC profile*

## 3.1.1 Resolution Storages

Storage name: Resolution**040**%04d
Class ID: 56616100-C154-11CE-8553-00AA00A1F95B

The decimal parameter in the storage name is the resolution number. Resolutions are numbered in sequence starting at 0, which is the lowest resolution.

**FIGURE 3.2** **Contents of a resolution storage**



This series of storages contains the image data for each resolution. Storage contents are the same for every resolution, as implied by the connection point in Figure 3.1 and Figure 3.2.

Future extensions to the *FlashPix* format may include additional subimages, but in the core definition, only one subimage is allowed. The format includes provisions for multiple subimages, which is accomplished by treating the one subimage in the core definition as subimage 0.

Each resolution storage contains a header stream and a data stream that contain all the image data for that resolution. (*Section 4: Image Data Format* describes the format of the header and data streams.) The subimage must be in the same color space and numerical format for all resolutions.

### 3.1.2 Summary Info Property Set (required)

This property set is an instance of the standard Summary Information property set, as described in Section 1.4.2. This property set must adhere to the definition specified in Section 1.4.1.

The thumbnail property value, if defined, must be representative of the image. Therefore, in the result instance of the image object, the thumbnail must have had all transforms applied to it.

### 3.1.3 CompObj Stream (required)

The CompObj stream is a standard Structured Storage stream and is not a *FlashPix* stream. The header of the stream is not extended for storage of a stream class ID. This stream is required and is defined in Section A.3. The Unicode versions of the CompObj stream fields are required.

The CompObj stream Clipboard Format field is used to store the class ID of the *FlashPix* image object. The *FlashPix* image object class ID is converted to a string for storage in the Clipboard Format field and must be bracketed by the bracket characters ‘{‘ and ‘}’ just as returned by the OLE function StringFromGUID2.

The CompObj stream User Type field is generally used to store the User Type information from the OLE registry for the class ID. In OLE-enabled environments, the string contents should be retrieved from the OLE registry. In non-OLE-enabled environments, a string which is a user-understandable brief description of the object contents should be used.

The CompObj stream ProgID field is generally used to store the ProgID information from the OLE registry for the class ID. In OLE-enabled environments, the string contents should be retrieved from the OLE registry. In non-OLE-enabled environments, a string which identifies the program associated with the class ID should be used. This string cannot contain any spaces.

## 3.1.4 Image Info Property Set (optional)

Stream name: **\005**Image**\040**Info
Class ID: 56616500-C154-11CE-8553-00AA00A1F95B
Format ID: 56616500-C154-11CE-8553-00AA00A1F95B

The image info property set contains properties that describe the actual image. (See *Section 6: Image Info Property Set* for a definition of this property set.)

## 3.1.5 Image Contents Property Set (required)

Stream name: **\005**Image**\040**Contents
Class ID: 56616400-C154-11CE-8553-00AA00A1F95B
Format ID: 56616400-C154-11CE-8553-00AA00A1F95B

The image contents property set contains properties that describe how the image data is stored. The properties may appear in any order, but they are conceptually divided into three groups. The *primary description group* describes the *FlashPix* image as a whole, specifying the number of resolutions, the size of the largest resolution, etc. The *resolution description group* describes the subimage at each resolution. The *compression description group* contains image compression information.

The image contents property set is stored in standard property set format, adhering to the restrictions in Section 1.4.1. All property ID codes not explicitly listed are reserved for registered extensions.

### 3.1.5.1 Primary description group

This group contains properties (Table 3.1) describing the *FlashPix* image object as a whole.

**TABLE 3.1** **Valid properties in the primary description group**

| Property name | ID Code | Type |
|---|---|---|
| Number of resolutions | 0x01000000 | VT_UI4 |
| Highest resolution width | 0x01000002 | VT_UI4 |
| Highest resolution height | 0x01000003 | VT_UI4 |
| Default display height | 0x01000004 | VT_R4 |
| Default display width | 0x01000005 | VT_R4 |
| Display height/width units | 0x01000006 | VT_UI4 |

**Number of resolutions property (required)**

This property specifies the number of resolutions contained in the *FlashPix* image. This property is required. The value must be the number of resolutions in the fully populated hierarchy or one for a non-hierarchical *FlashPix* image. That single stored resolution must be the highest resolution image. Its resolution number must be the number that would be assigned to the highest resolution image if the hierarchy were fully populated (number of resolutions property value - 1 since lowest resolution is stored as resolution 0). The number of resolutions stored for a fully populated hierarchy can be calculated as shown in Equation 3.1, where tile size is 64.

$$\left\lceil \log_2 \frac{max(width, height)}{TileSize} \right\rceil \tag{3.1}$$

**Highest resolution width and height properties (required)**

These properties specify in pixels the height and width of the highest resolution image. Values do not include the padded area if the image data is padded to tile boundaries.

**Default display height and width properties (optional)**

These properties specify the default height and width for displaying the image.

**Display height/width units property (optional)**

If the default display height and width properties are present, this property is used to define their unit of measurement. Legal values are listed in Table 3.2. If this property is not present, *FlashPix* reader software must treat the image as though the value were Inches (0x0).

**TABLE 3.2** **Legal display height/width units property values**

| Value | Meaning |
|---|---|
| 0x0 | Inches |
| 0x1 | Meters |
| 0x2 | Centimeters |
| 0x3 | Millimeters |

### 3.1.5.2 Resolution Description Groups

These groups, one for each stored resolution, contain properties to describe the subimage at that resolution. Table 3.3 lists the properties, where "*ii*" in the ID code is the resolution number.

**TABLE 3.3** **Valid properties in a resolution description group**

| Property name | ID code | Type |
|---|---|---|
| Subimage width | 0x02*ii*0000 | VT_UI4 |
| Subimage height | 0x02*ii*0001 | VT_UI4 |
| Subimage color | 0x02*ii*0002 | VT_BLOB |
| Subimage numerical format | 0x02*ii*0003 | VT_UI4 \| VT_VECTOR |
| Decimation method | 0x02*ii*0004 | VT_I4 |
| Decimation prefilter width | 0x02*ii*0005 | VT_R4 |
| Subimage ICC profile | 0x02*ii*0007 | VT_UI2 \| VT_VECTOR |

**Subimage width and height properties (required)**

These properties specify the width and height of subimage in pixels. Values do not include the padded area if the image data is padded to tile boundaries.

**Subimage color property (required)**

This property specifies the color of the subimage channels at this resolution. The format of the data portion of the VT_BLOB is shown in Table 3.4.

**TABLE 3.4** **Format and fields of the subimage color property**

| Field name | Length | Byte(s) |
|---|---|---|
| Number of subimages | 4 | 0-3 |
| Number of channels of subimage | 4 | 4-7 |
| Color of channel 0 of subimage | 4 | 8-11 |
| … | | |
| Color of channel *last* of subimage | 4 | variable |

Each color code is divided into two sections: the color space and the color. The upper 16 bits of the field specify the color space. The lower 16 bits specify the color.

Valid values for each of the color space subfields are shown in Table 3.5. If the most significant bit of the color space subfield is not set, then the image is calibrated to the color space as defined in *Section 5: Color Space Specifications*. If the most significant bit of the color space subfield is set, then the image channel definitions are that of the color space, but the image is not calibrated. Core reader software should provide a means for warning the application user that the image color is non-standard and unpredictable color results may occur.

**TABLE 3.5** **Valid color space subfield values**

| Value | Meaning |
|---|---|
| 0x0 | Colorless (CL) |
| 0x1 | Monochrome (M) |
| 0x2 | PhotoYCC (YCC) |
| 0x3 | NIF RGB (NRGB) |

Each color space is defined in *Section 5: Color Space Specifications*.

All the channels in the subimage must have the same color space value, including the most significant bit. It is not legal to create the subimage with a PhotoYCC luminance channel and a NIF RGB green channel. Core reader software must verify that the entire subimage color property value is the same for each resolution of the *FlashPix* image object.

If the subimage contains an opacity channel in addition to other color channels, the opacity channel color space codes should be that of the other channels in the subimage. For example, if an opacity channel is placed in the subimage with NIF RGB data, the complete color code for the opacity channel should be 0x00037FFE. If the subimage contains only an opacity channel, the color space should be colorless (0x0000). For example, an opacity channel alone in the subimage should have a complete color code of 0x00007FFE.

Valid values for each of the color subfields are shown in Table 3.6.

**TABLE 3.6** **Valid color subfield values**

| Color | Value | Allowed color spaces |
|---|---|---|
| Monochrome (M) | 0x0 | M |
| Red (R) | 0x0 | NRGB |
| Green (G) | 0x1 | NRGB |
| Blue (B) | 0x2 | NRGB |
| PhotoYCC luminance (Y) | 0x0 | YCC |
| PhotoYCC chrominance1 (C1) | 0x1 | YCC |
| PhotoYCC chrominance2 (C2) | 0x2 | YCC |
| Opacity (A) | 0x7FFE | all |

If the most significant bit of the color subfield (0x8000) is set, that channel is part of a subimage containing opacity data, which, in addition to being included as an additional channel, has been premultiplied into the color channels. If one color channel has been premultiplied, all channels except the opacity channel must be premultiplied. An opacity channel may never have the premultiplied flag set. In color character codes (such as R-NRGB), a lowercase "a" at the beginning of the character code indicates a channel with premultiplied opacity (for example, aR-NRGB). If the primary subimage contains an opacity channel, it must be premultiplied into the other channels.

Table 3.7 shows the valid subimage color value combinations. The channel color definitions are expected to appear in the order in which they are listed under Color codes in the table.

**TABLE 3.7** **Legal subimage color values**

| Description | Color codes |
|---|---|
| PhotoYCC | Y-YCC, C1-YCC, C2-YCC |
| PhotoYCC with premultiplied opacity | aY-YCC, aC1-YCC, aC2-YCC, A-YCC |
| NIF RGB | R-NRGB, G-NRGB, B-NRGB |
| NIF RGB with premultiplied opacity | aR-NRGB, aG-NRGB, aB-NRGB, A-NRGB |
| Monochrome | M-M |
| Monochrome with premultiplied opacity | aM-M, A-M |
| Opacity | A-CL |

#### Subimage numerical format property (required)

This property specifies the numerical formats of the image data at this resolution. The value and the image data may only be of the type VT_UI1 (8-bit unsigned integer). All channels in the subimage must have the same numerical format. Core readers must verify that the same subimage numerical format value is given for all channels of each resolution of each subimage.

#### Decimation method property (required)

This property characterizes the quality of the decimation performed to create the images at this resolution from the next higher resolution. If the value is 0x7FFFFFFF, this resolution was decimated from the next larger resolution using the following 8-point decimation prefilter:

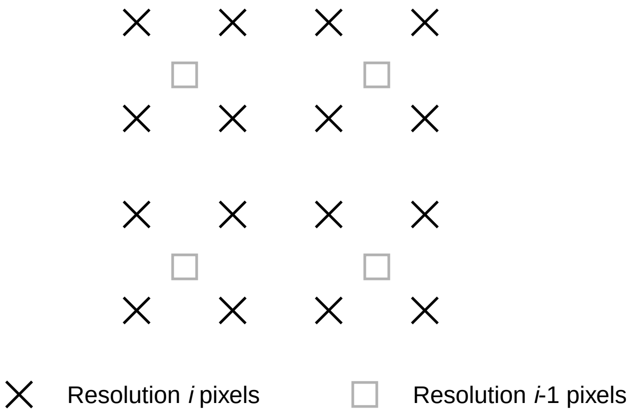
(-0.046734,-0.059009,0.156544,0.449199,0.449199,0.156544,-0.059009,-0.046734).

Other values indicate the number of elements in the prefilter. For example, if a 6-point filter was used to prefilter the image before decimation, this value would be 6. If this resolution is the full resolution image, the decimation method should have a value of zero.

If this resolution was artificially created by interpolating from a smaller resolution (it is not recommended for images to be interchanged with other applications), the value of the decimation method property should be negative. The absolute value is the width of the filter applied to sharpen the image prior to interpolating the data. Then -1 indicates a simple interpolation without sharpening.

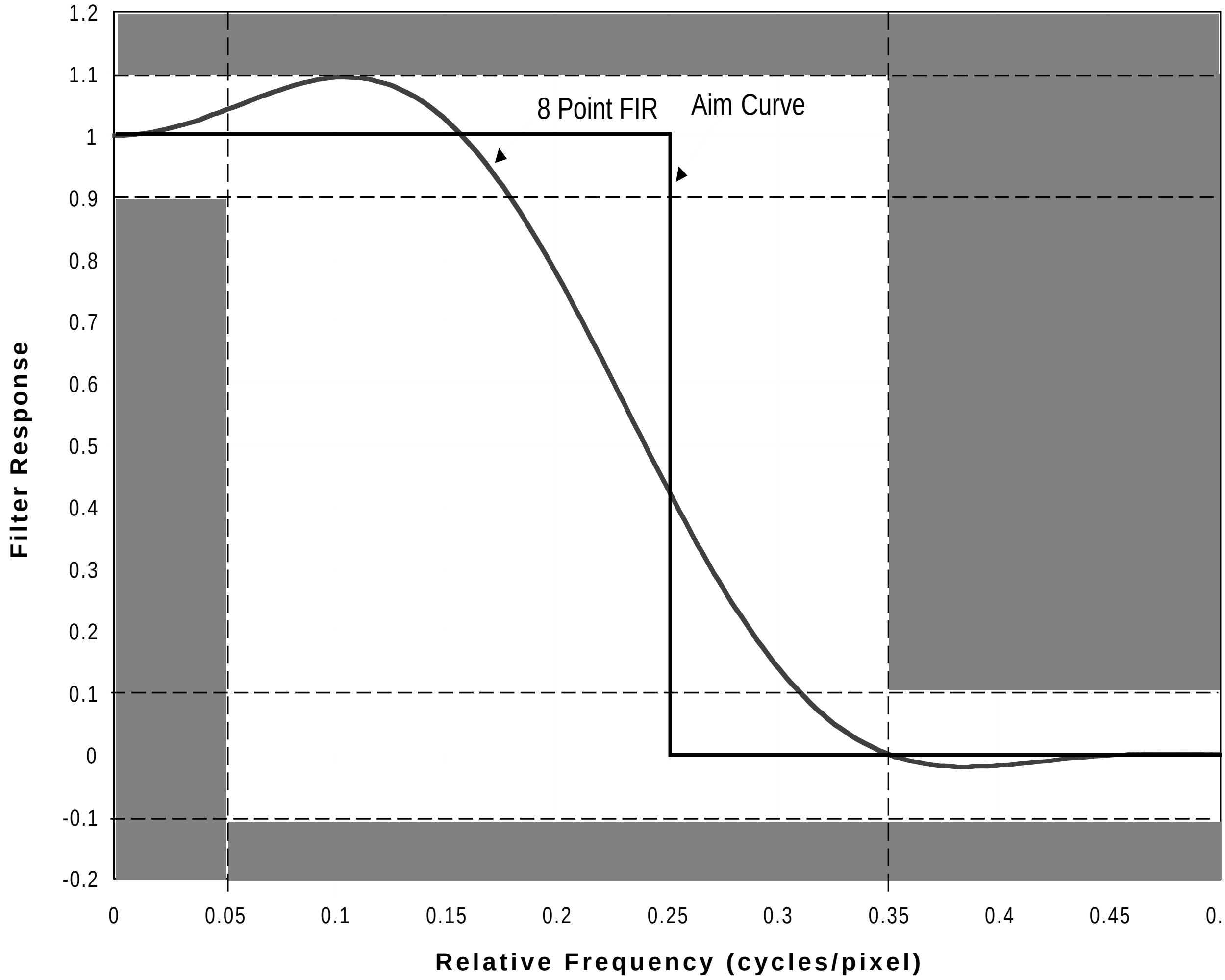
Correct decimation filter design depends on the choice of pixel location (Figure 3.3). The *FlashPix* format convention requires that the positions of pixels in a resolution layer (except for the full resolution layer) are offset by one half a pixel unit with respect to the resolution layer above. This can be obtained by using an even-width symmetric filter to prefilter the image before decimating. Note that most convolution code places the center of the filter at the location of the filtered pixel and that a nearest neighbor decimation (such as the Windows StretchBlt function) produces incorrect pixel alignment and should not be used.

**FIGURE 3.3** **Example of pixel-centered alignment between adjacent resolutions**



A developer may choose to design their own decimation prefilter. The filter must be a finite impulse response filter (FIR) and should be designed according to the aim frequency response curve and error bounds shown in Figure 3.4. The filter design process should achieve a prefilter frequency response curve which approximates the aim curve without ever entering the shaded region shown in the figure. A reader should assume that the image was decimated by a filter approximating this frequency response; and a writer should use a filter having the same aim frequency response.

**FIGURE 3.4** **Frequency response curve and error bounds**



**Decimation prefilter width property (optional)**

To perform resolution-independent image filtering, the algorithm must know the decimation prefilter degree of blurring, which is expressed as the effective filter width, *q*. This property specifies the value *q*.

The procedure is as follows: Approximate the prefilter MTF by the form $e^{-qs^2}$, where *q* is the width and *s* is the spatial frequency measured in cycles per pixel of the image before filtering. If the MTF is far from a Gaussian form, fit the low-frequency portion best. If this property is not found, it is assumed that the prefilter used had an aim frequency response as specified in the decimation method property.

**Subimage ICC profile property (optional)**

This property specifies an optional ICC profile for the subimage. It is a 1-element array whose value must be 1. The existence of this property indicates that an ICC profile exists in the *FlashPix* image object. The profile specifies the conversion of the subimage into the ICC-PCS (Profile Connection Space). If an ICC profile is identified for the subimage, the same profile must be associated with the subimage at all resolutions and core reader software must verify that this is the case.

### 3.1.5.3 Compression Description Group

This group of the image contents property set contains compression header information. Only those properties that contain valid data must be present.

Table 3.8 specifies the properties in the compression description group. In the table, "*ii*" in the ID code is the index of a JPEG table set selection.

**TABLE 3.8** **Valid properties in the compression information properties group**

| Property name | ID code | Type |
|---|---|---|
| JPEG tables | 0x03*ii*0001 | VT_BLOB |
| Maximum JPEG table index | 0x03000002 | VT_UI4 |

**JPEG tables property (optional)**

This property (as specified in Table 3.8) contains the JPEG quantization tables and the JPEG Huffman tables used across all resolutions of the *FlashPix* image. The format of the data in each of these properties should conform to the JPEG abbreviated format for table specification data, consisting of JPEG markers surrounding the actual table data, per the JPEG specification[9] Annex B.5 and Annex C. Note that it is possible to use quantizers or Huffman tables not defined here by including them in the JPEG data stream for the tile in which to apply them (as described in Section 4.1.2). The format of a JPEG abbreviated header table is shown in Table 3.9.

**TABLE 3.9** **Format and entries of a JPEG abbreviated header table**

| Field name | Length | Byte(s) | Value |
|---|---|---|---|
| Start of image marker (SOI) | 2 | 0-1 | 0xFFD8 |
| Define quantization table segment marker (DQT) | 2 | 2-3 | 0xFFDB |
| Quantization table data | variable | 4-variable | variable |
| Define Huffman table segment marker (DHT) | 2 | variable | 0xFFC4 |
| Huffman table data | variable | variable | variable |
| End of image marker (EOI) | 2 | variable | 0xFFD9 |

Each *FlashPix* image tile using JPEG compression must define at least one quantization table and two Huffman tables. A typical JPEG abbreviated table stream includes two quantization tables (numbered 0 and 1) for the luminance and chrominance components, and two Huffman tables (numbered 0 and 1) for the DC and AC entropy coder. The standard JPEG table specification syntax allows the definition of up to four quantization tables and four Huffman tables.

By storing multiple JPEG abbreviated header tables (each in a uniquely identified property), different sets of tables can be used by different tiles and multiple tiles can utilize the same table. Up to 255 table streams, with indices ranging from 1 to 255, may be defined in the compression property group.

#### Maximum JPEG table index property (optional)

This field specifies the maximum JPEG table index for the JPEG table properties. This property is optional, but must exist if there are any JPEG table properties in this *FlashPix* image object. It is recommended that when a JPEG table property is added that the index used be this property's value + 1. When a JPEG table property is added, the maximum JPEG table index property must be adjusted if the new index value in use is greater than the current property value.

### 3.1.6 ICC Profile (optional)

| | |
|---|---|
| Stream name: | ICC\**040**Profile\**040**0001 |
| Class ID: | 56616600-C154-11CE-8553-00AA00A1F95B |

This stream contains an ICC profile describing the conversion between the *FlashPix* image color space and the ICC PCS. The data portion of the *FlashPix* stream is stored in standard ICC profile format[8]. The ICC profile may contain only standard PhotoYCC to PCS or NIF RGB to PCS transforms.

### 3.1.7 Extension List Property Set (optional)

| | |
|---|---|
| Stream name: | \**005**Extension\**040**List |
| Class ID: | 56616010-C154-11CE-8553-00AA00A1F95B |
| Format ID: | 56616010-C154-11CE-8553-00AA00A1F95B |

This property set identifies extensions present in the *FlashPix* image object by class ID, name, and description as well as the data elements changed or added by each extension. The property set is optional, however, if the *FlashPix* image object contains any extensions, the extension list property set must be present and all extension, registered and private, in the *FlashPix* image object must be described.

The way in which the data associated with an extension is structured can take one or more of the following forms:

- New storage(s) may be added
- New stream(s) may be added
- New *FlashPix* stream(s) may be added
- New subimage(s) may be added to a *FlashPix* image object
- New property set(s) may be added
- New property(s) may be added to an existing property set section

- Element(s) may be added to core *FlashPix* property set vector properties that are defined as variable length
- Value of a core *FlashPix* stream field may be changed
- Value of a core property set property may be changed

There are five restrictions to structuring the data elements of an extension. First, new fields may not be added to existing *FlashPix* streams. Second, due to the inability to independently ensure property ID code uniqueness, only registered extensions may add properties to an existing property set section. Third, private extensions may not change the value of a core *FlashPix* stream field or a core property set property. Fourth, extensions can only add vector elements that are not already used by core or other extensions present in the file. Upon removal of an extension, the vector element values associated with the extension must be replaced with NULL and the vector must not be reordered. Fifth, only registered extensions can add elements to core property set vector properties.

Although there are a few practical examples where reasonable core reader actions could be defined for when an extension has changed the value of a core *FlashPix* stream field or a core property set property, these core reader actions must be considered in defining the core *FlashPix* specification. It is impractical to expect all core reader software to be updated to incorporate default actions identified in the course of developing new extensions. The definition of extensions must not impact the core definition unless some compelling feature set is identified which the *FlashPix* format Advisory Council agrees to include in a revised definition of the core *FlashPix* format. Therefore, efforts to define public extensions will avoid impacting core *FlashPix* stream field and core property set property values.

The valid properties of the extension list property set are listed in Table 3.10. The extensions present in the *FlashPix* image object are numbered for the convenience of grouping the descriptive information about each extension. Property ID codes 0xiiiixxxx describe the extension numbered 0x*iiii*.

**Used extension numbers property (required)**

This property lists all extension numbers iiii used in the extension list property set for the *FlashPix* image object. The property value is an unordered array of iiii values.

All applications must update this property each time an extension is added to or removed from a *FlashPix* image object.

**Extension name property (required)**

This property identifies the name of the extension. If the extension is registered, the name used must be that which is published in the official*FlashPix* Extension Specification. For private extensions, the name is the whatever short, descriptive label the authoring application chooses.

All applications must retain this property upon a save or copy function by default, or in accordance with the extension persistence.

**TABLE 3.10** **Valid properties for the extension list property set**

| Property name | ID code | Type |
|---|---|---|
| Used extension numbers | 0x10000000 | VT_UI2 \| VT_VECTOR |
| Extension name | 0x*iiii*0001 | VT_LPWSTR |
| Extension class ID | 0x*iiii*0002 | VT_CLSID |
| Extension persistence | 0x*iiii*0003 | VT_UI2 |
| Extension creation date | 0x*iiii*0004 | VT_FILETIME |
| Extension modification date | 0x*iiii*0005 | VT_FILETIME |
| Creating application | 0x*iiii*0006 | VT_LPWSTR |
| Extension description | 0x*iiii*0007 | VT_LPWSTR |
| Storage / stream pathname | 0x*iiii*1000 | VT_LPWSTR \| VT_VECTOR |
| *FlashPix* stream pathname | 0x*iiii*2000 | VT_LPWSTR \| VT_VECTOR |
| *FlashPix* stream field offset | 0x*iiii*2001 | VT_UI4 \| VT_VECTOR |
| Property set pathname | 0x*iiii*3000 | VT_LPWSTR \| VT_VECTOR |
| Property set ID codes | 0x*iiii*3jj1 | VT_LPWSTR \| VT_VECTOR |
| Property vector elements | 0x*iiii*3jj2 | VT_LPWSTR \| VT_VECTOR |
| Subimage number/resolution | 0xiiii4000 | VT_LPWSTR \| VT_VECTOR |

### Extension class ID property (required)

This property identifies a unique class ID for the extension. If the extension is registered, the class ID must be that which is published in the official*FlashPix* Extension Specification. For private extensions, the class ID is assigned by the authoring application.

All applications must retain this property upon a save or copy function by default, or in accordance with the extension persistence.

### Extension persistence property (required)

This property identifies the persistence of the extension with respect to edits to the core data elements of the *FlashPix* image object. The legal values for the extension persistence property are defined in Table 3.11.

All applications must retain this property upon a save or copy function by default, or in accordance with the extension persistence.

It is the responsibility of the reader/writer application upon save or copy functions to retain the extension data elements by default, or in accordance with the extension persistence property.

**TABLE 3.11** **Legal values of the existence persistence property**

| Value | Meaning |
|---|---|
| 0x0 | Extension is valid independent of core element modifications |
| 0x1 | Extension is invalid upon core element modifications |
| 0x2 | Extension is potentially invalid upon core element modifications |

**Extension creation date property (optional unless extension persistence property is 0x2)**
This property specifies the time and date the authoring application added the extension to the *FlashPix* image object. If the property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence value.

**Extension modification date property (optional unless extension persistence property is 0x2)**
This property specifies the time and date of the last modification to the extension. If the property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence value.

**Creating application property (optional)**
This property specifies the name of the application that authored the extension in the file. If the property exists, any application editing the extension must update the value and all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence value.

**Extension description property (optional)**
The description property is a short (<80 character) description of the extension. If the property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence value.

**Storage/stream pathname property (optional)**
This property lists the full storage or non-*FlashPix* stream name, including the path in the structured storage file from the *FlashPix* image object storage, for each storage or non-*FlashPix* stream the extension added to the *FlashPix* image object. The path is specified using the standard Unix file specification tokens: "/" represents a directory separator and must be the first character of the property value. Wildcard characters “*” and “?” (where “*” matches any 0 or more characters and “?” matches any 1 character) are permitted in the path portion of the property value. If a storage is listed in the extension list property set, its contents should not also be listed as they are assumed to also be associated with that extension. If this property is omitted it is assumed that no storages are added to the *FlashPix* image object for the extension. If the property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence property value.

***FlashPix* stream pathname property (optional)**
This property lists the full *FlashPix* stream name, including the path from the *FlashPix* image object storage, for each *FlashPix* stream the extension added to or modified in the

*FlashPix* image object. The path is specified using the standard Unix file specification tokens: "/" represents a directory separator and must be the first character of the property value. Wildcard characters “*” and “?” (where “*” matches any 0 or more characters and “?” matches any 1 character) are permitted in the path portion of the property value. The array of values for the *FlashPix* stream pathname property and the *FlashPix* stream field offset property array of values for extension iiii are associated as described in Table 3.12. If this property is omitted it is assumed that no *FlashPix* streams are added to or modified in the *FlashPix* image object for the extension. If the property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence property value.

**_FlashPix_ stream field offset property (optional)**

This property lists the byte offsets (after the header) into the *FlashPix* stream identified with the *FlashPix* stream pathname property array of fields modified by the extension. The array of values for the *FlashPix* stream field offset property and the *FlashPix* stream pathname property array of values for extension 0xiiii are associated as described in Table 3.12. This property is required only if the *FlashPix* stream pathname property exists. If this property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence property value.

In the Table 3.12 example, there are two *FlashPix* stream data elements associated with extension 0x17. The first, at Index = 0, is an added *FlashPix* stream as there is a *FlashPix* stream pathname value but the *FlashPix* stream field offset is 0xFFFFFFFF. The second, at index 0xjj = 1, is a field in a core *FlashPix* stream who's value is not among those defined in core the *FlashPix* format. This is indicated by the presence of a non-0xFFFFFFFF *FlashPix* stream field offset value in addition to a *FlashPix* stream pathname value.

**TABLE 3.12** **Example values of _FlashPix_ stream identification**

| Property | Index = 0 | Index = 1 |
|---|---|---|
| 0x00172000 | stream x pathname | stream y pathname |
| 0x00172001 | 0xFFFFFFFF | 64 |

**Property set pathname property (optional)**

This property is an array that lists the full property set name, including the path from the *FlashPix* image object storage, for each property set the extension 0xiiii added, added to, or modified in the *FlashPix* image object. The path is specified using the standard Unix file specification tokens: "/" represents a directory separator and must be the first character of the property value. Wildcard characters "*" and "?" (where "*" matches any 0 or more characters and "?" matches any 1 character) are permitted in the path portion of the property value.  Table 3.13 shows an example of how the property set pathname, property set ID codes, and property set vector elements for extension 0xiiii are associated. The array index of the property set pathname property corresponds to 0xjj in the properties 0xiiii3jj1 and 0xiiii3jj2.

This property set is optional and if omitted it is assumed that no property sets are added, added to, or modified in the *FlashPix* image object for the extensions. If the property exists all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence property value.

### Property set ID codes property (optional)

This property lists the ID codes of properties which have been added to a core property set, or defined with non-core values by an extension to the *FlashPix* image object. The value of each array posistion of the property is a VT_LPWSTR that may be composed of comma separated values each of which are either an individual property ID code or hyphen-separated pair of property ID codes. The array of values for the property set ID codes and the property vector elements for particular property set 0xjj and extension 0xiiii are associated as described in Table 3.13. When a new property set is added by an extension, the property set ID codes property is not required. This property is required if an extension adds properties to a core property set or modifies core property set properties. If the property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence property value.

### Property vector elements property (optional)

Extensions can add vector elements to core properties that are defined as variable length vectors. This property lists the vector index for the values added to a particular vector property. The value of each array posistion of the property is a VT_LPWSTR that may be composed of comma separated values each of which are either an individual vector element or hyphen-separated pair of vector elements. The array of values for the property vector elements and the property set ID codes for a particual property set 0xjj and extension 0xiiii are associated as described in Table 3.13. This property is only required when an extension adds vector elements to a core property set property. If the vector elements property is present, and vector elements have not been added to its associated property set ID code(s), then the value of this property must be NULL . If the property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence property value.

In the Table 3.13 example, there are three property sets associated with extension 0x19. The first index of the property 0x000193000, which corresponds to 0xjj=00, is a new property set being added by the extension as there is a property set pathname value, but the property set ID codes and property vector element properties for 0xjj=00 are not listed. The second index of the property 0x00193000, which corresponds to 0xjj=01, is a core property set in which property ID Codes 0x00011001-0x00011005 and 0x00001200 are being added by the extension. The third index of the property 0x00193000, which corresponds to 0xjj=02, is a core property set in which property ID code 0x00033000 is of type VT_VECTOR and the extension has added values in elements 3,4, and 5 of that vector. Property ID codes $00044001-$00044004 are new ID codes being added to the property set by the extension as well. In this case since the property ID codes are new, the value of 0x00193022 for this array posistion is assigned to NULL. This example also shows that the extension has added a value in element 2 of both the vectors defined by existing property ID codes, 0x00055000 and 0x00066000.

**TABLE 3.13** **Example values of property set identification**

| Property | Index=0 | Index=1 | Index=2 |
|---|---|---|---|
| 000193000 | PS x pathname (0xjj=00) | PS y pathname (0xjj=01) | PS z pathname (0xjj=02) |
| 000193011 | $000011001-$000011005, $000012000 | | |
| 000193021 | $000033000 | $000044001-$000044004 | $000055000, $000066000 |
| 000193022 | 3,4,5 | NULL | 2 |

### Subimage number/resolution property (optional)

This property is used to indicate that an extension has added new subimages to a *FlashPix* image object. It must be not used to indicate that an extension has modified core *FlashPix* subimage 0. The value of each array position is a VT_LPWSTR that is composed of comma-separate values each of which is either an individual subimage number or a hyphen-separated pair of subimage numbers. The array element position indicates the resolution indices to which the particular subimage(s) are added. If subimages are not added to a particular resolution, the value of the corresponding array element must be set to NULL.

The extension list must also list any additional storages, streams, property sets and properties added by the extension to store the subimage. This property is required if an extension adds a subimage to a *FlashPix* image object. All applications must retain this property upon a save or copy function by default, or in accordance with the extension persistence. Table 3.14 is an example of an extension that adds subimages to the *FlashPix* image object.This example shows that extension 0x49 has added subimages to resolution 0, 1, and 2 of the core *FlashPix* image object. The first element, at index 0xjj=0, shows that subimages 1 and 3 have been added to resolution 0x00. No subimages have been added to resolution 0x01. The third element, at index 0xjj=2, shows that the extension has added subimages 1, 3, and 4 through 6 to resolution 0x02.

**TABLE 3.14** **Example of subimage identification**

| Property | 0xjj=0 | 0xjj=1 | 0xjj=2 |
|---|---|---|---|
| 0x00494000 | 1,3 | NULL | 1,3,4-6 |

# SECTION 4 *Image Data Format*

The subimage is stored as a separate entity within the *FlashPix* image object. This chapter specifies the format of the subimage, as described in Section 3.1.1.

The subimage is stored in two *FlashPix* streams: a header stream and a data stream.

## 4.1 The Subimage Header Stream

| | |
|---|---|
| Stream name: | Subimage**\040**0000\**040**Header |
| Class ID: | 00010000-C154-11CE-8553-00AA00A1F95B |

The subimage header stream determines image data location in the data stream and contains information for decoding the data stream into uncompressed pixel values. Values are stored in little endian byte order.

### 4.1.1 Subimage Header Stream Data

The format of the data portion of the subimage header stream is given in Table 4.1.

**TABLE 4.1** **Format and fields of the subimage header stream**

| Field name | Length | Byte(s) |
|---|---|---|
| Length of header stream header | 4 | 0-3 |
| Image width | 4 | 4-7 |
| Image height | 4 | 8-11 |
| Number of tiles | 4 | 12-15 |
| Tile width | 4 | 16-19 |
| Tile height | 4 | 20-23 |
| Number of channels | 4 | 24-27 |
| Offset to tile header table | 4 | 28-31 |
| Length of tile header entry | 4 | 32-35 |
| Tile header table | variable | variable |

**Length of header stream header field**
This field specifies the length of the header stream header (from the beginning of the length field to the end of the length of tile header entry field) in bytes.

**Image width and height field**
These fields specify the width and height of the subimage in pixels. These values do not include any padding at the right or bottom of the image to fill partial tiles. The values of these fields must be the same as the values of the subimage width and height in the primary description group of the image contents stream (Section 3.1.5.1, Table 3.1).

**Number of tiles field**
This field specifies the number of tiles in the subimage.

**Tile width field**
This field specifies the width of a tile in pixels. The tile width must be 64 pixels.

**Tile height field**
This field specifies the height of a tile in pixels. The tile height must be 64 pixels.

**Number of channels field**
This field specifies the number of channels in the subimage. This value is also specified in the image contents property set (Section 3.1.5, Table 3.4).

**Offset to tile header table field**
This field specifies the offset in bytes from the beginning of the data portion of the *FlashPix* stream to the tile header table (Section 4.1.2).

**Length of a tile header entry field**
This field specifies the length of a single entry in the tile header table (Section 4.1.2).

**Tile header table field**
This field specifies the header data for each tile. The format of a tile header table is specified in Section 4.1.2.

## 4.1.2 Tile header table

The format of the tile header table is given in Table 4.2.

**TABLE 4.2** **Format and fields in the tile header table**

| Field Name | Length | Byte(s) |
|---|---|---|
| Tile header 0 | 16 | 0-15 |
| … | | |
| Tile header *last* | 16 | variable |

**Tile header 0-*last***
These fields specify the location and encoded form of the image data tiles. Tiles are ordered from top to bottom, left to right, in row major order. The tile containing pixel (0, 0) is first, followed by the tile containing pixel (*tile width*, 0). This order continues across the row, through the tile containing pixel (*width - tile width*, 0). Subsequent rows of tiles follow in the same order. Table 4.3 specifies the format of a tile header.

**TABLE 4.3** **Format and fields of a tile header**

| Field name | Length | Byte(s) |
|---|---|---|
| Tile offset | 4 | 0-3 |
| Tile size | 4 | 4-7 |
| Compression type | 4 | 8-11 |
| Compression subtype | 4 | 12-15 |

**Tile offset field**
This field specifies the offset of the tile data from the beginning of data portion of the subimage data *FlashPix* stream in bytes. This value is zero if the compression algorithm requires no data other than the compression type and compression subtype fields.

**Tile size field**
This field specifies the size of the tile data for this tile, in bytes. This value is zero if the compression algorithm requires no data other than the compression type and compression subtype fields.

### Compression type field

This field specifies the compression algorithm used to encode the data for this tile. Valid compression type values are given in Table 4.4.

**TABLE 4.4** **Valid compression type values**

| Value | Meaning |
|---|---|
| 0x0 | Uncompressed data |
| 0x1 | Single color compression (4-byte) |
| 0x2 | JPEG (8-bit) |
| 0xFFFFFFFF | Invalid tile |

The invalid tile compression type may be used to temporarily indicate that the tile has no valid data. Situations where marking a tile as invalid may be useful are during resolution hierarchy regeneration or during a partial resynchronize operation between resolutions. Images with invalid tiles should not be saved permanently.

Images with any tile having the invalid tile compression type are considered to be invalid. If reader software encounters the invalid tile compression type when preparing to access a tile, it has no responsibility to attempt to create usable image data from higher resolutions. It is permitted to respond as though a read error occurred.

### Compression subtype field

This field specifies compression algorithm information for this tile. The format of this field depends on the value of the compression type field. The different formats are described below.

If the compression type is set to uncompressed data (0x0) or invalid tile (0xFFFFFFFF), the compression subtype is unused and must be set to 0x0.

If the compression type is set to single color compression (0x1), the compression subtype identifies the actual color value of all pixels in the tile. Individual channel values are stored in little endian format, in the same order and bit depth as specified by the subimage color and subimage numerical format properties (Section 3.1.5.3), aligned at the 0th bit of the field.

If the compression type is set to JPEG compression (0x2), the compression subtype field will contain additional information needed by the reader to process the JPEG compressed data. The format of the compression subtype subfields is shown in Table 4.5.

**TABLE 4.5** **Format and entries of the compression subtype field for JPEG compressed tiles**

| Field name | Length | Byte(s) |
|---|---|---|
| Interleave type | 1 | 0 |
| Chroma subsampling | 1 | 1 |
| Internal color conversion | 1 | 2 |
| JPEG tables selector | 1 | 3 |

### Interleave type subfield

This field specifies the interleaving of the data within the JPEG data stream. If the value is 0x0, all channels in the tile are stored in a single scan with the 8×8 blocks for each channel interleaved. If the value is 0x1, each channel is stored as a separate scan. All other values are illegal. In either case, the channels are found in the same order as specified by the subimage color property of the image contents property set.

### Chroma subsampling subfield

This field specifies the amount of subsampling performed on chroma components of the image (either the native components of some YCrCb image or those generated by rotating some RGB image through the standard JPEG CCIR 601 RGB to YCrCb conversion). The most significant nibble of the field indicates the horizontal subsampling ratio. The least significant nibble of the field indicates the vertical subsampling ratio. Legal values of the subsampling fields are 1 and 2.

Both the values for horizontal and vertical subsampling must be either 1 or 2, and if horizontal subsampling is 1, then vertical subsampling must also be 1. The specific horizontal and vertical subsampling pairs (h.v) allowed are (2,2), (2,1), and (1,1). Subsampling in the horizontal direction by 2x and the vertical direction by 1x is allowed for compatibility with digital video standards.

Under no circumstances should an Opacity channel be subsampled.

### Internal color conversion subfield

This field specifies whether a color conversion was performed in the JPEG compression process. Valid values are 0x0 and 0x1. All other values are illegal.

If the value is 0x0, no color conversion was performed and the pixel values output from the JPEG decoder are in the color space specified by the subimage color value from the Image Contents property set.  If the field value is 0x1, the effect of this field is dependent on the existence of an Opacity channel as described below.

For NIF RGB subimage color value:

If the color space specified by the subimage color value (Table 3.7) is NIF RGB (whether calibrated or not calibrated), i.e. NIF RGB with no opacity channel, then the following standard RGB to YCrCb conversion (or an equivalent integer implementation) is performed on the input data in the JPEG encoding process:

$$Y = 0.29900*R + 0.58700*G + 0.11400*B \qquad (4.1)$$

$$Cb = (B - Y)/1.772 + 0.5 = -0.16874*R - 0.33126*G + 0.50000*B + 0.5 \qquad (4.2)$$

$$Cr = (R - Y)/1.402 + 0.5 = 0.50000*R - 0.41869*G - 0.08131*B + 0.5 \qquad (4.3)$$

where R, G, B, Y, Cb and Cr values are in the 0 ... 1 range.

When decoding, the inverse transformation from YCrCb to RGB is done according to the following equations (or an equivalent integer implementation):

$$R = Y + 1.40200*Cr - 0.70100 \qquad (4.4)$$

$$G = Y - 0.34414*Cb - 0.71414*Cr + 0.52914 \qquad (4.5)$$

$$B = Y + 1.77200*Cb - 0.88600 \qquad (4.6)$$

For NIF RGB with premultiplied opacity subimage color value:

If the value is 0x1 and the color space specified by the subimage color value is NIF RGB with premultiplied opacity (whether calibrated or not calibrated), then to retain interoperability with early *FlashPix* applications, the RGB input was 'inverted' before the standard RGB to YCbCr transform was applied. Please note that the opacity channel is not affected by this operation and must not be inverted or color converted. The sequence of conversion steps is:

(a) Invert the RGB values, i.e., for RGB encoded with 8 bits calculate new color values R' = (255-R), G'= (255-G) and B'= (255-B).

(b) Transform R'G'B' to the new space Y'Cb'Cr' using equations (4.1), (4.2), (4.3).

Compression is done in the Y'Cb'Cr' space. On the decoder side, the inverse transformation should take place, i.e.,

(c) Transform Y'Cb'Cr' to R'G'B' using equations (4.4), (4.5), (4.6).

(d) Transform R'G'B' to RGB, e.g., R = (255-R').

Please note that the current requirement for this legacy 'inversion' results in minor differences when compared to compression done to RGB without opacity channels, but may preclude the use of some kinds of hardware acceleration.

**JPEG tables selector subfield**

This byte selects a set of quantizer and Huffman tables to use to decompress this tile. If the index is 0x0, the tables are included at the beginning of the tile data stream and it is not necessary to load a separate tile stream into the JPEG decompressor to decompress the tile. A value from 1 to 255 indicates that this tile uses a set of tables stored in one of the JPEG tables properties in the compression property group (Section 3.1.5.3). Specifically, if the value is 0x*ii*, a *FlashPix* reader should load the value of property 0x03ii0001 into the JPEG decompressor.

# 4.2 The Subimage Data Stream

Stream name: Subimage\0400000\**040**Data
Class ID: 00010100-C154-11CE-8553-00AA00A1F95B

The subimage data stream contains the data referenced by the tile headers in the header stream.

## 4.2.1 Channel Ordering

The channels of multi-channel tiles are ordered as specified in the color space property in the image contents property set (Section 3.1.5.2).

## 4.2.2 Tile Data Format

### 4.2.2.1 Uncompressed

Data in uncompressed tiles is stored in row major order in a pixel interleaved fashion. Pixel channels are ordered as specified by the color space property in the image contents property set. Pixel values are stored in little endian format in the type specified by the numerical format property.

### 4.2.2.2 Single Color Compressed

No tile data is needed for single-color compressed tiles. Both the tile data offset and tile size in the tile header table must be set to zero.

### 4.2.2.3 JPEG Compressed

The format of compressed tile data conforms to the “Abbreviated Format for Compressed Image Data” described in Annex B, Section B.4 of the ISO JPEG Specifications [9]. At a minimum, this format contains the following JPEG markers and marker segments, as well as the entropy-coded data for the tile (Table 4.6).

Quantizer tables and Huffman table marker segments are not required, but may be included to force a decoder to use tables other than those defined in Section 3.1.5.3.

**TABLE 4.6** **Format and entries of a JPEG abbreviated format stream for tile data**

| Field name | Length | Byte(s) | Value |
|---|---|---|---|
| Start of image marker (SOI) | 2 | 0-1 | 0xFFD8 |
| Start of frame marker (SOF) | 2 | 2-3 | 0xFFC0 |
| Frame header | variable | 4-variable | variable |
| Start of scan marker (SOS) 0 | 2 | variable | 0xFFDA |
| Scan header 0 | variable | variable | variable |
| Entropy coded data 0 | variable | variable | variable |
| … | | | |
| Start of scan marker (SOS) *last* | 2 | variable | 0xFFDA |
| Scan header *last* | variable | variable | variable |
| Entropy coded data *last* | variable | variable | variable |
| End of image marker (EOI) | 2 | variable | 0xFFD9 |

Note that some JPEG CODECS may identify the encoded color space through JPEG specific markers. The *FlashPix* format provides other mechanisms for identifying color in the Image Contents property set which are the subimage color property and the color space subfield value.  No other mechanism or values, besides those found in the Image Contents property set, can be utilized to make any decisions about what color space is intended for a given *FlashPix* file.

# SECTION 5 *Color Space Specifications*

## 5.1 Introduction

The method of encoding for color imagery is critical to how consistently the colors in an image will be reproduced across different systems and different media types. The *FlashPix* format defines two colorspaces, PhotoYCC and NIF RGB, along with respective reference viewing environments for the flexible and unambiguous encoding. The *FlashPix* format also provides a well-defined monochrome encoding space for the storage of greyscale imagery and optional support for InterColor Consortium(ICC) color management used in conjunction with the *FlashPix* color encoding.

PhotoYCC and NIF RGB color values represent color appearance with respect to a defined reference viewing environment. For color stimuli that are meant to be viewed in the reference viewing environment, PhotoYCC and NIF RGB values are computed by a series of simple mathematical operations from standard CIE colorimetric values. For color stimuli that are meant to be viewed in an actual viewing environment that is different from the reference environment, it is necessary to include appropriate colorimetric transformations to determine visually corresponding CIE colorimetric values for the reference environment (the corresponding CIE colorimetric values define a stimulus that, if viewed in the reference viewing environment, would produce the same color appearance as the actual stimulus viewed in the actual environment). These transformations account for differences in the amount of viewing flare in the actual and reference environments, as well as for alterations in observer perception that would be induced by the differences in the environments. The corresponding CIE colorimetric values resulting from these transformations are then encoded in terms of PhotoYCC or encoded in terms of NIF RGB.

# 5.2 PhotoYCC and NIF RGB Reference Viewing Environments

Reference viewing environments are defined for both PhotoYCC and NIF RGB in Table 5.1. The reference viewing environments are provided to give a single aim for each color space to allow for the unambiguous definition of color for use in interchange. The two definitions serve different purposes, yet with proper colorimetric and color appearance transformations, it is possible to encode values in one space which were originally encoded in the other.

**TABLE 5.1** **Comparison of PhotoYCC and NIF RGB viewing environments**

| Condition | PhotoYCC | NIF RGB |
|---|---|---|
| Viewing flare | None | 1.0% |
| Image surround | Average | 20% |
| Illuminance level/Luminance level | > 5,000 lux | 80 cd/m$^2$ |
| Adaptive white | $x = 0.3127$, $y = 0.3290$ | $x = 0.3127$, $y = 0.3290$ |

## 5.2.1 PhotoYCC Reference Viewing Environment

The PhotoYCC reference viewing environment corresponds to conditions typical of outdoor scenes.

- *Viewing flare* is specified as "none." Any flare light in an original-scene environment is part of the scene itself.
- The *image surround* is defined as "average." Scene objects typically are surrounded by other similarly illuminated objects.
- The *illuminance level* is representative of typical daylight levels. Note that the illuminance is at least an order of magnitude higher than average indoor levels.
- The chromaticities of the *adaptive white* are those of CIE D65. An adaptive white is defined here as a color stimulus that an observer would judge as perfectly achromatic, with a luminance corresponding to that of a perfect white diffuser. While the chromaticities of the adaptive white will most often be those of the scene illuminant, they may, in certain cases, also be quite different. For example, the observer may be only partially adapted to the illuminant. The adaptive white therefore defines only the chromatic adaptive state of the observer. The adaptive white does *not* define the chromaticities or the spectral power distribution of the scene illuminant.

## 5.2.2 NIF RGB Reference Viewing Environment

The NIF RGB reference viewing environment corresponds to conditions typical of indoor viewing of computer CRT monitors.

- *Viewing flare* is specified to be 1.0% of the maximum white-luminance level.
- The *image surround* is defined as “20%” of the maximum white luminance. This is close to a CIELAB L* value of 50, while maintaining computational simplicity. The areas surrounding the image being viewed are similar in luminance and chrominance to the image itself. This surround condition would correspond, for example, to an image displayed on a computer monitor where the image on the CRT screen is surrounded by a gray background equivalent to twenty percent of the maximum white.
- The *luminance level* is representative of typical CRT display levels. Note that the illuminance is at least an order of magnitude lower than average outdoor levels.
- The chromaticities of the *adaptive white* are those of CIE D65. An adaptive white is defined here as a color stimulus that an observer would judge as perfectly achromatic, with a luminance corresponding to that of a perfect white diffuser. While the chromaticities of the adaptive white will most often be those of the viewing illuminant, they may also, in certain cases, be quite different. For example, the observer may be only partially adapted to the illuminant. The adaptive white therefore defines only the chromatic adaptive state of the observer. It does *not* define the chromaticities or the spectral power distribution of the viewing illuminant.

## 5.3 Colorimetric Definitions and Digital Encodings

PhotoYCC and NIF RGB in combination with their reference viewing environments can be defined from standard CIE colorimetric values through simple mathematical transformations. Resulting colorimetric values can then be encoded in terms of digital code values for storage in a *FlashPix* Image.

While NIF RGB and PhotoYCC encode colors using similar standards and equations, the definitions presented in this specification do not constitute a means of conversion between the two color spaces. The conversions are an implementation topic.

### 5.3.1 PhotoYCC Colorimetric Definition and Digital Encoding

PhotoYCC is defined from standard CIE colorimetric values and the PhotoYCC reference viewing environment which corresponds to daylight-illuminated outdoor scenes. This definition describes the encoding of a daylight-illuminated scene, captured using a Photo CD Reference Image-Capture Device, when the observer adaptive white corresponds to D65 chromaticities. Examples of the encoding of colors not represented by this definition are given in the *FlashPix* Implementation Guide.

The red, green, and blue spectral responsivities of the Reference Image-Capture Device in Figure 5.1 correspond to the color-matching functions for the reference primaries defined in CCIR Recommendation 709[1]. The CIE chromaticities for the red, green, and blue CCIR-709 reference primaries, and for CIE Standard Illuminant D65, are given in Table 5.2.

**FIGURE 5.1** **Spectral responsivities of the Reference Image-Capture Device**

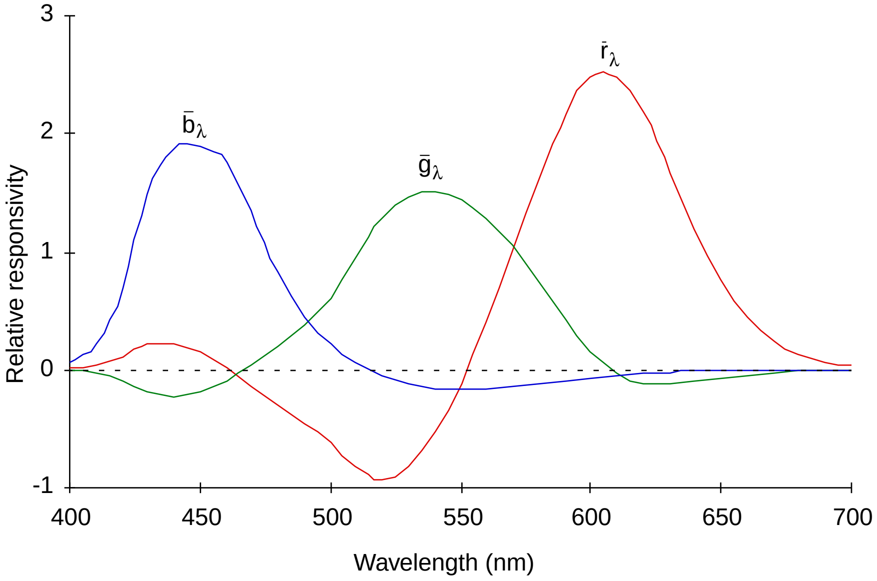


**TABLE 5.2** **CIE chromaticities for CCIR-709 reference primaries and CIE standard illuminant**

| | Red | Green | Blue | D65 |
|---|---|---|---|---|
| *x* | 0.6400 | 0.3000 | 0.1500 | 0.3127 |
| *y* | 0.3300 | 0.6000 | 0.0600 | 0.3290 |
| *z* | 0.0300 | 0.1000 | 0.7900 | 0.3583 |
| *u'* | 0.4507 | 0.1250 | 0.1754 | 0.1978 |
| *v'* | 0.5229 | 0.5625 | 0.1579 | 0.4683 |

Reference Image-Capture Device RGB$_{709}$ tristimulus values for the illuminated objects of the scene can be calculated using the spectral responsivities of the Reference Image-Capture Device:

$$
\begin{aligned}
R_{709} &= k_r \sum_{\lambda} P_{\lambda} R_{\lambda} \bar{r}_{\lambda} \\
G_{709} &= k_g \sum_{\lambda} P_{\lambda} R_{\lambda} \bar{g}_{\lambda} \qquad (5.1) \\
B_{709} &= k_b \sum_{\lambda} P_{\lambda} R_{\lambda} \bar{b}_{\lambda}
\end{aligned}
$$

where $P_{\lambda}$ is the spectral power of the scene illuminant at each wavelength $\lambda$; $R_{\lambda}$ is the spectral reflectance or transmittance of a scene object; and $\bar{r}_{\lambda}$, $\bar{g}_{\lambda}$, and $\bar{b}_{\lambda}$ are the spectral responsivities of the Reference Image-Capture Device. Normalizing factors $k_r$, $k_g$, and $k_b$ are determined such that *R*, *G*, and *B* tristimulus values of 1.00 will result for a perfect white diffuser. It is assumed that the reference image capture device produces flareless measurements; it is therefore unnecessary to adjust the resulting RGB values for instrument flare.

Since the spectral responsivities of the Reference Image-Capture Device are simply linear combinations of the 1931 CIE color-matching functions, $\bar{x}_{\lambda}$, $\bar{y}_{\lambda}$ and $\bar{z}_{\lambda}$ [3] its *RGB* tristimulus values can also be computed using the following relationship:

$$
\begin{bmatrix} R_{709} \\ G_{709} \\ B_{709} \end{bmatrix} = \begin{bmatrix} 3.2410 & -1.5374 & -0.4986 \\ -0.9692 & 1.8760 & 0.0416 \\ 0.0556 & -0.2040 & 1.0570 \end{bmatrix} \begin{bmatrix} X_{scene} \\ Y_{scene} \\ Z_{scene} \end{bmatrix} \qquad (5.2)
$$

where

$$
\begin{aligned}
X_{scene} &= k_r \sum_{\lambda} P_{\lambda} R_{\lambda} \bar{x}_{\lambda} \\
Y_{scene} &= k_g \sum_{\lambda} P_{\lambda} R_{\lambda} \bar{y}_{\lambda} \qquad (5.3) \\
Z_{scene} &= k_b \sum_{\lambda} P_{\lambda} R_{\lambda} \bar{z}_{\lambda}
\end{aligned}
$$

In the PhotoYCC encoding process, negative RGB$_{709}$ tristimulus values, and RGB$_{709}$ tristimulus values greater than 1.00 are retained. The luminance dynamic range and the color gamut defined by the RGB tristimulus values of the Reference Image-Capture Device are therefore unlimited.

Reference Image-Capture Device RGB$_{709}$ tristimulus values are next transformed to nonlinear R'G'B'$_{709}$ values as follows:

**APPENDIX II**

For $R_{709}$, $G_{709}$, $B_{709} \geq 0.018$:

$$\begin{aligned} R'_{709} &= 1.099 \times R_{709}{}^{0.45} - 0.099 \\ G'_{709} &= 1.099 \times G_{709}{}^{0.45} - 0.099 \\ B'_{709} &= 1.099 \times B_{709}{}^{0.45} - 0.099 \end{aligned} \qquad (5.4a)$$

For $R_{709}$, $G_{709}$, $B_{709} \leq -0.018$:

$$\begin{aligned} R'_{709} &= -1.099 \times \left|R_{709}\right|^{0.45} + 0.099 \\ G'_{709} &= -1.099 \times \left|G_{709}\right|^{0.45} + 0.099 \\ B'_{709} &= -1.099 \times \left|B_{709}\right|^{0.45} + 0.099 \end{aligned} \qquad (5.4b)$$

For $-0.018 < R_{709}$, $G_{709}$, $B_{709} < 0.018$:

$$\begin{aligned} R'_{709} &= 4.50 \times R_{709} \\ G'_{709} &= 4.50 \times G_{709} \\ B'_{709} &= 4.50 \times B_{709} \end{aligned} \qquad (5.4c)$$

For PhotoYCC, the nonlinear $R'G'B'_{709}$ values are rotated to luma and chromas as in Equation 5.5:

$$\begin{bmatrix} \mathrm{Luma} \\ \mathrm{Chroma}_1 \\ \mathrm{Chroma}_2 \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.299 & -0.587 & 0.886 \\ 0.701 & -0.587 & -0.114 \end{bmatrix} \begin{bmatrix} R'_{709} \\ G'_{709} \\ B'_{709} \end{bmatrix} \qquad (5.5)$$

For PhotoYCC, luma/chroma signals are converted to digital values. For 24-bit (8-bits/channel) encoding, PhotoYCC values are formed according to Equation 5.6:

$$\begin{aligned} Y &= (255/1.402) \times \mathrm{Luma} \\ C_1 &= 111.40 \times \mathrm{Chroma}_1 + 156 \\ C_2 &= 135.64 \times \mathrm{Chroma}_2 + 137 \end{aligned} \qquad (5.6)$$

### 5.3.2 NIFRGB Colorimetric Definition and Digital Encoding

NIFRGB is defined from standard CIE colorimetric values and the NIFRGB reference viewing environment which corresponds to indoor viewing of computer CRT displays. This definition describes the encoding of the appearance of the colors displayed on a reference monitor based on the reference primaries and transfer function implied in

CCIR Recommendation 709[1] when the observer adaptive white corresponds to D65 chromaticities. This transfer function is consistent with a large variety of legacy images including video and Microsoft Windows based imagery.

The CIE chromaticities for the red, green, and blue CCIR-709 reference primaries, and for CIE Standard Illuminant D65[2], are given in Table 5.2.

For NIFRGB, the goal is to communicate the appearance of the presentation of the appearance of the colors as displayed on a reference monitor in terms of 8-bit digital code values. Given the CIE $XYZ_{D65}$ tristimulus values for the colors represented on the monitor, a transformation can be made to reference monitor $RGB_{NIF}$ tristimulus values.

$$\begin{bmatrix} R_{NIF} \\ G_{NIF} \\ B_{NIF} \end{bmatrix} = \begin{bmatrix} 3.2410 & -1.5374 & -0.4986 \\ -0.9692 & 1.8760 & 0.0416 \\ 0.0556 & -0.2040 & 1.0570 \end{bmatrix} \begin{bmatrix} X_{D65} \\ Y_{D65} \\ Z_{D65} \end{bmatrix} \tag{5.7}$$

In the NIFRGB encoding process, $RGB_{NIF}$ values between 0.0 and 1.0 are encoded, while values outside that range are clipped and not retained. Therefore the luminance dynamic range and color gamut defined by the RGB tristimulus values of the reference monitor are limited.

Reference monitor $RGB_{NIF}$ tristimulus values are next transformed to nonlinear $RGB'_{NIF}$ values as follows:

For $R_{NIF}$, $G_{NIF}$, $B_{NIF} \geq 0.00304$:

$$\begin{aligned} R'_{NIF} &= 1.055 \times R_{NIF}^{0.42} - 0.055 \\ G'_{NIF} &= 1.055 \times G_{NIF}^{0.42} - 0.055 \\ B'_{NIF} &= 1.055 \times B_{NIF}^{0.42} - 0.055 \end{aligned} \tag{5.8a}$$

For $0.0 < R_{709}$, $G_{709}$, $B_{709} < 0.00304$:

$$\begin{aligned} R'_{NIF} &= 12.92 \times R_{NIF} \\ G'_{NIF} &= 12.92 \times G_{NIF} \\ B'_{NIF} &= 12.92 \times B_{NIF} \end{aligned} \tag{5.8b}$$

For NIF RGB, the nonlinear $R'G'B'_{NIF}$ values are converted to digital code values. For 24-bit (8-bits/channel) encoding, NIF RGB values are formed according to the following Equation 5.9:

$$\begin{aligned} R_{8bit} &= 255.0 \times R'_{NIF} \\ G_{8bit} &= 255.0 \times G'_{NIF} \\ B_{8bit} &= 255.0 \times B'_{NIF} \end{aligned} \tag{5.9}$$

## 5.4 Monochrome Encoding Definition

The *FlashPix* format supports the encoding and storage of 8-bit monochrome imagery. NIF monochrome is defined in terms of luminance, *$Luma_{NIF}$*, and is the defined in terms of luminance, *$Luma_{NIF}$*, and is designed to encode the appearance of a monochrome image on a reference monitor based on the primaries and tone transfer function defined in CCIR Recommendation 709.

Section 2.1 of CCIR Recommendation 601-4 defines a relationship between an analog luminance signal, $E'_y$ and red, green, and blue analog color signals ($E'_R$, $E'_G$, $E'_B$) and that relationship is given here as Equation 5.10.

$$E'_y = 0.299E'_R + 0.587E'_G + 0.114E'_B \tag{5.10}$$

The definition of NIF monochrome uses this relationship, however, the definitions and values of $E'_y$, $E'_R$, $E'_G$, $E'_B$ are not used from CCIR Recommendation 601-4. Instead, NIF monochrome is defined in terms of a luminance, *$Luma_{NIF}$*, and builds from the non-linear, *$R'_{NIF}$*, *$G'_{NIF}$*, and *$B'_{NIF}$* signals given in Equation 5.8a and Equation 5.8b of the definitions of NIFRGB. The relationship is given in Equation 5.11.

$$\mathrm{Luma}_{\mathrm{NIF}} = 0.299\mathrm{R'}_{\mathrm{NIF}} + 0.587\mathrm{G'}_{\mathrm{NIF}} + 0.114\mathrm{B'}_{\mathrm{NIF}} \tag{5.11}$$

The 8-bit digital encoding of *Luma* for NIF Monochrome is given in Equation 5.12.

$$\mathrm{Y}_{\mathrm{NIF}} = 255.0 \times \mathrm{Luma}_{\mathrm{NIF}} \tag{5.12}$$

# SECTION 6 *Image Info Property Set*

In addition to image data, a *FlashPix* image object also contains information to enhance the use of the image (information, for example, about the image itself, as well as how it was captured and how it might be used). This non-image data is stored in the image info property set in the *FlashPix* image object storage.

| | |
|---|---|
| Stream name: | **\005**Image**\040**Info |
| Class ID: | 56616500-C154-11CE-8553-00AA00A1F95B |
| Format ID: | 56616500-C154-11CE-8553-00AA00A1F95B |

## 6.1 Informational Groups

Though the properties may appear in any order, the property set is divided into several conceptual groups, each describing a different aspect of the image. The property groups are:

- File source
- Intellectual property
- Content description
- Camera information
- Per picture camera settings
- Digital camera characterization
- Film description
- Original document scan description
- Scan device

The information in these groups provides the framework to document facts about image capture, intellectual property concerns, and descriptive information about the image itself. With some images, users need to know who is in the picture, where and when it was taken, and so on, to understand the significance of the image.

For instance, a photograph of an automobile accident is useless to an insurance company unless it is known to which accident the picture applies. Similarly, an old family picture is far more interesting if it is known which ancestor is in the picture, and when and where it was taken. One problem with traditional methods of dealing with images is that it is easy for this data to become separated from the images, greatly diminishing the value of the images.

A fundamental concept of the *FlashPix* format is that an image should be as self-describing as possible. As an image moves across a network, or is written to various types of media, the self-describing data should move with the image.

Any property may be omitted. If omitted, that property should be treated as if the value is unknown. All property ID codes not explicitly listed are reserved for registered extensions.

## 6.2 File Source Group

This group of properties specify how the image was created. Table 6.1 lists the properties in this group:

**TABLE 6.1** **Properties in the file source group**

| Property name | ID code | Type |
|---|---|---|
| File source | 0x21000000 | VT_UI4 |
| Scene type | 0x21000001 | VT_UI4 |
| Creation path vector | 0x21000002 | VT_UI4 \| VT_VECTOR |
| Software Name/Manufacturer/Release | 0x21000003 | VT_LPWSTR |
| User defined ID | 0x21000004 | VT_LPWSTR |
| Sharpness approximation | 0x21000005 | VT_R4 |

### File source property (optional)

This property specifies the device source of the digital file, such as a film scanner, reflection print scanner, or digital camera. Possible values are listed in Table 6.2. Values greater than 0x5 must be handled by core reader software as though they were Unidentified (0x0).

**TABLE 6.2** **Valid file source property values**

| Value | Meaning |
|---|---|
| 0x0 | Unidentified |
| 0x1 | Film scanner |
| 0x2 | Reflection print scanner |
| 0x3 | Digital camera |
| 0x4 | Still from video |
| 0x5 | Computer graphics |

### Scene type property (optional)

This property specifies the type of scene that was captured. It differentiates "original scenes" (direct capture of real-world scenes) from "second generation scenes" (images captured from pre-existing hardcopy images). It provides further differentiation for scenes that are digitally composed. Values greater than 0x3must be handled by core reader software as though they were Unidentified (0x0). Possible values are listed in Table 6.3.

**TABLE 6.3** **Valid scene type property values**

| Value | Meaning |
|---|---|
| 0x0 | Unidentified |
| 0x1 | Original scene |
| 0x2 | Second generation scene |
| 0x3 | Digital scene generation |

### Creation path vector property (optional)

This property encodes the conversion path that an image takes as defined by both analog and digital capture processes. Each element of the vector is a property ID that corresponds to a property in the non-image data *FlashPix* properties. The first element of the vector is the *last* capture of the scene, corresponding to the property for File Source. Some examples are as follows.

A reflection print from an original scene would be listed as "File source, Type of reflection original, Film type, Camera model name, Scene type."

A reflection print from a second generation scene would be listed as "File source, Type of reflection original, Film type, Camera model name, Original medium, Scene type."

Film from the original scene would be listed as "File source, Film type, Camera model name, Scene type."

Film from a second generation scene would be listed as "File source, Film type, Camera model name, Original medium, Scene type."

An image from a digital camera capture of the original scene would be listed as “File source, Camera model name, Scene type.”

An image from a digital camera capture of a second generation scene would be listed as “File source, Camera model name, Original medium, Scene type.”

A still from a video camera would be listed as “File source, Camera model name, Scene type.”

A computer generated image would be listed as “File source, Software name/manufacturer/release.”

### Software name/release property (optional)

This property encodes the name of the software, its manufacturer’s name, and the version of the software used to create the *FlashPix* image.

### User defined ID property (optional)

This property encodes the values of an identification system assigned to an image by the user. This property is useful when users have their *own* filing or accounting scheme with an identification system already in place, and enables users to cross-reference their digital files to a pre-existing analog one.

### Sharpness approximation property (optional)

To perform image filtering in a resolution independent manner (Section 7.3.2), the algorithm must have information on the degree of blurring introduced by the system components which generated the digital image (digital camera, scanner, etc.). This is expressed as the effective filter width, $q$. Approximate the total capture MTF by the form $e^{-qs^2}$, where $q$ is the width and $s$ is the spatial frequency measured in cycles per pixel at the captured resolution delivered by the input device. If the MTF is far from Gaussian form, fit the low-frequency portion best. This property specifies the value $q$.

## 6.3 Intellectual Property Group

The intellectual property group contains information about the ownership and copyright status of the image. Rights for an original artifact may be stated, along with the rights for the digital file. Table 6.4 lists the properties in this group.

### Copyright message property (optional)

This property encodes the copyright notice of the Legal Broker for the digital file. The complete copyright statement should be listed in this field, including any dates and statements of claims. If desired, this property can also list details concerning the Legal Broker.

TABLE 6.4 **Properties in the intellectual property group**

| Property name | ID code | Type |
|---|---|---|
| Copyright message | 0x22000000 | VT_LPWSTR |
| Legal broker for the original image | 0x22000001 | VT_LPWSTR |
| Legal broker for the digital image | 0x22000002 | VT_LPWSTR |
| Authorship | 0x22000003 | VT_LPWSTR |
| Intellectual property notes | 0x22000004 | VT_LPWSTR |

**Legal broker for the original image property (optional)**
This property encodes the name of the person or organization that holds the legal right to grant permissions or restrict use of the original image. The original image is either the analog source scanned to create the digital file or the original digital capture of a scene.

**Legal broker for the digital image property (optional)**
This property encodes the name of the person or organization that holds the legal right to grant permissions or restrict use of the digital file.

**Authorship property (optional)**
This property encodes the name of the camera owner, photographer or image creator.

**Intellectual property notes property (optional)**
This property encodes additional information beyond the scope of other properties in this group.

## 6.4 Content Description Group

These properties describe the content of the image. Typically it is text that the user enters, either when the pictures are taken or later in the process. Table 6.5 lists the properties in this group.

**Test target in the image property (optional)**
This property encodes information about the type of scale or test target that is captured within the image frame. The values are in Table 6.6.

**Group caption property (optional)**
This property is text that describes the subject or purpose of a group of images (e.g., a roll of film). The image in the digital file is one member of the “group.”

**Caption text property (optional)**
This property is text that describes the subject or purpose of the image. It may be additionally used to provide any other type of information related to the image.

**TABLE 6.5** **Properties in the content description group**

| Property name | ID code | Type |
|---|---|---|
| Test target in the image | 0x23000000 | VT_UI4 |
| Group caption | 0x23000002 | VT_LPWSTR |
| Caption text | 0x23000003 | VT_LPWSTR |
| People in the image | 0x23000004 | VT_LPWSTR \| VT_VECTOR |
| Things in the image | 0x23000007 | VT_LPWSTR \| VT_VECTOR |
| Date of the original image | 0x2300000A | VT_FILETIME |
| Events in the image | 0x2300000B | VT_LPWSTR \| VT_VECTOR |
| Places in the image | 0x2300000C | VT_LPWSTR \| VT_VECTOR |
| Content description notes | 0x2300000F | VT_LPWSTR |

**TABLE 6.6** **Valid test target in the image property values**

| Value | Meaning |
|---|---|
| 0x0 | Unidentified |
| 0x1 | Color chart |
| 0x2 | Grey card |
| 0x3 | Greyscale |
| 0x4 | Resolution chart |
| 0x5 | Inch scale |
| 0x6 | Centimeter scale |
| 0x7 | Millimeter scale |
| 0x8 | Micrometer scale |

### People in the image property (optional)

This property encodes the personal or “role” names of people in the image. Personal names are any variation of FirstName, Initial, LastName, Titles of Address denotations (for example, Dr. Jane Smith). Roles may be occupational or situational denotations (for example, doctor). Multiple entries are allowed.

### Things in the image property (optional)

This property encodes the names of tangible objects depicted in the image, (Washington Monument, for example). Multiple entries are allowed.

### Date of the original image property (optional)

This property encodes the date and time the image was originally captured. In the case of a scanned photograph, this would be the date and time of the original photograph, not the date and time it was scanned. The date and time the digital file was created is stored in the property Scan Date. In the case of other printed materials, this would be the date the item was originally published.

**Events in the image property (optional)**
This property encodes the events depicted in the image. Events may be personal or societal (e.g., birthday, anniversary, New Year's Eve). Editorial applications may use this property to describe historical, political, or natural events (e.g., a coronation, the Crimean War, Hurricane Andrew).

**Places in the image property (optional)**
This property encodes the place depicted in the image (Chicago, Illinois). Multiple entries are allowed (e.g., the image may contain a map or an aerial view of a region).

**Content description notes property (optional)**
This property encodes additional user/application defined information beyond the scope of other properties in this group.

## 6.5 Camera Information Group

This group of properties describes the camera used to take a photograph. Table 6.7 lists the properties in this group.

**TABLE 6.7** **Properties in the camera information group**

| Property name | ID code | Type |
|---|---|---|
| Camera manufacturer name | 0x24000000 | VT_LPWSTR |
| Camera model name | 0x24000001 | VT_LPWSTR |
| Camera serial number | 0x24000002 | VT_LPWSTR |

**Camera manufacturer name property (optional)**
This property encodes the name of the manufacturer or vendor of the camera or original-scene capture device.

**Camera model name property (optional)**
This property encodes the model name or number of the camera, and can include the serial number of the camera.

**Camera serial number property (optional)**
This property encodes the manufacturer's serial number of the camera as a text string.

## 6.6 Per Picture Camera Settings Group

This group of properties describes the camera settings used when the image was captured.

New generations of digital and film cameras make it possible to capture more information about the conditions under which a picture was taken. This may include information about the lens aperture and exposure time, whether a flash was used, which lens was used, etc. This technical information is useful to professional and serious amateur photographers. In addition, some of these properties are useful to image database applications for populating values useful to image analysis and retrieval. Table 6.8 lists the properties in this group.

**TABLE 6.8** **Properties in the per picture camera settings group**

| Property name | ID code | Type |
|---|---|---|
| Capture date | 0x25000000 | VT_FILETIME |
| Exposure time | 0x25000001 | VT_R4 |
| F-number | 0x25000002 | VT_R4 |
| Exposure program | 0x25000003 | VT_UI4 |
| Brightness value | 0x25000004 | VT_R4 \| VT_VECTOR |
| Exposure bias value | 0x25000005 | VT_R4 |
| Subject distance | 0x25000006 | VT_R4 \| VT_VECTOR |
| Metering mode | 0x25000007 | VT_UI4 |
| Scene illuminant | 0x25000008 | VT_UI4 |
| Focal length | 0x25000009 | VT_R4 |
| Maximum aperture value | 0x2500000A | VT_R4 |
| Flash | 0x2500000B | VT_UI4 |
| Flash energy | 0x2500000C | VT_R4 |
| Flash return | 0x2500000D | VT_UI4 |
| Back light | 0x2500000E | VT_UI4 |
| Subject location | 0x2500000F | VT_R4 \| VT_VECTOR |
| Exposure index | 0x25000010 | VT_R4 |
| Special effects optical filter | 0x25000011 | VT_UI4 \| VT_VECTOR |
| Per picture notes | 0x25000012 | VT_LPWSTR |

### Capture date property (optional)

This property encodes the date and time the image was captured.

### Exposure time property (optional)

This property encodes the exposure time used when the image was captured. The units are seconds.

### F-number property (optional)

This property encodes the lens f-number (ratio of lens aperture to focal length) used when the image was captured.

### Exposure program property (optional)

This property encodes the class of exposure program that the camera used at the time the image was captured. Typical exposure programs include normal-program (general-purpose auto-exposure), aperture-priority (user sets aperture, camera selects shutter speed to properly expose), shutter-priority (user sets shutter speed, camera selects aperture to properly expose), etc. Values greater than 0x8 must be handled by core reader software as though they were Unidentified (0x0). Possible values are listed in Table 6.9.

**TABLE 6.9** **Valid exposure program property values**

| Value | Meaning |
|---|---|
| 0x0 | Unidentified |
| 0x1 | Manual |
| 0x2 | Program normal |
| 0x3 | Aperture priority |
| 0x4 | Shutter priority |
| 0x5 | Program creative (biased toward greater depth of field) |
| 0x6 | Program action (biased toward faster shutter speed) |
| 0x7 | Portrait mode (intended for close-up photos with the background out of focus) |
| 0x8 | Landscape mode (intended for landscapes with the background in good focus) |

### Brightness value property (optional)

This property encodes the Brightness Value (BV) measured when the image was captured, using APEX units. The expected maximum value is approximately 13.00 corresponding to a picture taken of a snow scene on a sunny day, and the expected minimum value is approximately -3.00 corresponding to a night scene.

If the value supplied by the capture device represents a range of values rather than a single value, it is encoded as a VT_VECTOR of two VT_R4 real numbers. The first value represents the lower value of the range, and the second represents the higher value. If the capture device supplies an exact value, it is encoded as a VT_VECTOR with a single VT_R4 value in the vector.

### Exposure bias value property (optional)

This property encodes the actual exposure bias (the amount of over- or under-exposure relative to a normal exposure, as determined by the camera’s exposure system) used when capturing the image, using APEX units. The range is between -99.99 and 99.99.

The value is the number of exposure values (stops). For example, -1.00 indicates 1 eV (1 stop) underexposure, or half the normal exposure.

### Subject distance property (optional)

This property encodes the distance (in meters) between the front nodal plane of the lens and the position at which the camera was focusing when the image was captured. Note that the camera may have focused on a subject within the scene which may not have been the primary subject.

If the value supplied by the capture device represents a range of values rather than a single value, it is encoded as a VT_VECTOR of two VT_R4 real numbers. The first value represents the lower value of the range, and the second represents the higher value. If the capture device supplies an exact value, it is encoded as a VT_VECTOR with a single VT_R4 value in the vector. Focus at infinity is encoded as -1.

### Metering mode property (optional)

This property encodes the metering mode (the camera’s method of spatially weighting the scene luminance values to determine the sensor exposure) used when capturing the image. Values greater than 0x4 must be handled by core reader software as though they were Unidentified (0x0). Possible values are listed in Table 6.10.

**TABLE 6.10** **Valid metering mode property values**

| Value | Meaning |
|---|---|
| 0x0 | Unidentified |
| 0x1 | Average |
| 0x2 | Center weighted average |
| 0x3 | Spot |
| 0x4 | Multi-spot |

### Scene illuminant property (optional)

This property encodes the light source (scene illuminant) that was present when the image was captured. Values between 0xB and 0x7FFF must be handled by core reader software as though they were Unidentified (0x0).

Note: Bit 15 of this 16-bit word is used as the key to whether or not a color temperature value is being stored. If bit 15 is 0, the value described within bits 0-14 will provide one of the prescribed color values depicted within the table below. If bit 15 is 1, bits 0-14 contain the actual color temperature value stored in units of Kelvin. In this case, color temperatures are limited to values in the range of 0 to 32767 Kelvin. Valid values are listed in Table 6.11. Values between 0xB and 0x7FFF must be handled by core reader software as though they were Unidentified (0x0).

### Focal length property (optional)

This property encodes the lens focal length (in millimeters) used to capture the image.

### Max aperture value property (optional)

This property encodes the maximum possible aperture opening (minimum lens f-number) of the camera or image capturing device, using APEX units. The allowed range is 1.00 to 99.99.

### Flash property (optional)

This property encodes whether flash was used. Possible values are listed in Table 6.12. Values greater than 0x2 must be handled by core reader software as though they were Unidentified (0x0).

**TABLE 6.11** **Valid scene illuminant property values**

| Value | Meaning |
|---|---|
| 0x0 | Unidentified |
| 0x1 | Daylight |
| 0x2 | Fluorescent light |
| 0x3 | Tungsten lamp |
| 0x4 | Flash |
| 0x5 | Standard illuminant A |
| 0x6 | Standard illuminant B |
| 0x7 | Standard illuminant C |
| 0x8 | D55 illuminant |
| 0x9 | D65 illuminant |
| 0xA | D75 illuminant |
| > 0x7FFF | The encoded actual temperature |

**TABLE 6.12** **Valid flash property values**

| Value | Meaning |
|---|---|
| 0x0 | Unidentified |
| 0x1 | No flash used |
| 0x2 | Flash used |

**Flash energy property (optional)**

This property encodes the amount of flash energy that was used. The measurement units are Beam Candle Power Seconds (BCPS).

**Flash return property (optional)**

This property encodes whether the camera judged that the flash was not effective at the time of exposure. Values greater than 0x2 must be handled by core reader software as though they were Unidentified (0x0). Possible values are listed in Table 6.13.

**TABLE 6.13** **Valid flash return property values**

| Value | Meaning |
|---|---|
| 0x0 | Unidentified |
| 0x1 | Subject outside flash range |
| 0x2 | Subject inside flash range |

**Back light property (optional)**

This property encodes the camera's evaluation of the lighting conditions at the time of exposure. The definitions of the conditions are:

- Front lit: the subject is illuminated from the front side.
- Back lit 1: The brightness value difference between the subject center and the surrounding area is greater than one full step (APEX). The frame is exposed for the subject center.
- Back lit 2: The brightness value difference between the subject center and the surrounding area is greater than one full step (APEX). The frame is exposed for the surrounding area.

Values greater than 0x3 must be handled by core reader software as though they were Unidentified (0x0). Possible values are listed in Table 6.14.

**TABLE 6.14 Valid back light property values**

| Value | Meaning |
|---|---|
| 0x0 | Unidentified |
| 0x1 | Front lit |
| 0x2 | Back lit 1 |
| 0x3 | Back lit 2 |

**Subject location property (optional)**

This property identifies the approximate location of the subject in the scene. It provides an X column number and Y row number that corresponds to the center of the subject location. It is stored as a VT_VECTOR of two VT_R4 values, where the first value of the vector is the X location and the second value of the vector is the Y location, in resolution-independent coordinates where the height of the image is 1.0 and the width is the aspect ratio.

**Exposure index property (optional)**

This property encodes the exposure index setting the camera selected.

**Special effects optical filter property (optional)**

This property encodes the type of filter used. The property contains an array of filter values, where the order of the elements in the array indicates the stacking order of the filters. The first value in the array is the filter closest to the original scene. Possible values are listed in Table 6.15. Values greater than 0x7 must be handled by core reader software as though they were Unidentified (0x0).

**Per picture camera settings notes property (optional)**

This property encodes additional information not provided by the other properties. Both professional and amateur photographers may want to keep track of a variety of miscellaneous technical information, such as the use of extension tubes, bellows, close-up lenses, and other specialized accessories.

**TABLE 6.15** **Valid special effects optical filter property values**

| Value | Meaning |
|---|---|
| 0x0 | Unidentified |
| 0x1 | None |
| 0x2 | Colored |
| 0x3 | Diffusion |
| 0x4 | Multi-image |
| 0x5 | Polarizing |
| 0x6 | Split-field |
| 0x7 | Star |

## 6.7 Digital Camera Characterization Group

This group of properties stores technical data specific to digital cameras. Table 6.16 lists the properties in the group.

**TABLE 6.16** **Properties in the digital camera characterization group**

| Property name | ID code | Type |
|---|---|---|
| Sensing method | 0x26000000 | VT_UI4 |
| Focal plane X resolution | 0x26000001 | VT_R4 |
| Focal plane Y resolution | 0x26000002 | VT_R4 |
| Focal plane resolution unit | 0x26000003 | VT_UI4 |
| Spatial frequency response | 0x26000004 | VT_VARIANT \| VT_VECTOR |
| CFA pattern | 0x26000005 | VT_VARIANT \| VT_VECTOR |
| Spectral sensitivity | 0x26000006 | VT_LPWSTR |
| ISO speed ratings | 0x26000007 | VT_UI2 \| VT_VECTOR |
| OECF | 0x26000008 | VT_VARIANT \| VT_VECTOR |

### Sensing method property (optional)

This property encodes the type of image sensor used in the camera or image capturing device. Possible values are listed in Table 6.17. Values greater than 0x8 must be handled by core reader software as though they were Unidentified (0x0).

**TABLE 6.17** **Valid sensing method property values**

| Value | Meaning |
|---|---|
| 0x0 | Undefined |
| 0x1 | Monochrome area sensor |
| 0x2 | One-chip color area sensor |
| 0x3 | Two-chip color area sensor |
| 0x4 | Three-chip color area sensor |
| 0x5 | Color sequential area sensor |
| 0x6 | Monochrome linear sensor |
| 0x7 | Trilinear sensor |
| 0x8 | Color sequential linear sensor |

### Focal plane X resolution property (optional)

This property encodes the number of pixels per *FocalPlaneResolutionUnit* in the ImageWidth direction for the main image. This property specifies the actual *FocalPlaneXResolution* at the focal plane of the camera. If this property is stored, the Focal length property in the per picture camera settings group must also be stored.

### Focal plane Y resolution property (optional)

This property encodes the number of pixels per *FocalPlaneResolutionUnit* in the ImageLength direction for the main image. This property specifies the actual *FocalPlaneYResolution* at the focal plane of the camera. If this property is stored, the Focal length property in the per picture camera settings group must also be stored.

### Focal plane resolution unit property (optional)

This property encodes the unit of measurement for the *FocalPlaneXResolution* and *FocalPlaneYResolution*. This property is mandatory if *FocalPlaneXResolution* or *FocalPlaneYResolution* exist. If this property is stored, the Focal length property in the per picture camera settings group must also be stored. Values other than those explicitly listed in Table 6.18 are not supported.

**TABLE 6.18** **Valid focal plane resolution unit property values**

| Value | Meaning |
|---|---|
| 0x0 | Inches |
| 0x1 | Meters |
| 0x2 | Centimeters |
| 0x3 | Millimeters |

### Spatial frequency response property (optional)

This property encodes the spatial frequency response (SFR) of the camera or image capturing device. The camera measured SFR data, described in ISO/TC42/WG18 Work Item [188] Working Draft 6.0, “Photography - Electronic still picture cameras - Resolution measurements,” can be stored as a table of spatial frequencies, horizontal SFR values, vertical SFR values, and diagonal SFR values. The following is a simple example of measured SFR data table (Table 6.19):

**TABLE 6.19** **Sample frequency response**

| Spatial frequency (lw/ph[a]) | Horizontal SFR | Vertical SFR |
|---|---|---|
| 0.1 | 1.00 | 1.00 |
| 0.2 | 0.90 | 0.95 |
| 0.3 | 0.80 | 0.85 |

[a] line widths per picture height

The spatial frequency response is stored as a VT_VARIANT | VT_VECTOR in the format shown in Table 6.20.

**TABLE 6.20** **Structure and entries of spatial frequency response VT_VARIANT | VT_VECTOR block**

| Field | Type |
|---|---|
| Number of columns | VT_UI4 |
| Number of rows | VT_UI4 |
| Column headings | VT_LPWSTR \| VT_VECTOR |
| Data | VT_R4 \| VT_VECTOR |

The number of entries in the column headings vector is the same as the number of columns, and the number of entries in the data field is the product of the number of rows and columns. Data entries are stored in row major order.

### CFA pattern property (optional)

This property encodes the actual color filter array (CFA) geometric pattern of the image sensor used to capture a single-sensor color image. It is not relevant for all sensing methods.

The first value, *CFARepeatRows*, encodes the number of rows in the vertical direction needed to uniquely define the repeat pattern of the CFA. The second value, *CFARepeatCols*, encodes the number of columns in the horizontal direction that are needed to uniquely define the repeat pattern of the CFA. These two values are followed by a list of integer values of length (*CFARepeatRows* x *CFARepeatCols*) that define the color filter pattern, using the integers given in Table 6.21.

**TABLE 6.21** **Valid CFA pattern property values**

| Value | Meaning |
|---|---|
| 0x0 | Red |
| 0x1 | Green |
| 0x2 | Blue |
| 0x3 | Cyan |
| 0x4 | Magenta |
| 0x5 | Yellow |
| 0x6 | White |

This property is stored in the form of a VT_VARIANT | VT_VECTOR, as shown in Table 6.20.

**TABLE 6.22** **Structure and entries of CFA pattern VT_VARIANT | VT_VECTOR block**

| Field | Type |
|---|---|
| *CFARepeatRows* | VT_UI2 |
| *CFARepeatCols* | VT_UI2 |
| *CFAArray* | VT_UI1 | VT_VECTOR |

where *CFARepeatRows* and *CFARepeatCols* are the minimum number of rows and columns, respectively, needed to uniquely define the CFA pattern, and where *CFAArray* is a list of unsigned 1 byte integers, in row major order that define the pattern. For example, the property:

| | |
|---|---|
| *CFARepeatRow* | = 2 |
| *CFARepeatCol* | = 2 |
| *CFAArray* | = 1 0 2 1 |

corresponds to the Bayer CFA pattern shown below:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Line 0 | = | G | R | G | R | G | R | … |
| Line 1 | = | B | G | B | G | B | G | … |
| Line 2 | = | G | R | G | R | G | R | … |
| Line 1 | = | B | G | B | G | B | G | … |

**Spectral sensitivity property (optional)**

This property field can be used to describe the spectral sensitivity of each channel of the camera used to capture the image. It is useful for certain scientific applications.

The property field is an ASCII string compatible with the “New Standard Practice for the Electronic Interchange of Color and Appearance Data” being developed within an ASTM Technical Committee. The ASCII string consists of a mandatory keyword list followed by the associated data values. Mandatory keywords include NUMBER_OF_FIELDS, which equals the number of channels (spectral bands) + 1, and NUMBER_OF_SETS, which specifies the number of spectral frequency (wavelength) entries.

### ISO speed ratings property (optional)

The property field is a VT_VECTOR of two VT_UI2 values. The first value is the ISO saturation speed rating classification and the second value is the ISO noise-based speed rating classification as defined in [21] tables 1 and 2.

### OECF property (optional)

This property encodes the "Opto-Electronic Conversion Function" (OECF). The OECF is the relationship between the optical input and the image file code value outputs of an electronic camera. The property allows OECF values defined in [22] to be stored as a table of values. Table 6.23 shows a simple example of measured OECF data.

**TABLE 6.23** **An example of measured OECF data**

| Log exposure | Red output level | Green output level | Blue output level |
|---|---|---|---|
| -3.0 | 10.2 | 12.5 | 8.9 |
| -2.0 | 48.1 | 47.5 | 48.3 |
| -1.0 | 150.2 | 152.0 | 149.8 |

The OECF is stored as a VT_VARIANT | VT_VECTOR in the following format (Table 6.20).

**TABLE 6.24** **Structure and entries of OECF VT_VARIANT | VT_VECTOR block**

| Field | Type |
|---|---|
| Number of columns | VT_UI2 |
| Number of rows | VT_UI2 |
| Column headings | VT_LPWSTR \| VT_VECTOR |
| Data | VT_R4 \| VT_VECTOR |

The number of entries in the column headings vector is the same as the number of columns, and the number of entries in the data field is the product of the number of rows and columns. Data entries are stored in row major order.

# 6.8 Film Description Group

This group of properties is used for images originating on photographic film. Table 6.25 lists the properties in the group.

**TABLE 6.25** **Properties in the film description group**

| Property name | ID code | Type |
|---|---|---|
| Film brand | 0x27000000 | VT_LPWSTR |
| Film category | 0x27000001 | VT_UI4 |
| Film size | 0x27000002 | VT_VARIANT \| VT_VECTOR |
| Film roll number | 0x27000003 | VT_UI4 |
| Film frame number | 0x27000004 | VT_UI4 |

### Film brand property (optional)

This property encodes the name of the film manufacturer, the brand name, product code and generation code (for example, Kodak Gold100, Kodak Aerial 100).

### Film category property (optional)

This property encodes the category of film used. Legal values are listed in Table 6.26.

**TABLE 6.26** **Valid film category property values**

| Value | Meaning |
|---|---|
| 0x0 | Unidentified |
| 0x1 | Negative B/W |
| 0x2 | Negative color |
| 0x3 | Reversal B/W |
| 0x4 | Reversal color |
| 0x5 | Chromagenic |
| 0x6 | Internegative B/W |
| 0x7 | Internegative color |

Values greater than 0x7 must be handled by core reader software as though they were Unidentified (0x0).

Note: Chromagenic refers to B/W negative film that is developed with a C41 process (i.e., color negative chemistry).

### Film size property (optional)

This property encodes the size of the X and Y dimension of the film used, and the unit of measurement. These properties are encoded as VT_VARIANT | VT_VECTOR, and internally consists of two VT_R4 dimensions and one VT_UI2 unit value indicator as shown in Table 6.27.

**TABLE 6.27** **Structure and entries of original scanned image size VT_VARIANT | VT_VECTOR block**

| Field | Type |
|---|---|
| Film size X | VT_R4 |
| Film size Y | VT_R4 |
| Film size unit | VT_UI2 |

Film size X and Y are the width and height of the original film used, respectively, represented in the unit specified by Film size unit. Film size unit has the same values as the focal plane resolution unit (Table 6.18).

**Film roll number property (optional)**
This property encodes the roll number of the film. For some film, this number is encoded on the film cartridge as a bar code.

**Film frame number property (optional)**
This property encodes the frame number from the roll of film.

## 6.9 Original Document Scan Description Group

This group of properties is used for images originating as documents or prints. Table 6.28 lists the properties in the group.

**TABLE 6.28** **Properties in the original document scan description group**

| Property name | ID code | Type |
|---|---|---|
| Original scanned image size | 0x29000000 | VT_VARIANT \| VT_VECTOR |
| Original document size | 0x29000001 | VT_VARIANT \| VT_VECTOR |
| Original medium | 0x29000002 | VT_UI4 |
| Type of original | 0x29000003 | VT_UI4 |

**Original scanned image size property (optional)**
This property encodes the lengths of the X and Y dimension of the scanned area, and the unit of measurement. These properties are encoded as VT_VARIANT | VT_VECTOR, and internally consists of two VT_R4 dimensions and one VT_UI2 unit value indicator as shown in Table 6.29.

**TABLE 6.29** **Structure and entries of original scanned image size VT_VARIANT | VT_VECTOR block**

| Field | Type |
|---|---|
| Original size X | VT_R4 |
| Original size Y | VT_R4 |
| Original size unit | VT_UI2 |

Original size X and Y are the width and height of the original scanned image, respectively, represented in the unit specified by Original size unit. Original size unit has the same values as the focal plane resolution unit (Table 6.18).

### Original document size property (optional)

This property encodes the lengths of the X and Y dimension of the original photograph or document, and the unit of measurement. These values are encoded as VT_VARIANT | VT_VECTOR, and internally consist of two VT_R4 dimensions and one VT_UI2 unit value indicator. It has the same format as the original scanned image size property (Table 6.29).

### Original medium property (optional)

This property encodes the medium of the original photograph, document, or artifact. Possible values are shown in Table 6.30.

**TABLE 6.30** **Valid original medium property values**

| Value | Meaning |
|---|---|
| 0x0 | Unidentified |
| 0x1 | Continuous tone image |
| 0x2 | Halftone image |
| 0x3 | Line art |

**Type of reflection original property (optional)**
This property encodes the type of the original document or photographic print. Possible values are shown in Table 6.31.

**TABLE 6.31** **Valid type of reflection original property values**

| Value | Meaning |
|---|---|
| 0x0 | Unidentified |
| 0x1 | B/W print |
| 0x2 | Color print |
| 0x3 | B/W document |
| 0x4 | Color document |

## 6.10 Scan Device Property Group

This group of properties is used for images scanned from reflection prints, documents, photographic slides, or negatives. It contains the properties listed in Table 6.32.

**TABLE 6.32** **Properties in the scan device property group**

| Property name | ID code | Type |
|---|---|---|
| Scanner manufacturer name | 0x28000000 | VT_LPWSTR |
| Scanner model name | 0x28000001 | VT_LPWSTR |
| Scanner serial number | 0x28000002 | VT_LPWSTR |
| Scan software | 0x28000003 | VT_LPWSTR |
| Scan software revision date | 0x28000004 | VT_DATE |
| Service bureau/organization name | 0x28000005 | VT_LPWSTR |
| Scan operator ID | 0x28000006 | VT_LPWSTR |
| Scan date | 0x28000008 | VT_FILETIME |
| Last modified date | 0x28000009 | VT_FILETIME |
| Scanner pixel size | 0x2800000A | VT_R4 |

**Scanner manufacturer name property (optional)**
This property encodes the manufacturer or vendor of the scanner.

**Scanner model name property (optional)**
This property encodes model name or number of the scanner. It can also include the serial number of the scanner.

**Scanner serial number property (optional)**
This property encodes the manufacturer's serial number of the scanner as a text string.

**Scan software property (optional)**
This property encodes the name and version of the scanner software or firmware.

**Scan software revision date property (optional)**
This property encodes the revision date of the scanner software or firmware. The date should be in GMT.

**Service bureau/organization name property (optional)**
This property encodes the name of the service bureau, photofinisher, or organization performing the scan.

**Scan operator ID property (optional)**
This property encodes a name or ID for the person operating the scanner.

**Scan date property (optional)**
This property encodes the date and time the image was originally captured and digitized. This property should never be changed after it is written in the image capture device.

**Last modified date property (optional)**
This property encodes the last modification date of the scanned data.

**Scanner pixel size property (optional)**
This property specifies the pixel size, in micrometers, of the scanner.

# SECTION 7 *FlashPix Image View Object*

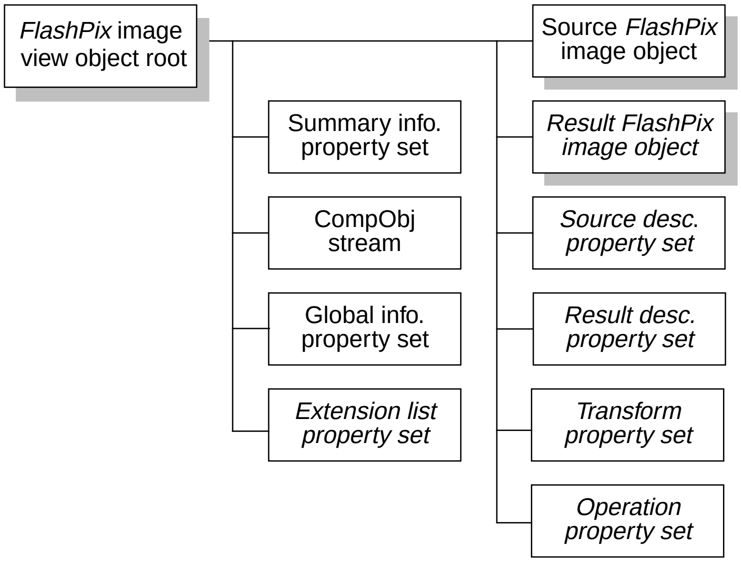
## 7.1 *FlashPix* Image View Object

The *FlashPix* format allows the specification of a viewing transform through a *FlashPix* image view object which references a *FlashPix* image. The viewing transform enables applications to represent a set of simple edits as a list of “commands” which are applied to the image in real time without altering the original image.

| | |
|---|---|
| Storage name: | Any valid storage name (recommend the file name in host file system) |
| Class ID: | 56616700-C154-11CE-8553-00AA00A1F95B |

This class ID is to be used for all *FlashPix* image view objects whether or not they contain any extensions. Figure 7.1 shows the storages and streams in a *FlashPix* image view. Those streams and storages in italics are optional or optional under certain circumstances.

**FIGURE 7.1** ***FlashPix* image view storages and streams**


*FlashPix* image view object root
Summary info. property set
CompObj stream
Global info. property set
*Extension list property set*
Source *FlashPix* image object
*Result FlashPix image object*
*Source desc. property set*
*Result desc. property set*
*Transform property set*
*Operation property set*

A *FlashPix* image view object contains the following storages and streams which are defined in more detail in subsequent sections. All property sets must abide by the restrictions specified in Section 1.4.1.

**Source *FlashPix* image (required)**

This storage is an instance of a *FlashPix* image object as defined in *Section 3: The FlashPix Image Object*. Transforms may be applied to the source (original) image to produce the result image (below).

**Result *FlashPix* image (optional)**

This storage is an instance of a *FlashPix* image object as defined in *Section 3: The FlashPix Image Object*. Transforms may be applied to the source image (above) to produce the result image.

**Summary info property set (required)**

This property set is an instance of the standard Summary Information property set, as described in Section 1.4.2. The viewing transform must be applied in creating the thumbnail image which is optional if the *FlashPix* file is authored with a non-hierarchical source image object in an embedded capture environment. It is otherwise required.

**CompObj stream (required)**

This stream contains the class ID of the *FlashPix* image view object.

**Global info property set (required)**

This property set contains generic information about the image view.

**Source description property set (required)**
This property set describes location and type of the source *FlashPix* image.

**Result description property set (optional)**
This property set describes the location and type of the result *FlashPix* images. It is required if the result *FlashPix* image object exists or if there is a viewing transform specified.

**Transform property set (optional)**
This property set describes the viewing transform specified for this image view.

**Operation property set (optional)**
This property set describes the actual view transform operation.

**Extension list property set (optional)**
This property set identifies extensions present in the *FlashPix* image view object including the data structures modified or added by each extension.

### 7.1.1 CompObj Stream (required)

The CompObj stream is a standard Structured Storage stream and is not a *FlashPix* stream. The header of the stream is not extended for storage of a stream class ID. This stream is required and is defined in Section A.3. The unicode versions of the CompObj stream fields are required.

The CompObj stream Clipboard Format field is used to store the class ID of the *FlashPix* image view object. The *FlashPix* image view object class ID is converted to a string for storage in the Clipboard Format field and must be bracketed by the bracket characters '{'and '}' just as returned by the OLE function StringFromGUID2.

The CompObj stream User Type field is generally used to store the User Type information from the OLE registry for the class ID. In OLE-enabled environments, the string contents should be retrieved from the OLE registry. In non-OLE-enabled environments, a string which is a user-understandable brief description of the object contents should be used.

The CompObj stream ProgID field is generally used to store the ProgID information from the OLE registry for the class ID. In OLE-enabled environments, the string contents should be retrieved from the OLE registry. In non-OLE-enabled environments, a string which identifies the program associated with the class ID should be used. This string cannot contain any spaces.

### 7.1.2 Source and Result *FlashPix* Image Objects

| | |
|---|---|
| Names: | Data\**040**Object\**040**Store\**040**%06d |
| Class ID: | 56616000-C154-11CE-8553-00AA00A1F95B |

The source and result *FlashPix* image objects are instances of the *FlashPix* image object as defined in *Section 3: The FlashPix Image Object*. The source *FlashPix* image object is required and the result *FlashPix* image object is optional. The single numeric parameter in the name represents the index of the source and result image objects. Upon creation, the index used must be unique in the *FlashPix* image view object. The maximum image index property of the global info property set must always be the maximum *FlashPix* image object index in the *FlashPix* image view.

The source image object represents the image to be processed through the viewing transform. For example, it may be an image that needs to be cropped and rotated or its color balance adjusted. The result image object is the image generated by applying the viewing transform to the source image object. The result image object cannot exist unless the *FlashPix* image view contains a viewing transform.

## 7.1.3 Source and Result Description Property Sets

| | |
|---|---|
| Name: | **\005**Data**040**Object**040**%06d |
| Class ID (for both): | 56616080-C154-11CE-8553-00AA00A1F95B |
| Format ID: | 56616080-C154-11CE-8553-00AA00A1F95B |

These property sets have only one section, which has a format ID that is the same as the property set class ID.

These property sets are associated with the source and result image objects in this image view object. This association is indicated by matching index values in the *FlashPix* image object storage and description property set names.

Source description properties describe the source image object. Result description properties describe the result image object. Both property sets have the same format, as described below.

If the *FlashPix* image view does not contain a viewing transform, the result description property set is unused and may not exist.

The *FlashPix* image object to be used as the input to the image view must be characterized in the source description property set. After applying the viewing transform to the source image object, an actual *FlashPix* image may be stored in a result *FlashPix* image object. Even if the result image object is not stored, the result description property set must exist if there is a viewing transform to specify how the result is created.

The valid properties for these property sets are shown in Table 7.1.

**Data object ID property (required)**

This property specifies a unique ID used to identify the associated *FlashPix* image object. These values may be used, for example, in a networked system, to determine if a local copy of the images exist or if they must be pulled across the network.

**TABLE 7.1** **Valid properties for the source and result description property sets**

| Property name | ID code | Type |
|---|---|---|
| Data object ID | 0x00010000 | VT_CLSID |
| Locked property list | 0x00010002 | VT_UI4 \| VT_VECTOR |
| Data object title | 0x00010003 | VT_LPWSTR |
| Last modifier | 0x00010004 | VT_LPWSTR |
| Revision number | 0x00010005 | VT_UI4 |
| Creation time and date | 0x00010006 | VT_FILETIME |
| Modification time and date | 0x00010007 | VT_FILETIME |
| Creating application | 0x00010008 | VT_LPWSTR |
| Status | 0x00010100 | VT_UI4 |
| Creator | 0x00010101 | VT_UI4 |
| Users | 0x00010102 | VT_UI4 \| VT_VECTOR |
| Cached image height | 0x10000000 | VT_UI4 |
| Cached image width | 0x10000001 | VT_UI4 |

### Locked property list property (optional)

This property specifies a list of properties that are locked. Each value in the list is taken as a property ID of a property found in this instance of the property set. If an editing application finds a property which is also found in the locked property list, it may not modify the value of the property. If the value of a locked property is modified, the results of rendering the *FlashPix* image view from another application will be undefined. If this property exists, it may not be deleted. This property is used to provide guidance to an editing application in situations where the *FlashPix* image view is a template to be “filled out” by the user.

### Data object title (optional)

This property specifies a title for the associated image object. If this property exists, an editing application must keep the value updated.

### Last modifier property (optional)

This property specifies the name of the last person (or system if the last modification was made by an automatic editing system) to modify the contents of the associated image object. If this property exists, an editing application must keep the value updated.

### Revision number property (optional)

This property specifies the number of times the associated image object has been modified since its creation. If this property exists, an editing application must keep the value updated.

### Creation time and date property (optional)

This property specifies the time and date of creation of the associated image object. If this property exists, an editing application must keep the value updated.

**Modification time and date property (optional)**

This property specifies the time and date of the last modification to the associated image object. If this property exists, an editing application must keep the value updated.

**Creating application property (optional)**

This property specifies the name of the application that created the associated image object. If this property exists, an editing application may not delete it.

**Status property (required)**

This property (Table 7.2) indicates the status of the value of the associated image. Possible values of existence/location field are shown in Table 7.3.

**TABLE 7.2** **Structure and entries of the status property**

| Field name | Length | Byte(s) |
|---|---|---|
| Existence data | 2 | 0-1 |
| Permissions set for data | 2 | 2-3 |

**TABLE 7.3** **Valid status property values of the existence/location field**

| Value | Meaning |
|---|---|
| 0x0 | The data object associated with this property set does not exist. |
| 0x1 | The data object associated with this property set does exist. |

The existence field indicates whether the associated image object exists or is stored for direct application access. If the existence field is 0x0 (not cached), the permissions field is ignored. The source image object must be cached (the value of the existence field must be 0x1). The result image object may or may not be cached, at the discretion of the writer. Possible values of the permissions field are shown in Table 7.4.

**TABLE 7.4** **Valid status property permissions field values**

| Value | Meaning |
|---|---|
| 0x0 | The data object is purgeable |
| 0x1 | The data object is not purgeable |

If the associated image object is marked purgeable, a clean-up utility may delete the image object to recover storage space if it can be recreated from a source transform. Therefore, the source image object must be set to not purgeable (0x1). The result image object should be set to purgeable (0x0).

**Creator property (required)**

This property specifies the number of the transform node that created the image object. For the source image object, the creator should be set to zero (NULL). For the result image object, the creator should be set to the index of the viewing transform.

**Users property (required)**

This property specifies a list of transforms that take this image object as an input. Each entry in the list specifies the index of a transform. The array is unordered. If the associated image object is not used by any transforms, the number of elements in the array should be zero. Therefore, the users property must be zero for the result image object.

In the source description property set, the users property must be an array with one and only one element, the index used to name the transform property set. For the result description, a *FlashPix* image view writer should write an array with zero elements.

**Cached image height and width properties (required if associated image exists)**

These properties specify the height and width, respectively, of the image cached in the associated *FlashPix* image object. These properties are the height and width of the largest resolution in the associated *FlashPix* image object. These properties are required if the status property existence flag is set to cached (0x1). If the status property existence flag is set to not cached (0x0), these properties may not exist. Note that the height property precedes the width property contrary to other height and width property pairs in the format. Note also that the cached image height and width properties for the result *FlashPix* image object take precedence over the rectangle of interest and result aspect ratio properties should they exist in the transform property set.

## 7.1.4 Transform Property Set (optional)

| | |
|---|---|
| Name: | **\005**Transform**\040**%06d |
| Class ID: | 56616A00-C154-11CE-8553-00AA00A1F95B |
| Format ID: | 56616A00-C154-11CE-8553-00AA00A1F95B |

This property set describes the viewing transform specified for the image view. The single numeric parameter in the name represents the index of the transform. In a *FlashPix* image view, this array has only one element, the viewing transform. Upon creation, the index used must be unique in the *FlashPix* image view object. The maximum transform index property of the global info property set must always be the maximum transform index in use in the *FlashPix* image view object. The index is referenced by both the creator property of the result description property set and the users property of the source description property set, which must both have the same value. The transform property set is unused and must not exist if the *FlashPix* image view does not contain a viewing transform. Table 7.5 lists the possible properties of the transform list property set.

**Transform node ID property (required)**

This property specifies a unique ID used to identify the viewing transform. Note that this identifies the transform itself, not the value of the parameters. For example, this ID specifies that this transform node is performing the viewing transform on a particular

**TABLE 7.5** **Valid properties for the transform property set**

| Property name | ID code | Type |
|---|---|---|
| Transform node ID | 0x00010000 | VT_CLSID |
| Operation Class ID | 0x00010001 | VT_CLSID |
| Locked property list | 0x00010002 | VT_UI4 \| VT_VECTOR |
| Transform title | 0x00010003 | VT_LPWSTR |
| Last modifier | 0x00010004 | VT_LPWSTR |
| Revision number | 0x00010005 | VT_UI4 |
| Creation time and date | 0x00010006 | VT_FILETIME |
| Modification time and date | 0x00010007 | VT_FILETIME |
| Creating application | 0x00010008 | VT_LPWSTR |
| Input data object list | 0x00010100 | VT_UI4 \| VT_VECTOR |
| Output data object list | 0x00010101 | VT_UI4 \| VT_VECTOR |
| Operation number | 0x00010102 | VT_UI4 |
| Result aspect ratio | 0x10000000 | VT_R4 |
| Rectangle of interest | 0x10000001 | VT_R4 \| VT_VECTOR |
| Filtering | 0x10000002 | VT_R4 |
| Spatial orientation | 0x10000003 | VT_R4 \| VT_VECTOR |
| Colortwist matrix | 0x10000004 | VT_R4 \| VT_VECTOR |
| Contrast adjustment | 0x10000005 | VT_R4 |

source image. This ID does not change if the actual viewing parameters change (the transform is reexecuted).

#### Operation class ID property (required)

This property specifies the class ID of the operation to be performed by this transform node. This property (along with the class ID of this stream) specifies the code that actually executes the viewing transform. This property must have the value 56616A00-C154-11CE-8553-00AA00A1F95B.

#### Locked property list property (optional)

This property specifies a list of properties that are locked for the viewing transform. Each value in the list is a property ID of a property found in the transform property set. Editing applications may not modify the value of properties found in the locked property list. If the value of a locked property is modified, the results of rendering the *FlashPix* image view from another application will be undefined. If this property exists, it may not be deleted. This property is used to provide guidance to an editing application in situations where the *FlashPix* image view is a template to be “filled out” by the user.

#### Transform title property (optional)

This property specifies a title for the viewing transform. If this property exists, an editing application must keep the value updated.

**Last modifier property (optional)**
This property specifies the name of the last person (or system if the last modification was made by an automatic editing system) to modify the contents of the viewing transform. If this property exists, an editing application must keep the value updated.

**Revision number property (optional)**
This property specifies the number of times the viewing transform has been modified since its creation. If this property exists, an editing application must keep the value updated.

**Creation time and date property (optional)**
This property specifies the time and date of creation of the viewing transform. If this property exists, an editing application may not delete it.

**Modification time and date property (optional)**
This property specifies the time and date of the last modification to the viewing transform. If this property exists, an editing application must keep the value updated.

**Creating application property (optional)**
This property specifies the index of the application that created the viewing transform. If this property exists, an editing application must keep the value updated.

**Input data object list property (required)**
This property specifies the index used in naming the source *FlashPix* image that is input to the viewing transform. There may be only one element in the array.

**Output data object list property (required)**
This property specifies the index of the result image (in the sparse array of images). There may be only one element in the array.

**Operation number property (required)**
This property specifies the index used in naming the viewing operation.

**Result aspect ratio property (optional)**
The result aspect ratio property allows applications to specify the desired aspect ratio for image output. The value is an IEEE 4-byte floating point number.

The value (R) defines a rectangle with the top-left corner at (0,0) and the bottom-right corner at (R,1). The result aspect ratio must be applied to the output of the spatial orientation matrix as a cropping function. Pixels outside the rectangle are cropped and 100% transparent. If the property is not present, applications must operate as though the result aspect ratio is the same as the raw image aspect ratio. Note that the cached image height and width properties in the result *FlashPix* image object result description property set take precedence over the rectangle of interest and result aspect ratio properties.

**Rectangle of interest property (optional)**
The rectangle of interest property lets applications select part of the image. Only a rectangular region with sides parallel to the edges of the image can be specified. Each element of the rectangle of interest property array is an IEEE 4-byte floating point number. The format of the rectangle of interest property is given in Table 7.6. It is discussed in

more detail in Section 7.2.1. Applications must operate as though this property is defined with the array values (0,0,R,1) if the property is not present. This indicates that the entire image is selected. Note that the cached image height and width properties in the result *FlashPix* image object result description property set take precedence over the rectangle of interest and result aspect ratio properties.

**TABLE 7.6** **Format and fields of the rectangle of interest property**

| Field | Length | Vector Element |
|---|---|---|
| left edge (*x*) | 4 | 0 |
| top edge (*y*) | 4 | 1 |
| width (*w*) | 4 | 2 |
| height (*h*) | 4 | 3 |

### Filtering property (optional)

The filtering property specifies the degree of filtering (sharpening / blurring) applied to the raw image data. The value is an IEEE 4-byte floating point number. The interpretation of the value is discussed in Section 7.2.2. Applications must operate as though this property has a zero value if it is not present to indicate that the raw image is not filtered.

### Spatial orientation property (optional)

The spatial orientation property allows applications to rotate, flip, stretch, and shear an image. The value is an array of 16 IEEE 4-byte floating point numbers, with the first number starting at vector element 0. The position of the sixteen elements is as follows:

$$\begin{aligned} a_{ij} &\equiv \mathrm{ARRAY}[k] \\ k &= (i-1)\times 4 + j - 1 \end{aligned} \qquad (7.1)$$

The interpretation of the value is discussed in Section 7.2.3. Applications must operate as though this property is defined with the array value (1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1) if the property is not present. This indicates that there is no spatial transformation applied to the raw image.

### Colortwist matrix property (optional)

The colortwist matrix property allows applications to make minor changes to the tone and color of a raw image. It is not intended to support correcting faults in the imaging chain. The value is an array of 16 IEEE 4-byte floating point numbers, with the first number starting at vector element 0. The position of the sixteen elements is as follows:

$$\begin{aligned} a_{ij} &\equiv \mathrm{ARRAY}[k] \\ k &= (i-1)\times 4 + j - 1 \end{aligned} \qquad (7.2)$$

The interpretation of the value is discussed in Section 7.2.4. Applications must operate as though this property is defined with the array value (1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1) if the property is not present. This indicates that there is no tone or color correction applied to the raw image.

**Contrast adjustment property (optional)**

The contrast adjustment property allows applications to modify the contrast of a raw image. The value is an IEEE 4-byte floating point number. The interpretation of the value is discussed in Section 7.2.5. Applications must operate as though this property has a value of 1.0 if the property is not present. This indicates that there is no contrast adjustment applied to the raw image.

### 7.1.5 Operation Property Set (optional)

| | |
|---|---|
| Name: | **\005**Operation**\040**%06d |
| Class ID: | 56616E00-C154-11CE-8553-00AA00A1F95B |
| Format ID: | 56616E00-C154-11CE-8553-00AA00A1F95B |

The operation property set specifies the software to execute the viewing transform. The single numeric parameter in the name represents the index of the operations in a sparse array. Upon creation, the index used must be greater than the maximum operation index property of the Global Info property set. In a *FlashPix* image view, the array has only one element, the viewing operation. The index is referenced by the operation number property in the transform property set.

If this *FlashPix* image view does not contain a viewing transform, the operation property set is unused and may not exist. Table 7.7 lists the possible properties for the operation property set.

**TABLE 7.7** **Valid properties for the operation property set**

| Property name | ID code | Type |
|---|---|---|
| Operation ID | 0x00010000 | VT_CLSID |

**Operation ID property (required)**

This property specifies the class ID of the viewing operation. This value is used by either the *FlashPix* reader, an OLE server or an OpenDoc part to identify the actual software to implement the viewing transform. For a *FlashPix* image view, the value of this property must be 56616A00-C154-11CE-8553-00AA00A1F95B. This value specifies the actual code that executes the viewing transform.

### 7.1.6 Global Info Property Set (required)

| | |
|---|---|
| Stream name: | **\005**Global**\040**Info |
| Class ID: | 56616F00-C154-11CE-8553-00AA00A1F95B |
| Format ID: | 56616F00-C154-11CE-8553-00AA00A1F95B |

This property set provides global information about the image view. Table 7.8 lists the possible properties for the global info property set.

**TABLE 7.8** **Valid properties in the global info property set**

| Property name | ID code | Type |
|---|---|---|
| Locked property list | 0x00010002 | VT_UI4 \| VT_VECTOR |
| Transformed image title | 0x00010003 | VT_LPWSTR |
| Last modifier | 0x00010004 | VT_LPWSTR |
| Visible outputs | 0x00010100 | VT_UI4 \| VT_VECTOR |
| Maximum image index | 0x00010101 | VT_UI4 |
| Maximum transform index | 0x00010102 | VT_UI4 |
| Maximum operation index | 0x00010103 | VT_UI4 |

### Locked property list property (optional)

This property specifies a list of properties that are locked for this property set. Each value in the list is a property ID of a property found in this property set. Editing applications may not modify the value of properties found in the locked property list. If the value of a locked property is modified, the results of rendering the *FlashPix* image view from another application will be undefined. If this property exists, it may not be deleted. This property is used to provide guidance to an editing application in situations where the *FlashPix* image view is a template to be “filled out” by the user.

### Transformed image title property (optional)

This property specifies a title for the viewing transform. If this property exists, an editing application must keep the value updated.

### Last modifier property (optional)

This property specifies the name of the last person (or system if the last modification was made by an automatic editing system) to modify the contents of this *FlashPix* image view. If this property exists, an editing application must keep the value updated.

### Visible outputs property (required)

This property specifies the output of the image view. The value of this property indicates the image that is to be considered the output of this file. There may be only one value in this array. If the *FlashPix* image view contains a viewing transform, this value must be the index used to name the result image. If a viewing transform is not specified, this value must be the index used to name the source image.

### Maximum image, transform node, and operation index properties (required)

These properties specify the highest index in use for data objects, transforms, and operations. When an application creates a new entity, it is recommended that it use the value of the appropriate of these index properties + 1 as the index value for that entity. Then the appropriate index property must be updated if necessary so that it is the maximum of the indices in use for the type of entity. The values of these properties are 0 prior to creating any image, transform, and operation entities.

### 7.1.7 Extension List Property Set (optional)

| | |
|---|---|
| Stream name: | **\005**Image**\040**Contents |
| Class ID: | 56616010-C154-11CE-8553-00AA00A1F95B |
| Format ID: | 56616010-C154-11CE-8553-00AA00A1F95B |

This property set identifies extensions present in the *FlashPix* image view object by class ID, name, and description as well as the data elements changed or added by each extension. The property set is optional, however, if the *FlashPix* image view object contains any extensions, the extension list property set must be present and all extensions, registered and private, in the *FlashPix* image view object must be described. The way in which the data associated with an extension is structured can take one or more of the following forms:

- New storage(s) may be added
- New stream(s) may be added
- New *FlashPix* stream(s) may be added
- New subimage(s) may be added to a *FlashPix* image object
- New property set(s) may be added
- New property(s) may be added to an existing property set section
- Element(s) may be added to core *FlashPix* property set vector properties that are defined as variable length
- Value of a core *FlashPix* stream field may be changed
- Value of a core property set property may be changed

There are five restrictions to structuring the data elements of an extension. First, new fields may not be added to existing *FlashPix* streams. Second, due to the inability to independently ensure property ID code uniqueness, only registered extensions may add properties to an existing property set section. Third, private extensions may not change the value of a core *FlashPix* stream field or a core property set property. Fourth, extensions can only add vector elements that are not already used by core or other extensions present in the file. Upon removal of an extension, the vector element values associated with the extension must be replaced with NULL and the vector must not be reordered. Fifth, only registered extensions can add elements to core property set vector properties.

Although there are a few practical examples where reasonable core reader actions could be defined for when an extension has changed the value of a core *FlashPix* stream field or a core property set property, these core reader actions must be considered in defining the core *FlashPix* specification. It is impractical to expect all core reader software to be updated to incorporate default actions identified in the course of developing new extensions. The definition of extensions must not impact the core definition unless some compelling feature set is identified which the *FlashPix* format Advisory Council agrees to include in a revised definition of the core *FlashPix* format. Therefore, efforts to define public extensions will avoid impacting core *FlashPix* stream field and core property set property values.

If the *FlashPix* image view object contains an extended *FlashPix* image object, the extended *FlashPix* image object must be listed in the extension list of the *FlashPix* image view object. The details of how the *FlashPix* image object is extended are left to the extension list property set of the *FlashPix* image object itself. The extension list entry for the extended *FlashPix* image object must use the *FlashPix* image object class ID as the extension ID and the *FlashPix* image object storage name as the extension description.

If an extension is present in the *FlashPix* image view object which affects the output image appearance, an intermediate core *FlashPix* data object must be created as an intermediate source image object for core reader use. The creator transform of this source image object may not be a core viewing transform so it is clear to a core reader that this image object is truly its source and it will not attempt to resolve the creating transform. The intermediate source image object must be hierarchical, but is not required to be at the full resolution potential of images from which it is created. The extended authoring application may choose the resolution to make available. As core reader software cannot access data of a higher resolution than provided in the intermediate source image object, it is strongly recommended that data corresponding to at least 200dpi is provided. Further, the intermediate source image object does not have to be an exact representation of the output that is created from a reader supporting the extension. That may not be possible. Although the image content of the intermediate source image is also at the discretion of the authoring application, it is recommended that the closest feasible representation is provided.

The valid properties of the extension list property set are listed in Table 7.9. The extensions present in the *FlashPix* image view object are numbered for the convenience of grouping the descriptive information about each extension. Property ID codes

**TABLE 7.9** **Valid properties for the extension property list property set**

| **Property name** | **ID code** | **Type** |
|---|---|---|
| Used extension numbers | 0x10000000 | VT_UI2 \| VT_VECTOR |
| Extension name | 0x*iiii*0001 | VT_LPWSTR |
| Extension class ID | 0x*iiii*0002 | VT_CLSID |
| Extension persistence | 0x*iiii*0003 | VT_UI2 |
| Extension creation date | 0x*iiii*0004 | VT_FILETIME |
| Extension modification date | 0x*iiii*0005 | VT_FILETIME |
| Creating application | 0x*iiii*0006 | VT_LPWSTR |
| Extension description | 0x*iiii*0007 | VT_LPWSTR |
| Storage / stream pathname | 0x*iiii*1000 | VT_LPWSTR \| VT_VECTOR |
| *FlashPix* stream pathname | 0x*iiii*2000 | VT_LPWSTR \| VT_VECTOR |
| *FlashPix* stream field offset | 0x*iiii*2001 | VT_UI4 \| VT_VECTOR |
| Property set pathname | 0x*iiii*3000 | VT_LPWSTR \| VT_VECTOR |
| Property set ID codes | 0x*iiii*3jj1 | VT_LPWSTR \| VT_VECTOR |
| Property vector elements | 0x*iiii*3jj2 | VT_LPWSTR \| VT_VECTOR |

0xiiiixxxx describe the extension numbered 0x*iiii*.

**Used extension numbers property (required)**

This property lists all extension numbers *0xiiii* used in the extension list property set for the *FlashPix* image view object. The property value is an unordered array of *0xiiii* values.

All applications must update this property each time an extension is added to or removed from a *FlashPix* image view object.

**Extension name property (required)**

This property identifies the name of the extension. If the extension is registered, the name used must be that which is published in the official*FlashPix* Extension Specification. For private extensions, the name is the whatever short, descriptive label the authoring application chooses.

All applications must retain this property upon a save or copy function by default, or in accordance with the extension persistence.

**Extension class ID property (required)**

This property identifies a unique class ID for the extension. If the extension is registered, the class ID must be that which is published in the official*FlashPix* Extension Specification. For private extensions, the class ID is assigned by the authoring application.

All applications must retain this property upon a save or copy function by default, or in accordance with the extension persistence.

**Extension persistence property (required)**

This property identifies the persistence of the extension with respect to edits to the core data elements of the *FlashPix* image view object. The legal values for the extension persistence property are defined in Table 7.10.

It is the responsibility of the reader/writer application upon save or copy functions to retain the extension data elements by default, or in accordance with the extension persistence property.

All applications must retain this property upon a save or copy function by default, or in accordance with the extension persistence.

**Extension creation date property (optional unless extension persistence property is 0x2)**

This property specifies the time and date the authoring application added the extension to the *FlashPix* image view object. If the property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence value.

**TABLE 7.10** **Legal values of the existence persistence property**

| Value | Meaning |
|---|---|
| 0x0 | Extension is valid independent of core element edits |
| 0x1 | Extension is invalid upon core element edits |
| 0x2 | Extension is potentially invalid upon core element edits |

**Extension modification date property (optional unless extension persistence property is 0x2)**
This property specifies the time and date of the last modification to the extension. If the property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence value.

**Creating application property (optional)**
This property specifies the name of the application that authored the extension in the file. If the property exists, any application editing the extension must update the value and all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence value.

**Extension description property (optional)**
The description property is a short (<80 character) description of the extension. If the property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence value.

**Storage/stream pathname property (optional)**
This property lists the full storage or non-*FlashPix* stream name, including the path in the structured storage file from the *FlashPix* image view object storage, for each storage or non-*FlashPix* stream the extension added to the *FlashPix* image view object. The path is specified using the standard Unix file specification tokens: "/" represents a directory separator and must be the first character of the property value. Wildcard characters “*” and “?” (where “*” matches any 0 or more characters and “?” matches any 1 character) are permitted in the path portion of the property value. If a storage is listed in the extension list property set, its contents should not also be listed as they are assumed to also be associated with that extension. If this property is omitted it is assumed that no storages are added to the *FlashPix* image view object for the extension. If the property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence property value.

***FlashPix* stream pathname property (optional)**
This property lists the full *FlashPix* stream name, including the path from the *FlashPix* image view object storage, for each *FlashPix* stream the extension added to or modified in the *FlashPix* image view object. The path is specified using the standard Unix file specification tokens: "/" represents a directory separator and must be the first character of the property value. Wildcard characters “*” and “?” (where “*” matches any 0 or more characters and “?” matches any 1 character) are permitted in the path portion of

the property value. The array of values for the *FlashPix* stream pathname property and the *FlashPix* stream field offset property array of values for extension *iiii* are associated as described in Table 7.11. If this property is omitted it is assumed that no *FlashPix* streams are added to or modified in the *FlashPix* image view object for the extension. If the property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence property value.

#### ***FlashPix*** **stream field offset property (optional)**

This property lists the byte offsets (after the header) into the *FlashPix* stream identified with the *FlashPix* stream pathname property array of fields modified by the extension. The array of values for the *FlashPix* stream field offset property and the *FlashPix* stream pathname property array of values for extension *0xiiii* are associated as described in Table 7.11. This property is required only if the *FlashPix* stream pathname property exists. If this property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence property value.

In the Table 7.11 example, there are two *FlashPix* stream data elements associated with extension 0x17. The first, at Index = 0, is an added *FlashPix* stream as there is a *FlashPix* stream pathname value but the *FlashPix* stream field offset is 0xFFFFFFFF. The second, at index 0xjj = 1, is a field in a core *FlashPix* stream who's value is not among those defined in the core *FlashPix* format. This is indicated by the presence of a non-0xFFFFFFFF *FlashPix* stream field offset value in addition to a *FlashPix* stream pathname value.

**TABLE 7.11** **Example values of *FlashPix* stream identification**

| **Property** | **Index = 0** | **Index = 1** |
|---|---|---|
| 0x00172000 | stream x pathname | stream y pathname |
| 0x00172001 | 0xFFFFFFFF | 64 |

#### **Property set pathname property (optional)**

This property is an array that lists the full property set name, including the path from the *FlashPix* image object storage, for each property set the extension 0xiiii added, added to, or modified in the *FlashPix* image object. The path is specified using the standard Unix file specification tokens: "/" represents a directory separator and must be the first character of the property value. Wildcard characters "*" and "?" (where "*" matches any 0 or more characters and "?" matches any 1 character) are permitted in the path portion of the property value. Table 7.12 shows an example of how the property set pathname, property set ID codes, and property set vector elements for extension 0xiiii are associated. The array index of the property set pathname property corresponds to 0xjj in the properties 0xiiii3jj1 and 0xiiii3jj2.

This property set is optional and if omitted it is assumed that no property sets are added, added to, or modified in the *FlashPix* image object for the extensions. If the property exists all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence property value.

**Property set ID codes property (optional)**

This property lists the ID codes of properties which have been added to a core property set, or defined with non-core values by an extension to the *FlashPix* image object. The value of each array position of the property is a VT_LPWSTR that may be composed of comma separated values each of which are either an individual property ID code or hyphen-separated pair of property ID codes. The array of values for the property set ID codes and the property vector elements for particular property set 0xjj and extension 0xiiii are associated as described in Table 7.12. When a new property set is added by an extension, the property set ID codes property is not required. This property is required if an extension adds properties to a core property set or modifies core property set properties. If the property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence property value.

**Property vector elements property (optional)**

Extensions can add vector elements to core properties that are defined as variable length vectors. This property lists the vector index for the values added to a particular vector property. The value of each array position of the property is a VT_LPWSTR that may be composed of comma separated values each of which are either an individual vector element or hyphen-separated pair of vector elements. The array of values for the property vector elements and the property set ID codes for a particular property set 0xjj and extension 0xiiii are associated as described in Table 7.12. This property is only required when an extension adds vector elements to a core property set property. If the vector elements property is present, and vector elements have not been added to its associated property set ID code(s), then the value of this property must be NULL. If the property exists, all applications must retain it upon save or copy functions by default, or in accordance with the extension persistence property value.

In the Table 7.12 example, there are three property sets associated with extension 0x19. The first index of the property 0x000193000, which corresponds to 0xjj=00, is a new property set being added by the extension as there is a property set pathname value, but the property set ID codes and property vector element properties for 0xjj=00 are not listed. The second index of the property 0x00193000, which corresponds to 0xjj=01, is a core property set in which property ID Codes 0x00011001-0x00011005 and 0x00001200 are being added by the extension. The third index of the property 0x00193000, which corresponds to 0xjj=02, is a core property set in which property ID code 0x00033000 is of type VT_VECTOR and the extension has added values in elements 3,4, and 5 of that vector. Property ID codes $00044001-$00044004 are new ID codes being added to the property set by the extension as well. In this case since the property ID codes are new, the value of 0x00193022 for this array position is assigned to NULL. This example also shows that the extension has added a value in element 2 of both the vectors defined by existing property ID codes, 0x00055000 and 0x00066000.

**TABLE 7.12** **Example values of property set identification**

| Property | Index=0 | Index=1 | Index=2 |
|---|---|---|---|
| 000193000 | PS x pathname (0xjj=00) | PS y pathname (0xjj=01) | PS z pathname (0xjj=02) |
| 000193011 | $000011001-$000011005, $000012000 | | |
| 000193021 | $000033000 | $000044001-$000044004 | $000055000, $000066000 |
| 000193022 | 3,4,5 | NULL | 2 |

# 7.2 Viewing Transform Parameters

The viewing transform parameters allow a view other than the raw image data itself. There are four classes of viewing parameters: selection, filtering, spatial orientation, and color reproduction. The application must provide a user interface that will help users work with and preview the viewing parameters.

## 7.2.1 Selection via Rectangle of Interest

The rectangle of interest property lets applications specify which part of the image to retain. Only a rectangular region with sides applications specify parallel to the edges of the image can be specified.

The rectangle of interest is specified in the rectangle of interest property in the transform property set as a horizontal, rectangular box. Four values are needed to specify the rectangle: left edge (*x*), top edge (*y*), width (*w*), and height (*h*). Each of these values is specified as a floating point number in the resolution-independent coordinate system described in Section 2.1.1.

The rectangle of interest is always interpreted in the context of a specific resolution layer. For example, if the layer has *N* pixels across (in *x*), numbered 0 to *N*-1, the range of columns is given by:

$$\lfloor N \times x + 0.5 \rfloor \le \text{columns} \le \lfloor N \times (x + w) + 0.5 \rfloor \tag{7.3}$$

Note that while pixel locations should be used to identify a pixel neighborhood for spatial operations, the rectangle of interest box should run from the left edge of the unit square surrounding the first pixel to the right edge of the unit square surrounding the last pixel. This combination of parameters provides a rectangle of interest which is robust to resolution changes. If the user selects a region manually at a specific resolution, the rectangle parameters should be calculated as follows, using the left edge as an example: calculate the location of the leftmost pixel in the continuous, resolution-dependent coordinate system described in Section 2.1.2. Scale this value to the resolution-independent system described inSection 2.1.1. This process will provide a stable description of the region when it is expressed at a higher or lower resolution.

The rectangle of interest does not imply any scaling or shifting into a “standard coordinate space.” The rectangle only specifies that pixels outside the rectangle should be considered fully transparent. If the user desires that the area inside the rectangle of interest be displayed as “the whole image,” the spatial orientation matrix (described in Section 7.2.3) should map the top-left corner of the rectangle to (0,0) and the bottom-right of the rectangle to (*w/h*,1).

## 7.2.2 Filtering

The *FlashPix* format viewing parameter that sharpens or blurs the image is referred to as image filtering. The degree of filtering is controlled by the filtering property in the transform property set. Positive values produce a sharper image; negative values produce a smoother, more blurry image, with less detail. A value of 0 leaves the image unchanged. Control is scaled so that one unit of filtering makes a just-noticeable change. Values between -20 and 20 may be expected to produce reasonable results. The default value of the filtering parameter is zero. It is stored as an IEEE 4-byte floating point number.

The influence of the filtering control is independent of image resolution. Proper operation of image filtering is closely tied to the resolution-independent rendering algorithms of the *FlashPix* format. When a reader requests actual *FlashPix* image data, it selects the best resolution layer to build the data from, and creates a digital filter to apply the requested degree of sharpening or blurring to that data.

To provide a degree of filtering independent of resolution, a reader also needs information about the imaging capabilities of the physical subsystems that generated the digital image and the subsystem that will be used to print the image. If this information is not available, a reader will use defaults that will yield good results in non-critical situations. Peripheral manufacturers could add value to systems that use the *FlashPix* format by providing this information with their devices (as part of their device driver or profiles).

### 7.2.2.1 The Measure

Filtering is defined as the change in the acutance of the complete imaging system. Acutance is a “one number” description of system sharpness. A number of definitions have been used, but in general acutance is a measure of the degree of blur introduced by all the elements of an imaging system, including the human eye that views the image. The only element missing from acutance is the degree of detail in the scene itself. Most modern measures of acutance are based on some integral of the overall MTF (modulation transfer function) of the system, including some eye MTF, scaled so that one unit of acutance change is a just-noticeable difference. Higher acutances are associated with sharper imaging systems. A specific definition of acutance has been adopted for the *FlashPix* format.

The digital image is an intermediate result in the imaging chain that starts with image capture and ends with physical reproduction. Acutance is a measure of the ability of the chain to produce a sharp image. The customer can specify an increase or decrease in the acutance of this system. This will determine the digital filter used to increase or decrease the sharpness of the image.

#### 7.2.2.2 Subsystem information

To precisely calculate the change in acutance of the system, some information about real devices is required. In particular, a reader requires an estimate of the capability of the input and output devices to reproduce fine image details. It also helps to have some estimate of how large the image will appear to the viewer, since the human eye is the final element in the imaging system.

Information on device sharpness is not generally available on the desktop today, even though manufacturers have measured it. An estimate of relative image size can often be made based on how the image is being used in a composition. In any case, the system is relatively insensitive to these factors, and the internal default values generally give good results.

Accurate information passed to a reader will improve the predictability and reliability of filtering as a change in acutance. Knowledge of these parameters also may permit a reader to work from a lower resolution level of the hierarchy, enabling faster rendering with no loss in image quality.

Four pieces of information may be utilized by a reader to determine acutance:

- A number describing the MTF (sharpness capability) of the capture operation sequence (stored in the sharpness approximation property of the file source group of the image info property set, Section 6.2)
- A number describing the MTF (sharpness capability) of the printer
- A number describing the MTF (blurring) of the prefilter used in building the *FlashPix* format hierarchy (stored for each resolution in the decimation prefilter width property of the resolution description groups of the image contents property set, Section 3.1.5.2).
- A number describing the relative size of the image as used by the customer

#### 7.2.2.3 User Sharpening Adjustment

One challenge with providing image sharpening on the desktop is the difficulty novice users have in selecting the correct value. Generally, some sharpening is required to correct for the input/output devices, and some is added by the user for artistic effect. (A reader does not automatically add sharpening to correct for input and output devices, even when it has that information, but an application could do so, for example, by suggesting a filtering parameter setting.)

Handling sharpening adjustment for artistic effect is more difficult. However, the *FlashPix* filtering system can enable a very simple means for users to preview the sharpening effects if the subsystem information listed above is known. Specifically, numbers describing the MTFs of the printer and the monitor must be available.

### 7.2.3 Spatial Orientation

The *FlashPix* format provides spatial orientation parameters to rotate, scale, shear, and translate an image. These parameters are stated in terms of a 4×4 matrix, which maps image points from the displayed form of the image to the original image. In this form, the source points for resampling can be directly calculated.

Two-dimensional affine transformations are used to implement the spatial changes needed for image viewing. The general transform requires interpolation of the image data. A subset of the operations can be executed much faster because there is no change in the definition of the pixels (they only move around). It is the responsibility of the viewing engine to recognize these operations and to execute them efficiently.

The spatial orientation viewing parameters are specified by a 4×4 matrix as described by Equation 7.4:

$$\begin{bmatrix} x' \\ y' \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix} \tag{7.4}$$

Each parameter is a floating point number. The 2-D affine transformations that map from the displayed form of the image to the original image are encoded in the six matrix elements $a_{11,}$ $a_{12}$, $a_{14}$, $a_{21}$, $a_{22}$, $a_{24}$

In general, applying the affine transformations to the image will require resampling of the image. Note that the offset parameters $a_{14}$ and $a_{24}$ do matter for rotations and flips. In particular, rapid execution of these operations depends on certain values of the offsets.

The 4x4 matrix is stored, in the spatial orientation property of the Transform Property Set (Section 7.1.4), as an array of 16 IEEE 4-byte floating point numbers.

### 7.2.4 Tone and Color Corrections

The *FlashPix* format provides a colortwist matrix property in the transform property set to allow users to make small changes to the tone and color of an image. The changes are not designed to correct specific faults in the imaging chain, but to provide the user a simple way to specify common color corrections.

### 7.2.4.1 Color Images

Tone and color changes are applied in two ways. First, an affine matrix (colortwist matrix) on the vector (*Luma*, $Chroma_1$, $Chroma_2$, 1) of normalized PhotoYCC values is used to adjust lightness, saturation, and color balance. Many other kinds of simple changes can be effected with this matrix. The use of a matrix for this operation makes it easy to combine color operations done at different levels of an image composite. The matrix elements may be identified as follows:

$$\begin{bmatrix} Luma' \\ Chroma'_1 \\ Chroma'_2 \\ \end{bmatrix} = \begin{bmatrix} b_{YY} & b_{YC_1} & b_{YC_2} & b_{Yoff} \\ b_{C_1Y} & b_{C_1C_1} & b_{C_1C_2} & b_{C_1off} \\ b_{C_2Y} & b_{C_2C_1} & b_{C_2C_2} & b_{C_2off} \\ 0 & 0 & 0 & b_{off} \end{bmatrix} \begin{bmatrix} Luma \\ Chroma_1 \\ Chroma_2 \\ 1 \end{bmatrix} \tag{7.5}$$

Each element of the matrix is a floating point number. No limitation is placed on the matrix elements except for the left most three elements on the bottom row, which should all be zero. The input data to and the output data from the colortwist matrix MUST be normalized PhotoYCC. Although matrices for conversion into and out of normalized PhotoYCC may be concatenated with the colortwist matrix for efficiency, the colortwist matrix stored in the *FlashPix* image view object must NOT contain the matrix component for those conversions.The terms of the vector resulting from the matrix multiplication should be clipped to the correct limits for the desired output color space.

The values in the matrix are defined in terms of modifications to normalized PhotoYCC. Note that the tone and color correction matrix maps the source image to the destination image, which is different than the spatial orientation affine matrix.

### 7.2.4.2 Monochrome Images

Monochrome images should be treated as NIF RGB images where $R = G = B = X$. This allows a “tint” to be applied to a monochrome image using the viewing parameters, without requiring multichannel data to be stored in the *FlashPix* image.

## 7.2.5 Contrast adjustments

Contrast adjustments are controlled by the contrast adjustment property in the transform property set. It specifies a function through which the image data is passed. This operation may be implemented as a 1D lookup table or an equation applied to the image data once transformed to the proper color representation. The contrast change must be performed on an RGB representation of the image data which depends on the implementation method chosen.   Additional (fixed) matrix operations (aside from the colortwist matrix) will be required to convert to and from the RGB values used in the contrast adjustment.

The first matrix implements the conversion from the input color space to normalized PhotoYCC where the colortwist matrix is applied. The matrices following the colortwist converts normalized PhotoYCC data to the proper RGB representation for application of the contrast parameters. The choice of RGB representations includes normalized RGB(*rgb*) and a greater than 8 bit integer $\mathrm{RGB_m}$ representation. Once applied, some additional matrices are required to return the image data back to its pre-color/tone correction representation which can then be converted to a different output color space if desired.

The color and tone corrections can be applied to both NIFRGB and PhotoYCC data. While the colortwist matrix is applied to a normalized PhotoYCC data representation and contrast is applied in one of two RGB data representations, the matrices provided to do these conversions alone are not sufficient to convert from the NIFRGB color space to the PhotoYCC space or visa versa. They are sufficient to do the metric conversions required for the tone and color corrections, but are not for color space transformations. Therefore when the tone and color corrections are applied, the modified image data should be in the same color space as the original input color space at which point additional processing can be performed to convert the color space if needed. It may be possible, however, to concatenate some of the additional color space transformation processing with the color and tone correction matrices and LUTs to simplify the processing.

The desired contrast is specified by a single floating point number. A value of 1.0 indicates no contrast change; values greater than 1 provide higher contrast.

The contrast adjustment is stored as an IEEE 4-byte floating point number. Its default value is 1.0.

If the contrast parameter is other than 1.0 and the equation method for contrast is used, the procedure for implementing tone and color viewing parameters is as follows:

1. Combine a normalized PhotoYCC→normalized RGB conversion matrix with the given colortwist matrix and the input→normalized PhotoYCC matrix, producing a new matrix *M*'.
2. Pass the image data through *M*'
3. Apply the contrast modification through equation (7.6) to the image data resulting from *M*'.
4. Pass the contrast modified data through a series of matrices to convert the contrast modified normalized RGB back to the original color space which can then be converted to a different output space if needed.

If the contrast parameter is other than 1.0 and the LUT method for contrast is used, the procedure for implementing tone and color viewing parameters is as follows:

1. Combine a normalized PhotoYCC→normalized RGB conversion matrix and a normalized RGB→$\mathrm{RGB_m}$with the given colortwist matrix and the input→normalized PhotoYCC matrix, producing a new matrix *M*'.
2. Pass the image data through *M*' limiting the $\mathrm{RGB}_m$ values to the range 0<=X<2M.
3. Create the LUT, $K_{LUT}$, using equation (7.7). The output of $K_{LUT}$ is normalized RGB. If the $\mathrm{RGB_m}$ representation is preferred, cascade equation (7.8) with equation (7.7) to create $K'_{LUT}$.
4. Apply the contrast modification through either $K_{LUT}$ or $K'_{LUT}$.

5. Pass the contrast modified data through a series of matrices to convert the contrast modified normalized RGB or RGB$_m$ data back to the original color space which can then be converted to a different output space if needed.

For example, to increase the contrast by 20%, $K = 1.2$, where $K$ is the contrast parameter. The following equations specify the contrast modification function, $f_K$, to be applied to *rgb* data:

$$p = 0.43$$

$$f_K(K, j) = \begin{cases} j < 0 & -p \times \left(\frac{-j}{p}\right)^K \\ j = 0 & 0 \\ 0 < j & p \times \left(\frac{j}{p}\right)^K \end{cases} \tag{7.6}$$

If the contrast modification is to be applied as a LUT, the input data must be converted to *RGB*$_m$ space. The following equation can then be used to generate the LUT, $K_{LUT}$, where $j'$ is the input pixel value in *RGB*$_m$ space, where $M = 2^{k-1}$:

$$K_{LUT}[j'] = f_K\left(K, \frac{j' - \frac{M}{2}}{M}\right) \tag{7.7}$$

The output of $K_{LUT}$ is *rgb* space. If *RGB*$_m$ space is desired, the following equation can be cascaded with the equation for $K_{LUT}$, creating $K'_{LUT}$:

$$K'_{LUT}[j'] = K_{LUT}[j'] \times M + \frac{M}{2} \tag{7.8}$$

Note that if the “nine LUT method” is used for executing matrix multiplications, the 1D LUT and the final matrix can be combined into a single step.

## 7.3 Sequence of Viewing Parameters

The viewing parameters are not commutative. They must be applied in the following order: selection, filtering, spatial orientation, specification of result aspect ratio. The tone and color operation can be applied at any stage after the filtering step.

An application must carefully record edits to the image once it has been loaded. If the application wishes to modify the viewing parameters of the original image to reflect the edits, it must determine how to express the changes in terms of the allowed parameters.

### 7.3.1 Coordinate System

Both the selection and spatial orientation operations require clear specification of a coordinate system for the image. A *FlashPix* image can have any size, but the viewing parameters have no knowledge of the number of pixels in the image. The image is stored with an implicit orientation. The upper-left hand corner of the image has the coordinates (0, 0). The height of the image is 1.0, so that the lower-left corner has the coordinates (0, 1.0). The lower-right corner has the coordinates ($R$, 1.0), where $R$ is the aspect ratio of the image.

The pixel locations are specified exactly for spatial operations: in a layer with $N$ pixels in the x direction, the first pixel is centered at $0.5/N$, the second is centered at $1.5/N$, etc.

The affine spatial transform may rotate and shift the image data. Still, the coordinates implied by the specification of the result aspect ratio refer to the original coordinate system of the image.

For each pixel in the result rectangle (specified by the result aspect ratio), apply the affine transform to identify the corresponding location in the original image.

IF this point falls within the rectangle of interest,

THEN: This point will probably not fall in the middle of an original pixel. Use an interpolation scheme to calculate the values for the new pixel. Map these values through the tone and color transform to get the final image values.

ELSE: The point is outside of the defined image area. In the context of a viewer, nothing should be displayed—the viewer must define its background level. In the context of compositing, this pixel has 100% transparency.

### 7.3.2 Image Size and Limits

The *FlashPix* format is a resolution-independent format. There is no implicit scale to the image. The image should be displayed fully at whatever size and resolution suits the viewing device.

After a spatial orientation change leaves the image tilted or sheared, the resulting image is defined to be the area in the rectangle (0,0) to ($R_{result}$,1), even though there may be legitimate image data outside this region. If the application wants the image to be rotated and displayed in its entirety, the affine matrix must also perform a scale to shrink the image such that it fits entirely inside the output rectangle.

September 11, 1996

# APPENDIX A Structured Storage

Note: This document is meant to accompany the Microsoft OLE Structured Storage Reference Implementation, hereafter referred to as the ‘Software’. If this document and functionality of the Software conflict, the actual functionality of the Software represents the correct functionality. Microst assumes no responsibility for any damages that might occur either directly or indirectly from these discrepancies or inaccuracies. Microsoft may have trademarks, copyrights, patents or pending patent applications, or other intellectual property rights covering subject matter in this document and in the Software. The furnishing of this document does not give you a license to these trademarks, copyrights, patents, or other intellectual property rights and any license rights granted are limited to those set forth in the End User License Agreement accompanying this document.

# A.1 Compound File Binary Format

## A.1.0 Overview

A Compound File is made up of a number of **virtual streams**. These are collections of data that behave as a linear stream, although their on-disk format may be fragmented. Virtual streams can be user data, or they can be control structures used to maintain the file. Note that the file itself can also be considered a virtual stream.

All allocations of space within a Compound File are done in units called **sectors**. The size of a sector is definable at creation time of a Compound File, but for the purposes of this document will be 512 bytes. A virtual stream is made up of a sequence of sectors.

The Compound File uses several different types of sector: *Fat*, *Directory*, *Minifat*, *DIF*, and *Storage*. A separate type of 'sector' is a *Header*, the primary difference being that a Header is always 512 bytes long (regardless of the sector size of the rest of the file) and is always located at offset zero (0). With the exception of the header, sectors of any type can be placed anywhere within the file. The function of the various sector types is discussed below.

In the discussion below, the term **SECT** is used to describe the location of a sector within a virtual stream (in most cases this virtual stream is the file itself). Internally, a SECT is represented as a ULONG.

## A.1.1 Sector Types

```
[4 bytes]   typedef unsigned long ULONG;
[2 bytes]   typedef unsigned short USHORT;
[2 bytes]   typedef short OFFSET;
[4 bytes]   typedef ULONG SECT;
[4 bytes]   typedef ULONG FSINDEX;
[2 bytes]   typedef USHORT FSOFFSET;
[4 bytes]   typedef ULONG DFSIGNATURE;
[1 byte]    typedef unsigned char BYTE;
[2 bytes]   typedef unsigned short WORD;
[4 bytes]   typedef unsigned long DWORD;
[2 bytes]   typedef WORD DFPROPTYPE;
[4 bytes]   typedef ULONG SID;
[16 bytes]  typedef CLSID GUID;

[8 bytes]   typedef struct tagFILETIME {
                                   DWORD dwLowDateTime;
                                   DWORD dwHighDateTime;
                                    } FILETIME, TIME_T;


[4 bytes]   const SECT DIFSECT= 0xFFFFFFFC;
[4 bytes]   const SECT FATSECT= 0xFFFFFFFD;
[4 bytes]   const SECT ENDOFCHAIN= 0xFFFFFFFE;
[4 bytes]   const SECT FREESECT= 0xFFFFFFFF;
```

### A.1.1.1 Header

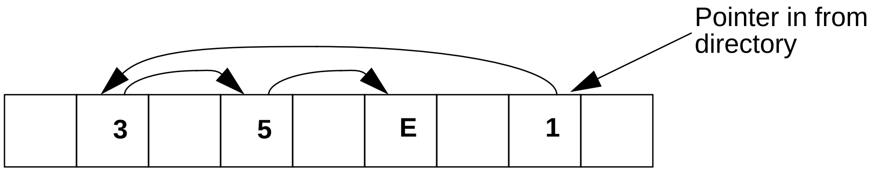
```
struct StructuredStorageHeader{// [offset from start in bytes, length
                                  // in bytes]
    BYTE      _abSig[8];          // [000H,08] {0xd0, 0xcf, 0x11, 0xe0,
                                  // 0xa1, 0xb1, 0x1a, 0xe1} for current
                                  // version, was {0x0e, 0x11, 0xfc,
                                  // 0x0d, 0xd0, 0xcf, 0x11, 0xe0} on old,
                                  // beta 2 files (late ’92) which are also
                                  // supported by the reference
                                  // implementation
    CLSID     _clid;              // [008H,16] class id (set with
                                  // WriteClassStg, retrieved with
                                  // GetClassFile/ReadClassStg)
    USHORT    _uMinorVersion;     // [018H,02] minor version of the
                                  // format: 33 is written by reference
                                  // implementation
    USHORT    _uDllVersion;       // [01AH,02] major version of the dll/
                                  // format: 3 is written by reference
                                  // implementation
    USHORT    _uByteOrder;        // [01CH,02] 0xFFFE: indicates Intel
                                  // byte-ordering
    USHORT    _uSectorShift;      // [01EH,02] size of sectors in power-
                                  // of-two (typically 9, indicating 512-
                                  // byte sectors)
    USHORT    _uMiniSectorShift;  // [020H,02] size of mini-sectors
                                  // in power-of-two (typically 6,
                                  // indicating 64-byte mini-sectors)
    USHORT    _usReserved;        // [022H,02] reserved, must be zero
    ULONG     _ulReserved1;       // [024H,04] reserved, must be zero
    ULONG     _ulReserved2;       // [028H,04] reserved, must be zero
    FSINDEX   _csectFat;          // [02CH,04] number of SECTs in the FAT
                                  // chain
    SECT      _sectDirStart;      // [030H,04] first SECT in the FAT
                                  // Directory chain
    DFSIGNATURE_signature;        // [034H,04] signature used for transac
                                  // tioning must be zero. The reference
                                  // implementation does not support
                                  // transactioning
    ULONG     _ulMiniSectorCutoff;// [038H,04] maximum size for
                                  // mini-streams: typically 4096 bytes
    SECT      _sectMiniFatStart;  // [03CH,04] first SECT in the
                                  // mini-FAT chain
    FSINDEX   _csectMiniFat;      // [040H,04] number of SECTs in the
                                  // mini-FAT chain
    SECT      _sectDifStart;      // [044H,04] first SECT in the DIF
                                  // chain
    FSINDEX   _csectDif;          // [048H,04] number of SECTs in the DIF
                                  // chain
    SECT      _sectFat[109];      // [04CH,436] the SECTs of the first
                                  // 109 FAT sectors
    };
```

The *Header* contains vital information for the instantiation of a Compound File. Its total length is 512 bytes. There is exactly one *Header* in any Compound File, and it is always located beginning at offset zero in the file.

### A.1.1.2 Fat Sectors

The **Fat** is the main allocator for space within a Compound File. Every sector in the file is represented within the Fat in some fashion, including those sectors that are unallocated (free). The Fat is a virtual stream made up of one or more Fat Sectors.

Fat sectors are arrays of SECTs that represent the allocation of space within the file. Each stream is represented in the Fat by a **chain**, in much the same fashion as a DOS file-allocation-table (FAT). To elaborate, the set of Fat Sectors can be considered together to be a single array -- each cell in that array contains the SECT of the next sector in the chain, and this SECT can be used as an index into the Fat array to continue along the chain. Special values are reserved for chain terminators (ENDOFCHAIN = 0xFFFFFFFE), free sectors (FREESECT = 0xFFFFFFFF), and sectors that contain storage for Fat Sectors (FATSECT = 0xFFFFFFFD) or DIF Sectors (DIFSECT = 0xFFFFFFC), which are not chained in the same way as the others.



Chaining

The locations of Fat Sectors are read from the DIF (Double-indirect Fat), which is described below. The Fat is represented in itself, but not by a chain – a special reserved SECT value (FATSECT = 0xFFFFFFFD) is used to mark sectors allocated to the Fat.

A SECT can be converted into a byte offset into the file by using the following formula: SECT << ssheader._uSectorShift + sizeof(ssheader). This implies that sector 0 of the file begins at byte offset 512, not at 0.

### A.1.1.3 MiniFat Sectors

Since space for streams is always allocated in sector-sized blocks, there can be considerable waste when storing objects much smaller than sectors (typically 512 bytes). As a solution to this problem, we introduced the concept of the **MiniFat**. The MiniFat is structurally equivalent to the Fat, but is used in a different way. The virtual sector size for objects represented in the Minifat is 1 << ssheader._uMiniSectorShift (typically 64 bytes) instead of 1 << ssheader._uSectorShift (typically 512 bytes). The storage for these objects comes from a virtual stream within the Multistream (called the **Ministream**).

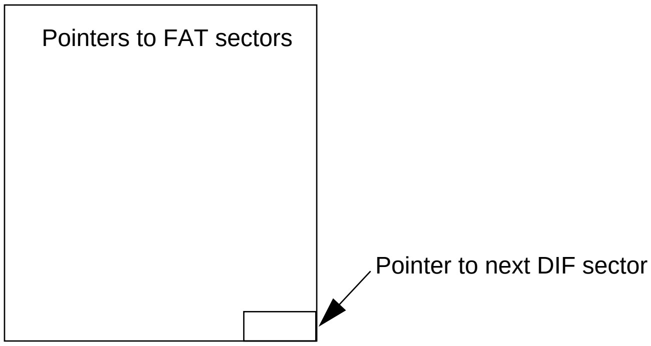
The locations for MiniFat sectors are stored in a standard chain in the Fat, with the beginning of the chain stored in the header.

A Minifat sector number can be converted into a byte offset into the ministream by using the following formula: SECT << ssheader._uMiniSectorShift. (This formula is different from the formula used to convert a SECT into a byte offset in the file, since no header is stored in the Ministream)

The Ministream is chained within the Fat in exactly the same fashion as any normal stream. It is referenced by the first Directory Entry (SID 0).

### A.1.1.4 DIF Sectors

**DIF Sector**



The **Double-Indirect Fat** is used to represent storage of the Fat. The DIF is also represented by an array of SECTs, and is chained by the terminating cell in each sector array (see the diagram above). As an optimization, the first 109 Fat Sectors are represented within the header itself, so no DIF sectors will be found in a small (< 7 MB) Compound File.

The DIF represents the Fat in a different manner than the Fat represents a chain. A given index into the DIF will contain the SECT of the Fat Sector found at that offset in the Fat virtual stream. For instance, index 3 in the DIF would contain the SECT for Sector #3 of the Fat.

The storage for DIF Sectors is reserved in the Fat, but is not chained there (space for it is reserved by a special SECT value , DIFSECT=0xFFFFFFFC). The location of the first DIF sector is stored in the header.

A value of ENDOFCHAIN=0xFFFFFFFE is stored in the pointer to the next DIF sector of the last DIF sector.

### A.1.1.5 Directory Sectors

```
typedef enum tagSTGTY {
    STGTY_INVALID= 0,
    STGTY_STORAGE= 1,
    STGTY_STREAM= 2,
    STGTY_LOCKBYTES= 3,
    STGTY_PROPERTY= 4,
    STGTY_ROOT= 5,
    } STGTY;

typedef enum tagDECOLOR {
    DE_RED= 0,
    DE_BLACK= 1,
    } DECOLOR;

struct StructuredStorageDirectoryEntry {// [offset from start in bytes,
                                          // length in bytes]
    BYTE       _ab[32*sizeof(WCHAR)];     // [000H,64] 64 bytes. The
                                          // Element name in Unicode,
                                          // padded with zeros to fill
                                          // this byte array
    WORD       _cb;                       // [040H,02] Length of the
                                          // Element name in characters,
                                          // not bytes
    BYTE       _mse;                      // [042H,01] Type of object:
```

```
                                              // value taken from the STGTY
                                              // enumeration
    BYTE       _bflags;                       // [043H,01] Value taken from
                                              // DECOLOR enumeration.
    SID        _sidLeftSib;                   // [044H,04] SID of the left-
                                              // sibling of this entry in the
                                              // directory tree
    SID        _sidRightSib;                  // [048H,04] SID of the right-
                                              // sibling of this entry in the
                                              // directory tree
    SID        _sidChild;                     // [04CH,04] SID of the first
                                              // child acting as the root of
                                              // all the children of this el-
                                              // ement(if_mse=STGTY_STORAGE)
    GUID       _clsId;                        // [050H,16]CLSID of this stor-
                                              // age (if_mse=STGTY_STORAGE)
    DWORD      _dwUserFlags;                  // [060H,04] User flags of this
                                              // storage
                                              // (if_mse=STGTY_STORAGE)
    TIME       _T_time[2];                    // [064H,16] Create/Modify
                                              // time-stamps
                                              // (if_mse=STGTY_STORAGE)
    SECT       _sectStart;                    // [074H,04] starting SECT of
                                              // the stream
                                              // (if_mse=STGTY_STREAM)
    ULONG      _ulSize;                       // [078H,04] size of stream in
                                              // bytes (if_mse=STGTY_STREAM)
    DFPROPTYPE _dptPropType;                  // [07CH,02] Reserved for fu
                                              // ture use. Must be zero.
    };
```

The **Directory** is a structure used to contain per-stream information about the streams in a Compound File, as well as to maintain a tree-styled containment structure. It is a virtual stream made up of one or more Directory Sectors. The Directory is represented as a standard chain of sectors within the Fat. The first sector of the Directory chain (the Root Directory Entry)

Each level of the containment hierarchy (i.e. each set of siblings) is represented as a red-black tree. The parent of this set of sibilings will have a pointer to the top of this tree. This red-black tree must maintain the following conditions in order for it to be valid:

1. The root node must always be black.

2. No two consecutive nodes may both be red.

3. The left child must always be less than the right child. This relationship is defined as:

   - A node with a shorter name is less than a node with a longer name (i.e. compare the length of the name)
   - For nodes with the same length names, compare the two names.

The simplest implementation of the above invariants would be to mark every node as black, in which case the tree is simply a binary tree.

A Directory Sector is an array of Directory Entries, a structure represented in the diagram below. Each user stream within a Compound File is represented by a single Directory Entry. The Directory is considered as a large array of Directory Entries. It is useful to note that the Directory Entry for a stream remains at the same index in the Directory array for the life of the stream – thus, this index (called an **SID**) can be used to readily identify a given stream.

The directory entry is then padded out with zeros to make a total size of 128 bytes.

Directory entries are grouped into blocks of four to form Directory Sectors.

### A.1.1.5.1 Root Directory Entry

The first sector of the Directory chain (also referred to as the first element of the Directory array, or SID 0) is known as the **Root Directory Entry** and is reserved for two purposes: First, it provides a root parent for all objects stationed at the root of the multi-stream. Second, its function is overloaded to store the size and starting sector for the Mini-stream.

The Root Directory Entry behaves as both a stream and a storage. All of the fields in the Directory Entry are valid for the root. The Root Directory Entry's Name field typically contains the string "RootEntry" in Unicode, although some versions of structured storage (particularly the preliminary reference implementation and the Macintosh version) store only the first letter of this string, "R" in the name. This string is always ignored, since the Root Directory Entry is known by its position at SID 0 rather than by its name, and its name is not otherwise used. New implementations should write "RootEntry" properly in the Root Directory Entry for consistency and support manipulating files created with only the "R" name.

### A.1.1.5.2 Other Directory Entries

Non-root directory entries are marked as either stream (STGTY_STREAM) or storage (STGTY_STORAGE) elements. Storage elements have a _clsid, _time[ ], and _sidChild values; stream elements may not. Stream elements have valid _sectStart and _ulSize members, whereas these fields are set to zero for storage elements (except as noted above for the Root Directory Entry).

To determine the physical file location of actual stream data from a stream directory entry, it is necessary to determine which FAT (normal or mini) the stream exists within. Streams whose _ulSize member is less than the _ulMiniSectorCutoff value for the file exist in the ministream, and so the _startSect is used as an index into the MiniFat (which starts at _sectMiniFatStart) to track the chain of mini-sectors through the mini-stream (which is, as noted earlier, the standard (non-mini) stream referred to by the Root Directory Entry's _sectStart value). Streams whose _ulSize member is greater than the _ulMiniSectorCutoff value for the file exist as standard streams – their _sectStart value is used as an index into the standard FAT which describes the chain of full sectors containing their data).

### A.1.1.6 Storage Sectors

Storage sectors are simply collections of arbitrary bytes. They are the building blocks of user streams, and no restrictions are imposed on their contents. Storage sectors are represented as chains in the Fat, and each storage chain (stream) will have a single Directory Entry associated with it.

## A.1.2 Examples

This section contains a hexadecimal dump of an example structured storage file to clarify the binary file format.

### A.1.2.1 Sector 0: Header

```
    _abSig     = DOCF 11E0 A1B1 1AE1
    _clid      = 0000 0000 0000 0000 0000 0000 0000 0000
    _uMinorVersion= 003B
    _uDllVersion= 3
    _uByteOrder= FFFE (Intel byte order)
    _uSectorShift= 9 (512 bytes)
    _uMiniSectorShift= 6 (64 bytes)
    _usReserved= 0000
    _ulReserved1= 00000000
    _ulReserved2= 00000000
    _csectFat  = 00000001
    _sectDirStart= 00000001
    _signature = 00000000
    _ulMiniSectorCutoff= 00001000 (4096 bytes)
    _sectMiniFatStart= 00000002
    _csectMiniFat= 00000001
    _sectDifStart= FFFFFFFE (no DIF, file is < 7Mb)
    _csectDIF  = 00000000
    _sectFat[] = 00000000 FFFFFFFF...(continues with FFFFFFFF)

000000:  D0CF 11E0 A1B1 1AE1  0000 0000 0000 0000   ................
000010:  0000 0000 0000 0000  3B00 0300 FEFF 0900   ................
000020:  0600 0000 0000 0000  0000 0000 0100 0000   ................
000030:  0100 0000 0000 0000  0010 0000 0200 0000   ................
000040:  0100 0000 FEFF FFFF  0000 0000 0000 0000   ................
000050:  FFFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF   ................
...
0001F0:  FFFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF   ................
```

### A.1.2.2 SECT 0: First (Only) FAT Sector

```
SECT 0: FFFFFFFD = FATSECT: marks this sector as a FAT sector.
              Referred to in header by _sectFat[0]
SECT 1: FFFFFFFE = ENDOFCHAIN: marks the end of the directory chain,
              referred to in header by _sectDirStart
SECT 2: FFFFFFFE = ENDOFCHAIN: marks the end of the mini-fat, re
              ferred to in header by _sectMiniFatStart
SECT 3: 00000004 = pointer to the next sector in the “Stream 1” data.
              This sector is the first sector of “Stream 1”, it is re
              ferred to by the Directory Entry
SECT 4: ENDOFCHAIN (0xFFFFFFFE): marks the end of the “Stream 1”
              stream data. Further Entries are empty (FREESECT =
              0xFFFFFFFF)

000200:  FDFF FFFF FEFF FFFF  FEFF FFFF 0400 0000   ................
000210:  FEFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF   ................
...
0003F0:  FFFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF   ................
```

### A.1.2.3 SECT 1: First (Only) Directory Sector

```
SID 0: Root SID: Root Name = "R"
SID 1: Element 1 SID: Name = "Storage 1"
SID 2: Element 2 SID: Name = "Stream 1"
SID 3: Unused
```

#### A.1.2.3.1 SID 0: Root Directory Entry

```
     _ab        = ("R")(this should be “Root Entry”)
     _cb        = 00042(42 bytes,does not include double-null termi
     nator)
     _mse       = 05 (STGTY_ROOT)
     _bflags    = 00 (DE_RED)
     _sidLeftSib= FFFFFFFF (none)
     _sidRightSib= FFFFFFFF (none)
     _sidChild  = 00000001 (SID 1: “Storage 1”)
     _clsid     = 0067 6156 54C1 CE11 8553 00AA 00A1 F95B
     _dwUserFlags= 00000000 (n/a for STGTY_ROOT)
     _time[0]   = CreateTime   = 0000 0000 0000 0000 (none set)
     _time[1]   = ModifyTime   = 801E 9213 4BB4 BA01 (??)
     _sectStart = 00000003 (starting sector of MiniStream)
     _ulSize    = 00000240 (length of MiniStream in bytes)
     _dptPropType= 0000 (n/a)

000400:  0052 0000 0000 0000  0000 0000 0000 0000   .R..............
000410:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000420:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000430:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000440:  04200 0500 FFFF FFFF  FFFF FFFF 0100 0000   ................
000450:  0067 6156 54C1 CE11  8553 00AA 00A1 F95B   .gaVT....S.....[
000460:  0000 0000 0000 0000  0000 0000 801E 9213   ................
000470:  4BB4 BA01 0300 0000  4002 0000 0000 0000   K.......@.......
```

#### A.1.2.3.2 SID 1: “Storage 1”

```
     _ab        = ("Storage 1")
     _cb        = 0014 (20 bytes, including double-null terminator)
     _mse       = 01 (STGTY_STORAGE)
     _bflags    = 01 (DE_BLACK)
     _sidLeftSib= FFFFFFFF (none)
     _sidRightSib= FFFFFFFF (none)
     _sidChild  = 00000002 (SID 2: “Stream 1”)
     _clsid     = 0000 0000 0000 0000 0000 0000 0000 0000 (none set)
     _dwUserFlags= 00000000 (none set)
     _time[0]   = CreateTime   = 00000000 00000000 (none set)
     _time[1]   = ModifyTime   = 00000000 00000000 (none set)
     _sectStart = 00000000 (n/a)
     _ulSize    = 00000000 (n/a)
     _dptPropType= 0000 (n/a)

000480:  5300 7400 6F00 7200  6100 6700 6500 2000   S.t.o.r.a.g.e. .
000490:  3100 0000 0000 0000  0000 0000 0000 0000   1...............
0004A0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
0004B0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
0004C0:  1400 0101 FFFF FFFF  FFFF FFFF 0200 0000   ................
0004D0:  0061 6156 54C1 CE11  8553 00AA 00A1 F95B   .aaVT....S.....[
0004E0:  0000 0000 0088 F912  4BB4 BA01 801E 9213   ........K.......
0004F0:  4BB4 BA01 0000 0000  0000 0000 0000 0000   K...............
```

### A.1.2.3.3SID 2: “Stream 1”

```
    _ab        = ("Stream 1")
    _cb        = 0012 (18 bytes, including double-null terminator)
    _mse       = 02 (STGTY_STREAM)
    _bflags    = 01 (DE_BLACK)
    _sidLeftSib= FFFFFFFF (none)
    _sidRightSib= FFFFFFFF (none)
    _sidChild  = FFFFFFFF (n/a for STGTY_STREAM)
    _clsid     = 0000 0000 0000 0000 0000 0000 0000 0000 (n/a)
    _dwUserFlags= 00000000 (n/a)
    _time[0]   = CreateTime   = 00000000 00000000 (n/a)
    _time[1]   = ModifyTime   = 00000000 00000000 (n/a)
    _startSect = 00000000 (SECT in mini-fat, since _ulSize is
    smaller than _ulMiniSectorCutoff)
    _ulSize    = 00000220 (< ssheader._ulMiniSectorCutoff, so
    _sectStart is in Mini)
    _dptPropType= 0000 (n/a)

000500:  5300 7400 7200 6500  6100 6D00 2000 3100   S.t.r.e.a.m. .1.
000510:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000520:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000530:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000540:  1200 0201 FFFF FFFF  FFFF FFFF FFFF FFFF   ................
000550:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000560:  0000 0000 0000 0000  0000 0000 0000 0000   ................
000570:  0000 0000 0000 0000  2002 0000 0000 0000   ........ .......
000580:  0000 0000 0000 0000  0000 0000 0000 0000   ................
```

### A.1.2.3.4 SID 3: Unused

```
000590:  0000 0000 0000 0000  0000 0000 0000 0000   ................
0005A0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
0005B0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
0005C0:  0000 0000 FFFF FFFF  FFFF FFFF FFFF FFFF   ................
0005D0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
0005E0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
0005F0:  0000 0000 0000 0000  0000 0000 0000 0000   ................
```

### A.1.2.4 SECT 3: MiniFat Sector

```
    SECT 0: 00000001: pointer to the second sector in the “Stream 1”
    data. This sector is the first sector of “Stream 1”,
    it is referred to by _sectStart of SID 2
    SECT 1: 00000002: pointer to the third sector in the “Stream 1”
    data. This sector is the second sector of “Stream 1”,
    it is referred to in MiniFat SECT 0, above.
...
    SECT 8: FFFFFFFE = ENDOFCHAIN: marks the end of the “Stream 1”
    data.

    Further Entries are empty (FREESECT = 0xFFFFFFFF)

000600:  0100 0000 0200 0000  0300 0000 0400 0000   ................
000610:  0500 0000 0600 0000  0700 0000 0800 0000   ................
000620:  FEFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF   ................
...
0007F0:  FFFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF   ................
```

### A.1.2.5 SECT 4: MiniStream (Data of “Stream 1”)

```
// referred to by SECTs in MiniFat of SECT 3, above

000800:  4461 7461 2066 6F72  2073 7472 6561 6D20   Data for stream
000810:  3144 6174 6120 666F  7220 7374 7265 616D   1Data for stream
000820:  2031 4461 7461 2066  6F72 2073 7472 6561    1Data for strea
...
000A00:  7461 2066 6F72 2073  7472 6561 6D20 3144   ta for stream 1D
000A10:  6174 6120 666F 7220  7374 7265 616D 2031   ata for stream 1

// data ends at 000A1F, MiniSector is filled to the end with known data
// (a copy of the header or FFFFFFF to prevent random disk or memory
// contents from contaminating the file on-disk.

000A20:  0000 0000 0000 0000  3B00 03FF FE00 0900   ........;.......
000A30:  0600 0000 0000 0000  0000 0000 0000 0100   ................
000A40:  D0CF 11E0 A1B1 1AE1  0000 0000 0000 0000   ................
000A50:  0000 0000 0000 0000  003B 0003 FFFE 0009   .........;......
000A60:  0006 0000 0000 0000  0000 0000 0000 0001   ................
000A70:  0000 0001 0000 0000  0000 1000 0000 0002   ................
000A80:  0000 0001 FFFF FFFE  0000 0000 0000 0000   ................
000A90:  FFFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF   ................
...
000BF0:  FFFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF   ................
```

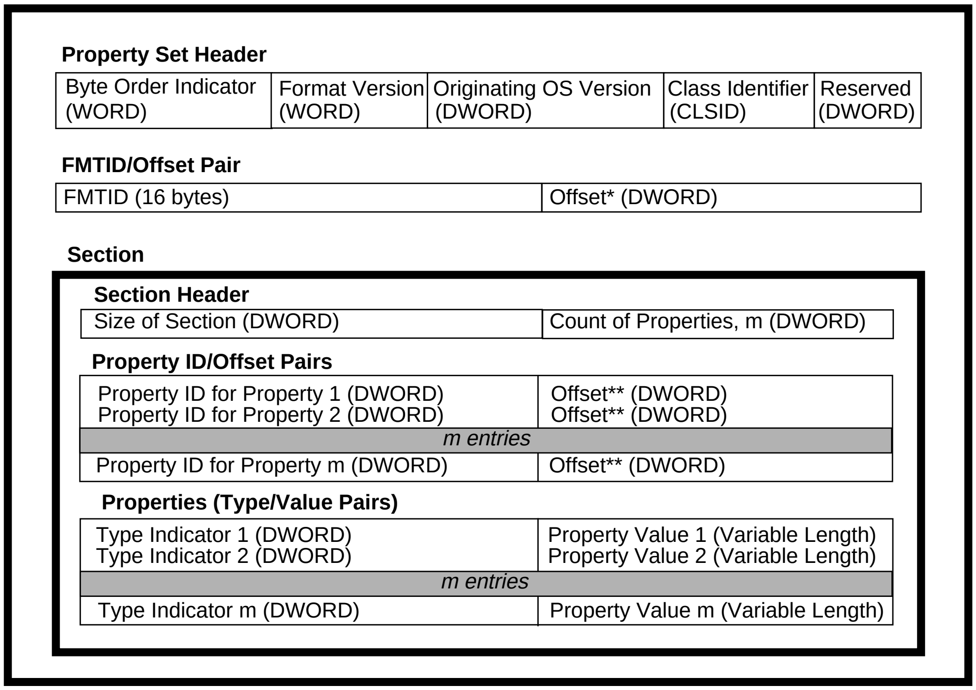
# A.2 OLE Property Set Binary Format

## A.2.0 Document Properties in Storage

In an IStorage, a serialized property set is stored in either a single stream or in a nested IStorage instance. In the latter case, the contained stream named “Contents” is the primary stream containing property values. The format of the primary stream, the same in either case, is described in the next section below. None of the property types VT_STREAM, VT_STORAGE, VT_STREAMED_OBJECT, or VT_STORED_OBJECT may be used in a stream-based property set; these types may only be used in storage-based sets. It is the person who invents / defines a new property set who gets to choose whether the set is always stream-based, is always storage-based, or at times can be either.

Names in an IStorage that begin with the value '\0x05' are reserved exclusively for the storage of property sets. Streams or storages that begin with '\0x05' must therefore be in the format described below; storages so named must contain a “Contents” stream in the format.[1] One of the things that a person who invents a new standard property set does is specify the standard string name under which instances of that type are stored. For example, the summary information property set defined by OLE2 is always found under the name "\005SummaryInformation". OLE2 provided no conventions for choosing this name; however, a convention for choosing such names is now strongly recommended below.

**Primary stream of a serialized property set**



**Property Set Header**

| Byte Order Indicator (WORD) | Format Version (WORD) | Originating OS Version (DWORD) | Class Identifier (CLSID) | Reserved (DWORD) |
|---|---|---|---|---|

**FMTID/Offset Pair**

| FMTID (16 bytes) | Offset* (DWORD) |
|---|---|

**Section**

**Section Header**

| Size of Section (DWORD) | Count of Properties, m (DWORD) |
|---|---|

**Property ID/Offset Pairs**

| | |
|---|---|
| Property ID for Property 1 (DWORD)<br>Property ID for Property 2 (DWORD) | Offset** (DWORD)<br>Offset** (DWORD) |
| *m entries* | |
| Property ID for Property m (DWORD) | Offset** (DWORD) |

**Properties (Type/Value Pairs)**

| | |
|---|---|
| Type Indicator 1 (DWORD)<br>Type Indicator 2 (DWORD) | Property Value 1 (Variable Length)<br>Property Value 2 (Variable Length) |
| *m entries* | |
| Type Indicator m (DWORD) | Property Value m (Variable Length) |

*Offset in bytes from the start of the stream to the start of the section
**Offset in bytes from the start of the section to the start of the type/value pair

**Figure 1. Stream containing a serialized property set**

[1] Properties may of course be stored in streams or storages that do not begin with '\0x05', but such properties are completely private to the application manipulating the storage; there is little reason to do this.

## A.2.1 Format of the primary property set stream

The overall structure of a stream containing a serialized property set is as illustrated in Figure 2. The format consists of a property set header, a sequence of size exactly one of format id / offset pair, and a corresponding sequence of sections containing the actual property values.[1]

Absolutely all the fields of a serialized property set specified here are *always* stored in storage in little-endian (Intel) byte order.[2]

The overall length of this property set stream is limited to 256k bytes.

### A.2.1.1 Property Set Header

At the beginning of the property set stream is a header. The following structure illustrates the header:

```
typedef struct PROPERTYSETHEADER {
                                  WORD   wByteOrder;// Always 0xFFFE
                                  WORD   wFormat;// Should be 0
                                  DWORD  dwOSVer;// System version
                                  CLSID  clsid; // Application CLSID
                                  DWORD  reserved;// Should be 1
                                         } PROPERTYSETHEADER;
```

The definition of the members of this structure as as follows.

| **Member** | **Meaning** |
|---|---|
| wByteOrder | The byte-order indicator is a WORD and should always hold the value 0xFFFE. This is the same as the Unicode© byte-order indicator. When written in little-endian (Intel) byte order, as is always done, this appears in the stream as 0xFE, 0xFF. |
| wFormat | The format version is a WORD and indicates the format version of this stream. Property set writers should write zero for this value. Property set readers should check this value; if it is non-zero, then they should refuse to read the set, for it is in a format that they don't in fact understand. |
| dwOSVer | The OS version number is encoded as OS kind in the high order word (0 for Windows on DOS, 1 for Macintosh, 2 for Windows 32-bit, 3 for UNIX) and the OS-supplied version number in the low order word. For Windows on DOS and Windows 32-bit, the latter is the low order word of the result of GetVersion(). |
| clsid | The class identifier is the CLSID of a class that can display and/or provide programmatic access to the property values. If there is no such class, it is recommended that the Format ID be used (see below), though a value of all zeros is also acceptable; the former simply allows for greater future extensibility. |
| reserved | Reserved for future use. A writer of a property set should write the value one here; a reader of a property set should only however check that the value is at least one. |

[1] The original OLE2 format allowed for more than one section, but use of that functionality is discourarged and no longer supported.

[2] Notwithstanding the fact that there is a byte-order tag of 0xFFFE at the start of the format. This tag was intended to allow for future extensibility that has been subsequently determined to be very unlikely to be done.

### A.2.1.2 Format ID / Offset Pairs

This part of the serialized property set indicates two things: the FMTID that scopes the property values contained in the set, and the location within the stream at which those values are stored.

```
typedef struct FORMATIDOFFSET {
                                 FMTID  fmtid; // semantic name of a section
                                 DWORD  dwOffset;// offset from start of whole
                                                 //property set stream to the
                                                 //section
                                        } FORMATIDOFFSET;
```

The offset is the distance of bytes from the start of the whole stream to where the section begins. The format id (FMTID) is the semantic name of its corresponding section, telling how to interpret the property values therein.

### A.2.1.3 Sections

Each section is made of up a property section header followed by an array that locates each property value within the section. It is specifically *not* the case that the properties in this array are sorted in any particular order Offsets within this array are the distance from the start of the section to the start of the property (type, value) pair. This allows entire sections to be copied as an array of bytes without any translation of internal structure.

```
typedef struct PROPERTYSECTIONHEADER {
                                        DWORD  cbSection;// size of section in
                                                          //bytes, which is
                                                          //inclusive of the byte
                                                          //count itself
                                        DWORD  cProperties;// count of properties
                                                            //in section
                                        PROPERTYIDOFFSETrgprop[];// array of
                                                                   //property
                                                                   //locations
                                             } PROPERTYSECTIONHEADER;

typedef struct PROPERTYIDOFFSET {
                                        DWORD  propid;// name of a property
                                        DWORD  dwOffset;// offset from the start
                                                        //of the section to that
                                                        //property
                                               } PROPERTYIDOFFSET;
```

Each property value contains a type tag followed by the bytes of the actual property value (at last!). All type/value pairs begin on a 32-bit boundary. Thus values may be followed with null bytes to align the subsequent pair on a 32-bit boundary (note though that there is no guarantee that property values are in fact as tightly packed in a section as this restriction permits; that is, there may be additional gratuitous padding).

```
typedef struct SERIALIZEDPROPERTYVALUE {
                                        DWORD  dwType;// type tag
                                        BYTE   rgb[]; // the actual property
                                                      //value
                                            } SERIALIZEDPROPERTYVALUE;
```

A consequence of these rules is that the smallest legal section, one containing zero properties, contains the following eight bytes: 08 00 00 00  00 00 00 00.

## A.2.2 Special property ids

A couple of property ids have special significance in all property sets.

### A.2.2.1 Property Id zero: Dictionary of property names

To enable users of property sets to attach meaning to properties beyond those provided by the type indicator, property id zero is reserved in all property sets for an optional dictionary giving human readable names for the properties in the set and for the property set itself. The value will be an array of (property id, string) pairs.

The value of property id zero is an array of propid / string pairs. Entries in the array are the ids and corresponding names of the properties; these are not in any particular order with respect to their property ids. Not all of the names of the properties in the set need appear in the dictionary: the dictionary may omit entries  for properties that are assumed to be universally known by clients that manipulate the property set. Typically names for the base property sets for widely accepted standards will be omitted.

Property names that begin with the binary Unicode characters 0x0001 through 0x001F are reserved for future use.

The name indicated as corresponding to property id zero is to be interpreted as the human readable name of the property set itself; like all property names, this may or may not be present.

The dictionary is stored as a list of Property ID/string pairs; the code page for the strings involved is as indicated in property id one. This can be illustrated using the following pseudo-structure definition for a dictionary entry (it's a pseudo-structure because the sz[] member is variable size).

```
 typedef struct tagENTRY {
                           DWORD    propid;    // Property ID
                           DWORD    cb;        // Count of bytes in the string,
                                               //including the null at the end
                           tchar    tsz[cb];   // Zero-terminated string. Code
                                               //page as indicated by property id
                                               //one.
                                    } ENTRY;

typedef struct tagDICTIONARY {
                              DWORD    cEntries;  // Count of entries in the list
                              ENTRY    rgEntry[cEntries];
                                       } DICTIONARY;
```

Note the following:

- Property ID zero does not have a type indicator. The DWORD that indicates the count of entries sits in the usual type indicator position.
- The count of bytes in the string (cb) includes the zero character that terminates the string.
- If the code page indicator is not 1200 (Unicode), there is no padding between entries to achieve reasonable alignment (sigh). However, if the code page indicator is Unicode, then each entry should be aligned on a DWORD boundary.
- If the code page indicator is not 1200 (Unicode), property names are stored dbcs strings. If the code page indicator does indicate Unicode, property name strings are stored as Unicode.
- Property name strings are restricted in length to 128 characters including the NULL terminating character.

### A.2.2.2 Property Id One: Code Page Indicator

Property id one (1) is reserved as an indicator of which code page or script any not-always-Unicode strings in the property set originated from (code pages are used in Windows and scripts are from the Macintosh world). All such string values in the entire property set, such as VT_LPSTRs, VT_BSTRs, and the names in the property name dictionary found in code page zero use characters from this one code page. If the code page indicator is not present, the prevailing code page on the reader's machine must be assumed. If an application cannot understand the indicated code page, it should not try to modify strings stored in the property set.

When an application that is not the author of a property set changes a property of type string in the set, it should examine the code page indicator and take one of the following courses of action:

1. Write the new value using the code page found in the code page indicator.

2. Rewrite all string values in the property set using the new code page (including the new value), and modify the code page indicator to reflect the new code page.

Possible values for the code page indicator are given in the Win32 API reference (see the NLSAPI functions, and specifically the GetACP function) and Inside Macintosh Volume VI, §14-111. For example, the code page US ANSI is represented by 0x04e4 (or 1252 in decimal); the code page for Unicode is 1200. Whether a Windows code page or a Macintosh script is found in property id one is determinted by the "originating OS version" (PROPERTYSETHEADER::dwOSVer) of the property set as a whole. Note that there exist Windows code page equivalents for the Macintosh scripts numbers (Windows code page 10000, for example, is the Macintosh Roman script).

By far, if it is at all possible, it is recommended that the Unicode code page (1200) be used. This is the only practical way to in fact achieve worldwide interoperable property sets. In code page 1200, note especially that the count at the start of a VT_LPSTR or VT_BSTR is to be interpreted as a *byte* count, not a character count. The byte count includes the two zero bytes at the end of the string.

Property id one is of type VT_I2, and therefore consists of a DWORD containing VT_I2 followed by a USHORT indicating the code page. For example, the type/value pair for property ID one representing the US ANSI code page is the following six bytes:

```
 02 00 00 00 e4 04
```

plus any necessary padding.

### A.2.2.3 Property Id 0x80000000: Locale Indicator

Property Id 0x80000000 (PID_LOCALE) is reserved as an indication of which locale the property set was written in. The default locale for a property set, in the event that PID_LOCALE does not exist in the property set will be the system's default locale (LOCALE_SYSTEM_DEFAULT).

Applications can choose to support locale or just get the default behaviour. Applications that allow users to specify a working locale should write that locale identifier to this property. Applications that use the user's default locale (LOCALE_USER_DEFAULT) should write the user's default locale identifier.

Applications should be concerned with the possibility of getting information from a property set which is of a different locale than the app's locale or the user's or the system's (i.e. a foreign object).

There is no provision in the OLE Property Set interfaces defined above to specifically read and write PID_LOCALE; in other words this property can be treated just like any property. Likewise the system will not attempt to automatically add or modify this property.

Property Id PID_LOCALE is of type VT_U4, and therefore consists of a DWORD containing VT_U4 followed by a DWORD containing the Locale Identifier (LCID) as defined by Appendix C of the Win32 SDK.

### A.2.2.4 Reserved property ids

Property ids with the high bit set (that is, which are negative) are reserved for future definition by Microsoft.

## A.2.3 Property Type Representations

A property (type, value) pair is a DWORD type indicator, followed by a value whose representation depends on the type. The serialized representations of each of the different types of values are as follows:

| **Type indicator** | **Value Representation** |
|---|---|
| VT_EMPTY | no bytes |
| VT_NULL | no bytes |
| VT_I2 | 2 byte signed integer |
| VT_I4 | 4 byte signed integer |
| VT_R4 | 32bit IEEE Floating point value |
| VT_R8 | 64bit IEEE Floating point value |
| VT_CY | 8 byte two's complement integer (scaled by 10,000) |
| VT_DATE | A 64bit floating point number representing the number of days (not seconds) since December 31, 1899 (thus, January 1, 1900 is 2.0, January 2, 1900 is 3.0, and so on). This is stored in the same representation as VT_R8. |

VT_BSTR — Counted, null terminated binary string; represented as a DWORD byte count of the number of bytes in the string (including the terminating null) followed by the bytes of the string. Character set is as indicated by the code page indicator.

VT_ERROR — A DWORD containing a status code.

VT_BOOL — Boolean value, a WORD containing 0 (false) or -1 (true).

VT_VARIANT — A type indicator (a DWORD) followed by the corresponding value. VT_VARIANT is only used in conjunction with VT_VECTOR: see below.

VT_UI1 — 1 byte unsigned integer

VT_UI2 — 2 byte unsigned integer

VT_UI4 — 4 byte unsigned integer

VT_I8 — 8 byte signed integer

VT_UI8 — 8 byte unsigned integer

VT_LPSTR — This is the representation of many strings. Stored in the same representation as VT_BSTR. Note therefore that the serialized representation of VT_LPSTR in fact has a preceding byte count, whereas the in-memory representation does not. Character set is as indicated by the code page indicator.

VT_LPWSTR — A counted and null terminated Unicode string; a DWORD character count (where the count includes the terminating null) followed by that many Unicode (16bit) characters. Note that the count is a character count, not a byte count.

VT_FILETIME — 64bit FILETIME structure as defined by Win32

VT_BLOB — A DWORD count of bytes, followed by that many bytes of data; the byte count does not include the four bytes for the length of the count itself: an empty blob would have a count of zero, followed by zero bytes. Thus, the serialized representation of a VT_BLOB is similar to that of a VT_BSTR but does not guarantee a null byte at the end of the data.

VT_STREAM — Indicates the value is stored in a stream which is sibling to the “Contents” stream. Following this type indicator is data in the format of a serialized VT_LPSTR which names the stream containing the data.

VT_STORAGE — Indicates the value is stored in an IStorage which is sibling to the “Contents” stream. Following this type indicator is data in the format of a serialized VT_LPSTR which names the IStorage containing the data.

VT_STREAMED_OBJECT — As in VT_STREAM but indicates that the stream contains a serialized object, which is a class id followed by initialization data for the class.

VT_STORED_OBJECT — As in VT_STORAGE but indicates that the designated IStorage contains a loadable object.

VT_BLOB_OBJECT

A BLOB containing a serialized object in the same representation as would appear in a VT_STREAMED_OBJECT. That is, following the VT_BLOB_OBJECT tag is a DWORD byte count of the remaining data (where the byte count does not include the size of itself) which is in the format of a class id followed by initialization data for that class.

The only significant difference between VT_BLOB_OBJECT and VT_STREAMED_OBJECT is that the former does not have the system-level storage overhead that the latter would have, and is therefore more suitable for scenarios involving numbers of small objects.

VT_CF

A BLOB containing a clipboard format identifier followed by the data in that format. That is, following the VT_CF tag is data in the format of a VT_BLOB: a DWORD count of bytes, followed by that many bytes of data in the format of a packed VTCFREP described just below, followed immediately by an array of bytes as appropriate for data in the clipboard format format (text, metafile, or whatever).

VT_CLSID

A class ID (or other GUID).

VT_VECTOR

If the type indicator is one of the above values with this bit on in addition, then the value is a DWORD count of elements, followed by that many repetitions of the value.

As an example, a type indicator of VT_LPSTR|VT_VECTOR has a DWORD element count, a DWORD byte count, the first string data, a DWORD byte count, the second string data, and so on.

Clipboard format identifiers, stored with the tag VT_CF, use one of five different representations:

```
typedef struct VTCFREP {
                        LONG   lTag;
                        BYTE   rgb[];
                        } VTCFREP;
```

The values for rgb are determined by the different values for lTag:

| **lTag Value** | **rgb value** |
|---|---|
| -1L | a DWORD containing a built-in Windows clipboard format value. |
| -2L | a DWORD containing a Macintosh clipboard format value. |
| -3L | a GUID containging a format identifier (this is in little usage). |
| any positive value | a null-terminated string containing a Windows clipboard format name, one suitable for passing to RegisterClipboardFormat. The code page used for characters in the string is per the code page indicator. The “positive value” here is the length of the string, including the null byte at the end. |
| 0L | no data (very rare usage) |

As was mentioned above, all type/value pairs begin on a 32-bit boundary. It follows that in turn, the type indicators and values of a type value pair are so aligned. This means that values may be necessarily followed by null bytes to align a subsequent type/value pair.

However, *within* a vector of values, each repetition of a value is to be aligned with its *natural* alignment rather than with 32-bit alignment. In practice, this is only significant for types VT_I2 and VT_BOOL (which have 2-byte natural alignment); all other types have 4-byte natural alignment. Therefore, a value with type tag VT_I2 | VT_VECTOR would be

- a DWORD element count, followed by
- an sequence of packed 2-byte integers with *no* padding between them, whereas a value of with type tag VT_LPSTR | VT_VECTOR would be a DWORD element count, followed by
- a sequence of (DWORD cch, char rgch[]) strings, each of which may be followed by null padding to round to a 32-bit boundary.

# A.3 ‘CompObj’ Stream Binary Format

## A.3.0 Overview

The ‘CompObj’ stream in a storage object provides generic information regarding the native data contained in this storage object. This generic information is manipulated through the OLE API functions WriteFmtUserTypeStg and ReadFmtUserTypeStg and includes:

- User Type: a user readable string that indicates the type of the object.
- Clipboard Format: implies the names and structure of streams and sub-storages.

This document exposes the binary format of the data written by WriteFmtUserTypeStg and interpreted by ReadFmtUserTypeStg.

## A.3.1 Format

The format consists of three basic parts, that represent versions of the stream written by different versions of the OLE2 libraries:

- Header, User Type (ANSI), Clipboard format (ANSI)
- ProgID (ANSI): optional, if not present, not Unicode information may follow
- Unicode versions of User Type, Clipboard format and ProgID: optional, if any Unicode information is present all three items have to be valid. Presence of the Unicode information is indicated by a “magic DWORD” value following the ANSI ProgID.

The following is a detailed description of the format using a pseudo C++ syntax where applicable.

### A.3.1.1 Mandatory part

### A.3.1.1.1 Stream name

```
// Stream name: L”\1CompObj”
```

### A.3.1.1.2 Header

```
struct CompObjHdr// The leading data in the CompObj stream
{
DWORDdwVersionAndByteOrder;// First DWORD: LOWORD Ver
                                sion=0x0001, HIWORD=FFFE (ignored by
                                reader!)
DWORDdwFormat = 0x00000a03;      // OS Version: always Win 3.1
DWORDunused=-1L;          // Always a -1L in the stream

CLSIDclsidClass;        // Class ID of this object, identical
                                to the CLSID in the parent storage of
                                the stream
};
```

### A.3.1.1.3 User Type

```
struct ANSIUserType
{
DWORDdwLenBytes;// length of User Type string in bytes
   including terminating 0
charszUserType[dwLenBytes];// User Type string (ANSI) terminated
                                with '\0'
}
```

### A.3.1.1.4 Clipboard Format (ANSI)

```
LONGdwCFLen;// Length of clipboard format name
   // special values:
   // 0  no clipboard format
   // -1 DWORD with standard Windows CF
   //follows:
   //DWORD cfStdWin;
   // -2 DWORD with standard Apple Mac
   //intosh CF follows:
   //DWORD cfStdMac;
   // >0 Length in bytes of clipboard
   //format name including terminating 0
char szCFName[dwCFLen]; // Clipboard Format Name (ANSI) te
                         //  minated with '\0'
```

### A.3.1.2 Optional: ProgID (ANSI)

The stream may end at this point. Versions of OLE before 2.01 provided only the data described in section 2.1.

If more data follows it is to be interpreted as follows:

```
struct ANSIProgID
{
DWORDdwLenBytes;// length of ProgID stream in bytes.
                // dwLenBytes<=40
charszProgID[dwLenBytes];// ProgID string (ANSI) terminated with '\0'
}
```

### A.3.1.3 Optional: Unicode versions

Only if a ANSI ProgID was provided (possibly with ANSIProgID::dwLenBytes=0), the following data may follow:

#### A.3.1.3.1 Magic Number

```
DWORD dwMagicNumber =0x71B239F4;     // indicates Unicode UserType, CF
                                    // and ProgID follow (all three!)
```

#### A.3.1.3.2 User Type (Unicode)

```
struct UNICODEUserType
{
DWORD dwLenBytes;// Size of Unicode User
    Type in bytes (not cha
    acters!) including te
    minating 0.
WCHARwszUserType[dwLenBytes/sizeof(WCHAR)];// Unicode User Type
                                               //string,terminated with
                                               // ‘\0’.
};
```

#### A.3.1.3.3 Clipboard Format (Unicode)

```
LONGdwUnicodeCFLen;// Length of Unicode clipboard format
    name in bytes
    // special values:
    // 0 no clipboard format
    // -1 DWORD with standard Windows CF
    //follows:
    //DWORD cfStdWin;
    // -2 DWORD with standard Apple Mac
    //intosh CF follows:
    //DWORD cfStdMac;
    // >0 Length in bytes of clipboard
    //format name including terminating 0
WCHARszCFName[dwUnicodeCFLen/sizeof(WCHAR)]; // Clipboard Format
                //Name (Unicode) terminated with ‘\0’
```

#### A.3.1.3.4 ProgID (Unicode)

```
struct UNICODEProgID
{
DWORD dwLenBytes;// Size of Unicode ProgID in bytes (not characters!) including
                  // terminating ‘\’0.
WCHARwszProgID[dwLenBytes/sizeof(WCHAR)];// Unicode ProgID string, terminated
                                             // with ‘\’0.
};
```

## References

1. CCIR Recommendation 709, Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange
2. Hunt, R.W.G. “Measuring Color”, Ellis Harwood Limited, Chichester, England, 1987
3. CIE Publication 15.2 “Colorimetry” second edition, CIE, Vienna, 1986
4. CCIR Recommendation 601–1, Encoding Parameters of Digital Television for Studios
5. ISO/TC 130/WG2 N438, International Color Profile format, March 26, 1995
6. *OLE 2 Programmers Reference, Volume One* Microsoft Press (1994)
7. Brockschmidt, Kraig. *Inside OLE2*, Microsoft Press (1994)
8. ICC Profile Format Specification version 3.2, International Color Consortium (1995)
9. ISO/IEC 10918-1 / ITU-T Recommendation T.81 “Information technology - Digital compression and coding of continuous-tone still images - Requirements and guidelines
10. *Computer graphics: principles and practice*, James D. Foley et al., 2nd ed. 1992, Addison-Wesley Systems programming series
11. *PostScript language reference manual*, Adobe Systems, Inc., 2nd ed., 1990, Addison-Wesley Publishing Company, Inc.
12. *Programmer’s Guide to the IVUE toolkit*, FITS Imaging, Copyright © 1993-1994, FITS Imaging
13. *The Unicode Standard*, The Unicode Consortium, 1991, Addison-Wesley.
14. Porter, T., and T. Duff. *Compositing digital images*, ACM Computer Graphics (SIGGRAPH), 1984, 18(3), 253-259, SIGGRAPH '84 Conference Proceedings.
15. Thompson, Kelvin. *Alpha Blending*, Graphic Gems. (1990) Academic Press, Inc. 210-211
16. Goldman, R. “Decomposing projective transformations,” *Graphics Gems 3*, 1992, Academic Press, pp 98-107.
17. Miller, Steven. *DEC/HP, Network Computing Architecture, Remote Procedure Call Run Time Extensions Specification*, Version OSF TX1.0.11, July 23, 1992, Appendix A “Universal Unique Indentifiers,” http://www.osf.org/dce.
18. Blinn, James F. *Jim Blinn’s Corner: Compositing Part 1: Theory*, IEEE Computer Graphics & Applications, September 1994, pp. 83-87. *Compositing Part 2: Practice*, IEEE Computer Graphics & Applications, November 1994, pp. 78-82.
19. “Storage naming conventions,” *OLE 2 Programmer’s reference*. Volume 1, Microsoft Press, 1995, pp. 596-596.
20. Kano, Nadine and A. Freytag. “The international character set conundrum: ANSI Unicode, and Microsoft Windows,” *Microsoft Systems Journal*, 1994 Volum 9, November 1994.
21. Photography - Electronic still picture cameras - determination of ISO speed (Working draft #6).
22. ISO 14524, Photography - Electronic still picture cameras - Methods for measuring the opto-electronic conversion functions (Working draft 4.0).

**23.** Programming Windows, Charles Petzold, 1990, Microsoft Press.

**24.** Inside Macintosh: Files, Addison-Wesley Publishing Co. (1992)

APPENDIX KK

VRML97

ISO/IEC 14772-1:1997

# The Virtual Reality Modeling Language

**International Standard ISO/IEC 14772-1:1997**

APPENDIX KK

# Copyright Information

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice, this paragraph and the title, URL, and authors of the Document as referenced below are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the VRML Consortium, except as needed for the purpose of developing VRML standards in which case the procedures for copyrights defined in the VRML Consortium standards process must be followed.

TITLE: ISO/IEC 14772-1:1997 Virtual Reality Modeling Language (VRML97)
URL: **http://www.vrml.org/Specifications/VRML97**
AUTHORS: Rikk Carey, Gavin Bell, Chris Marrin

The limited permissions granted above are perpetual and will not be revoked by the VRML Consortium or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE VRML CONSORTIUM DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Except as contained in this notice, the name of the VRML Consortium shall not be used in advertising or to otherwise promote the sale, use or other dealings of this document without prior written authorization from the VRML Consortium.

## INTELLECTUAL PROPERTY NOTICE

The VRML Consortium takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on the VRML Consortium's procedures with respect to rights in standards-track documents can be found in the VRML Consortium's Intellectual Property Rights Statement. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the VRML Consortium Executive Director.

The VRML Consortium invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the VRML Consortium Executive Director.

## Other copyrights and trademarks

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

# Acknowledgements

The VRML Consortium gratefully acknowledges the authors, Rikk Carey, Gavin Bell, and Chris Marrin, whose valuable efforts produced the VRML standard.

We would like to give special thanks to Steve Carson, chair of the ISO/IEC JTC 1/SC 24, *Computer Graphics and Image Processing* subcommittee, and to Dick Puk, liaison between the VRML Consortium and SC 24, for guiding the standards process as well as their significant contributions to the document itself. Also, thanks to all the members of ISO who participated in the review and editing of ISO/IEC 14772.

Special thanks to Kouichi Matsuda and the Sony VRML team for their work on the Java annex and to Jan Hardenbergh for his work on the ECMAScript annex. Thanks to Curtis Beason, Chris Fouts, John Gebhardt, Rich Gossweiler, Paul Isaacs, and Daniel Woods for writing key sections. Thanks to Justin Couch and the Script Working Group for drafting several improvements to the scripting sections. Thanks to all the others who drafted text for the standard, too numerous to name them all.

Thanks to Mark Pesce, Tony Parisi, Mitra, Brian Behlendorf, and Dave Raggett for their early pioneering work and continued efforts on VRML.

Thanks to the hundreds of participants who contributed ideas, reviews, and feedback on the VRML standard.

Thanks to Kevin Hughes for the VRML logo artwork.

And, last but not least, thanks to the members of the VRML community for their support, passion, and hard work that has made VRML into an International Standard.

APPENDIX KK

# The Virtual Reality Modeling Language

## International Standard ISO/IEC 14772-1:1997

This document is part 1 of ISO/IEC 14772-1:1997, the Virtual Reality Modeling Language (VRML), also referred to as "VRML97". The full title of this part of the International Standard is: *Information technology -- Computer graphics and image processing -- The Virtual Reality Modeling Language (VRML) -- Part 1: Functional specification and UTF-8 encoding.*

The ***Foreword*** provides background on the standards process for VRML. The ***Introduction*** describes the purpose, design criteria, and characteristics of VRML. The following clauses define part 1 of ISO/IEC 14772:

a. ***Scope*** defines the problem area that VRML addresses.

b. ***Normative references*** lists the normative standards referenced in this part of ISO/IEC 14772.

c. ***Definitions*** contains the glossary of terminology used in this part of ISO/IEC 14772.

d. ***Concepts*** describes various fundamentals of VRML.

e. ***Field and event reference*** specifies the datatypes used by nodes.

f. ***Node reference*** defines the syntax and semantics of VRML nodes.

g. ***Conformance and minimum support requirements*** describes the conformance requirements for VRML implementations.

APPENDIX KK

There are several annexes included in the specification:

A. ***Grammar definition*** presents the grammar for the VRML file format.

B. ***Java platform scripting reference*** describes how VRML scripting integrates with the Java platform.

C. ***ECMAScript scripting reference*** describes how VRML scripting integrates with ECMAScript.

D. ***Examples*** includes a variety of VRML example files.

E. ***Bibliography*** lists the informative, non-standard topics referenced in this part of ISO/IEC 14772.

F. ***Recommendations for non-normative extensions*** lists informative recommendations for extensions to VRML.

Questions or comments should be sent to rikk@wasabisoft.com.

APPENDIX KK

# Foreword

## Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form a specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. See http://www.iso.ch for information on ISO and http://www.iec.ch for information on IEC.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote. See http://www.iso.ch/meme/JTC1.html for information on JTC 1.

International Standard ISO/IEC 14772 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee 24, *Computer graphics and image processing*, in collaboration with The VRML Consortium, Inc. (http://www.vrml.org) and the VRML moderated email list (www-vrml@vrml.org).

ISO/IEC 14772 consists of the following part, under the general title *Information technology -- Computer graphics and image processing -- The Virtual Reality Modeling Language:*

*Part 1: Functional specification and UTF-8 encoding.*

Further parts will follow.

Annexes A to C form an integral part of this part of ISO/IEC 14772. Annexes D to F are for information only.

APPENDIX KK

# Introduction

## Purpose

The Virtual Reality Modeling Language (VRML) is a file format for describing interactive 3D objects and worlds. VRML is designed to be used on the Internet, intranets, and local client systems. VRML is also intended to be a universal interchange format for integrated 3D graphics and multimedia. VRML may be used in a variety of application areas such as engineering and scientific visualization, multimedia presentations, entertainment and educational titles, web pages, and shared virtual worlds.

## Design Criteria

VRML has been designed to fulfill the following requirements:

Authorability

> *Enable the development of computer programs capable of creating, editing, and maintaining VRML files, as well as automatic translation programs for converting other commonly used 3D file formats into VRML files.*

Composability

> *Provide the ability to use and combine dynamic 3D objects within a VRML world and thus allow re-usability.*

Extensibility

> *Provide the ability to add new object types not explicitly defined in VRML.*

Be capable of implementation

> *Capable of implementation on a wide range of systems.*

Performance

> *Emphasize scalable, interactive performance on a wide variety of computing platforms.*

Scalability

> *Enable arbitrarily large dynamic 3D worlds.*

## Characteristics of VRML

VRML is capable of representing static and animated dynamic 3D and multimedia objects with hyperlinks to other media such as text, sounds, movies, and images. VRML browsers, as well as authoring tools for the creation of VRML files, are widely available for many different platforms.

APPENDIX KK

VRML supports an extensibility model that allows new dynamic 3D objects to be defined allowing application communities to develop interoperable extensions to the base standard. There are mappings between VRML objects and commonly used 3D application programmer interface (API) features.

APPENDIX KK

VRML97

ISO/IEC 14772-1:1997

# Information technology -- Computer graphics and image processing -- The Virtual Reality Modeling Language -- Part 1: Functional specification and UTF-8 encoding

# 1 Scope

ISO/IEC 14772, the Virtual Reality Modeling Language (VRML), defines a file format that integrates 3D graphics and multimedia. Conceptually, each VRML file is a 3D time-based space that contains graphic and aural objects that can be dynamically modified through a variety of mechanisms. This part of ISO/IEC 14772 defines a primary set of objects and mechanisms that encourage composition, encapsulation, and extension.

The semantics of VRML describe an abstract functional behaviour of time-based, interactive 3D, multimedia information. ISO/IEC 14772 does not define physical devices or any other implementation-dependent concepts (e.g., screen resolution and input devices). ISO/IEC 14772 is intended for a wide variety of devices and applications, and provides wide latitude in interpretation and implementation of the functionality. For example, ISO/IEC 14772 does not assume the existence of a mouse or 2D display device.

Each VRML file:

a. implicitly establishes a world coordinate space for all objects defined in the file, as well as all objects included by the file;

b. explicitly defines and composes a set of 3D and multimedia objects;

c. can specify hyperlinks to other files and applications;

d. can define object behaviours.

An important characteristic of VRML files is the ability to compose files together through inclusion and to relate files together through hyperlinking. For example, consider the file *earth.wrl* which specifies a world that contains a sphere representing the earth. This file may also contain references to a variety of other VRML files representing cities on the earth (e.g., file *paris.wrl).* The enclosing file, *earth.wrl*, defines the coordinate system that all the cities reside in. Each city file defines the world coordinate system that the city resides in but that becomes a local coordinate system when contained by the earth file.

Hierarchical file inclusion enables the creation of arbitrarily large, dynamic worlds. Therefore, VRML ensures that each file is completely described by the objects contained within it.

Another essential characteristic of VRML is that it is intended to be used in a distributed environment such as the World Wide Web. There are various objects and mechanisms built into the language that support multiple distributed files, including:

g. in-lining of other VRML files;

h. hyperlinking to other files;

i. using established Internet and ISO standards for other file formats;

j. defining a compact syntax.

# 2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 14772. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of ISO/IEC 14772 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

Annex E, Bibliography, contains a list of informative documents and technology.

| Identifier | Reference |
|---|---|
| 1766 | IETF RFC 1766, Tags for the Identification of Languages, Internet standards track protocol.<br>http://ds.internic.net/rfc/rfc1766.txt |
| CGM | ISO/IEC 8632:1992 (all parts) Information technology -- Computer graphics -- Metafile for the storage and transfer of picture description information.<br>http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=8632 |
| ESCR | ISO/IEC DIS 16262 Information technology -- ECMAScript: A general purpose, cross-platform programming language.<br>http://www.ecma.ch<br>http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=16262 |
| HTML | HTML 3.2 Reference Specification.<br>http://www.w3.org/TR/REC-html32.html |
| I639 | ISO 639:1988 Code for the representation of names of languages.<br>http://www.iso.ch/isob/switch-engine-cate.pl?KEYWORDS=10918&searchtype=refnumber,<br>http://www.chemie.fu-berlin.de/diverse/doc/ISO_639.html |
| I3166 | ISO 3166:1997 (all parts) Codes for the representation of names of countries and their subdivisions. http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=3166 |
| I8859 | ISO/IEC 8859-1:1987 Information technology -- 8-bit single-byte coded graphic character sets -- Part 1: Latin alphabet No. 1.<br>http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=8859 |

| | |
|---|---|
| ISOC | ISO/IEC 9899:1990 Programming languages -- C.<br>http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=9899 |
| ISOG | ISO/IEC 10641:1993 Information technology -- Computer graphics and image processing -- Conformance testing of implementations of graphics standards.<br>http://www.iso.ch/isob/switch-engine-cate.pl?KEYWORDS=10641&searchtype=refnumber |
| JAVA | "The Java Language Specification" by James Gosling, Bill Joy and Guy Steele, Addison Wesley, Reading Massachusetts, 1996, ISBN 0-201-63451-1.<br>http://java.sun.com/docs/books/jls/index.html<br>"The Java Virtual Machine Specification" by Tim Lindhold and Frank Yellin, Addison Wesley, Reading Massachusetts, 1996, ISBN 0-201-63452-X.<br>http://java.sun.com/docs/books/vmspec/index.html |
| JPEG | "JPEG File Interchange Format," JFIF, Version 1.02, 1992.<br>http://www.w3.org/pub/WWW/Graphics/JPEG/jfif.txt<br>ISO/IEC 10918-1:1994 Information technology -- Digital compression and coding of continuous-tone still images: Requirements and guidelines.<br>http://www.iso.ch/isob/switch-engine-cate.pl?KEYWORDS=10918&searchtype=refnumber |
| MIDI | Complete MIDI 1.0 Detailed Specification, MIDI Manufacturers Association,<br>P.O. Box 3173, La Habra, CA 90632 USA 1996.<br>http://www.midi.org |
| MPEG | ISO/IEC 11172-1:1993 Information technology -- Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s -- Part 1: Systems.<br>http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=11172 |
| PNG | PNG (Portable Network Graphics), Specification Version 1.0, W3C Recommendation, 1 October 1996.<br>http://www.w3.org/pub/WWW/TR/REC-png-multi.html |
| RURL | IETF RFC 1808 Relative Uniform Resource Locator, Internet standards track protocol.<br>http://ds.internic.net/rfc/rfc1808.txt |
| URL | IETF RFC 1738 Uniform Resource Locator, Internet standards track protocol.<br>http://ds.internic.net/rfc/rfc1738.txt |

APPENDIX KK

| | |
|---|---|
| UTF8 | ISO/IEC 10646-1:1993 Information technology -- Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane, Internet standards track protocol. http://www.iso.ch/isob/switch-engine-cate.pl?searchtype=refnumber&KEYWORDS=10646, http://ds.internic.net/rfc/rfc2044.txt |

APPENDIX KK

# 3 Definitions

For the purposes of this part of ISO/IEC 14722, the following definitions apply.

## 3.1 activate

To cause a *sensor node* to generate an "isActive" *event*. The various types of sensor nodes are "activated" by *user* interactions, the passage of *time*, or other events. Only active sensor nodes affect the *user's* experience. A Script *node* is activated when it receives an event. A pointing device such as a *mouse* is activated when one of its buttons is depressed by a user. See 4.12.2, Script execution, for details.

## 3.2 ancestor

A *node* which is an antecedent of another node in the *transformation hierarchy*.

## 3.3 author

A person or agent that creates *VRML files*. Authors typically use *generators* to assist them.

## 3.4 authoring tool

See *generator*.

## 3.5 avatar

The abstract representation of the *user* in a VRML *world*. The physical dimensions of the avatar are used for collision detection and terrain following. See 6.29, NavigationInfo, for details.

## 3.6 bearing

A straight line passing through the *pointer* location in the direction of the pointer. If multiple sensors' geometry intersect this line, only the sensor nearest the viewer will be eligible to generate *events* regardless of material and texture properties (e.g., transparency).

## 3.7 bindable node

A *node* that may have many *instances* in a *scene graph*, but only one instance may be active at any instant of *time*. A node of type Background, Fog, NavigationInfo, or Viewpoint. See 4.6.10, Bindable children nodes, for details.

## 3.8 browser

A computer program that interprets *VRML files*, presents their content to a *user* on a *display device*, and allows the user to interact with *worlds* defined by VRML files by means of a user interface.

## 3.9 browser extension

*Nodes* defined using the prototyping mechanism that are understood only by certain *browsers*. See 4.9.3, Browser extensions, for details.

## 3.10 built-in node

A *node* of a *type* explicitly defined in this part of ISO/IEC 14772.

## 3.11 callback

A function defined in a *scripting language* to which *events* are passed. See 4.12.8, EventIn handling, for details.

## 3.12 candidate

One of potentially several choices. The *user* or the *browser* will select none or one of the choices when all candidates are identified. See 4.6.10, Bindable children nodes, and 6.2, Anchor, for details.

## 3.13 child

An instance of a *children node*.

## 3.14 children node

One of a set of *node types*, instances of which can be collected in a group to share specific properties dependent on the type of the *grouping node*. See 4.6.5, Grouping and children nodes, for a list of allowable children nodes.

## 3.15 client system

A computer system, attached to a *network*, that relies on another computer (the server) for essential processing functions. Many client systems also function as stand-alone computers.

## 3.16 collision proxy

A *node* used as a substitute for all of a Collision node's children during collision detection. See 6.8, Collision, for details.

## 3.17 colour model

Characterization of a colour space in terms of explicit parameters. ISO/IEC 14772 allows colours to be defined only with the RGB colour model. However, colour interpolation is performed in the HSV colour space.

## 3.18 culling

The process of identifying *objects* or parts of objects which do not need to be processed further by the *browser* in order to produce the desired view of a *world*.

## 3.19 descendant

A *node* which descends from another node in the *transformation hierarchy*. A *children node*.

## 3.20 display device

A graphics device on which VRML *worlds* may be rendered.

## 3.21 drag sensor

A *pointing device sensor* that causes *events* to be generated in response to sensor-dependent pointer motions. For example, the SphereSensor generates spherical rotation events. A *node* of type CylinderSensor, PlaneSensor, or SphereSensor. See 4.6.7, Sensor nodes, and 4.6.7.4, Drag sensors, for details.

## 3.22 environmental sensor

A sensor *node* that generates *events* based on the location of the viewpoint in the *world* or in relation to *objects* in the world. The TimeSensor node generates events at regular intervals in *time*. A node of type Collision, ProximitySensor, TimeSensor, or VisibilitySensor. See 4.6.7.2, Environmental sensors, for details.

## 3.23 event

A *message* sent from one *node* to another as defined by a *route*. *Events* signal external stimuli, changes to *field* values, and interactions between nodes. An event consists of a *timestamp* and a field value.

## 3.24 event cascade

A sequence of *events* initiated by a script or sensor event and propagated from *node* to node along one or more *routes*. All events in an event cascade are considered to have occurred simultaneously. See 4.10.3, Execution model, for details.

## 3.25 eventIn

A logical receptor attached to a *node* which receives *events*.

## 3.26 eventOut

A logical output terminal attached to a *node* from which *events* are sent. The eventOut also stores the event most recently sent.

## 3.27 execution model

The rules governing how *events* are processed by *browsers* and scripts.

## 3.28 exposed field

A *field* that is capable of receiving *events* via an *eventIn* to change its value(s), and generating events via an *eventOut* when its value(s) change.

## 3.29 external prototype

A *prototype* defined in an external file and referenced by a *URL*.

## 3.30 field

A property or attribute of a *node*. Each *node type* has a fixed set of fields. Fields may contain various kinds of data and one or many values. Each field has a default value.

## 3.31 field name

The identifier of a *field*. Field names are unique within the scope of the *node*.

## 3.32 file

A collection of related data. A file may be stored on physical media or may exist as a data stream or as data within a computer program.

## 3.33 frame

A single rendering of a *world* on a *display device* or a single time-step in a simulation.

## 3.34 generator

A computer program which creates *VRML files*. A generator may be used by a person or operate automatically. Synonymous with *authoring tool*.

## 3.35 geometric property node

A *node* defining the properties of a specific geometry node. A node of type Color, Coordinate, Normal, or TextureCoordinate. See 4.6.3.2, Geometric property nodes, for details.

## 3.36 geometric sensor node

A *node* that generates *events* based on *user* actions, such as a *mouse* click or navigating close to a particular *object*. A node of type CylinderSensor, PlaneSensor, ProximitySensor, SphereSensor, TouchSensor, VisibilitySensor, or Collision. See 4.6.7.1, Introduction to sensors, for details.

## 3.37 geometry node

A *node* containing mathematical descriptions of three-dimensional (3D) points, lines, surfaces, text strings and solids. A node of type Box, Cone, Cylinder, ElevationGrid, Extrusion, IndexedFaceSet, IndexedLineSet, PointSet, Sphere, or Text. See 4.6.3, Shapes and geometry, for details.

## 3.38 grab

To receive *events* from activated pointing devices (e.g., *mouse* or *wand*). A *pointing device sensor* becomes the exclusive recipient of pointing device events when one or more pointing devices are activated simultaneously.

## 3.39 gravity

In the context of ISO/IEC 14772, gravity may be simulated by constraining the motion of the viewpoint to the lowest possible path (smallest Y-coordinate in the local coordinate system of the viewpoint) consistent with following the surface of encountered *objects*. See 6.29, NavigationInfo, for details.

## 3.40 grouping node

One of a set of *node types* which include a list of nodes, referred to as its *children nodes*. These children nodes are collected together to share specific properties dependent on the type of the grouping node. Each grouping node defines a coordinate space for its children relative to its own coordinate space. The children may themselves be instances of grouping nodes, thus forming a *transformation hierarchy*. See 4.6.5, Grouping and children nodes, for details.

## 3.41 HSV

Hue, Saturation, and Value colour model. See E.[FOLE].

## 3.42 HTML

HyperText Markup Language. See 2.[HTML].

## 3.43 hyperlink

A reference to a *URL* that is associated with an Anchor *node*. See 6.2, Anchor, for details.

## 3.44 ideal VRML implementation

An implementation of VRML that presents all *objects* and simulates movement without approximation. Not realizable in practice.

## 3.45 IEC

International Electrotechnical Commission. See `http://www.iec.ch`.

## 3.46 IETF

Internet Engineering Task Force. The organization which develops *Internet* standards. See `http://www.ietf.org/overview.html`.

## 3.47 image

A two-dimensional (2D) rectangular array of pixel values. Pixel values may have from one to four components. See 5.5, SFImage, for details.

## 3.48 in-lining

The mechanism by which one *VRML file* is hierarchically included in another.

## 3.49 Internet

The world-wide named *network* of computers which communicate with each other using a common set of communication protocols known as TCP/IP. See *IETF*. The *World Wide Web* is implemented on the Internet.

## 3.50 instance

A reference to a previously defined and named *node*. Nodes are named by means of the DEF syntax and reference by USE syntax (see 4.6.2, DEF/USE semantics). Instances of nodes may be used in any context in which the defining node may be used.

## 3.51 interpolator node

A *node* that defines a piece-wise linear interpolation. A node of type ColorInterpolator, CoordinateInterpolator, NormalInterpolator, OrientationInterpolator, PositionInterpolator, or ScalarInterpolator. See 4.6.8, Interpolator nodes, for details.

## 3.52 intranet

A private *network* that uses the same protocols and standards as the *Internet*.

## 3.53 ISO

International Organization for Standardization. See `http://www.iso.ch/infoe/intro.html`.

## 3.54 JPEG

Joint Photographic Experts Group. See 2.[JPEG].

## 3.55 JTC 1

ISO/IEC Joint Technical Committee 1. See `http://www.iso.ch/meme/JTC1.html`.

## 3.56 level of detail

The amount of detail or complexity which is displayed at any particular *time* for any particular *object*. The level of detail for an object is controllable as a function of the distance of the object from the viewer. See 6.26, LOD, for details. (Abbreviated LOD)

## 3.57 line terminator

A linefeed character (0x0A) or a carriage return character (0x0D).

## 3.58 loop

A sequence of *events* which would result in a specific *eventOut* sending more than one event with the same *timestamp*.

## 3.59 message

A data string sent between *nodes* upon the occurrence of an *event*. See 4.10, Event processing, for details.

## 3.60 MIDI

Musical Instrument Digital Interface. A standard for digital music representation. See 2.[MIDI].

## 3.61 MIME

Multipurpose Internet Mail Extension. Used to specify filetyping rules for *Internet* applications, including *browsers*. See 4.5.1, File extension and MIME types, for details. See also E.[MIME].

## 3.62 mouse

A pointing device that moves in two dimensions and that enables a *user* to move a cursor on a *display device* in order to point at displayed *objects*. One or more push buttons on the mouse allow the user to indicate to the computer program that some action is to be taken.

## 3.63 MPEG

Moving Picture Experts Group. See `http://drogo.cselt.stet.it/mpeg/`.

## 3.64 multimedia

An integrated presentation, typically on a computer, of content of various types, such as computer graphics, audio, and video.

## 3.65 network

Set of interconnected computers.

## 3.66 node

The fundamental component of a *scene graph* in ISO/IEC 14772. Nodes are abstractions of various real-world objects and concepts. Examples include spheres, lights, and material descriptions. Nodes contain *fields* and *events*. *Messages* may be sent between nodes along *routes*.

## 3.67 node type

A characteristic of each *node* that describes, in general, its particular semantics. For example, Box, Group, Sound, and SpotLight are node types. See 4.6, Node semantics, and 6, Node reference, for details.

## 3.68 now

The present *time* as perceived by the *user*.

## 3.69 object

A collection of data and procedures, packaged according to the rules and syntax defined in ISO/IEC 14772. "Object" is usually synonymous with *node*.

## 3.70 object space

The coordinate system in which an *object* is defined.

## 3.71 panorama

A background texture that is placed behind all geometry in the scene and in front of the ground and sky. See 6.5, Background, for details.

## 3.72 parent

A *node* which is an instance of a *grouping node*.

## 3.73 PNG

Portable Network Graphics. A specification for representing two-dimensional images in *files*. See 2.[PNG].

## 3.74 pointer

A location and direction in the *virtual world* defined by the *pointing device* which the *user* is currently using to interact with the virtual world.

## 3.75 pointing device

A hardware device connected to the *user's* computer by which the user directly controls the location and direction of the *pointer*. Pointing devices may be either two-dimensional or three-dimensional and may have one or more control buttons. See 4.6.7.5, Activating and manipulating sensors, for details.

## 3.76 pointing device sensor

A sensor *node* that generates *events* based on *user* actions, such as *pointing device* motions or button activations. A node of type Anchor, CylinderSensor, PlaneSensor, SphereSensor, or TouchSensor. See 4.6.7.3, Pointing device sensors, for details.

## 3.77 polyline

A sequence of straight line segments where the end point of the first segment is coincident with the start point of the second segment, the endpoint of the second segment is coincident with the start point of the third segment, and so on. A piecewise linear curve.

## 3.78 profile

A named collection of criteria for functionality and conformance that defines an implementable subset of a standard.

## 3.79 prototype

The definition of a new *node type* in terms of the *nodes* defined in this part of ISO/IEC 14772. See 4.8, Prototype semantics, for details.

## 3.80 prototyping

The mechanism for extending the set of *node types* from within a *VRML file*.

## 3.81 public interface

The formal definition of a *node type* in this part of ISO/IEC 14772.

## 3.82 RGB

The colour model used within ISO/IEC 14772 for the specification of colours. Each colour is represented as a combination of the three primary colours red, green, and blue. See E.[FOLE].

## 3.83 route

The connection between a *node* generating an *event* and a node receiving the event. See 4.3.9, Route statement syntax, and 4.10.2, Route semantics, for details.

## 3.84 route graph

The set of connections between *eventOuts* and *eventIns* formed by ROUTE statements or addRoute method invocations.

## 3.85 run-time name scope

The extent to which a name defined within a VRML file applies and is visible. Several different run-time name scopes are recognized and are defined in 4.4.6, Run-time name scope.

## 3.86 RURL

Relative Uniform Resource Locator. See 2.[RURL].

## 3.87 scene graph

An ordered collection of *grouping nodes* and other nodes. Grouping nodes, (such as LOD, Switch, and Transform nodes) may have *children nodes*. See 4.2.3, Scene graph, and 4.4.2, Scene graph hierarchy, for details.

## 3.88 script

A set of procedural functions normally executed as part of an *event cascade* (see 6.40, Script). A script function may also be executed asynchronously (see 4.12.6, Asynchronous scripts).

## 3.89 scripting

The process of creating or referring to a script.

## 3.90 scripting language

A system of syntactical and semantic constructs used to define and automate procedures and processes on a computer. Typically, scripting languages are interpreted and executed sequentially on a statement-by-statement basis whereas programming languages are generally compiled prior to execution.

## 3.91 sensor node

A *node* that enables the *user* to interact with the *world* in the scene graph hierarchy. Sensor nodes respond to user interaction with geometric *objects* in the world, the movement of the user through the world, or the passage of *time*. See 4.6.7, Sensor nodes, for details.

## 3.92 separator character

A *UTF-8* character used to separate syntactical entities in a *VRML file*. Specifically, commas, spaces, tabs, linefeeds, and carriage-returns are separator characters wherever they appear outside of string *fields*. See 4.3.1, Clear text (UTF-8) encoding, for details.

## 3.93 sibling

A *node* which shares a *parent* with other nodes.

## 3.94 simulation tick

The smallest time unit capable of being identified in a digital simulation of analog time. *Time* in the context of ISO/IEC 14772 is conceptually analog but is realized by an implementation as a digital simulation of abstract analog time. See 4.11, Time, for details.

## 3.95 special group node

A *grouping node* that exhibits special behaviour. Examples of such special behaviour include selecting one of many *children nodes* to be rendered based on a dynamically changing parameter value and dynamically loading children nodes from an external file. A node of type Inline, LOD (level of detail), or Switch. See 4.6.5, Grouping and children nodes, for details.

## 3.96 texture

An *image* used in a *texture map* to create visual appearance effects when applied to *geometry nodes*.

## 3.97 texture coordinates

The set of two-dimensional coordinates used by some vertex-based *geometry nodes* (*e.g.*, IndexedFaceSet and ElevationGrid) and specified in the TextureCoordinate node to map textures to the vertices of those nodes. Texture coordinates range from 0 to 1 across each axis of the texture image. See 4.6.11, Texture maps, and 6.48, TextureCoordinate, for details.

## 3.98 texture map

A *texture* plus the general parameters necessary for mapping the texture to geometry.

## 3.99 time

A monotonically increasing value generated by a node. Time (0.0) starts at 00:00:00 GMT January 1, 1970. See 4.11, Time, for details.

## 3.100 timestamp

The part of a *message* that describes the *time* the *event* occurred and that caused the message to be sent. See 4.11, Time, for details.

## 3.101 transformation hierarchy

The subset of the *scene graph* consisting of *nodes* that have well-defined coordinate systems. The transformation hierarchy excludes nodes that are not *descendants* of the scene graph root nodes and nodes in SFNode or MFNode fields of Script nodes.

## 3.102 transparency chunk

A section of a PNG file containing transparency information (derived from 2.[PNG]).

## 3.103 traverse

To process the *nodes* in a *scene graph* in the correct order.

## 3.104 UCS

Universal multiple-octet coded Character Set. See 2.[UTF8].

## 3.105 URL

Uniform Resource Locator. See 2.[URL].

## 3.106 URN

Universal Resource Name. See E.[URN].

## 3.107 UTF-8

The character set used to encode *VRML files*. The 8-bit UCS Transformation Format. See 2.[UTF8].

## 3.108 user

A person or agent who uses and interacts with *VRML files* by means of a *browser*.

## 3.109 viewer

A location, direction, and viewing angle in a *virtual world* that determines the portion of the virtual world presented by the *browser* to the *user*.

## 3.110 virtual world

See *world*.

## 3.111 VRML browser

See *browser*.

## 3.112 VRML document server

A computer program that locates and transmits *VRML files* and supporting files in response to requests from *browsers*.

## 3.113 VRML file

A set of VRML nodes and statements as defined in this part of ISO/IEC 14772. This set of VRML nodes and statements may be in the form of a file, a data stream, or an in-line sequence of VRML information as defined by a particular VRML encoding.

## 3.114 wand

A pointing device that moves in three dimensions and that enables a *user* to indicate a position in the three-dimensional coordinate system of a world in order to point at displayed *objects*. One or more push buttons on the wand allow the user to indicate to the computer program that some action is to be taken.

## 3.115 white space

One or more consecutive occurrences of a *separator character*. See 4.3.1, Clear text (UTF-8) encoding, for details.

## 3.116 world

A collection of one or more *VRML files* and other multimedia content that, when interpreted by a *VRML browser*, presents an interactive experience to the *user* consistent with the *author's* intent.

## 3.117 world coordinate space

The coordinate system in which each VRML *world* is defined. The world coordinate space is an orthogonal right-handed Cartesian coordinate system. The units of length are metres.

## 3.118 World Wide Web

The collection of documents, data, and content typically encoded in HTML pages and accessible via the *Internet* using the HTTP protocol.

## 3.119 XY plane

The plane perpendicular to the Z-axis that passes through the point Z = 0.0.

## 3.120 YZ plane

The plane perpendicular to the X-axis that passes through the point X = 0.0.

## 3.121 ZX plane

The plane perpendicular to the Y-axis that passes through the point Y = 0.0.

APPENDIX KK

# 4 Concepts

## 4.1 Introduction and table of contents

### 4.1.1 Introduction

This clause describes key concepts in ISO/IEC 14772. This includes how nodes are combined into scene graphs, how nodes receive and generate events, how to create node types using prototypes, how to add node types to VRML and export them for use by others, how to incorporate scripts into a *VRML file*, and various general topics on nodes.

### 4.1.2 Table of contents

See Table 4.1 for the table of contents for this clause.

**Table 4.1 -- Table of contents, Concepts**

### 4.1.3 Conventions used

The following conventions are used throughout this part of ISO/IEC 14772:

*Italics* are used for event and field names, and are also used when new terms are introduced and equation variables are referenced.

A `fixed-space` font is used for URL addresses and source code examples. ISO/IEC 14772 UTF-8 encoding examples appear in **`bold, fixed-space`** font.

Node type names are appropriately capitalized (e.g., "The Billboard node is a grouping node..."). However, the concept of the node is often referred to in lower case in order to refer to the semantics of the node, not the node itself (e.g., "To rotate the billboard...").

The form "0xhh" expresses a byte as a hexadecimal number representing the bit configuration for that byte.

Throughout this part of ISO/IEC 14772, references are denoted using the "x.[ABCD]" notation, where "x" denotes which clause or annex the reference is described in and "[ABCD]" is an abbreviation of the reference title. For example, 2.[ABCD] refers to a reference described in clause 2 and E.[ABCD] refers to a reference described in annex E.

## 4.2 Overview

### 4.2.1 The structure of a VRML file

A *VRML file* consists of the following major functional components: the header, the *scene graph*, the prototypes, and *event routing*. The contents of this file are processed for presentation and interaction by a program known as a *browser*.

## 4.2.2 Header

For easy identification of VRML files, every VRML file shall begin with:

```
#VRML V2.0 <encoding type> [optional comment] <line terminator>
```

The header is a single line of UTF-8 text identifying the file as a VRML file and identifying the encoding type of the file. It may also contain additional semantic information. There shall be exactly one space separating "**#VRML**" from "**V2.0**" and "**V2.0**" from "<encoding type>". Also, the "<encoding type>" shall be followed by a linefeed (0x0a) or carriage-return (0x0d) character, or by one or more space (0x20) or tab (0x09) characters followed by any other characters, which are treated as a comment, and terminated by a linefeed or carriage-return character.

The <encoding type> is either "**utf8**" or any other authorized values defined in other parts of ISO/IEC 14772. The identifier "**utf8"** indicates a clear text encoding that allows for international characters to be displayed in ISO/IEC 14772 using the UTF-8 encoding defined in ISO/IEC 10646-1 (otherwise known as Unicode); see 2.[UTF8]. The usage of UTF-8 is detailed in 6.47, Text, node. The header for a UTF-8 encoded VRML file is

```
#VRML V2.0 utf8 [optional comment] <line terminator>
```

Any characters after the <encoding type> on the first line may be ignored by a browser. The header line ends at the occurrence of a <line terminator>. A <line terminator> is a linefeed character (0x0a) or a carriage-return character (0x0d) .

## 4.2.3 Scene graph

The scene graph contains nodes which describe objects and their properties. It contains hierarchically grouped geometry to provide an audio-visual representation of objects, as well as nodes that participate in the event generation and routing mechanism.

## 4.2.4 Prototypes

Prototypes allow the set of VRML node types to be extended by the user. Prototype definitions can be included in the file in which they are used or defined externally. Prototypes may be defined in terms of other VRML nodes or may be defined using a browser-specific extension mechanism. While ISO/IEC 14772 has a standard format for identifying such extensions, their implementation is browser-dependent.

## 4.2.5 Event routing

Some VRML nodes generate events in response to environmental changes or user interaction. Event routing gives authors a mechanism, separate from the scene graph hierarchy, through which these events can be propagated to effect changes in other nodes. Once generated, events are sent to their routed destinations in time order and processed by the receiving node. This processing can change the state of the node, generate additional events, or change the structure of the scene graph.

Script nodes allow arbitrary, author-defined event processing. An event received by a Script node causes the execution of a function within a script which has the ability to send events through the normal event routing mechanism, or bypass this mechanism and send events directly to any node to which the Script node has a reference. Scripts can also dynamically add or delete routes and thereby changing the event-routing topology.

The ideal event model processes all events instantaneously in the order that they are generated. A timestamp serves two purposes. First, it is a conceptual device used to describe the chronological flow of the event mechanism. It ensures that deterministic results can be achieved by real-world implementations that address processing delays and asynchronous interaction with external devices. Second, timestamps are also made available to Script nodes to allow events to be processed based on the order of user actions or the elapsed time between events.

### 4.2.6 Generating VRML files

A *generator* is a human or computerized creator of VRML files. It is the responsibility of the generator to ensure the correctness of the VRML file and the availability of supporting assets (e.g., images, audio clips, other VRML files) referenced therein.

### 4.2.7 Presentation and interaction

The interpretation, execution, and presentation of VRML files will typically be undertaken by a mechanism known as a browser, which displays the shapes and sounds in the scene graph. This presentation is known as a *virtual world* and is navigated in the browser by a human or mechanical entity, known as a *user*. The world is displayed as if experienced from a particular location; that position and orientation in the world is known as the *viewer*. The browser provides navigation paradigms (such as walking or flying) that enable the user to move the viewer through the virtual world.

In addition to navigation, the browser provides a mechanism allowing the user to interact with the world through sensor nodes in the scene graph hierarchy. Sensors respond to user interaction with geometric objects in the world, the movement of the user through the world, or the passage of time.

The visual presentation of geometric objects in a VRML world follows a conceptual model designed to resemble the physical characteristics of light. The VRML lighting model describes how appearance properties and lights in the world are combined to produce displayed colours (see 4.14, Lighting Model, for details).

Figure 4.1 illustrates a conceptual model of a VRML browser. The browser is portrayed as a presentation application that accepts user input in the forms of file selection (explicit and implicit) and user interface gestures (e.g., manipulation and navigation using an input device). The three main components of the browser are: Parser, Scene Graph, and Audio/Visual Presentation. The Parser component reads the VRML file and creates the Scene Graph. The Scene Graph component consists of the Transformation Hierarchy (the nodes) and the Route Graph. The Scene Graph also includes the Execution Engine that processes events, reads and edits the Route Graph, and makes changes to the Transform Hierarchy (nodes). User input generally affects sensors and navigation, and thus is wired to the Route Graph component (sensors) and the Audio/Visual Presentation component (navigation). The Audio/Visual Presentation component performs the graphics and audio rendering of the Transform Hierarchy that feeds back to the user.

### 4.2.8 Profiles

ISO/IEC 14772 supports the concept of profiles. A profile is a named collection of functionality and requirements which shall be supported in order for an implementation to conform to that profile. Only one profile is defined in this part of ISO/IEC 14772. The functionality and minimum support requirements described in ISO/IEC 14772-1 form the *Base* profile. Additional profiles may be defined in other parts of ISO/IEC 14772. Such profiles shall incorporate the entirety of the Base profile.

## 4.3 UTF-8 file syntax

### 4.3.1 Clear text (UTF-8) encoding

This section describes the syntax of UTF-8-encoded, human-readable VRML files. A more formal description of the syntax may be found in annex A, Grammar definition. The semantics of VRML in terms of the UTF-8 encoding are presented in this part of ISO/IEC 14772. Other encodings may be defined in other parts of ISO/IEC 14772. Such encodings shall describe how to map the UTF-8 descriptions to and from the corresponding encoding elements.

For the UTF-8 encoding, the # character begins a comment. The first line of the file, the header, also starts with a "#" character. Otherwise, all characters following a "#", until the next line terminator, are ignored. The only exception is within double-quoted SFString and MFString fields where the "#" character is defined to be part of the string.

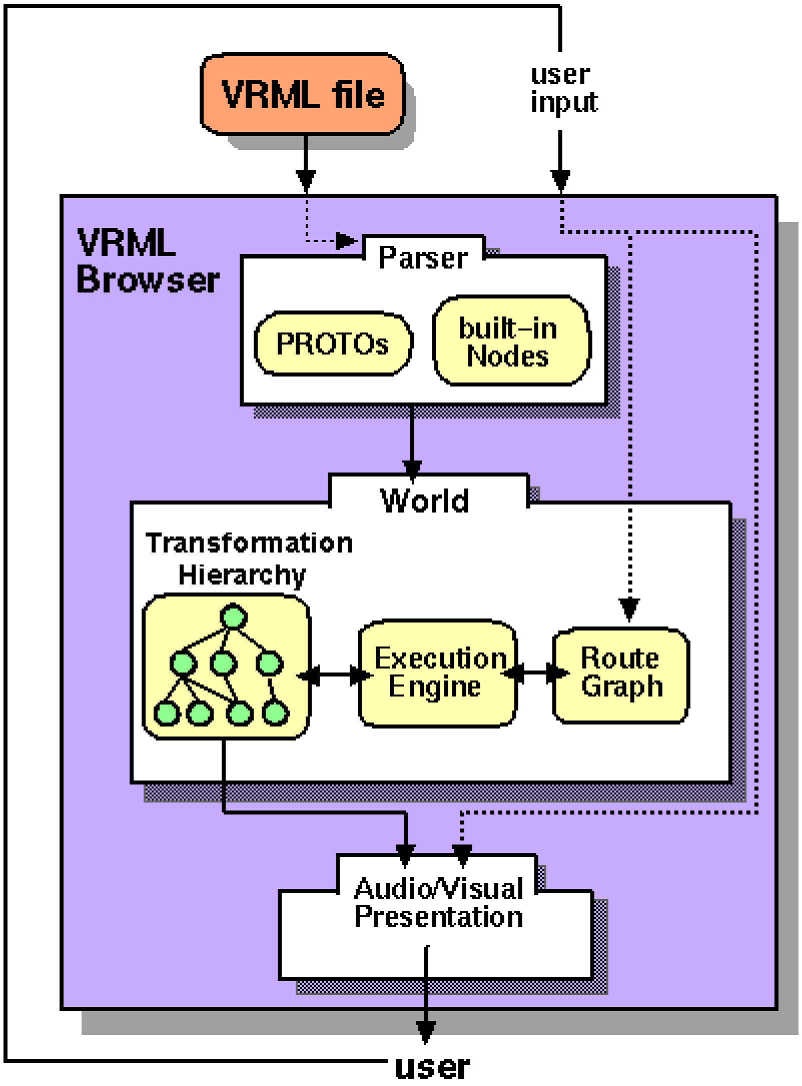


**Figure 4.1 -- Conceptual model of a VRML browser**

Commas, spaces, tabs, linefeeds, and carriage-returns are separator characters wherever they appear outside of string fields. Separator characters and comments are collectively termed *whitespace*.

A VRML document server may strip comments and extra separators including the comment portion of the header line from a VRML file before transmitting it. WorldInfo nodes should be used for persistent information such as copyrights or author information.

Field, event, PROTO, EXTERNPROTO, and node names shall not contain control characters (0x0-0x1f, 0x7f), space (0x20), double or single quotes (0x22: ", 0x27: '), sharp (0x23: #), comma (0x2c: ,), period (0x2e: .), brackets (0x5b, 0x5d: []), backslash (0x5c: \) or braces (0x7b, 0x7d: {}). Further, their first character shall not be a digit (0x30-0x39), plus (0x2b: +), or minus (0x2d: -) character. Otherwise, names may contain any ISO 10646 character encoded using UTF-8. VRML is case-sensitive; "Sphere" is different from "sphere" and "BEGIN" is different from "begin."

The following reserved keywords shall not be used for field, event, PROTO, EXTERNPROTO, or node names:

- DEF
- EXTERNPROTO
- FALSE
- IS
- NULL
- PROTO
- ROUTE
- TO
- TRUE
- USE
- eventIn
- eventOut
- exposedField
- field

## 4.3.2 Statements

After the required header, a VRML file may contain any combination of the following:

a. Any number of PROTO or EXTERNPROTO statements (see 4.8, Prototype semantics);

b. Any number of root node statements (see 4.4.1, Root nodes);

c. Any number of USE statements (see 4.6.2, DEF/USE semantics);

d. Any number of ROUTE statements (see 4.10.2, Route semantics).

## 4.3.3 Node statement syntax

A node statement consists of an optional name for the node followed by the node's type and then the body of the node. A node is given a name using the keyword DEF followed by the name of the node. The node's body is enclosed in matching braces ("**{ }**"). Whitespace shall separate the DEF, name of the node, and node type, but is not required before or after the curly braces that enclose the node's body. See A.3, Nodes, for details on node grammar rules.

```
[DEF <name>] <nodeType> { <body> }
```

A node's body consists of any number of field statements, IS statements, ROUTE statements, PROTO statements or EXTERNPROTO statements, in any order.

See 4.6.2, DEF/USE, sematnics for more details on node naming. See 4.3.4, Field statement syntax, for a description of field statement syntax and 4.7, Field, eventIn, and eventOut semantics, for a description of field statement semantics. See 4.6, Node semantics, for a description of node statement semantics.

### 4.3.4 Field statement syntax

A field statement consists of the name of the field followed by the field's value(s). The following illustrates the syntax for a single-valued field:

```
<fieldName> <fieldValue>
```

The following illustrates the syntax for a multiple-valued field:

```
<fieldName> [ <fieldValues> ]
```

See A.4, Fields, for details on field statement grammar rules.

Each node type defines the names and types of the fields that each node of that type contains. The same field name may be used by multiple node types. See 5, Field and event reference, for the definition and syntax of specific field types.

See 4.7, Field, eventIn, and eventOut semantics, for a description of field statement semantics.

### 4.3.5 PROTO statement syntax

A PROTO statement consists of the PROTO keyword, followed in order by the prototype name, prototype interface declaration, and prototype definition:

```
PROTO <name> [ <declaration> ] { <definition> }
```

See A.2, General, for details on prototype statement grammar rules.

A prototype interface declaration consists of eventIn, eventOut, field, and exposedField declarations (see 4.7, Field, eventIn, and eventOut semantics) enclosed in square brackets. Whitespace is not required before or after the brackets.

EventIn declarations consist of the keyword "eventIn" followed by an event type and a name:

```
eventIn <eventType> <name>
```

EventOut declarations consist of the keyword "eventOut" followed by an event type and a name:

```
eventOut <eventType> <name>
```

Field and exposedField declarations consist of either the keyword "field" or "exposedField" followed by a field type, a name, and an initial field value of the given field type.

```
field <fieldType> <name> <initial field value>

exposedField <fieldType> <name> <initial field value>
```

Field, eventIn, eventOut, and exposedField names shall be unique in each PROTO statement, but are not required to be unique between different PROTO statements. If a PROTO statement contains an exposedField with a given name (e.g., *zzz*), it shall not contain eventIns or eventOuts with the prefix *set_* or the suffix *_changed* and the given name (e.g., *set_zzz* or *zzz_changed*).

A prototype definition consists of at least one node statement and any number of ROUTE statements, PROTO statements, and EXTERNPROTO statements in any order.

See 4.8, Prototype semantics, for a description of prototype semantics.

## 4.3.6 IS statement syntax

The body of a node statement that is inside a prototype definition may contain IS statements. An IS statement consists of the name of a field, exposedField, eventIn or eventOut from the node's public interface followed by the keyword IS followed by the name of a field, exposedField, eventIn or eventOut from the prototype's interface declaration:

```
<field/eventName> IS <field/eventName>
```

See A.3, Nodes, for details on prototype node body grammar rules. See 4.8, Prototype semantics, for a description of IS statement semantics.

## 4.3.7 EXTERNPROTO statement syntax

An EXTERNPROTO statement consists of the EXTERNPROTO keyword followed in order by the prototype's name, its interface declaration, and a list (possibly empty) of double-quoted strings enclosed in square brackets. If there is only one member of the list, the brackets are optional.

```
EXTERNPROTO <name> [ <external declaration> ] URL or [ URLs ]
```

See A.2, General, for details on external prototype statement grammar rules.

An EXTERNPROTO interface declaration is the same as a PROTO interface declaration, with the exception that field and exposedField initial values are not specified and the prototype definition is specified in a separate VRML file to which the URL(s) refer.

## 4.3.8 USE statement syntax

A USE statement consists of the USE keyword followed by a node name:

```
USE <name>
```

See A.2, General, for details on USE statement grammar rules.

## 4.3.9 ROUTE statement syntax

A ROUTE statement consists of the ROUTE keyword followed in order by a node name, a period character, a field name, the TO keyword, a node name, a period character, and a field name. Whitespace is allowed but not required before or after the period characters:

```
ROUTE <name>.<field/eventName> TO <name>.<field/eventName>
```

See A.2, General, for details on ROUTE statement grammar rules.

# 4.4 Scene graph structure

## 4.4.1 Root nodes

A VRML file contains zero or more root nodes. The root nodes for a VRML file are those nodes defined by the node statements or USE statements that are not contained in other node or PROTO statements. Root nodes shall be children nodes (see 4.6.5, Grouping and children nodes).

## 4.4.2 Scene graph hierarchy

A VRML file contains a directed acyclic graph. Node statements can contain SFNode or MFNode field statements that, in turn, contain node (or USE) statements. This hierarchy of nodes is called the *scene graph*. Each arc in the graph from A to B means that node A has an SFNode or MFNode field whose value directly contains node B. See E.[FOLE] for details on hierarchical scene graphs.

## 4.4.3 Descendant and ancestor nodes

The *descendants* of a node are all of the nodes in its SFNode or MFNode fields, as well as all of those nodes' descendants. The *ancestors* of a node are all of the nodes that have the node as a descendant.

## 4.4.4 Transformation hierarchy

The transformation hierarchy includes all of the root nodes and root node descendants that are considered to have one or more particular locations in the virtual world. VRML includes the notion of *local coordinate systems*, defined in terms of transformations from ancestor coordinate systems (using Transform or Billboard nodes). The coordinate system in which the root nodes are displayed is called the *world coordinate system*.

A VRML browser's task is to present a VRML file to the user; it does this by presenting the transformation hierarchy to the user. The transformation hierarchy describes the directly perceptible parts of the virtual world.

The following node types are in the scene graph but not affected by the transformation hierarchy: ColorInterpolator, CoordinateInterpolator, NavigationInfo, NormalInterpolator, OrientationInterpolator, PositionInterpolator, Script, ScalarInterpolator, TimeSensor, and WorldInfo. Of these, only Script nodes may have descendants. A descendant of a Script node is not part of the transformation hierarchy unless it is also the descendant of another node that is part of the transformation hierarchy or is a root node.

Nodes that are descendants of LOD or Switch nodes are affected by the transformation hierarchy, even if the settings of a Switch node's *whichChoice* field or the position of the viewer with respect to a LOD node makes them imperceptible.

The transformation hierarchy shall be a directed acyclic graph; results are undefined if a node in the transformation hierarchy is its own ancestor.

## 4.4.5 Standard units and coordinate system

ISO/IEC 14772 defines the unit of measure of the world coordinate system to be metres. All other coordinate systems are built from transformations based from the world coordinate system. Table 4.2 lists standard units for ISO/IEC 14772.

**Table 4.2 -- Standard units**

| Category | Unit |
|---|---|
| Linear distance | Metres |
| Angles | Radians |
| Time | Seconds |
| Colour space | RGB ([0.,1.], [0.,1.], [0., 1.]) |

ISO/IEC 14772 uses a Cartesian, right-handed, three-dimensional coordinate system. By default, the viewer is on the Z-axis looking down the -Z-axis toward the origin with +X to the right and +Y straight up. A modelling transformation (see 6.52, Transform, and 6.6, Billboard) or viewing transformation (see 6.53, Viewpoint) can be used to alter this default projection.

### 4.4.6 Run-time name scope

Each VRML file defines a run-time name scope that contains all of the root nodes of the file and all of the descendent nodes of the root nodes, with the exception of:

a. descendent nodes that are inside Inline nodes;

b. descendent nodes that are inside a prototype instance and are not part of the prototype's interface (i.e., are not in an SF/MFNode field or eventOut of the prototype).

Each Inline node and prototype instance also defines a run-time name scope, consisting of all of the root nodes of the file referred to by the Inline node or all of the root nodes of the prototype definition, restricted as above.

Nodes created dynamically (using a Script node invoking the Browser.createVrml methods) are not part of any name scope, until they are added to the scene graph, at which point they become part of the same name scope of their parent node(s). A node may be part of more than one run-time name scope. A node shall be removed from a name scope when it is removed from the scene graph.

## 4.5 VRML and the World Wide Web

### 4.5.1 File extension and MIME types

The file extension for VRML files is `.wrl` (for *world*).

The official MIME type for VRML files is defined as:

```
model/vrml
```

where the MIME major type for 3D data descriptions is `model,` and the minor type for VRML documents is `vrml`.

For compatibility with earlier versions of VRML, the following MIME type shall also be supported:

```
x-world/x-vrml
```

where the MIME major type is `x-world,` and the minor type for VRML documents is `x-vrml`.

See E.[MIME] for details.

## 4.5.2 URLs

A *URL* (Uniform Resource Locator), described in 2.[URL], specifies a file located on a particular server and accessed through a specified protocol (e.g., http). In ISO/IEC 14772, the upper-case term URL refers to a Uniform Resource Locator, while the italicized lower-case version *url* refers to a field which may contain URLs or in-line encoded data.

All *url* fields are of type MFString. The strings in these fields indicate multiple locations to search for data in decreasing order of preference. If the browser cannot locate or interpret the data specified by the first location, it shall try the second and subsequent locations in order until a URL containing interpretable data is encountered. If no interpretable URL's are located, the node type defines the resultant default behaviour. The *url* field entries are delimited by double quotation marks " ". Due to 4.5.4, Scripting language protocols, *url* fields use a superset of the standard URL syntax defined in 2.[URL]. Details on the string field are located in 5.9, SFString and MFString.

More general information on URLs is described in 2.[URL].

## 4.5.3 Relative URLs

Relative URLs are handled as described in 2.[RURL]. The base document for EXTERNPROTO statements or nodes that contain URL fields is:

a. The VRML file in which the prototype is instantiated, if the statement is part of a prototype definition.

b. The file containing the script code, if the statement is part of a string passed to the createVrmlFromURL() or createVrmlFromString() browser calls in a Script node.

c. Otherwise, the VRML file from which the statement is read, in which case the RURL information provides the data itself.

## 4.5.4 Scripting language protocols

The Script node's *url* field may also support custom protocols for the various scripting languages. For example, a script *url* prefixed with *javascript:* shall contain ECMAScript source, with line terminators allowed in the string. The details of each language protocol are defined in the annex for each language. Browsers are not required to support any specific scripting language. However, browsers shall adhere to the protocol defined in the corresponding annex of ISO/IEC 14772 for any scripting language which is supported. The following example illustrates the use of mixing custom protocols and standard protocols in a single *url* field (order of precedence determines priority):

```
#VRML V2.0 utf8
Script {
  url [ "javascript: ...",           # custom protocol ECMAScript
        "http://bar.com/foo.js",      # std protocol ECMAScript
        "http://bar.com/foo.class" ] # std protocol Java platform bytecode
}
```

In the example above, the "..." represents in-line ECMAScript source code.

# 4.6 Node semantics

## 4.6.1 Introduction

Each node has the following characteristics:

a. **A type name.** Examples include Box, Color, Group, Sphere, Sound, or SpotLight.

b. **Zero or more fields that define how each node differs from other nodes of the same type.** Field values are stored in the VRML file along with the nodes, and encode the state of the virtual world.

c. **A set of events that it can receive and send.** Each node may receive zero or more different kinds of events which will result in some change to the node's state. Each node may also generate zero or more different kinds of events to report changes in the node's state.

d. **An implementation.** The implementation of each node defines how it reacts to events it can receive, when it generates events, and its visual or auditory appearance in the virtual world (if any). The VRML standard defines the semantics of built-in nodes (i.e., nodes with implementations that are provided by the VRML browser). The PROTO statement may be used to define new types of nodes, with behaviours defined in terms of the behaviours of other nodes.

e. **A name.** Nodes can be named. This is used by other statements to reference a specific instantiation of a node.

## 4.6.2 DEF/USE semantics

A node given a name using the DEF keyword may be referenced by name later in the same file with USE or ROUTE statements. The USE statement does not create a copy of the node. Instead, the same node is inserted into the scene graph a second time, resulting in the node having multiple parents. Using an instance of a node multiple times is called *instantiation*.

Node names are limited in scope to a single VRML file, prototype definition, or string submitted to either the CreateVrmlFromString browser extension or a construction mechanism for SFNodes within a script. Given a node named "NewNode" (i.e., `DEF NewNode`), any "`USE NewNode`" statements in SFNode or MFNode fields inside NewNode's scope refer to NewNode (see 4.4.4, Transformation hierarchy, for restrictions on self-referential nodes).

If multiple nodes are given the same name, each USE statement refers to the closest node with the given name preceding it in either the VRML file or prototype definition.

## 4.6.3 Shapes and geometry

### 4.6.3.1 Introduction

The Shape node associates a geometry node with nodes that define that geometry's appearance. Shape nodes shall be part of the transformation hierarchy to have any visible result, and the transformation hierarchy shall contain Shape nodes for any geometry to be visible (the only nodes that render visible results are Shape nodes and the Background node). A Shape node contains exactly one geometry node in its *geometry* field. The following node types are *geometry* nodes:

- Box
- Cone
- Cylinder
- ElevationGrid
- Extrusion
- IndexedFaceSet
- IndexedLineSet
- PointSet
- Sphere
- Text

#### 4.6.3.2 Geometric property nodes

Several geometry nodes contain Coordinate, Color, Normal, and TextureCoordinate as geometric property nodes. The geometric property nodes are defined as individual nodes so that instancing and sharing is possible between different geometry nodes.

#### 4.6.3.3 Appearance nodes

Shape nodes may specify an Appearance node that describes the appearance properties (material and texture) to be applied to the Shape's geometry. Nodes of the following type may be specified in the *material* field of the Appearance node:

- Material

Nodes of the following types may be specified by the *texture* field of the Appearance node:

- ImageTexture
- PixelTexture
- MovieTexture

Nodes of the following types may be specified in the *textureTransform* field of the Appearance node:

- TextureTransform

The interaction between such appearance nodes and the Color node is described in 4.14, Lighting Model.

#### 4.6.3.4 Shape hint fields

The Extrusion and IndexedFaceSet nodes each have three SFBool fields that provide hints about the geometry. These hints specify the vertex ordering, if the shape is solid, and if the shape contains convex faces. These fields are *ccw*, *solid*, and *convex*, respectively. The ElevationGrid node has the *ccw* and *solid* fields.

The *ccw* field defines the ordering of the vertex coordinates of the geometry with respect to user-given or automatically generated normal vectors used in the lighting model equations. If *ccw* is TRUE, the normals shall follow the right hand rule; the orientation of each normal with respect to the vertices (taken in order) shall be such that the vertices appear to be oriented in a counterclockwise order when the vertices are viewed (in the local coordinate system of the Shape) from the opposite direction as the normal. If *ccw* is FALSE, the normals shall be oriented in the opposite direction. If normals are not generated but are supplied using a Normal node, and the orientation of the normals does not match the setting of the *ccw* field, results are undefined.

The *solid* field determines whether one or both sides of each polygon shall be displayed. If *solid* is FALSE, each polygon shall be visible regardless of the viewing direction (i.e., no backface culling shall be done, and two-sided lighting shall be performed to illuminate both sides of lit surfaces). If *solid* is TRUE, the visibility of each polygon shall be determined as follows: Let ***V*** be the position of the viewer in the local coordinate system of the geometry. Let ***N*** be the geometric normal vector of the polygon, and let ***P*** be any point (besides the local origin) in the plane defined by the polygon's vertices. Then if (***V*** dot ***N***) **-** (***N*** dot ***P***) is greater than zero, the polygon shall be visible; if it is less than or equal to zero, the polygon shall be invisible (backface culled).

The *convex* field indicates whether all polygons in the shape are convex (TRUE). A polygon is convex if it is planar, does not intersect itself, and all of the interior angles at its vertices are less than 180 degrees. Non-planar and self-intersecting polygons may produce undefined results even if the *convex* field is FALSE.

### 4.6.3.5 Crease angle field

The *creaseAngle* field, used by the ElevationGrid, Extrusion, and IndexedFaceSet nodes, affects how default normals are generated. If the angle between the geometric normals of two adjacent faces is less than the crease angle, normals shall be calculated so that the faces are smooth-shaded across the edge; otherwise, normals shall be calculated so that a lighting discontinuity across the edge is produced. For example, a crease angle of 0.5 radians means that an edge between two adjacent polygonal faces will be smooth shaded if the geometric normals of the two faces form an angle that is less than 0.5 radians. Otherwise, the faces will appear faceted. Crease angles shall be greater than or equal to 0.0.

## 4.6.4 Bounding boxes

Several of the nodes include a bounding box specification comprised of two fields, *bboxSize* and *bboxCenter*. A bounding box is a rectangular parallelepiped of dimension *bboxSize* centred on the location *bboxCenter* in the local coordinate system. This is typically used by grouping nodes to provide a hint to the browser on the group's approximate size for culling optimizations. The default size for bounding boxes (-1, -1, -1) indicates that the user did not specify the bounding box and the effect shall be as if the bounding box were infinitely large. A *bboxSize* value of (0, 0, 0) is valid and represents a point in space (i.e., an infinitely small box). Specified *bboxSize* field values shall be >= 0.0 or equal to (-1, -1, -1). The *bboxCenter* fields specify a position offset from the local coordinate system.

The *bboxCenter* and *bboxSize* fields may be used to specify a maximum possible bounding box for the objects inside a grouping node (e.g., Transform). These are used as hints to optimize certain operations such as determining whether or not the group needs to be drawn. The bounding box shall be large enough at all times to enclose the union of the group's children's bounding boxes; it shall not include any transformations performed by the group itself (i.e., the bounding box is defined in the local coordinate system of the children). Results are undefined if the specified bounding box is smaller than the true bounding box of the group.

## 4.6.5 Grouping and children nodes

Grouping nodes have a field that contains a list of children nodes. Each grouping node defines a coordinate space for its children. This coordinate space is relative to the coordinate space of the node of which the group node is a child. Such a node is called a *parent* node. This means that transformations accumulate down the scene graph hierarchy.

The following node types are grouping nodes:

- Anchor
- Billboard
- Collision
- Group
- Inline

- LOD
- Switch
- Transform

The following node types are children nodes:

- Anchor
- Background
- Billboard
- Collision
- ColorInterpolator
- CoordinateInterpolator
- CylinderSensor
- DirectionalLight
- Fog
- Group
- Inline
- LOD
- NavigationInfo
- NormalInterpolator
- OrientationInterpolator
- PlaneSensor
- PointLight
- PositionInterpolator
- ProximitySensor
- ScalarInterpolator
- Script
- Shape
- Sound
- SpotLight
- SphereSensor
- Switch
- TimeSensor
- TouchSensor
- Transform
- Viewpoint
- VisibilitySensor
- WorldInfo

The following node types are not valid as children nodes:

- Appearance
- AudioClip
- Box
- Color
- Cone
- Coordinate
- Cylinder
- ElevationGrid
- Extrusion
- ImageTexture
- IndexedFaceSet
- IndexedLineSet
- Material
- MovieTexture
- Normal
- PointSet
- Sphere
- Text
- TextureCoordinate
- TextureTransform

All grouping nodes except Inline, LOD, and Switch also have *addChildren* and *removeChildren* eventIn definitions. The *addChildren* event appends nodes to the grouping node's *children* field. Any nodes passed to the *addChildren* event that are already in the group's children list are ignored. For example, if the *children* field contains the nodes Q, L and S (in order) and the group receives an *addChildren* eventIn containing (in order) nodes A, L, and Z, the result is a *children* field containing (in order) nodes Q, L, S, A, and Z.

The *removeChildren* event removes nodes from the grouping node's *children* field. Any nodes in the *removeChildren* event that are not in the grouping node's *children* list are ignored. If the *children* field contains the nodes Q, L, S, A and Z and it receives a *removeChildren* eventIn containing nodes A, L, and Z, the result is Q, S.

Note that a variety of node types reference other node types through fields. Some of these are parent-child relationships, while others are not (there are node-specific semantics). Table 4.3 lists all node types that reference other nodes through fields.

APPENDIX KK

**Table 4.3 -- Nodes with SFNode or MFNode fields**

| Node Type | Field | Valid Node Types for Field |
|---|---|---|
| Anchor | *children* | Valid children nodes |
| Appearance | *material* | Material |
| | *texture* | ImageTexture, MovieTexture, Pixel Texture |
| Billboard | *children* | Valid children nodes |
| Collision | *children* | Valid children nodes |
| ElevationGrid | *color* | Color |
| | *normal* | Normal |
| | *texCoord* | TextureCoordinate |
| Group | *children* | Valid children nodes |
| IndexedFaceSet | *color* | Color |
| | *coord* | Coordinate |
| | *normal* | Normal |
| | *texCoord* | TextureCoordinate |
| IndexedLineSet | *color* | Color |
| | *coord* | Coordinate |
| LOD | *level* | Valid children nodes |
| Shape | *appearance* | Appearance |
| | *geometry* | Box, Cone, Cylinder, ElevationGrid, Extrusion, IndexedFaceSet, IndexedLineSet, PointSet, Sphere, Text |
| Sound | *source* | AudioClip, MovieTexture |
| Switch | *choice* | Valid children nodes |
| Text | *fontStyle* | FontStyle |

| | | |
|---|---|---|
| Transform | *children* | Valid children nodes |

## 4.6.6 Light sources

Shape nodes are illuminated by the sum of all of the lights in the world that affect them. This includes the contribution of both the direct and ambient illumination from light sources. Ambient illumination results from the scattering and reflection of light originally emitted directly by light sources. The amount of ambient light is associated with the individual lights in the scene. This is a gross approximation to how ambient reflection actually occurs in nature.

The following node types are light source nodes:

- DirectionalLight
- PointLight
- SpotLight

All light source nodes contain an *intensity*, a *color*, and an *ambientIntensity* field. The *intensity* field specifies the brightness of the direct emission from the light, and the *ambientIntensity* specifies the intensity of the ambient emission from the light. Light intensity may range from 0.0 (no light emission) to 1.0 (full intensity). The *color* field specifies the spectral colour properties of both the direct and ambient light emission as an RGB value.

PointLight and SpotLight illuminate all objects in the world that fall within their volume of lighting influence regardless of location within the transformation hierarchy. PointLight defines this volume of influence as a sphere centred at the light (defined by a radius). SpotLight defines the volume of influence as a solid angle defined by a radius and a cutoff angle. DirectionalLight nodes illuminate only the objects descended from the light's parent grouping node, including any descendent children of the parent grouping nodes.

## 4.6.7 Sensor nodes

### 4.6.7.1 Introduction to sensors

The following node types are sensor nodes:

- Anchor
- Collision
- CylinderSensor
- PlaneSensor
- ProximitySensor
- SphereSensor
- TimeSensor
- TouchSensor
- VisibilitySensor

Sensors are children nodes in the hierarchy and therefore may be parented by grouping nodes as described in 4.6.5, Grouping and children nodes.

Each type of sensor defines when an event is generated. The state of the scene graph after several sensors have generated events shall be as if each event is processed separately, in order. If sensors generate events at the same time, the state of the scene graph will be undefined if the results depend on the ordering of the events.

It is possible to create dependencies between various types of sensors. For example, a TouchSensor may result in a change to a VisibilitySensor node's transformation, which in turn may cause the VisibilitySensor node's visibility status to change.

The following two sections classify sensors into two categories: *environmental sensors* and *pointing-device sensors*.

### 4.6.7.2 Environmental sensors

The following node types are environmental sensors:

- Collision
- ProximitySensor
- TimeSensor
- VisibilitySensor

The ProximitySensor detects when the user navigates into a specified region in the world. The ProximitySensor itself is not visible. The TimeSensor is a clock that has no geometry or location associated with it; it is used to start and stop time-based nodes such as interpolators. The VisibilitySensor detects when a specific part of the world becomes visible to the user. The Collision grouping node detects when the user collides with objects in the virtual world. Proximity, time, collision, and visibility sensors are each processed independently of whether others exist or overlap.

When environmental sensors are inserted into the transformation hierarchy and before the presentation is updated (i.e., read from file or created by a script), they shall generate events indicating any conditions which the sensor is intended to detect (see 4.10.3, Execution model). The conditions for individual sensor types to generate these initial events are defined in the individual node specifications in 6, Node reference.

### 4.6.7.3 Pointing-device sensors

Pointing-device sensors detect user pointing events such as the user clicking on a piece of geometry (i.e., TouchSensor). The following node types are pointing-device sensors:

- Anchor
- CylinderSensor
- PlaneSensor
- SphereSensor
- TouchSensor

A pointing-device sensor is activated when the user locates the pointing device over geometry that is influenced by that specific pointing-device sensor. Pointing-device sensors have influence over all geometry that is descended from the sensor's parent groups. In the case of the Anchor node, the Anchor node itself is considered to be the parent group. Typically, the pointing-device sensor is a sibling to the geometry that it influences. In other cases, the sensor is a sibling to groups which contain geometry (i.e., are influenced by the pointing-device sensor).

The appearance properties of the geometry do not affect activation of the sensor. In particular, transparent materials or textures shall be treated as opaque with respect to activation of pointing-device sensors.

For a given user activation, the lowest enabled pointing-device sensor in the hierarchy is activated. All other pointing-device sensors above the lowest enabled pointing-device sensor are ignored. The hierarchy is defined by the geometry node over which the pointing-device sensor is located and the entire hierarchy upward. If there are multiple pointing-device sensors tied for lowest, each of these is activated simultaneously and independently, possibly resulting in multiple sensors activating and generating output simultaneously. This feature allows combinations of pointing-device sensors (e.g., TouchSensor and PlaneSensor). If a pointing-device sensor appears in the transformation hierarchy multiple times (DEF/USE), it shall be tested for activation in all of the coordinate systems in which it appears.

If a pointing-device sensor is not enabled when the pointing-device button is activated, it will not generate events related to the pointing device until after the pointing device is deactivated and the sensor is enabled (i.e., enabling a sensor in the middle of dragging does not result in the sensor activating immediately).

The Anchor node is considered to be a pointing-device sensor when trying to determine which sensor (or Anchor node) to activate. For example, a click on *Shape3* is handled by *SensorD*, a click on *Shape2* is handled by *SensorC* and the *AnchorA*, and a click on *Shape1* is handled by *SensorA* and *SensorB*:

```
Group {
  children [
    DEF Shape1  Shape       { ... }
    DEF SensorA TouchSensor { ... }
    DEF SensorB PlaneSensor { ... }
    DEF AnchorA Anchor {
      url "..."
      children [
        DEF Shape2  Shape { ... }
        DEF SensorC TouchSensor { ... }
        Group {
          children [
            DEF Shape3  Shape { ... }
            DEF SensorD TouchSensor { ... }
          ]
        }
      ]
    }
  ]
}
```

#### 4.6.7.4 Drag sensors

*Drag sensors* are a subset of pointing-device sensors. There are three types of drag sensors: CylinderSensor, PlaneSensor, and SphereSensor. Drag sensors have two eventOuts in common, *trackPoint_changed* and *<value>_changed*. These eventOuts send events for each movement of the activated pointing device according to their "virtual geometry" (e.g., cylinder for CylinderSensor). The *trackPoint_changed* eventOut sends the intersection point of the *bearing* with the drag sensor's virtual geometry. The *<value>_changed* eventOut sends the sum of the relative change since activation plus the sensor's *offset* field. The type and name of *<value>_changed* depends on the drag sensor type: *rotation_changed* for CylinderSensor, *translation_changed* for PlaneSensor, and *rotation_changed* for SphereSensor.

To simplify the application of these sensors, each node has an *offset* and an *autoOffset* exposed field. When the sensor generates events as a response to the activated pointing device motion, *<value>_changed* sends the sum of the relative change since the initial activation plus the *offset* field value. If *autoOffset* is TRUE when the pointing-device is deactivated, the *offset* field is set to the sensor's last *<value>_changed* value and *offset* sends an *offset_changed* eventOut. This enables subsequent grabbing operations to accumulate the changes. If *autoOffset* is FALSE, the sensor does not set the *offset* field value at deactivation (or any other time).

APPENDIX KK

### 4.6.7.5 Activating and manipulating sensors

The pointing device controls a pointer in the virtual world. While activated by the pointing device, a sensor will generate events as the pointer moves. Typically the pointing device may be categorized as either 2D (e.g., conventional mouse) or 3D (e.g., wand). It is suggested that the pointer controlled by a 2D device is mapped onto a plane a fixed distance from the viewer and perpendicular to the line of sight. The mapping of a 3D device may describe a 1:1 relationship between movement of the pointing device and movement of the pointer.

The position of the pointer defines a bearing which is used to determine which geometry is being indicated. When implementing a 2D pointing device it is suggested that the bearing is defined by the vector from the viewer position through the location of the pointer. When implementing a 3D pointing device it is suggested that the bearing is defined by extending a vector from the current position of the pointer in the direction indicated by the pointer.

In all cases the pointer is considered to be indicating a specific geometry when that geometry is intersected by the bearing. If the bearing intersects multiple sensors' geometries, only the sensor nearest to the pointer will be eligible for activation.

## 4.6.8 Interpolator nodes

Interpolator nodes are designed for linear keyframed animation. An interpolator node defines a piecewise-linear function, *f(t)*, on the interval (*-infinity*, *+infinity)*. The piecewise-linear function is defined by *n* values of *t*, called *key*, and the *n* corresponding values of *f(t)*, called *keyValue*. The keys shall be monotonically non-decreasing, otherwise the results are undefined. The keys are not restricted to any interval.

An interpolator node evaluates *f(t)* given any value of *t* (via the *set_fraction* eventIn) as follows: Let the *n* keys $t_0$, $t_1$, $t_2$, ..., $t_{n-1}$ partition the domain (*-infinity*, *+infinity*) into the *n*+1 subintervals given by (*-infinity*, $t_0$), $[t_0, t_1)$, $[t_1, t_2)$, ... , $[t_{n-1}$, *+infinity*). Also, let the *n* values $v_0$, $v_1$, $v_2$, ..., $v_{n-1}$ be the values of *f(t)* at the associated key values. The piecewise-linear interpolating function, *f(t)*, is defined to be

$$\begin{aligned} f(t) &= v_0, \text{ if } t <= t_0, \\ &= v_{n-1}, \text{ if } t >= t_{n-1}, \\ &= linterp(t, v_i, v_{i+1}), \text{ if } t_i <= t <= t_{i+1} \end{aligned}$$

where *linterp(t,x,y)* is the linear interpolant,
*i* belongs to {0,1,..., n-2}.

The third conditional value of *f(t)* allows the defining of multiple values for a single key, (i.e., limits from both the left and right at a discontinuity in *f(t))*. The first specified value is used as the limit of *f(t)* from the left, and the last specified value is used as the limit of *f(t)* from the right. The value of *f(t)* at a multiply defined key is indeterminate, but should be one of the associated limit values.

The following node types are interpolator nodes, each based on the type of value that is interpolated:

- ColorInterpolator
- CoordinateInterpolator
- NormalInterpolator
- OrientationInterpolator
- PositionInterpolator
- ScalarInterpolator

All interpolator nodes share a common set of fields and semantics:

```
eventIn      SFFloat      set_fraction
exposedField MFFloat      key           [...]
exposedField MF<type>     keyValue      [...]
eventOut     [S|M]F<type> value_changed
```

The type of the *keyValue* field is dependent on the type of the interpolator (e.g., the ColorInterpolator's *keyValue* field is of type MFColor).

The *set_fraction* eventIn receives an SFFloat event and causes the interpolator function to evaluate, resulting in a *value_changed* eventOut with the same timestamp as the *set_fraction* event.

ColorInterpolator, OrientationInterpolator, PositionInterpolator, and ScalarInterpolator output a single-value field to *value_changed*. Each value in the *keyValue* field corresponds in order to the parameter value in the *key* field. Results are undefined if the number of values in the *key* field of an interpolator is not the same as the number of values in the *keyValue* field.

CoordinateInterpolator and NormalInterpolator send multiple-value results to *value_changed*. In this case, the *keyValue* field is an *n* x *m* array of values, where *n* is the number of values in the key field and *m* is the number of values at each keyframe. Each *m* values in the *keyValue* field correspond, in order, to a parameter value in the *key* field. Each *value_changed* event shall contain *m* interpolated values. Results are undefined if the number of values in the *keyValue* field divided by the number of values in the *key* field is not a positive integer.

If an interpolator node's *value* eventOut is read before it receives any inputs, *keyValue*[0] is returned if *keyValue* is not empty. If *keyValue* is empty (i.e., [ ]), the initial value for the eventOut type is returned (e.g., (0, 0, 0) for SFVec3f); see 5, Field and event reference, for initial event values.

The location of an interpolator node in the transformation hierarchy has no effect on its operation. For example, if a parent of an interpolator node is a Switch node with *whichChoice* set to -1 (i.e., ignore its children), the interpolator continues to operate as specified (receives and sends events).

## 4.6.9 Time-dependent nodes

AudioClip, MovieTexture, and TimeSensor are *time-dependent* nodes that activate and deactivate themselves at specified times. Each of these nodes contains the exposedFields: *startTime*, *stopTime*, and *loop*, and the eventOut: *isActive*. The values of the exposedFields are used to determine when the node becomes active or inactive Also, under certain conditions, these nodes ignore events to some of their exposedFields. A node ignores an eventIn by not accepting the new value and not generating an eventOut*_changed* event. In this subclause, an abstract time-dependent node can be any one of AudioClip, MovieTexture, or TimeSensor.

Time-dependent nodes can execute for 0 or more cycles. A cycle is defined by field data within the node. If, at the end of a cycle, the value of *loop* is FALSE, execution is terminated (see below for events at termination). Conversely, if *loop* is TRUE at the end of a cycle, a time-dependent node continues execution into the next cycle. A time-dependent node with *loop* TRUE at the end of every cycle continues cycling forever if *startTime* >= *stopTime*, or until *stopTime* if *startTime* < *stopTime*.

A time-dependent node generates an *isActive* TRUE event when it becomes active and generates an *isActive* FALSE event when it becomes inactive. These are the only times at which an *isActive* event is generated. In particular, *isActive* events are not sent at each tick of a simulation.

A time-dependent node is inactive until its *startTime* is reached. When time *now* becomes greater than or equal to *startTime*, an *isActive* TRUE event is generated and the time-dependent node becomes active (*now* refers to the time at which the browser is simulating and displaying the virtual world). When a time-dependent node is read from a VRML file and the ROUTEs specified within the VRML file have been established, the node should determine if it is active and, if so, generate an *isActive* TRUE event and begin generating any other necessary events. However, if a node would have become inactive at any time before the reading of the VRML file, no events are generated upon the completion of the read.

An active time-dependent node will become inactive when *stopTime* is reached if *stopTime* > *startTime.* The value of *stopTime* is ignored if *stopTime* <= *startTime.* Also, an active time-dependent node will become inactive at the end of the current cycle if *loop* is FALSE. If an active time-dependent node receives a *set_loop* FALSE event, execution continues until the end of the current cycle or until *stopTime* (if *stopTime* > *startTime*), whichever occurs first. The termination at the end of cycle can be overridden by a subsequent *set_loop* TRUE event.

Any *set_startTime* events to an active time-dependent node are ignored. Any *set_stopTime* event where *stopTime* <= *startTime* sent to an active time-dependent node is also ignored. A *set_stopTime* event where *startTime* < *stopTime* <= *now* sent to an active time-dependent node results in events being generated as if *stopTime* has just been reached. That is, final events, including an *isActive* FALSE, are generated and the node becomes inactive. The *stopTime_changed* event will have the *set_stopTime* value. Other final events are node-dependent (c.f., TimeSensor).

A time-dependent node may be restarted while it is active by sending a *set_stopTime* event equal to the current time (which will cause the node to become inactive) and a *set_startTime* event, setting it to the current time or any time in the future. These events will have the same time stamp and should be processed as *set_stopTime,* then *set_startTime* to produce the correct behaviour.

The default values for each of the time-dependent nodes are specified such that any node with default values is already inactive (and, therefore, will generate no events upon loading). A time-dependent node can be defined such that it will be active upon reading by specifying *loop* TRUE. This use of a non-terminating time-dependent node should be used with caution since it incurs continuous overhead on the simulation.

Figure 4.2 illustrates the behavior of several common cases of time-dependent nodes. In each case, the initial conditions of *startTime*, *stopTime*, *loop*, and the time-dependent node's cycle interval are labelled, the red region denotes the time period during which the time-dependent node is active, the arrows represent eventIns received by and eventOuts sent by the time-dependent node, and the horizontal axis represents time.

## 4.6.10 Bindable children nodes

The Background, Fog, NavigationInfo, and Viewpoint nodes have the unique behaviour that only one of each type can be bound (i.e., affecting the user's experience) at any instant in time. The browser shall maintain an independent, separate stack for each type of bindable node. Each of these nodes includes a *set_bind* eventIn and an *isBound* eventOut. The *set_bind* eventIn is used to move a given node to and from its respective top of stack. A TRUE value sent to the *set_bind* eventIn moves the node to the top of the stack; sending a FALSE value removes it from the stack. The *isBound* event is output when a given node is:

a. moved to the top of the stack;

b. removed from the top of the stack;

c. pushed down from the top of the stack by another node being placed on top.

That is, *isBound* events are sent when a given node becomes, or ceases to be, the active node. The node at the top of stack, (the most recently bound node), is the active node for its type and is used by the browser to set the world state. If the stack is empty (i.e., either the VRML file has no bindable nodes for a given type or the stack has been popped until empty), the default field values for that node type are used to set world state. The results are undefined if a multiply instanced (DEF/USE) bindable node is bound.

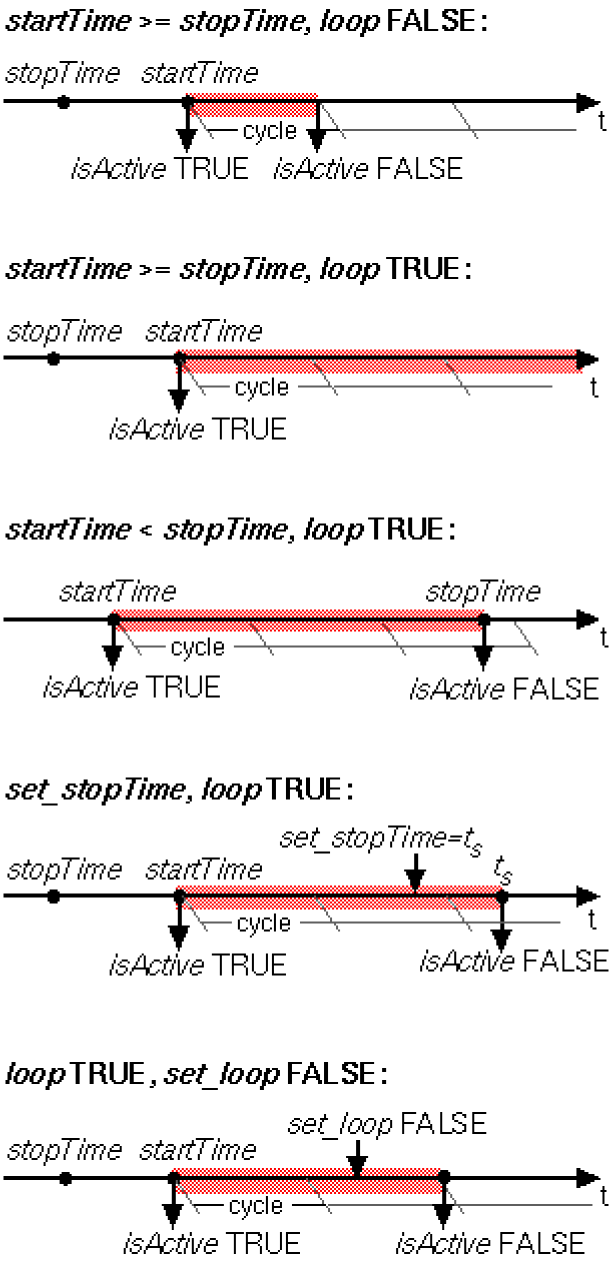


**Figure 4.2 -- Examples of time-dependent node execution**

The following rules describe the behaviour of the binding stack for a node of type <*bindable node*>, (Background, Fog, NavigationInfo, or Viewpoint):

a. During read, the first encountered <*bindable node*> is bound by pushing it to the top of the <*bindable node*> stack. Nodes contained within Inlines, within the strings passed to the Browser.createVrmlFromString() method, or within VRML files passed to the Browser.createVrmlFromURL() method (see 4.12.10, Browser script interface)are not candidates for the first encountered <*bindable node*>. The first node within a prototype instance is a valid candidate for the first encountered <*bindable node*>. The first encountered <*bindable node*> sends an *isBound* TRUE event.

b. When a *set_bind* TRUE event is received by a <*bindable node*>,

   1. If it is not on the top of the stack: the current top of stack node sends an *isBound* FALSE event. The new node is moved to the top of the stack and becomes the currently bound <*bindable node*>. The new <*bindable node*> (top of stack) sends an *isBound* TRUE event.

   2. If the node is already at the top of the stack, this event has no effect.

c. When a *set_bind* FALSE event is received by a <*bindable node*> in the stack, it is removed from the stack. If it was on the top of the stack,

   1. it sends an *isBound* FALSE event;

   2. the next node in the stack becomes the currently bound <*bindable node*> (i.e., pop) and issues an *isBound* TRUE event.

d. If a *set_bind* FALSE event is received by a node not in the stack, the event is ignored and *isBound* events are not sent.

e. When a node replaces another node at the top of the stack, the *isBound* TRUE and FALSE eventOuts from the two nodes are sent simultaneously (i.e., with identical timestamps).

f. If a bound node is deleted, it behaves as if it received a *set_bind* FALSE event (see f above).

## 4.6.11 Texture maps

### 4.6.11.1 Texture map formats

Four node types specify texture maps: Background, ImageTexture, MovieTexture, and PixelTexture. In all cases, texture maps are defined by 2D images that contain an array of colour values describing the texture. The texture map values are interpreted differently depending on the number of components in the texture map and the specifics of the image format. In general, texture maps may be described using one of the following forms:

a. *Intensity textures* (one-component)

b. *Intensity plus alpha opacity textures* (two-component)

c. *Full RGB textures* (three-component)

d. *Full RGB plus alpha opacity textures* (four-component)

Note that most image formats specify an alpha opacity, not transparency (where alpha = 1 - transparency).

See Table 4.5 and Table 4.6 for a description of how the various texture types are applied.

### 4.6.11.2 Texture map image formats

Texture nodes that require support for the PNG (see 2.[PNG]) image format (6.5, Background, and 6.22, ImageTexture) shall interpret the PNG pixel formats in the following way:

a. Greyscale pixels without alpha or simple transparency are treated as intensity textures.

b. Greyscale pixels with alpha or simple transparency are treated as intensity plus alpha textures.

c. RGB pixels without alpha channel or simple transparency are treated as full RGB textures.

d. RGB pixels with alpha channel or simple transparency are treated as full RGB plus alpha textures.

If the image specifies colours as indexed-colour (i.e., palettes or colourmaps), the following semantics should be used (note that `greyscale' refers to a palette entry with equal red, green, and blue values):

a. If all the colours in the palette are greyscale and there is no transparency chunk, it is treated as an intensity texture.

b. If all the colours in the palette are greyscale and there is a transparency chunk, it is treated as an intensity plus opacity texture.

c. If any colour in the palette is not grey and there is no transparency chunk, it is treated as a full RGB texture.

d. If any colour in the palette is not grey and there is a transparency chunk, it is treated as a full RGB plus alpha texture.

Texture nodes that require support for JPEG files (see 2.[JPEG], 6.5, Background, and 6.22, ImageTexture) shall interpret JPEG files as follows:

i. Greyscale files (number of components equals 1) are treated as intensity textures.

j. YCbCr files are treated as full RGB textures.

k. No other JPEG file types are required. It is recommended that other JPEG files are treated as a full RGB textures.

Texture nodes that support MPEG files (see 2.[MPEG] and 6.28, MovieTexture) shall treat MPEG files as full RGB textures.

Texture nodes that recommend support for GIF files (see E.[GIF], 6.5, Background, and 6.22, ImageTexture) shall follow the applicable semantics described above for the PNG format.

# 4.7 Field, eventIn, and eventOut semantics

Fields are placed inside node statements in a VRML file, and define the persistent state of the virtual world. Results are undefined if multiple values for the same field in the same node (e.g., **`Sphere { radius 1.0 radius 2.0 }`**) are declared.

EventIns and eventOuts define the types and names of events that each type of node may receive or generate. Events are transient and event values are not written to VRML files. Each node interprets the values of the events sent to it or generated by it according to its implementation.

Field, eventIn, and eventOut types, and field encoding syntax, are described in 5, Field and event reference.

An *exposedField* can receive events like an eventIn, can generate events like an eventOut, and can be stored in VRML files like a field. An exposedField named *zzz* can be referred to as '*set_zzz*' and treated as an eventIn, and can be referred to as '*zzz_changed*' and treated as an eventOut. The initial value of an exposedField is its value in the VRML file, or the default value for the node in which it is contained, if a value is not specified. When an exposedField receives an event it shall generate an event with the same value and timestamp. The following sources, in precedence order, shall be used to determine the initial value of the exposedField:

a. the user-defined value in the instantiation (if one is specified);

b. the default value for that field as specified in the node or prototype definition.

The rules for naming fields, exposedFields, eventOuts, and eventIns for the built-in nodes are as follows:

c. All names containing multiple words start with a lower case letter, and the first letter of all subsequent words is capitalized (e.g., *addChildren*), with the exception of *set_* and *_changed*, as described below.

d. All eventIns have the prefix "*set_*", with the exception of the *addChildren* and *removeChildren* eventIns.

e. Certain eventIns and eventOuts of type SFTime do not use the "*set_*" prefix or "*_changed*" suffix.

f. All other eventOuts have the suffix "*_changed*" appended, with the exception of eventOuts of type SFBool. Boolean eventOuts begin with the word "*is*" (e.g., *isFoo*) for better readability.

# 4.8 Prototype semantics

## 4.8.1 Introduction

The PROTO statement defines a new node type in terms of already defined (built-in or prototyped) node types. Once defined, prototyped node types may be instantiated in the scene graph exactly like the built-in node types.

Node type names shall be unique in each VRML file. The results are undefined if a prototype is given the same name as a built-in node type or a previously defined prototype in the same scope.

## 4.8.2 PROTO interface declaration semantics

The prototype interface defines the fields, eventIns, and eventOuts for the new node type. The interface declaration includes the types and names for the eventIns and eventOuts of the prototype, as well as the types, names, and default values for the prototype's fields.

The interface declaration may contain exposedField declarations, which are a convenient way of defining a field, eventIn, and eventOut at the same time. If an exposedField named *zzz* is declared, it is equivalent to declaring a field named *zzz*, an eventIn named *set_zzz*, and an eventOut named *zzz_changed*.

Each prototype instance can be considered to be a complete copy of the prototype, with its own fields, events, and copy of the prototype definition. A prototyped node type is instantiated using standard node syntax. For example, the following prototype (which has an empty interface declaration):

```
PROTO Cube [ ] { Box { } }
```

may be instantiated as follows:

```
Shape { geometry Cube { } }
```

It is recommended that user-defined field or event names defined in PROTO interface declarations statements follow the naming conventions described in 4.7, Field, eventIn, and eventOut semantics.

If an eventOut in the prototype declaration is associated with an exposedField in the prototype definition, the initial value of the eventOut shall be the initial value of the exposedField. If the eventOut is associated with multiple exposedFields, the results are undefined.

## 4.8.3 PROTO definition semantics

A prototype definition consists of one or more nodes, nested PROTO statements, and ROUTE statements. The first node type determines how instantiations of the prototype can be used in a VRML file. An instantiation is created by filling in the parameters of the prototype declaration and inserting copies of the first node (and its scene graph) wherever the prototype instantiation occurs. For example, if the first node in the prototype definition is a Material node, instantiations of the prototype can be used wherever a Material node can be used. Any other nodes and accompanying scene graphs are not part of the transformation hierarchy, but may be referenced by ROUTE statements or Script nodes in the prototype definition.

Nodes in the prototype definition may have their fields, eventIns, or eventOuts associated with the fields, eventIns, and eventOuts of the prototype interface declaration. This is accomplished using IS statements in the body of the node. When prototype instances are read from a VRML file, field values for the fields of the prototype interface may be given. If given, the field values are used for all nodes in the prototype definition that have IS statements for those fields. Similarly, when a prototype instance is sent an event, the event is delivered to all nodes that have IS statements for that event. When a node in a prototype instance generates an event that has an IS statement, the event is sent to any eventIns connected (via ROUTE) to the prototype instance's eventOut.

IS statements may appear inside the prototype definition wherever fields may appear. IS statements shall refer to fields or events defined in the prototype declaration. Results are undefined if an IS statement refers to a non-existent declaration. Results are undefined if the type of the field or event being associated by the IS statement does not match the type declared in the prototype's interface declaration. For example, it is illegal to associate an SFColor with an SFVec3f. It is also illegal to associate an SFColor with an MFColor or *vice versa*.

Results are undefined if an IS statement:

- eventIn is associated with a field or an eventOut;
- eventOut is associated with a field or eventIn;
- field is associated with an eventIn or eventOut.

An exposedField in the prototype interface may be associated only with an exposedField in the prototype definition, but an exposedField in the prototype definition may be associated with either a field, eventIn, eventOut or exposedField in the prototype interface. When associating an exposedField in a prototype definition with an eventIn or eventOut in the prototype declaration, it is valid to use either the shorthand exposedField name (e.g., *translation*) or the explicit event name (e.g., *set_translation* or *translation_changed*). Table 4.4 defines the rules for mapping between the prototype declarations and the primary scene graph's nodes (*yes* denotes a legal mapping, *no* denotes an error).

**Table 4.4 -- Rules for mapping PROTOTYPE declarations to node instances**

| | | **Prototype declaration** | | | |
|---|---|---|---|---|---|
| | | **exposedField** | **field** | **eventIn** | **eventOut** |
| **Prototype definition** | **exposedField** | yes | yes | yes | yes |
| | **field** | no | yes | no | no |
| | **eventIn** | no | no | yes | no |
| | **eventOut** | no | no | no | yes |

APPENDIX KK

Results are undefined if a field, eventIn, or eventOut of a node in the prototype definition is associated with more than one field, eventIn, or eventOut in the prototype's interface (i.e., multiple IS statements for a field, eventIn, and eventOut in a node in the prototype definition), but multiple IS statements for the fields, eventIns, and eventOuts in the prototype interface declaration is valid. Results are undefined if a field of a node in a prototype definition is both defined with initial values (i.e., field statement) and associated by an IS statement with a field in the prototype's interface. If a prototype interface has an eventOut $E$ associated with multiple eventOuts in the prototype definition $ED_i$ , the value of $E$ is the value of the eventOut that generated the event with the greatest timestamp. If two or more of the eventOuts generated events with identical timestamps, results are undefined.

### 4.8.4 Prototype scoping rules

Prototype definitions appearing inside a prototype definition (i.e., nested) are local to the enclosing prototype. IS statements inside a nested prototype's implementation may refer to the prototype declarations of the innermost prototype.

A PROTO statement establishes a DEF/USE name scope separate from the rest of the scene and separate from any nested PROTO statements. Nodes given a name by a DEF construct inside the prototype may not be referenced in a USE construct outside of the prototype's scope. Nodes given a name by a DEF construct outside the prototype scope may not be referenced in a USE construct inside the prototype scope.

A prototype may be instantiated in a file anywhere after the completion of the prototype definition. A prototype may not be instantiated inside its own implementation (i.e., recursive prototypes are illegal).

## 4.9 External prototype semantics

### 4.9.1 Introduction

The EXTERNPROTO statement defines a new node type. It is equivalent to the PROTO statement, with two exceptions. First, the implementation of the node type is stored externally, either in a VRML file containing an appropriate PROTO statement or using some other implementation-dependent mechanism. Second, default values for fields are not given since the implementation will define appropriate defaults.

### 4.9.2 EXTERNPROTO interface semantics

The semantics of the EXTERNPROTO are exactly the same as for a PROTO statement, except that default field and exposedField values are not specified locally. In addition, events sent to an instance of an externally prototyped node may be ignored until the implementation of the node is found.

Until the definition has been loaded, the browser shall determine the initial value of exposedFields using the following rules (in order of precedence):

a. the user-defined value in the instantiation (if one is specified);

b. the default value for that field type.

For eventOuts, the initial value on startup will be the default value for that field type. During the loading of an EXTERNPROTO, if an initial value of an eventOut is found, that value is applied to the eventOut and no event is generated.

The names and types of the fields, exposedFields, eventIns, and eventOuts of the interface declaration shall be a subset of those defined in the implementation. Declaring a field or event with a non-matching name is an error, as is declaring a field or event with a matching name but a different type.

It is recommended that user-defined field or event names defined in EXTERNPROTO interface statements follow the naming conventions described in 4.7, Field, eventIn, and eventOut semantics.

### 4.9.3 EXTERNPROTO URL semantics

The string or strings specified after the interface declaration give the location of the prototype's implementation. If multiple strings are specified, the browser searches in the order of preference (see 4.5.2, URLs).

If a URL in an EXTERNPROTO statement refers to a VRML file, the first PROTO statement found in the VRML file (excluding EXTERNPROTOs) is used to define the external prototype's definition. The name of that prototype does not need to match the name given in the EXTERNPROTO statement. Results are undefined if a URL in an EXTERNPROTO statement refers to a non-VRML file

To enable the creation of libraries of reusable PROTO definitions, browsers shall recognize EXTERNPROTO URLs that end with "#*name*" to mean the PROTO statement for "name" in the given VRML file. For example, a library of standard materials might be stored in a VRML file called "materials.wrl" that looks like:

```
#VRML V2.0 utf8
PROTO Gold   [] { Material { ... } }
PROTO Silver [] { Material { ... } }
...etc.
```

A material from this library could be used as follows:

```
#VRML V2.0 utf8
EXTERNPROTO GoldFromLibrary [] "http://.../materials.wrl#Gold"
...
Shape {
    appearance Appearance { material GoldFromLibrary {} }
    geometry   ...
}
...
```

## 4.10 Event processing

### 4.10.1 Introduction

Most node types have at least one eventIn definition and thus can receive *events*. Incoming events are data messages sent by other nodes to change some state within the receiving node. Some nodes also have eventOut definitions. These are used to send data messages to destination nodes that some state has changed within the source node.

If an eventOut is read before it has sent any events, the *initial value* as specified in 5, Field and event reference, for each field/event type is returned.

## 4.10.2 Route semantics

The connection between the node generating the event and the node receiving the event is called a *route*. Routes are not nodes. The ROUTE statement is a construct for establishing event paths between nodes. ROUTE statements may either appear at the top level of a VRML file, in a prototype definition, or inside a node wherever fields may appear. Nodes referenced in a ROUTE statement shall be defined before the ROUTE statement.

The types of the eventIn and the eventOut shall match exactly. For example, it is illegal to route from an SFFloat to an SFInt32 or from an SFFloat to an MFFloat.

Routes may be established only from eventOuts to eventIns. For convenience, when routing to or from an eventIn or eventOut (or the eventIn or eventOut part of an exposedField), the *set_* or *_changed* part of the event's name is optional. If the browser is trying to establish a ROUTE to an eventIn named *zzz* and an eventIn of that name is not found, the browser shall then try to establish the ROUTE to the eventIn named *set_zzz*. Similarly, if establishing a ROUTE from an eventOut named *zzz* and an eventOut of that name is not found, the browser shall try to establish the ROUTE from *zzz_changed*.

Redundant routing is ignored. If a VRML file repeats a routing path, the second and subsequent identical routes are ignored. This also applies for routes created dynamically via a scripting language supported by the browser.

## 4.10.3 Execution model

Once a sensor or Script has generated an *initial event*, the event is propagated from the eventOut producing the event along any ROUTEs to other nodes. These other nodes may respond by generating additional events, continuing until all routes have been honoured. This process is called an *event cascade*. All events generated during a given event cascade are assigned the same timestamp as the initial event, since all are considered to happen instantaneously.

Some sensors generate multiple events simultaneously. Similarly, it is possible that asynchronously generated events could arrive at the identical time as one or more sensor generated event. In these cases, all events generated are part of the same initial event cascade and each event has the same timestamp.

After all events of the initial event cascade are honored, post-event processing performs actions stimulated by the event cascade. The entire sequence of events occuring in a single timestamp are:

a. Perform event cascade evaluation.

b. Call *shutdown( )* on scripts that have received *set_url* events or are being removed from the scene.

c. Send final events from environmental sensors being removed from the transformation hierarchy.

d. Add or remove routes specified in *addRoute( )* or *deleteRoute( )* from any script execution in the preceeding event cascade.

e. Call *eventsProcessed( )* for scripts that have sent events in the just ended event cascade.

f. Send initial events from any dynamically created environmental sensors.

g. Call *initialize( )* of newly loaded script code.

h. If any events were generated from steps 2 through 7, go to step 2 and continue.

Figure 4.3 provides a conceptual illustration of the execution model.

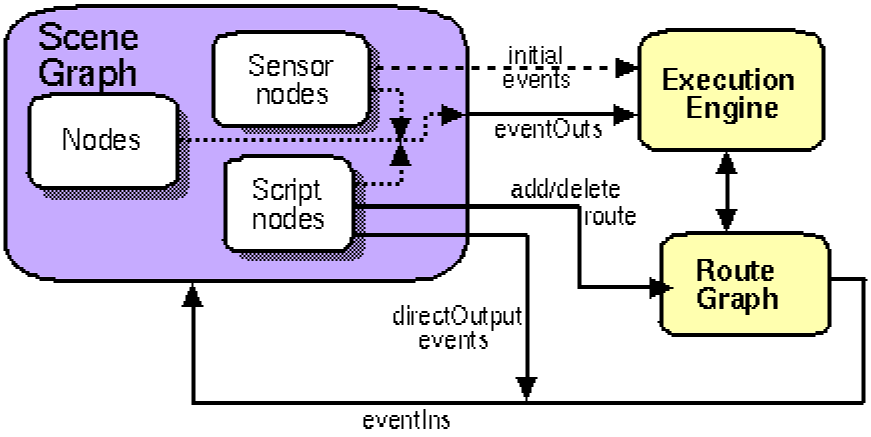


**Figure 4.3 -- Conceptual execution model**

Nodes that contain eventOuts or exposedFields shall produce at most one event per timestamp. If a field is connected to another field via a ROUTE, an implementation shall send only one event per ROUTE per timestamp. This also applies to scripts where the rules for determining the appropriate action for sending eventOuts are defined in 4.12.9.3, Sending eventOuts.

D.19, Execution model, provides an example that demonstrates the execution model. Figure 4.4 illustrates event processing for a single timestamp in example in D.19, Execution model:

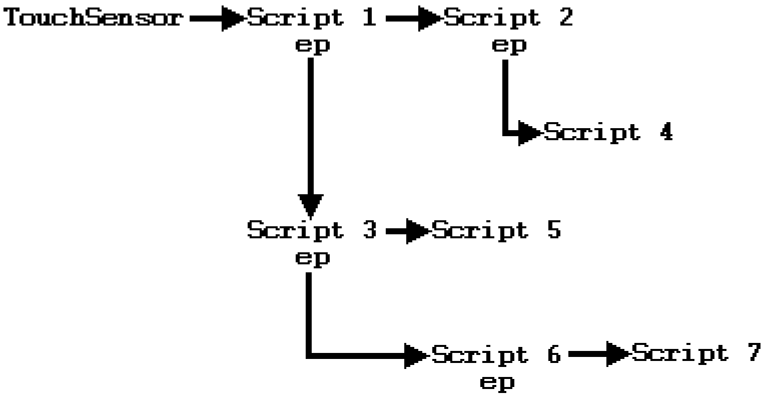


**Figure 4.4 -- Example D.19, event processing order**

In Figure 4.4, arrows coming out of a script at **ep** are events generated during the *eventsProcessed()* call for the script. The other arrows are events sent during an eventIn method. One possible compliant order of execution is as follows:

i. User activates **TouchSensor**

j. Run initial event cascade (step 1)

   1. **Script 1** runs, generates an event for **Script 2**
   2. **Script 2** runs

    3. end of initial event cascade

k. Execute eventsProcessed calls (step 5)

    1. *eventsProcessed* for **Script 1** runs, sends event to **Script 3**
    2. **Script 3** runs, generates events for **Script 5**
    3. **Script 5** runs
    4. *eventsProcessed* for **Script 2** runs, sends events to **Script 4**
    5. **Script 4** runs
    6. end of *eventsProcessed* processing

l. Go to step 2 for generated events (step 8)

m. Execute *eventsProcessed* calls (step 5)

    1. *eventsProcessed* for **Script 3** runs, sends event to **Script 6**
    2. **Script 6** runs, sends event to **Script 7**
    3. **Script 7** runs
    4. *eventsProcessed* for **Script 4** runs, does not generate any events
    5. *eventsProcessed* for **Script 5** runs, does not generate any events
    6. end of *eventsProcessed* processing

n. Go to step 2 for generated events (step 8)

o. Execute *eventsProcessed* calls (step 5)

    1. *eventsProcessed* for **Script 6** runs, does not generate any events
    2. *eventsProcessed* for **Script 7** runs, does not generate any events
    3. end of *eventsProcessed* processing

p. No more events to handle.

The above is not the only possible compliant order of execution. If multiple *eventsProcessed()* methods are pending when step 4 is executed, the order in which these methods is called is not defined. For instance, in the third step of the example, the *eventsProcessed* method is pending for both **Script** 1 and **Script 2**. The order of execution in this case is not defined, so executing the *eventsProcessed* method of **Script 2** before that of **Script 1** would have been compliant. However, executing the *eventsProcessed* method for **Script 3** before that of **Script 2** would not have been compliant because any methods made pending during processing must wait until the next iteration of the event cascade for execution.

## 4.10.4 Loops

Event cascades may contain *loops* where an event *E* is routed to a node that generates an event that eventually results in *E* being generated again. See 4.10.3, Execution model, for the loop breaking rule that limits each eventOut to one event per timestamp. This rule shall also be used to break loops created by cyclic dependencies between different sensor nodes.

### 4.10.5 Fan-in and fan-out

*Fan-in* occurs when two or more routes write to the same eventIn. Events coming into an eventIn from different eventOuts with the same timestamp shall be processed, but the order of evaluation is implementation dependent.

*Fan-out* occurs when one eventOut routes to two or more eventIns. This results in sending any event generated by the eventOut to all of the eventIns.

## 4.11 Time

### 4.11.1 Introduction

The browser controls the passage of time in a world by causing TimeSensors to generate events as time passes. Specialized browsers or authoring applications may cause time to pass more quickly or slowly than in the real world, but typically the times generated by TimeSensors will approximate "real" time. A world's creator should make no assumptions about how often a TimeSensor will generate events but can safely assume that each time event generated will have a timestamp greater than any previous time event.

### 4.11.2 Time origin

Time (0.0) is equivalent to 00:00:00 GMT January 1, 1970. Absolute times are specified in SFTime or MFTime fields as double-precision floating point numbers representing seconds. Negative absolute times are interpreted as happening before 1970.

Processing an event with timestamp *t* may only result in generating events with timestamps greater than or equal to *t*.

### 4.11.3 Discrete and continuous changes

ISO/IEC 14772 does not distinguish between discrete events (such as those generated by a TouchSensor) and events that are the result of sampling a conceptually continuous set of changes (such as the fraction events generated by a TimeSensor). An ideal VRML implementation would generate an infinite number of samples for continuous changes, each of which would be processed infinitely quickly.

Before processing a discrete event, all continuous changes that are occurring at the discrete event's timestamp shall behave as if they generate events at that same timestamp.

Beyond the requirements that continuous changes be up-to-date during the processing of discrete changes, the sampling frequency of continuous changes is implementation dependent. Typically a TimeSensor affecting a visible (or otherwise perceptible) portion of the world will generate events once per *frame*, where a frame is a single rendering of the world or one time-step in a simulation.

APPENDIX KK

# 4.12 Scripting

## 4.12.1 Introduction

Authors often require that VRML worlds change dynamically in response to user inputs, external events, and the current state of the world. The proposition "if the vault is currently closed AND the correct combination is entered, open the vault" illustrates the type of problem which may need addressing. These kinds of decisions are expressed as Script nodes (see 6.40, Script) that receive events from other nodes, process them, and send events to other nodes. A Script node can also keep track of information between subsequent executions (i.e., retaining internal state over time).

This subclause describes the general mechanisms and semantics of all scripting language access protocols. Note that no scripting language is required by ISO/IEC 14772. Details for two scripting languages are in annex B, Java platform scripting reference, and annex C, ECMAScript scripting reference, respectively. If either of these scripting languages are implemented, the Script node implementation shall conform with the definition described in the corresponding annex.

Event processing is performed by a program or script contained in (or referenced by) the Script node's *url* field. This program or script may be written in any programming language that the browser supports.

## 4.12.2 Script execution

A Script node is activated when it receives an event. The browser shall then execute the program in the Script node's *url* field (passing the program to an external interpreter if necessary). The program can perform a wide variety of actions including sending out events (and thereby changing the scene), performing calculations, and communicating with servers elsewhere on the Internet. A detailed description of the ordering of event processing is contained in 4.10, Event processing.

Script nodes may also be executed after they are created (see 4.12.3, Initialize() and shutdown()). Some scripting languages may allow the creation of separate processes from scripts, resulting in continuous execution (see 4.12.6, Asynchronous scripts).

Script nodes receive events in timestamp order. Any events generated as a result of processing an event are given timestamps corresponding to the event that generated them. Conceptually, it takes no time for a Script node to receive and process an event, even though in practice it does take some amount of time to execute a Script.

When a *set_url* event is received by a Script node that contains a script that has been previously initialized for a different URL, the *shutdown()* method of the current script is called (see 4.12.3, Initialize() and shutdown()). Until the new script becomes available, the script shall behave as though it has no executable content. When the new script becomes available, the *Initialize()* method is invoked as defined in 4.10.3, Execution model. The limiting case is when the URL contains inline code that can be immediately executed upon receipt of the *set_url* event (e.g., javascript: protocol). In this case, it can be assumed that the old code is unloaded and the new code loaded instantaneously, after any dynamic route requests have been performed.

## 4.12.3 *Initialize()* and *shutdown()*

The scripting language binding may define an *initialize()* method. This method shall be invoked before the browser presents the world to the user and before any events are processed by any nodes in the same VRML file as the Script node containing this script. Events generated by the *initialize()* method shall have timestamps less than any other events generated by the Script node. This allows script initialization tasks to be performed prior to the user interacting with the world.

Likewise, the scripting language binding may define a *shutdown()* method. This method shall be invoked when the corresponding Script node is deleted or the world containing the Script node is unloaded or replaced by another world. This method may be used as a clean-up operation, such as informing external mechanisms to remove temporary files. No other methods of the script may be invoked after the *shutdown()* method has completed, though the *shutdown()* method may invoke methods or send events while shutting down. Events generated by the *shutdown()* method that are routed to nodes that are being deleted by the same action that caused the *shutdown()* method to execute will not be delivered. The deletion of the Script node containing the *shutdown()* method is not complete until the execution of its *shutdown()* method is complete.

## 4.12.4 *EventsProcessed()*

The scripting language binding may define an *eventsProcessed()* method that is called after one or more events are received. This method allows Scripts that do not rely on the order of events received to generate fewer events than an equivalent Script that generates events whenever events are received. If it is used in some other time-dependent way, *eventsProcessed()* may be nondeterministic, since different browser implementations may call *eventsProcessed()* at different times.

For a single event cascade, a given Script node's eventsProcessed method shall be called at most once. Events generated from an *eventsProcessed()* method are given the timestamp of the last event processed.

## 4.12.5 Scripts with direct outputs

Scripts that have access to other nodes (via SFNode/MFNode fields or eventIns) and that have their *directOutput* field set to TRUE may directly post eventIns to those nodes. They may also read the last value sent from any of the node's eventOuts.

When setting a value in another node, implementations are free to either immediately set the value or to defer setting the value until the Script is finished. When getting a value from another node, the value returned shall be up-to-date; that is, it shall be the value immediately before the time of the current timestamp (the current timestamp returned is the timestamp of the event that caused the Script node to execute).

If multiple *directOutput* Scripts read from and/or write to the same node, the results are undefined.

## 4.12.6 Asynchronous scripts

Some languages supported by VRML browsers may allow Script nodes to spontaneously generate events, allowing users to create Script nodes that function like new Sensor nodes. In these cases, the Script is generating the initial events that causes the event cascade, and the scripting language and/or the browser shall determine an appropriate timestamp for that initial event. Such events are then sorted into the event stream and processed like any other event, following all of the same rules including those for looping.

## 4.12.7 Script languages

The Script node's *url* field may specify a URL which refers to a file (e.g., using protocol http:) or incorporates scripting language code directly in-line. The MIME-type of the returned data defines the language type. Additionally, instructions can be included in-line using 4.5.4, Scripting language protocol, defined for the specific language (from which the language type is inferred).

For example, the following Script node has one eventIn field named *start* and three different URL values specified in the *url* field: Java, ECMAScript, and inline ECMAScript:

```
Script {
  eventIn SFBool start
  url [ "http://foo.com/fooBar.class",
    "http://foo.com/fooBar.js",
    "javascript:function start(value, timestamp) { ... }"
  ]
}
```

In the above example when a *start* eventIn is received by the Script node, one of the scripts found in the *url* field is executed. The Java platform bytecode is the first choice, the ECMAScript code is the second choice, and the inline ECMAScript code the third choice. A description of order of preference for multiple valued URL fields may be found in 4.5.2, URLs.

## 4.12.8 EventIn handling

Events received by the Script node are passed to the appropriate scripting language method in the script. The method's name depends on the language type used. In some cases, it is identical to the name of the eventIn; in others, it is a general callback method for all eventIns (see the scripting language annexes for details). The method is passed two arguments: the event value and the event timestamp.

## 4.12.9 Accessing fields and events

The fields, eventIns, and eventOuts of a Script node are accessible from scripting language methods. Events can be routed to eventIns of Script nodes and the eventOuts of Script nodes can be routed to eventIns of other nodes. Another Script node with access to this node can access the eventIns and eventOuts just like any other node (see 4.12.5, Scripts with direct outputs).

It is recommended that user-defined field or event names defined in Script nodes follow the naming conventions described in 4.7, Field, eventIn, and eventOut semantics.

### 4.12.9.1 Accessing fields and eventOuts of the script

Fields defined in the Script node are available to the script through a language-specific mechanism (e.g., a variable is automatically defined for each field and event of the Script node). The field values can be read or written and are persistent across method calls. EventOuts defined in the Script node may also be read; the returned value is the last value sent to that eventOut.

### 4.12.9.2 Accessing eventIns and eventOuts of other nodes

The script can access any eventIn or eventOut of any node to which it has access. The syntax of this mechanism is language dependent. The following example illustrates how a Script node accesses and modifies an exposed field of another node (i.e., sends a *set_translation* eventIn to the Transform node) using ECMAScript:

```
DEF SomeNode Transform { }
Script {
  field   SFNode  tnode USE SomeNode
  eventIn SFVec3f pos
  directOutput TRUE
  url "javascript:
    function pos(value, timestamp) {
      tnode.set_translation = value;
    }"
}
```

The language-dependent mechanism for accessing eventIns or eventOuts (or the eventIn or eventOut part of an exposedField) shall support accessing them without their "*set_*" or "*_changed*" prefix or suffix, to match the ROUTE statement semantics. When accessing an eventIn named "*zzz*" and an eventIn of that name is not found, the browser shall try to access the eventIn named "*set_zzz*". Similarly, if accessing an eventOut named "*zzz*" and an eventOut of that name is not found, the browser shall try to access the eventOut named "*zzz_changed*".

### 4.12.9.3 Sending eventOuts

Each scripting language provides a mechanism for allowing scripts to send a value through an eventOut defined by the Script node. For example, one scripting language may define an explicit method for sending each eventOut, while another language may use assignment statements to automatically defined eventOut variables to implicitly send the eventOut. Sending multiple values through an eventOut during a single script execution will result in the "last" event being sent, where "last" is determined by the semantics of the scripting language being used.

## 4.12.10 Browser script interface

### 4.12.10.1 Introduction

The browser interface provides a mechanism for scripts contained by Script nodes to get and set browser state (e.g., the URL of the current world). This subclause describes the semantics of methods that the browser interface supports. An arbitrary syntax is used to define the type of parameters and returned values. The specific annex for a language contains the actual syntax required. In this abstract syntax, types are given as VRML field types. Mapping of these types into those of the underlying language (as well as any type conversion needed) is described in the appropriate language annex.

### 4.12.10.2 SFString getName( ) and SFString getVersion( )

The **getName()** and **getVersion()** methods return a string representing the "name" and "version" of the browser currently in use. These values are defined by the browser writer, and identify the browser in some (unspecified) way. They are not guaranteed to be unique or to adhere to any particular format and are for information only. If the information is unavailable these methods return empty strings.

### 4.12.10.3 SFFloat getCurrentSpeed( )

The **getCurrentSpeed()** method returns the average navigation speed for the currently bound NavigationInfo node in meters per second, in the coordinate system of the currently bound Viewpoint node. If speed of motion is not meaningful in the current navigation type, or if the speed cannot be determined for some other reason, 0.0 is returned.

### 4.12.10.4 SFFloat getCurrentFrameRate( )

The **getCurrentFrameRate()** method returns the current frame rate in frames per second. The way in which frame rate is measured and whether or not it is supported at all is browser dependent. If frame rate measurement is not supported or cannot be determined, 0.0 is returned.

### 4.12.10.5 SFString getWorldURL( )

The **getWorldURL()** method returns the URL for the root of the currently loaded world.

### 4.12.10.6 void replaceWorld( MFNode nodes )

The **replaceWorld()** method replaces the current world with the world represented by the passed nodes. An invocation of this method will usually not return since the world containing the running script is being replaced. Scripts that may call this method shall have *mustEvaluate* set to TRUE.

### 4.12.10.7 void loadURL( MFString url, MFString parameter )

The **loadURL()** method loads the first recognized URL from the specified *url* field with the passed parameters. The *parameter* and *url* arguments are treated identically to the Anchor node's *parameter* and *url* fields (see 6.2, Anchor). This method returns immediately. However, if the URL is loaded into this browser window (e.g., there is no TARGET parameter to redirect it to another frame), the current world will be terminated and replaced with the data from the specified URL at some time in the future. Scripts that may call this method shall set *mustEvaluate* to TRUE. If **loadUrl()** is invoked with a URL of the form "#name", the Viewpoint node with the given name ("name") in the Script' node's run-time name scope(s) shall be bound. However, if the Script node containing the script that invokes **loadURL("#name")** is not part of any run-time name scope or is part of more than one run-time name scope, results are undefined. See 4.4.6, Run-time name scope, for a description of run-time name scope.

### 4.12.10.8 void setDescription( SFString description )

The **setDescription()** method sets the passed string as the current description. This message is displayed in a browser dependent manner. An empty string clears the current description. Scripts that call this method shall have *mustEvaluate* set to TRUE.

### 4.12.10.9 MFNode createVrmlFromString( SFString vrmlSyntax )

The **createVrmlFromString**() method parses a string consisting of VRML statements, establishes any PROTO and EXTERNPROTO declarations and routes, and returns an MFNode value containing the set of nodes in those statements. The string shall be self-contained (i.e., USE statements inside the string may refer only to nodes DEF'ed in the string, and non-built-in node types used by the string shall be prototyped using EXTERNPROTO or PROTO statements inside the string).

### 4.12.10.10 void createVrmlFromURL( MFString url, SFNode node, SFString event )

The **createVrmlFromURL()** instructs the browser to load a VRML scene description from the given URL or URLs. The VRML file referred to shall be self-contained (i.e., USE statements inside the string may refer only to nodes DEF'ed in the string, and non-built-in node types used by the string shall be prototyped using EXTERNPROTO or PROTO statements inside the string). After the scene is loaded, *event* is sent to the passed *node* returning the root nodes of the corresponding VRML scene. The *event* parameter contains a string naming an MFNode eventIn on the passed node.

### 4.12.10.11 void addRoute(...) and void deleteRoute(...)

```
void addRoute( SFNode fromNode, SFString fromEventOut,
               SFNode toNode, SFString toEventIn );

void deleteRoute( SFNode fromNode, SFString fromEventOut,
                  SFNode toNode, SFString toEventIn );
```

These methods respectively add and delete a route between the given event names for the given nodes. Scripts that call this method shall have *directOutput* set to TRUE. Routes that are added and deleted shall obey the execution order defined in 4.10.3, Execution model.

# 4.13 Navigation

## 4.13.1 Introduction

Conceptually speaking, every VRML world contains a *viewpoint* from which the world is currently being viewed. Navigation is the action taken by the user to change the position and/or orientation of this viewpoint thereby changing the user's view. This allows the user to move through a world or examine an object. The NavigationInfo node (see 6.29, NavigationInfo) specifies the characteristics of the desired navigation behaviour, but the exact user interface is browser-dependent. The Viewpoint node (see 6.53, Viewpoint) specifies key locations and orientations in the world to which the user may be moved via scripts or browser-specific user interfaces.

## 4.13.2 Navigation paradigms

The browser may allow the user to modify the location and orientation of the viewer in the virtual world using a navigation paradigm. Many different navigation paradigms are possible, depending on the nature of the virtual world and the task the user wishes to perform. For instance, a walking paradigm would be appropriate in an architectural walkthrough application, while a flying paradigm might be better in an application exploring interstellar space. Examination is another common use for VRML, where the world is considered to be a single object which the user wishes to view from many angles and distances.

The NavigationInfo node has a *type* field that specifies the navigation paradigm for this world. The actual user interface provided to accomplish this navigation is browser-dependent. See 6.29, NavigationInfo, for details.

## 4.13.3 Viewing model

The browser controls the location and orientation of the viewer in the world, based on input from the user (using the browser-provided navigation paradigm) and the motion of the currently bound Viewpoint node (and its coordinate system). The VRML author can place any number of viewpoints in the world at important places from which the user might wish to view the world. Each viewpoint is described by a Viewpoint node. Viewpoint nodes exist in their parent's coordinate system, and both the viewpoint and the coordinate system may be changed to affect the view of the world presented by the browser. Only one viewpoint is bound at a time. A detailed description of how the Viewpoint node operates is described in 4.6.10, Bindable children nodes, and 6.53, Viewpoint.

Navigation is performed relative to the Viewpoint's location and does not affect the location and orientation values of a Viewpoint node. The location of the viewer may be determined with a ProximitySensor node (see 6.38, ProximitySensor).

## 4.13.4 Collision detection and terrain following

A VRML file can contain Collision nodes (see 6.8, Collision) and NavigationInfo nodes that influence the browser's navigation paradigm. The browser is responsible for detecting collisions between the viewer and the objects in the virtual world, and is also responsible for adjusting the viewer's location when a collision occurs. Browsers shall not disable collision detection except for the special cases listed below. Collision nodes can be used to generate events when viewer and objects collide, and can be used to designate that certain objects should be treated as transparent to collisions. Support for inter-object collision is not specified. The NavigationInfo types of WALK, FLY, and NONE shall strictly support collision detection. However, the NavigationInfo types ANY and EXAMINE may temporarily disable collision detection during navigation, but shall not disable collision detection during the normal execution of the world. See 6.29, NavigationInfo, for details on the various navigation types.

NavigationInfo nodes can be used to specify certain parameters often used by browser navigation paradigms. The size and shape of the viewer's avatar determines how close the avatar may be to an object before a collision is considered to take place. These parameters can also be used to implement *terrain following* by keeping the avatar a certain distance above the ground. They can additionally be used to determine how short an object must be for the viewer to automatically step up onto it instead of colliding with it.

# 4.14 Lighting model

## 4.14.1 Introduction

The VRML lighting model provides detailed equations which define the colours to apply to each geometric object. For each object, the values of the Material node, Color node and texture currently being applied to the object are combined with the lights illuminating the object and the currently bound Fog node. These equations are designed to simulate the physical properties of light striking a surface.

## 4.14.2 Lighting ’off’

A Shape node is unlit if either of the following is true:

a. The shape's *appearance* field is NULL (default).

b. The *material* field in the Appearance node is NULL (default).

×Note the special cases of geometry nodes that do not support lighting (see 6.24, IndexedLineSet, and 6.36, PointSet, for details).

If the shape is unlit, the colour ($I_{rgb}$) and alpha (A, 1-transparency) of the shape at each point on the shape's geometry is given in Table 4.5.

**Table 4.5 -- Unlit colour and alpha mapping**

| Texture type | Colour per-vertex or per-face | Colour NULL |
|---|---|---|
| No texture | $I_{rgb}= I_{Crgb}$<br>$A = 1$ | $I_{rgb}= (1, 1, 1)$<br>$A = 1$ |
| Intensity<br>(one-component) | $I_{rgb}= I_T \times I_{Crgb}$<br>$A = 1$ | $I_{rgb} = (I_T,I_T,I_T )$<br>$A = 1$ |
| Intensity+Alpha<br>(two-component) | $I_{rgb}= I_T \times I_{Crgb}$<br>$A = A_T$ | $I_{rgb}= (I_T,I_T,I_T )$<br>$A = A_T$ |
| RGB<br>(three-component) | $I_{rgb}= I_{Trgb}$<br>$A = 1$ | $I_{rgb}= I_{Trgb}$<br>$A = 1$ |
| RGBA<br>(four-component) | $I_{rgb}= I_{Trgb}$<br>$A = A_T$ | $I_{rgb}= I_{Trgb}$<br>$A = A_T$ |

where:

$A_T$ = normalized [0, 1] alpha value from 2 or 4 component texture image
$I_{Crgb}$ = interpolated per-vertex colour, or per-face colour, from Color node
$I_T$ = normalized [0, 1] intensity from 1 or 2 component texture image
$I_{Trgb}$= colour from 3-4 component texture image

## 4.14.3 Lighting ’on’

If the shape is lit (i.e., a Material and an Appearance node are specified for the Shape), the Color and Texture nodes determine the diffuse colour for the lighting equation as specified in Table 4.6.

**Table 4.6 -- Lit colour and alpha mapping**

| Texture type | Colour per-vertex or per-face | Color node NULL |
|---|---|---|
| No texture | $O_{Drgb} = I_{Crgb}$<br>$A = 1\text{-}T_M$ | $O_{Drgb} = I_{Drgb}$<br>$A = 1\text{-}T_M$ |
| Intensity texture (one-component) | $O_{Drgb} = I_T \times I_{Crgb}$<br>$A = 1\text{-}T_M$ | $O_{Drgb} = I_T \times I_{Drgb}$<br>$A = 1\text{-}T_M$ |
| Intensity+Alpha texture (two-component) | $O_{Drgb} = I_T \times I_{Crgb}$<br>$A = A_T$ | $O_{Drgb} = I_T \times I_{Drgb}$<br>$A = A_T$ |
| RGB texture (three-component) | $O_{Drgb} = I_{Trgb}$<br>$A = 1\text{-}T_M$ | $O_{Drgb} = I_{Trgb}$<br>$A = 1\text{-}T_M$ |
| RGBA texture (four-component) | $O_{Drgb} = I_{Trgb}$<br>$A = A_T$ | $O_{Drgb} = I_{Trgb}$<br>$A = A_T$ |

where:

$I_{Drgb}$ = material *diffuseColor*
$O_{Drgb}$ = diffuse factor, used in lighting equations below
$T_M$ = material *transparency*

All other terms are as defined in 4.14.2, Lighting `off'.

## 4.14.4 Lighting equations

An ideal VRML implementation will evaluate the following lighting equation at each point on a lit surface. RGB intensities at each point on a geometry ($I_{rgb}$) are given by:

$$I_{rgb} = I_{Frgb} \times (1 - f_0) + f_0 \times (O_{Ergb} + \text{SUM}( \text{on}_i \times \text{attenuation}_i \times \text{spot}_i \times I_{Lrgb} \\ \times (\text{ambient}_i + \text{diffuse}_i + \text{specular}_i)))$$

where:

$\text{attenuation}_i = 1 / \max(c_1 + c_2 \times d_L + c_3 \times d_L^{\&sup2;} , 1 )$

$\text{ambient}_i = I_{ia} \times O_{Drgb} \times O_a$

$\text{diffuse}_i = I_i \times O_{Drgb} \times ( \mathbf{N} \bullet \mathbf{L} )$

$\text{specular}_i = I_i \times O_{Srgb} \times ( \mathbf{N} \bullet ((\mathbf{L} + \mathbf{V}) / |\mathbf{L} + \mathbf{V}|))^{\text{shininess} \times 128}$

and:

$\bullet$ *= modified vector dot product: if dot product < 0, then 0.0, otherwise, dot product*

$c_1$ , $c_2$, $c_3$ *= light i* attenuation

$d_V$ *= distance from point on geometry to viewer's position, in coordinate system of current fog node*

$d_L$ *= distance from light to point on geometry, in light's coordinate system*

$f_0$ *= Fog interpolant, see* Table 4.8 *for calculation*

$I_{Frgb}$ *= currently bound fog's* color

$I_{Lrgb}$ *= light i* color

$I_i$ *= light i* intensity

$I_{ia}$ *= light i* ambientIntensity

$\mathbf{L}$ *= (Point/SpotLight) normalized vector from point on geometry to light source i position*

$\mathbf{L}$ *= (DirectionalLight) -direction of light source i*

$\mathbf{N}$ *= normalized normal vector at this point on geometry (interpolated from vertex normals specified in Normal node or calculated by browser)*

$O_a$ *= Material* ambientIntensity

$O_{Drgb}$ *= diffuse colour, from Material node, Color node, and/or texture node*

$O_{Ergb}$ *= Material* emissiveColor

$O_{Srgb}$ *= Material* specularColor

$on_i$ *= 1, if light source i affects this point on the geometry,*

> *0, if light source i does not affect this geometry (if farther away than* radius *for PointLight or SpotLight, outside of enclosing Group/Transform for DirectionalLights, or* on *field is FALSE)*

*shininess = Material* shininess

*spotAngle = acos(* $-\mathbf{L} \bullet \mathbf{spotDir}_i$*)*

$spot_{BW}$ *= SpotLight i beamWidth*

$spot_{CO}$ *= SpotLight i* cutOffAngle

$spot_i$ *= spotlight factor, see* Table 4.7 *for calculation*

$\mathbf{spotDir}_i$ *= normalized SpotLight i* direction

*SUM: sum over all light sources i*

$\mathbf{V}$ *= normalized vector from point on geometry to viewer's position*

**Table 4.7 -- Calculation of the spotlight factor**

| ***Condition (in order)*** | **$spot_i$ =** |
|---|---|
| $light_i$ is PointLight or DirectionalLight | 1 |
| spotAngle >= $spot_{CO}$ | 0 |
| spotAngle <= $spot_{BW}$ | 1 |
| $spot_{BW}$ < spotAngle < $spot_{CO}$ | (spotAngle - $spot_{CO}$ ) / ($spot_{BW}$-$spot_{CO}$) |

**Table 4.8 -- Calculation of the fog interpolant**

| Condition | $f_0$ = |
|---|---|
| no fog | 1 |
| fogType "LINEAR", $d_V$ < fogVisibility | (fogVisibility-$d_V$) / fogVisibility |
| fogType "LINEAR", $d_V \geq$ fogVisibility | 0 |
| fogType "EXPONENTIAL", $d_V$ < fogVisibility | exp(-$d_V$ / (fogVisibility-$d_V$ ) ) |
| fogType "EXPONENTIAL", $d_V \geq$ fogVisibility | 0 |

## 4.14.5 References

*The VRML lighting equations are based on the simple illumination equations given in E.[FOLE] and E.[OPEN].*

# 5 Field and event reference

## 5.1 Introduction

### 5.1.1 Table of contents

### 5.1.2 Description

This clause describes the syntax and general semantics of *fields* and *events*, the elemental data types used by VRML nodes to define objects (see 6, Node reference). Nodes are composed of fields and events (see 4, Concepts). The types defined in this annex are used by both fields and events.

There are two general classes of fields and events: fields and events that contain a single value (where a value may be a single number, a vector, or even an image), and fields and events that contain an ordered list of multiple values. Single-valued fields and events have names that begin with **SF.** Multiple-valued fields and events have names that begin with **MF**.

Multiple-valued fields/events are written as an ordered list of values enclosed in square brackets and separated by whitespace. If the field or event has zero values, only the square brackets ("[ ]") are written. The last value may optionally be followed by whitespace. If the field has exactly one value, the brackets may be omitted. For example, all of the following are valid for a multiple-valued MFInt32 field named *foo* containing the single integer value 1:

```
    foo 1
    foo [1,]
    foo [ 1 ]
```

## 5.2 SFBool

*The SFBool field or event specifies a single boolean value. SFBools are written as TRUE or FALSE. For example,*

```
fooBool FALSE
```

is an SFBool field, *fooBool*, defining a FALSE value.

The initial value of an SFBool eventOut is FALSE.

## 5.3 SFColor and MFColor

The SFColor field or event specifies one RGB (red-green-blue) colour triple. MFColor specifies zero or more RGB triples. Each colour is written to the VRML file as an RGB triple of floating point numbers in ISO C floating point format (see 2.[ISOC]) in the range 0.0 to 1.0. For example:

```
fooColor [ 1.0 0. 0.0, 0 1 0, 0 0 1 ]
```

is an MFColor field, *fooColor*, containing the three primary colours red, green, and blue.

The initial value of an SFColor eventOut is (0 0 0). The initial value of an MFColor eventOut is [ ].

## 5.4 SFFloat and MFFloat

The SFFloat field or event specifies one single-precision floating point number. MFFloat specifies zero or more single-precision floating point numbers. SFFloats and MFFloats are written to the VRML file in ISO C floating point format (see 2.[ISOC]). For example:

```
fooFloat [ 3.1415926, 12.5e-3, .0001 ]
```

is an MFFloat field, *fooFloat*, containing three floating point values.

The initial value of an SFFloat eventOut is 0.0. The initial value of an MFFloat eventOut is [ ].

## 5.5 SFImage

The SFImage field or event specifies a single uncompressed 2-dimensional pixel image. SFImage fields and events are written to the VRML file as three integers representing the width, height and number of components in the image, followed by width*height hexadecimal or integer values representing the pixels in the image, separated by whitespace:

```
fooImage <width> <height> <num components> <pixels values>
```

Pixel values are limited to 256 levels of intensity (i.e., 0-255 decimal or 0x00-0xFF hexadecimal). A one-component image specifies one-byte hexadecimal or integer values representing the intensity of the image. For example, `0xFF` is full intensity in hexadecimal (255 in decimal), `0x00` is no intensity (0 in decimal). A two-component image specifies the intensity in the first (high) byte and the alpha opacity in the second (low) byte. Pixels in a three-component image specify the red component in the first (high) byte, followed by the green and blue components (e.g., `0xFF0000` is red, `0x00FF00` is green, `0x0000FF` is blue). Four-component images specify the alpha opacity byte after red/green/blue (e.g., `0x0000FF80` is semi-transparent blue). A value of `0x00` is completely transparent, 0xFF is completely opaque. Note that alpha equals (1.0 - transparency), if alpha and transparency range from 0.0 to 1.0.

Each pixel is read as a single unsigned number. For example, a 3-component pixel with value `0x0000FF` may also be written as `0xFF` (hexadecimal) or `255` (decimal). Pixels are specified from left to right, bottom to top. The first hexadecimal value is the lower left pixel and the last value is the upper right pixel.

For example,

```
fooImage 1 2 1 0xFF 0x00
```

is a 1 pixel wide by 2 pixel high one-component (i.e., greyscale) image, with the bottom pixel white and the top pixel black. As another example,

```
fooImage 2 4 3 0xFF0000 0xFF00 0 0 0 0 0xFFFFFF 0xFFFF00
                # red    green  black.. white    yellow
```

is a 2 pixel wide by 4 pixel high RGB image, with the bottom left pixel red, the bottom right pixel green, the two middle rows of pixels black, the top left pixel white, and the top right pixel yellow.

The initial value of an SFImage eventOut is (0 0 0).

## 5.6 SFInt32 and MFInt32

The SFInt32 field and event specifies one 32-bit integer. The MFInt32 field and event specifies zero or more 32-bit integers. SFInt32 and MFInt32 fields and events are written to the VRML file as an integer in decimal or hexadecimal (beginning with '0x') format. For example:

```
fooInt32 [ 17, -0xE20, -518820 ]
```

is an MFInt32 field containing three values.

The initial value of an SFInt32 eventOut is 0. The initial value of an MFInt32 eventOut is [ ].

## 5.7 SFNode and MFNode

The SFNode field and event specifies a VRML node. The MFNode field and event specifies zero or more nodes. The following example illustrates valid syntax for an MFNode field, *fooNode*, defining four nodes:

```
       fooNode [ Transform { translation 1 0 0 }
                 DEF CUBE Box { }
                 USE CUBE
                 USE SOME_OTHER_NODE  ]
```

The SFNode field and event may contain the keyword NULL to indicate that it is empty.

The initial value of an SFNode eventOut is NULL. The initial value of an MFNode eventOut is [ ].

## 5.8 SFRotation and MFRotation

The SFRotation field and event specifies one arbitrary rotation. The MFRotation field and event specifies zero or more arbitrary rotations. An SFRotation is written to the VRML file as four ISO C floating point values (see 2.[ISOC]) separated by whitespace. The first three values specify a normalized rotation axis vector about which the rotation takes place. The fourth value specifies the amount of right-handed rotation about that axis in radians. For example, an SFRotation containing a PI radians rotation about the Y axis is:

```
   fooRot 0.0 1.0 0.0 3.14159265
```

The 3x3 matrix representation of a rotation (x y z a) is

$$\left[\begin{matrix} tx^2+c & txy+sz & txz-sy \\ txy-sz & ty^2+c & tyz+sx \\ txz+sy & tyz-sx & tz^2+c \end{matrix}\right]$$

```
    where c = cos(a), s = sin(a), and t = 1-c
```

The initial value of an SFRotation eventOut is (0 0 1 0). The initial value of an MFRotation eventOut is [ ].

## 5.9 SFString and MFString

The SFString and MFString fields and events contain strings formatted with the UTF-8 universal character set (see 2.[UTF8]). SFString specifies a single string. The MFString specifies zero or more strings. Strings are written to the VRML file as a sequence of UTF-8 octets enclosed in double quotes (e.g., **"string"**).

Any characters (including linefeeds and '#') may appear within the quotes. A double quote character within the string is preceded with a backslash. A backslash character within the string is also preceded with a backslash forming two backslashes. For example:

```
   fooString [ "One, Two, Three", "He said, \"Immel did it!\"" ]
```

is an MFString field, *fooString*, with two valid strings.

The initial value of an SFString eventOut is "" (the empty string). The initial value of an MFString eventOut is [ ].

APPENDIX KK

# 5.10 SFTime and MFTime

The SFTime field or event specifies a single time value. The MFTime field or event specifies zero or more time values. Time values are written to the VRML file as a double-precision floating point number in ISO C floating point format (see 2.[ISOC]). Time values are specified as the number of seconds from a specific time origin. Typically, SFTime fields and events represent the number of seconds since Jan 1, 1970, 00:00:00 GMT. For example:

```
    fooTime 0.0
```

is an SFTime field, *fooTime*, representing a time of 0.0 seconds.

The initial value of an SFTime eventOut is -1. The initial value of an MFTime eventOut is [ ].

# 5.11 SFVec2f and MFVec2f

The SFVec2f field or event specifies a two-dimensional (2D) vector. An MFVec2f field or event specifies zero or more 2D vectors. SFVec2f's and MFVec2f's are written to the VRML file as a pair of ISO C floating point values (see 2.[ISOC]) separated by whitespace. For example:

```
    fooVec2f [ 42 666, 7 94 ]
```

is an MFVec2f field, *fooVec2f*, with two valid vectors.

The initial value of an SFVec2f eventOut is (0 0). The initial value of an MFVec2f eventOut is [ ].

# 5.12 SFVec3f and MFVec3f

The SFVec3f field or event specifies a three-dimensional (3D) vector. An MFVec3f field or event specifies zero or more 3D vectors. SFVec3f's and MFVec3f's are written to the VRML file as three ISO C floating point values (see 2.[ISOC]) separated by whitespace. For example:

```
    fooVec3f [ 1 42 666, 7 94 0 ]
```

is an MFVec3f field, *fooVec3f*, with two valid vectors.

The initial value of an SFVec3f eventOut is (0 0 0). The initial value of an MFVec3f eventOut is [ ].

APPENDIX KK

# 6 Node reference

## 6.1 Introduction

This clause provides a detailed definition of the syntax and semantics of each node in this part of ISO/IEC 14772. Table 6.1 lists the topics in this clause.

**Table 6.1 -- Table of contents**

In this clause, the first item in each subclause presents the public declaration for the node. This syntax is not the actual UTF-8 encoding syntax. The parts of the interface that are identical to the UTF-8 encoding syntax are in **bold**. The node declaration defines the names and types of the fields and events for the node, as well as the default values for the fields.

The node declarations also include value ranges for the node's fields and exposedFields (where appropriate). Parentheses imply that the range bound is exclusive, while brackets imply that the range value is inclusive. For example, a range of (-∞, 1] defines the lower bound as -∞ exclusively and the upper bound as 1 inclusively.

For example, the following defines the Collision node declaration:

```
Collision {
      eventIn      MFNode   addChildren
      eventIn      MFNode   removeChildren
      exposedField MFNode   children     []
      exposedField SFBool   collide      TRUE
      field        SFVec3f  bboxCenter   0 0 0     # (-∞,∞)
      field        SFVec3f  bboxSize     -1 -1 -1  # (0,∞) or -1,-1,-1
      field        SFNode   proxy        NULL
      eventOut     SFTime   collideTime
    }
```

APPENDIX KK

The fields and events contained within the node declarations are ordered as follows:

e. eventIns, in alphabetical order;

f. exposedFields, in alphabetical order;

g. fields, in alphabetical order;

h. eventOuts, in alphabetical order.

## 6.2 Anchor

```
Anchor {
  eventIn      MFNode   addChildren
  eventIn      MFNode   removeChildren
  exposedField MFNode   children        []
  exposedField SFString description     ""
  exposedField MFString parameter       []
  exposedField MFString url             []
  field        SFVec3f  bboxCenter      0 0 0     # (-∞,∞)
  field        SFVec3f  bboxSize        -1 -1 -1  # (0,∞) or -1,-1,-1
}
```

The Anchor grouping node retrieves the content of a URL when the user activates (e.g., clicks) some geometry contained within the Anchor node's children. If the URL points to a valid VRML file, that world replaces the world of which the Anchor node is a part (except when the *parameter* field, described below, alters this behaviour). If non-VRML data is retrieved, the browser shall determine how to handle that data; typically, it will be passed to an appropriate non-VRML browser.

Exactly how a user activates geometry contained by the Anchor node depends on the pointing device and is determined by the VRML browser. Typically, clicking with the pointing device will result in the new scene replacing the current scene. An Anchor node with an empty *url* does nothing when its children are chosen. A description of how multiple Anchors and pointing-device sensors are resolved on activation is contained in 4.6.7, Sensor nodes.

More details on the *children*, *addChildren*, and *removeChildren* fields and eventIns can be found in 4.6.5, Grouping and children nodes.

The *description* field in the Anchor node specifies a textual description of the Anchor node. This may be used by browser-specific user interfaces that wish to present users with more detailed information about the Anchor.

The *parameter* exposed field may be used to supply any additional information to be interpreted by the browser. Each string shall consist of "keyword=value" pairs. For example, some browsers allow the specification of a 'target' for a link to display a link in another part of an HTML document. The *parameter* field is then:

```
Anchor {
  parameter [ "target=name_of_frame" ]
  ...
}
```

An Anchor node may be used to bind the initial Viewpoint node in a world by specifying a URL ending with "#ViewpointName" where "ViewpointName" is the name of a viewpoint defined in the VRML file. For example:

```
Anchor {
  url "http://www.school.edu/vrml/someScene.wrl#OverView"
  children  Shape { geometry Box {} }
}
```

specifies an anchor that loads the VRML file "someScene.wrl" and binds the initial user view to the Viewpoint node named "OverView" when the Anchor node's geometry (Box) is activated. If the named Viewpoint node is not found in the VRML file, the VRML file is loaded using the default Viewpoint node binding stack rules (see 6.53, Viewpoint).

If the *url* field is specified in the form "#ViewpointName" (i.e. no file name), the Viewpoint node with the given name ("ViewpointName") in the Anchor's run-time name scope(s) shall be bound (*set_bind* TRUE). The results are undefined if there are multiple Viewpoints with the same name in the Anchor's run-time name scope(s). The results are undefined if the Anchor node is not part of any run-time name scope or is part of more than one run-time name scope. See 4.4.6, Run-time name scope, for a description of run-time name scopes. See 6.53, Viewpoint, for the Viewpoint transition rules that specify how browsers shall interpret the transition from the old Viewpoint node to the new one. For example:

```
Anchor {
  url "#Doorway"
  children Shape { geometry Sphere {} }
}
```

binds the viewer to the viewpoint defined by the "Doorway" viewpoint in the current world when the sphere is activated. In this case, if the Viewpoint is not found, no action occurs on activation.

More details on the *url* field are contained in 4.5, VRML and the World Wide Web.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Anchor's children. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. The default *bboxSize* value, (-1, -1, -1), implies that the bounding box is not specified and if needed shall be calculated by the browser. More details on the *bboxCenter* and *bboxSize* fields can be found in 4.6.4, Bounding boxes.

## 6.3 Appearance

```
Appearance {
  exposedField SFNode material          NULL
  exposedField SFNode texture           NULL
  exposedField SFNode textureTransform  NULL
}
```

The Appearance node specifies the visual properties of geometry. The value for each of the fields in this node may be NULL. However, if the field is non-NULL, it shall contain one node of the appropriate type.

The *material* field, if specified, shall contain a Material node. If the *material* field is NULL or unspecified, lighting is off (all lights are ignored during rendering of the object that references this Appearance) and the unlit object colour is (1, 1, 1). Details of the VRML lighting model are in 4.14, Lighting model.

The *texture* field, if specified, shall contain one of the various types of texture nodes (ImageTexture, MovieTexture, or PixelTexture). If the texture node is NULL or the *texture* field is unspecified, the object that references this Appearance is not textured.

The *textureTransform* field, if specified, shall contain a TextureTransform node. If the *textureTransform* is NULL or unspecified, the *textureTransform* field has no effect.

# 6.4 AudioClip

```
AudioClip {
  exposedField   SFString description      ""
  exposedField   SFBool   loop             FALSE
  exposedField   SFFloat  pitch            1.0        # (0,∞)
  exposedField   SFTime   startTime        0          # (-∞,∞)
  exposedField   SFTime   stopTime         0          # (-∞,∞)
  exposedField   MFString url              []
  eventOut       SFTime   duration_changed
  eventOut       SFBool   isActive
}
```

An AudioClip node specifies audio data that can be referenced by Sound nodes.

The *description* field specifies a textual description of the audio source. A browser is not required to display the *description* field but may choose to do so in addition to playing the sound.

The *url* field specifies the URL from which the sound is loaded. Browsers shall support at least the *wavefile* format in uncompressed PCM format (see E.[WAV]). It is recommended that browsers also support the MIDI file type 1 sound format (see 2.[MIDI]); MIDI files are presumed to use the General MIDI patch set. Subclause 4.5, VRML and the World Wide Web, contains details on the *url* field. The results are undefined when no URLs refer to supported data types

The *loop, startTime,* and *stopTime* exposedFields and the *isActive* eventOut, and their effects on the AudioClip node, are discussed in detail in 4.6.9, Time-dependent nodes. The "*cycle"* of an AudioClip is the length of time in seconds for one playing of the audio at the specified *pitch*.

The *pitch* field specifies a multiplier for the rate at which sampled sound is played. Values for the *pitch* field shall be greater than zero. Changing the *pitch* field affects both the pitch and playback speed of a sound. A *set_pitch* event to an active AudioClip is ignored and no *pitch_changed* eventOut is generated. If *pitch* is set to 2.0, the sound shall be played one octave higher than normal and played twice as fast. For a sampled sound, the *pitch* field alters the sampling rate at which the sound is played. The proper implementation of pitch control for MIDI (or other note sequence sound clips) is to multiply the tempo of the playback by the *pitch* value and adjust the MIDI Coarse Tune and Fine Tune controls to achieve the proper pitch change.

A *duration_changed* event is sent whenever there is a new value for the "normal" duration of the clip. Typically, this will only occur when the current *url* in use changes and the sound data has been loaded, indicating that the clip is playing a different sound source. The duration is the length of time in seconds for one cycle of the audio for a *pitch* set to 1.0. Changing the *pitch* field will not trigger a *duration_changed* event. A duration value of "-1" implies that the sound data has not yet loaded or the value is unavailable for some reason. A *duration_changed* event shall be generated if the AudioClip node is loaded when the VRML file is read or the AudioClip node is added to the scene graph.

The *isActive* eventOut may be used by other nodes to determine if the clip is currently active. If an AudioClip is active, it shall be playing the sound corresponding to the sound time (i.e., in the sound's local time system with sample 0 at time 0):

```
    t = (now - startTime) modulo (duration / pitch)
```

## 6.5 Background

```
Background {
  eventIn      SFBool   set_bind
  exposedField MFFloat  groundAngle  []         # [0,π/2]
  exposedField MFColor  groundColor  []         # [0,1]
  exposedField MFString backUrl      []
  exposedField MFString bottomUrl    []
  exposedField MFString frontUrl     []
  exposedField MFString leftUrl      []
  exposedField MFString rightUrl     []
  exposedField MFString topUrl       []
  exposedField MFFloat  skyAngle     []         # [0,π]
  exposedField MFColor  skyColor     0 0 0      # [0,1]
  eventOut     SFBool   isBound
}
```

The Background node is used to specify a colour backdrop that simulates ground and sky, as well as a background texture, or *panorama*, that is placed behind all geometry in the scene and in front of the ground and sky. Background nodes are specified in the local coordinate system and are affected by the accumulated rotation of their ancestors as described below.

Background nodes are bindable nodes as described in 4.6.10, Bindable children nodes. There exists a Background stack, in which the top-most Background on the stack is the currently active Background. To move a Background to the top of the stack, a TRUE value is sent to the *set_bind* eventIn. Once active, the Background is then bound to the browsers view. A FALSE value sent to *set_bind* removes the Background from the stack and unbinds it from the browser's view. More detail on the bind stack is described in 4.6.10, Bindable children nodes.

The backdrop is conceptually a partial sphere (the ground) enclosed inside of a full sphere (the sky) in the local coordinate system with the viewer placed at the centre of the spheres. Both spheres have infinite radius and each is painted with concentric circles of interpolated colour perpendicular to the local Y-axis of the sphere. The Background node is subject to the accumulated rotations of its ancestors' transformations. Scaling and translation transformations are ignored. The sky sphere is always slightly farther away from the viewer than the ground partial sphere causing the ground to appear in front of the sky where they overlap.

The *skyColor* field specifies the colour of the sky at various angles on the sky sphere. The first value of the *skyColor* field specifies the colour of the sky at 0.0 radians representing the zenith (i.e., straight up from the viewer). The *skyAngle* field specifies the angles from the zenith in which concentric circles of colour appear. The zenith of the sphere is implicitly defined to be 0.0 radians, the natural horizon is at π/2 radians, and the nadir (i.e., straight down from the viewer) is at π radians. *skyAngle* is restricted to non-decreasing values in the range [0.0,π]. There shall be one more *skyColor* value than there are *skyAngle* values. The first colour value is the colour at the zenith, which is not specified in the *skyAngle* field. If the last *skyAngle* is less than *pi*, then the colour band between the last *skyAngle* and the nadir is clamped to the last *skyColor*. The sky colour is linearly interpolated between the specified *skyColor* values.

The *groundColor* field specifies the colour of the ground at the various angles on the ground partial sphere. The first value of the *groundColor* field specifies the colour of the ground at 0.0 radians representing the nadir (i.e., straight down from the user). The *groundAngle* field specifies the angles from the nadir that the concentric circles of colour appear. The nadir of the sphere is implicitly defined at 0.0 radians. *groundAngle* is restricted to non-decreasing values in the range [0.0, π/2]. There shall be one more *groundColor* value than there are *groundAngle* values. The first colour value is for the nadir which is not specified in the *groundAngle* field. If the last *groundAngle* is less than π/2, the region between the last *groundAngle* and the equator is non-existant. The ground colour is linearly interpolated between the specified *groundColor* values.

The *backUrl*, *bottomUrl*, *frontUrl*, *leftUrl*, *rightUrl*, and *topUrl* fields specify a set of images that define a background panorama between the ground/sky backdrop and the scene's geometry. The panorama consists of six images, each of which is mapped onto a face of an infinitely large cube contained within the backdrop spheres and centred in the local coordinate system. The images are applied individually to each face of the cube. On the front, back, right, and left faces of the cube, when viewed from the origin looking down the negative Z-axis with the Y-axis as the view up direction, each image is mapped onto the corresponding face with the same orientation as if the image were displayed normally in 2D (*backUrl* to back face, *frontUrl* to front face, *leftUrl* to left face, and *rightUrl* to right face*)*. On the top face of the cube, when viewed from the origin looking along the +Y-axis with the +Z-axis as the view up direction, the *topUrl* image is mapped onto the face with the same orientation as if the image were displayed normally in 2D. On the bottom face of the box, when viewed from the origin along the negative Y-axis with the negative Z-axis as the view up direction, the *bottomUrl* image is mapped onto the face with the same orientation as if the image were displayed normally in 2D.

Figure 6.1 illustrates the Background node backdrop and background textures.

Alpha values in the panorama images (i.e., two or four component images) specify that the panorama is semi-transparent or transparent in regions, allowing the *groundColor* and *skyColor* to be visible.

See 4.6.11, Texture maps, for a general description of texture maps.

Often, the *bottomUrl* and *topUrl* images will not be specified, to allow sky and ground to show. The other four images may depict surrounding mountains or other distant scenery. Browsers shall support the JPEG (see 2.[JPEG]) and PNG (see 2.[PNG]) image file formats, and in addition, may support any other image format (e.g., CGM) that can be rendered into a 2D image. Support for the GIF (see E.[GIF]) format is recommended (including transparency) . More detail on the *url* fields can be found in 4.5, VRML and the World Wide Web.

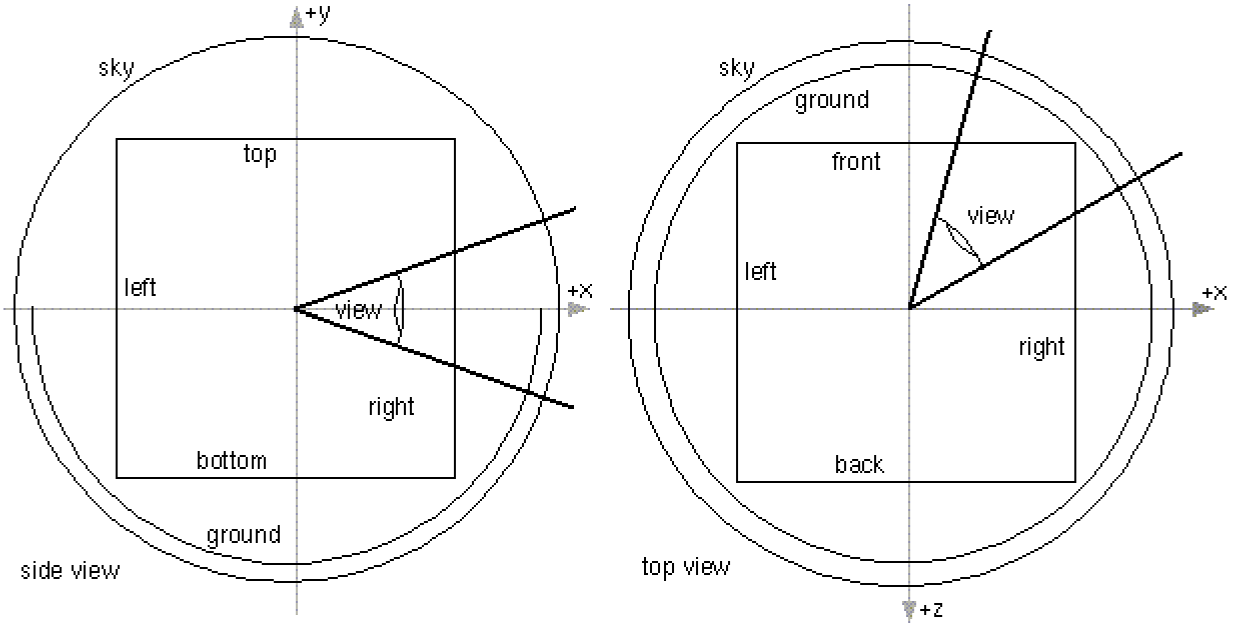


**Figure 6.1 -- Background node**

Panorama images may be one component (greyscale), two component (greyscale plus alpha), three component (full RGB colour), or four-component (full RGB colour plus alpha).

Ground colours, sky colours, and panoramic images do not translate with respect to the viewer, though they do rotate with respect to the viewer. That is, the viewer can never get any closer to the background, but can turn to examine all sides of the panorama cube, and can look up and down to see the concentric rings of ground and sky (if visible).

Background nodes are not affected by Fog nodes. Therefore, if a Background node is active (i.e., bound) while a Fog node is active, then the Background node will be displayed with no fogging effects. It is the author's responsibility to set the Background values to match the Fog values (e.g., ground colours fade to fog colour with distance and panorama images tinted with fog colour). Background nodes are not affected by light sources.

## 6.6 Billboard

```
Billboard {
  eventIn      MFNode   addChildren
  eventIn      MFNode   removeChildren
  exposedField SFVec3f  axisOfRotation 0 1 0     # (-∞,∞)
  exposedField MFNode   children       []
  field        SFVec3f  bboxCenter     0 0 0     # (-∞,∞)
  field        SFVec3f  bboxSize       -1 -1 -1  # (0,∞) or -1,-1,-1
}
```

The Billboard node is a grouping node which modifies its coordinate system so that the Billboard node's local Z-axis turns to point at the viewer. The Billboard node has children which may be other children nodes.

The *axisOfRotation* field specifies which axis to use to perform the rotation. This axis is defined in the local coordinate system.

When the *axisOfRotation* field is not (0, 0, 0), the following steps describe how to rotate the billboard to face the viewer:

a. Compute the vector from the Billboard node's origin to the viewer's position. This vector is called the *billboard-to-viewer* vector.

b. Compute the plane defined by the *axisOfRotation* and the billboard-to-viewer vector.

c. Rotate the local Z-axis of the billboard into the plane from b., pivoting around the *axisOfRotation*.

When the axisOfRotation field is set to (0, 0, 0), the special case of *viewer-alignment* is indicated. In this case, the object rotates to keep the billboard's local Y-axis parallel with the Y-axis of the viewer. This special case is distinguished by setting the *axisOfRotation* to (0, 0, 0). The following steps describe how to align the billboard's Y-axis to the Y-axis of the viewer:

d. Compute the billboard-to-viewer vector.

e. Rotate the Z-axis of the billboard to be collinear with the billboard-to-viewer vector and pointing towards the viewer's position.

f. Rotate the Y-axis of the billboard to be parallel and oriented in the same direction as the Y-axis of the viewer.

If the *axisOfRotation* and the billboard-to-viewer line are coincident, the plane cannot be established and the resulting rotation of the billboard is undefined. For example, if the *axisOfRotation* is set to (0,1,0) (Y-axis) and the viewer flies over the billboard and peers directly down the Y-axis, the results are undefined**.**

Multiple instances of Billboard nodes (DEF/USE) operate as expected: each instance rotates in its unique coordinate system to face the viewer.

Subclause 4.6.5, Grouping and children nodes, provides a description of the *children*, *addChildren*, and *removeChildren* fields and eventIns.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Billboard node's children. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. A default *bboxSize* value, (-1, -1, -1), implies that the bounding box is not specified and if needed shall be calculated by the browser. A description of the *bboxCenter* and *bboxSize* fields is contained in 4.6.4, Bounding boxes.

## 6.7 Box

```
Box {
  field    SFVec3f size  2 2 2        # (0,∞)
}
```

The Box node specifies a rectangular parallelepiped box centred at (0, 0, 0) in the local coordinate system and aligned with the local coordinate axes. By default, the box measures 2 units in each dimension, from -1 to +1. The *size* field specifies the extents of the box along the X-, Y-, and Z-axes respectively and each component value shall be greater than zero. Figure 6.2 illustrates the Box node.

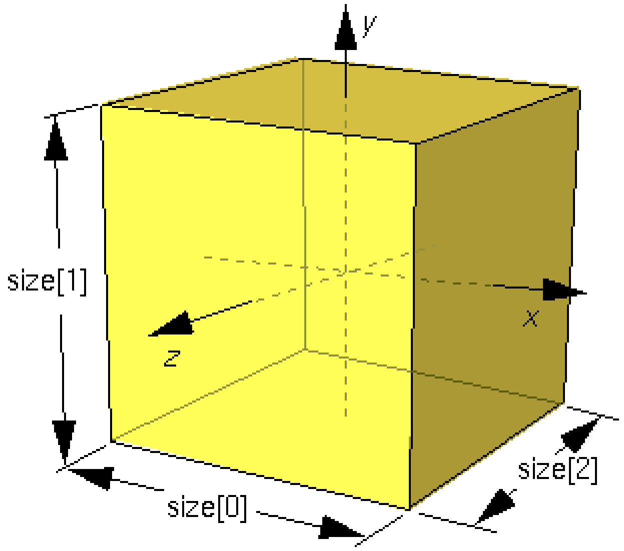


**Figure 6.2 -- Box node**

Textures are applied individually to each face of the box. On the front (+Z), back (-Z), right (+X), and left (-X) faces of the box, when viewed from the outside with the +Y-axis up, the texture is mapped onto each face with the same orientation as if the image were displayed normally in 2D. On the top face of the box (+Y), when viewed from above and looking down the Y-axis toward the origin with the -Z-axis as the view up direction, the texture is mapped onto the face with the same orientation as if the image were displayed normally in 2D. On the bottom face of the box (-Y), when viewed from below looking up the Y-axis toward the origin with the +Z-axis as the view up direction, the texture is mapped onto the face with the same orientation as if the image were displayed normally in 2D. TextureTransform affects the texture coordinates of the Box.

The Box node's geometry requires outside faces only. When viewed from the inside the results are undefined.

## 6.8 Collision

```
Collision {
  eventIn      MFNode   addChildren
  eventIn      MFNode   removeChildren
  exposedField MFNode   children        []
  exposedField SFBool   collide         TRUE
  field        SFVec3f  bboxCenter      0 0 0      # (-∞,∞)
  field        SFVec3f  bboxSize        -1 -1 -1   # (0,∞) or -1,-1,-1
  field        SFNode   proxy           NULL
  eventOut     SFTime   collideTime
}
```

The Collision node is a grouping node that specifies the collision detection properties for its children (and their descendants), specifies surrogate objects that replace its children during collision detection, and sends events signalling that a collision has occurred between the avatar and the Collision node's geometry or surrogate. By default, all geometric nodes in the scene are collidable with the viewer except IndexedLineSet, PointSet, and Text. Browsers shall detect geometric collisions between the avatar (see 6.29, NavigationInfo) and the scene's geometry and prevent the avatar from 'entering' the geometry. See 4.13.4, Collision detection and terrain following, for general information on collision detection.

If there are no Collision nodes specified in a VRML file, browsers shall detect collisions between the avatar and all objects during navigation.

Subclause 4.6.5, Grouping and children nodes, contains a description of the *children*, *addChildren*, and *removeChildren* fields and eventIns.

The Collision node's *collide* field enables and disables collision detection. If *collide* is set to FALSE, the children and all descendants of the Collision node shall not be checked for collision, even though they are drawn. This includes any descendent Collision nodes that have *collide* set to TRUE (i.e., setting *collide* to FALSE turns collision off for every node below it).

Collision nodes with the *collide* field set to TRUE detect the nearest collision with their descendent geometry (or proxies). When the nearest collision is detected, the collided Collision node sends the time of the collision through its *collideTime* eventOut. If a Collision node contains a child, descendant, or proxy (see below) that is a Collision node, and both Collision nodes detect that a collision has occurred, both send a *collideTime* event at the same time. A *collideTime* event shall be generated if the avatar is colliding with collidable geometry when the Collision node is read from a VRML file or inserted into the transformation hierarchy.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Collision node's children. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. A default *bboxSize* value, (-1, -1, -1), implies that the bounding box is not specified and if needed shall be calculated by the browser. More details on the *bboxCenter* and *bboxSize* fields can be found in 4.6.4, Bounding boxes.

The collision proxy, defined in the *proxy* field, is any legal children node as described in 4.6.5, Grouping and children nodes, that is used as a substitute for the Collision node's children during collision detection. The proxy is used strictly for collision detection; it is not drawn.

If the value of the *collide* field is TRUE and the *proxy* field is non-NULL, the *proxy* field defines the scene on which collision detection is performed. If the *proxy* value is NULL, collision detection is performed against the *children* of the Collision node.

If *proxy* is specified, any descendent children of the Collision node are ignored during collision detection. If *children* is empty, *collide* is TRUE, and *proxy* is specified, collision detection is performed against the proxy but nothing is displayed. In this manner, invisible collision objects may be supported.

The *collideTime* eventOut generates an event specifying the time when the avatar (see 6.29, NavigationInfo) makes contact with the collidable children or proxy of the Collision node. An ideal implementation computes the exact time of collision. Implementations may approximate the ideal by sampling the positions of collidable objects and the user. The NavigationInfo node contains additional information for parameters that control the avatar size.

## 6.9 Color

```
Color {
  exposedField MFColor color  []         # [0,1]
}
```

This node defines a set of RGB colours to be used in the fields of another node.

Color nodes are only used to specify multiple colours for a single geometric shape, such as colours for the faces or vertices of an IndexedFaceSet. A Material node is used to specify the overall material parameters of lit geometry. If both a Material node and a Color node are specified for a geometric shape, the colours shall replace the diffuse component of the material.

RGB or RGBA textures take precedence over colours; specifying both an RGB or RGBA texture and a Color node for geometric shape will result in the Color node being ignored. Details on lighting equations can be found in 4.14, Lighting model.

## 6.10 ColorInterpolator

```
ColorInterpolator {
  eventIn      SFFloat set_fraction        # (-∞,∞)
  exposedField MFFloat key          []     # (-∞,∞)
  exposedField MFColor keyValue     []     # [0,1]
  eventOut     SFColor value_changed
}
```

This node interpolates among a list of MFColor key values to produce an SFColor (RGB) *value_changed* event. The number of colours in the *keyValue* field shall be equal to the number of keyframes in the *key* field. The *keyValue* field and *value_changed* events are defined in RGB colour space. A linear interpolation using the value of *set_fraction* as input is performed in HSV space (see E.[FOLE] for description of RGB and HSV colour spaces). The results are undefined when interpolating between two consecutive keys with complementary hues.

4.6.8, Interpolator nodes, contains a detailed discussion of interpolators.

# 6.11 Cone

```
Cone {
  field     SFFloat   bottomRadius 1        # (0,∞)
  field     SFFloat   height       2        # (0,∞)
  field     SFBool    side         TRUE
  field     SFBool    bottom       TRUE
}
```

The Cone node specifies a cone which is centred in the local coordinate system and whose central axis is aligned with the local Y-axis. The *bottomRadius* field specifies the radius of the cone's base, and the *height* field specifies the height of the cone from the centre of the base to the apex. By default, the cone has a radius of 1.0 at the bottom and a height of 2.0, with its apex at y = *height*/2 and its bottom at y = -*height*/2. Both *bottomRadius* and *height* shall be greater than zero. Figure 6.3 illustrates the Cone node.

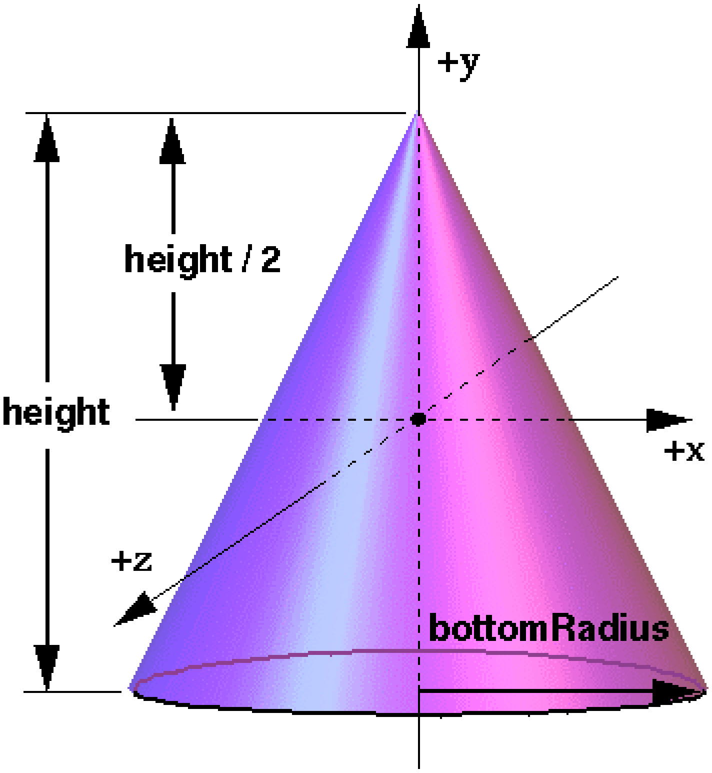


**Figure 6.3 -- Cone node**

The *side* field specifies whether sides of the cone are created and the *bottom* field specifies whether the bottom cap of the cone is created. A value of TRUE specifies that this part of the cone exists, while a value of FALSE specifies that this part does not exist (not rendered or eligible for collision or sensor intersection tests).

When a texture is applied to the sides of the cone, the texture wraps counterclockwise (from above) starting at the back of the cone. The texture has a vertical seam at the back in the X=0 plane, from the apex (0, *height*/2, 0) to the point (0, -*height*/2, -*bottomRadius*). For the bottom cap, a circle is cut out of the texture square centred at (0,-*height*/2, 0) with dimensions (2 × *bottomRadius)* by (2 × *bottomRadius)*. The bottom cap texture appears right

APPENDIX KK

side up when the top of the cone is rotated towards the -Z-axis. TextureTransform affects the texture coordinates of the Cone.

The Cone geometry requires outside faces only. When viewed from the inside the results are undefined.

## 6.12 Coordinate

```
Coordinate {
  exposedField MFVec3f point  []      # (-∞,∞)
}
```

This node defines a set of 3D coordinates to be used in the *coord* field of vertex-based geometry nodes including IndexedFaceSet, IndexedLineSet, and PointSet.

## 6.13 CoordinateInterpolator

```
CoordinateInterpolator {
  eventIn      SFFloat set_fraction        # (-∞,∞)
  exposedField MFFloat key           []    # (-∞,∞)
  exposedField MFVec3f keyValue      []    # (-∞,∞)
  eventOut     MFVec3f value_changed
}
```

This node linearly interpolates among a list of MFVec3f values. The number of coordinates in the *keyValue* field shall be an integer multiple of the number of keyframes in the *key* field. That integer multiple defines how many coordinates will be contained in the *value_changed* events.

4.6.8, Interpolator nodes, contains a more detailed discussion of interpolators.

## 6.14 Cylinder

```
Cylinder {
  field    SFBool    bottom  TRUE
  field    SFFloat   height  2         # (0,∞)
  field    SFFloat   radius  1         # (0,∞)
  field    SFBool    side    TRUE
  field    SFBool    top     TRUE
}
```

The Cylinder node specifies a capped cylinder centred at (0,0,0) in the local coordinate system and with a central axis oriented along the local Y-axis. By default, the cylinder is sized at "-1" to "+1" in all three dimensions. The *radius* field specifies the radius of the cylinder and the *height* field specifies the height of the cylinder along the central axis. Both *radius* and *height* shall be greater than zero. Figure 6.4 illustrates the Cylinder node.

The cylinder has three *parts*: the *side*, the *top* (Y = +height/2) and the *bottom* (Y = -height/2). Each part has an associated SFBool field that indicates whether the part exists (TRUE) or does not exist (FALSE). Parts which do not exist are not rendered and not eligible for intersection tests (e.g., collision detection or sensor activation).

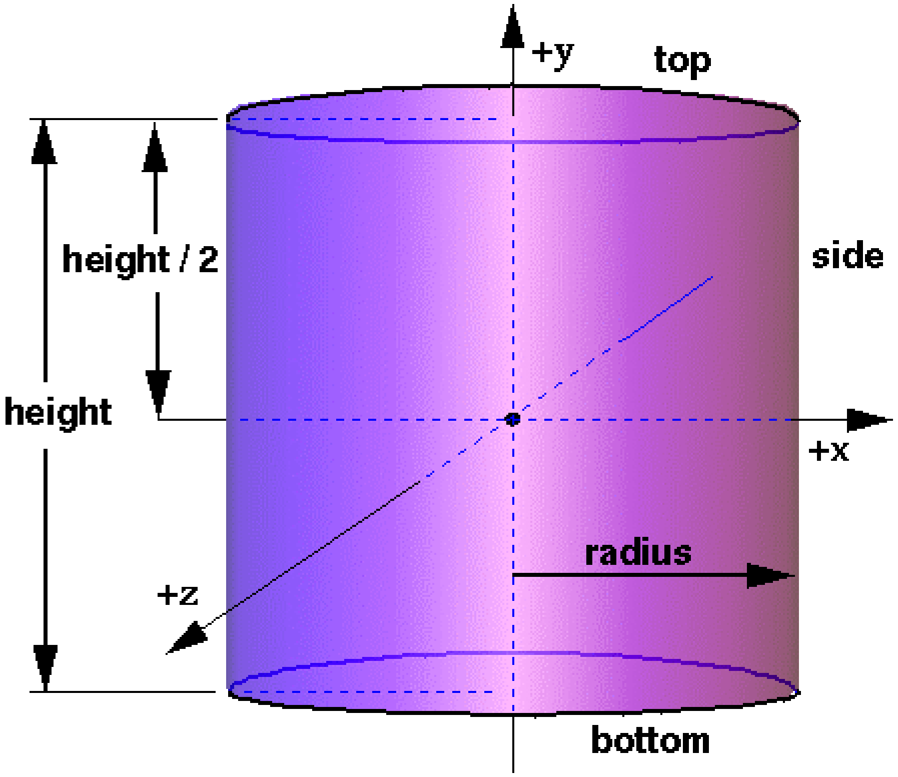


**Figure 6.4 -- Cylinder node**

When a texture is applied to a cylinder, it is applied differently to the sides, top, and bottom. On the sides, the texture wraps counterclockwise (from above) starting at the back of the cylinder. The texture has a vertical seam at the back, intersecting the X=0 plane. For the top and bottom caps, a circle is cut out of the unit texture squares centred at (0, +/- *height/2*, 0) with dimensions 2 × *radius* by 2 × *radius*. The top texture appears right side up when the top of the cylinder is tilted toward the +Z-axis, and the bottom texture appears right side up when the top of the cylinder is tilted toward the -Z-axis. TextureTransform affects the texture coordinates of the Cylinder node.

The Cylinder node's geometry requires outside faces only. When viewed from the inside the results are undefined.

# 6.15 CylinderSensor

```
CylinderSensor {
  exposedField SFBool     autoOffset TRUE
  exposedField SFFloat    diskAngle  0.262       # (0,π/2)
  exposedField SFBool     enabled    TRUE
  exposedField SFFloat    maxAngle   -1          # [-2π,2π]
  exposedField SFFloat    minAngle   0           # [-2π,2π]
  exposedField SFFloat    offset     0           # (-∞,∞)
  eventOut     SFBool     isActive
  eventOut     SFRotation rotation_changed
  eventOut     SFVec3f    trackPoint_changed
}
```

The CylinderSensor node maps pointer motion (e.g., a mouse or wand) into a rotation on an invisible cylinder that is aligned with the Y-axis of the local coordinate system. The CylinderSensor uses the descendent geometry of its parent node to determine whether it is liable to generate events.

The *enabled* exposed field enables and disables the CylinderSensor node. If TRUE, the sensor reacts appropriately to user events. If FALSE, the sensor does not track user input or send events. If *enabled* receives a FALSE event and *isActive* is TRUE, the sensor becomes disabled and deactivated, and outputs an *isActive* FALSE event. If *enabled* receives a TRUE event the sensor is enabled and ready for user activation.

A CylinderSensor node generates events when the pointing device is activated while the pointer is indicating any descendent geometry nodes of the sensor's parent group. See 4.6.7.5, Activating and manipulating sensors, for more details on using the pointing device to activate the CylinderSensor.

Upon activation of the pointing device while indicating the sensor's geometry, an *isActive* TRUE event is sent. The initial acute angle between the bearing vector and the local Y-axis of the CylinderSensor node determines whether the sides of the invisible cylinder or the caps (disks) are used for manipulation. If the initial angle is less than the *diskAngle*, the geometry is treated as an infinitely large disk lying in the local Y=0 plane and coincident with the initial intersection point. Dragging motion is mapped into a rotation around the local +Y-axis vector of the sensor's coordinate system. The perpendicular vector from the initial intersection point to the Y-axis defines zero rotation about the Y-axis. For each subsequent position of the bearing, a *rotation_changed* event is sent that equals the sum of the rotation about the +Y-axis vector (from the initial intersection to the new intersection) plus the *offset* value. *trackPoint_changed* events reflect the unclamped drag position on the surface of this disk. When the pointing device is deactivated and *autoOffset* is TRUE, *offset* is set to the last value of *rotation_changed* and an *offset_changed* event is generated. See 4.6.7.4, Drag sensors, for a more general description of *autoOffset* and *offset* fields.

If the initial acute angle between the bearing vector and the local Y-axis of the CylinderSensor node is greater than or equal to *diskAngle*, then the sensor behaves like a cylinder. The shortest distance between the point of intersection (between the bearing and the sensor's geometry) and the Y-axis of the parent group's local coordinate system determines the radius of an invisible cylinder used to map pointing device motion and marks the zero rotation value. For each subsequent position of the bearing, a *rotation_changed* event is sent that equals the sum of the right-handed rotation from the original intersection about the +Y-axis vector plus the *offset* value. *trackPoint_changed* events reflect the unclamped drag position on the surface of the invisible cylinder. When the pointing device is deactivated and *autoOffset* is TRUE, *offset* is set to the last rotation angle and an *offset_changed* event is generated. More details are available in 4.6.7.4, Drag sensors.

When the sensor generates an *isActive* TRUE event, it grabs all further motion events from the pointing device until it is released and generates an *isActive* FALSE event (other pointing-device sensors shall not generate events during this time). Motion of the pointing device while *isActive* is TRUE is referred to as a "drag." If a 2D pointing device is in use, *isActive* events will typically reflect the state of the primary button associated with the device (i.e., *isActive* is

TRUE when the primary button is pressed and FALSE when it is released). If a 3D pointing device (e.g., a wand) is in use, *isActive* events will typically reflect whether the pointer is within or in contact with the sensor's geometry.

While the pointing device is activated, *trackPoint_changed* and *rotation_changed* events are output and are interpreted from pointing device motion based on the sensor's local coordinate system at the time of activation. *trackPoint_changed* events represent the unclamped intersection points on the surface of the invisible cylinder or disk. If the initial angle results in cylinder rotation (as opposed to disk behaviour) and if the pointing device is dragged off the cylinder while activated, browsers may interpret this in a variety of ways (e.g., clamp all values to the cylinder and continuing to rotate as the point is dragged away from the cylinder). Each movement of the pointing device while *isActive* is TRUE generates *trackPoint_changed* and *rotation_changed* events.

The *minAngle* and *maxAngle* fields clamp *rotation_changed* events to a range of values. If *minAngle* is greater than *maxAngle*, *rotation_changed* events are not clamped. The *minAngle* and *maxAngle* fields are restricted to the range [-2π, 2π].

More information about this behaviour is described in 4.6.7.3, Pointing-device sensors, 4.6.7.4, Drag sensors, and 4.6.7.5, Activating and manipulating sensors.

## 6.16 DirectionalLight

```
DirectionalLight {
  exposedField SFFloat ambientIntensity  0        # [0,1]
  exposedField SFColor color             1 1 1    # [0,1]
  exposedField SFVec3f direction         0 0 -1   # (-∞,∞)
  exposedField SFFloat intensity         1        # [0,1]
  exposedField SFBool  on                TRUE
}
```

The DirectionalLight node defines a directional light source that illuminates along rays parallel to a given 3-dimensional vector. A description of the *ambientIntensity*, *color*, *intensity*, and *on* fields is in 4.6.6, Light sources.

The *direction* field specifies the direction vector of the illumination emanating from the light source in the local coordinate system. Light is emitted along parallel rays from an infinite distance away. A directional light source illuminates only the objects in its enclosing parent group. The light may illuminate everything within this coordinate system, including all children and descendants of its parent group. The accumulated transformations of the parent nodes affect the light.

DirectionalLight nodes do not attenuate with distance. A precise description of VRML's lighting equations is contained in 4.14, Lighting model.

APPENDIX KK

# 6.17 ElevationGrid

```
ElevationGrid {
  eventIn      MFFloat  set_height
  exposedField SFNode   color             NULL
  exposedField SFNode   normal            NULL
  exposedField SFNode   texCoord          NULL
  field        MFFloat  height            []       # (-∞,∞)
  field        SFBool   ccw               TRUE
  field        SFBool   colorPerVertex    TRUE
  field        SFFloat  creaseAngle       0        # [0,∞)
  field        SFBool   normalPerVertex   TRUE
  field        SFBool   solid             TRUE
  field        SFInt32  xDimension        0        # [0,∞)
  field        SFFloat  xSpacing          1.0      # [0,∞)
  field        SFInt32  zDimension        0        # [0,∞)
  field        SFFloat  zSpacing          1.0      # [0,∞)
}
```

The ElevationGrid node specifies a uniform rectangular grid of varying height in the Y=0 plane of the local coordinate system. The geometry is described by a scalar array of height values that specify the height of a surface above each point of the grid.

The *xDimension* and *zDimension* fields indicate the number of elements of the grid *height* array in the X and Z directions. Both *xDimension* and *zDimension* shall be greater than or equal to zero. If either the *xDimension* or the *zDimension* is less than two, the ElevationGrid contains no quadrilaterals. The vertex locations for the rectangles are defined by the *height* field and the *xSpacing* and *zSpacing* fields:

- The *height* field is an *xDimension* by *zDimension* array of scalar values representing the height above the grid for each vertex.

- The *xSpacing* and *zSpacing* fields indicate the distance between vertices in the X and Z directions respectively, and shall be greater than zero.

Thus, the vertex corresponding to the point P[i, j] on the grid is placed at:

```
    P[i,j].x = xSpacing × i
    P[i,j].y = height[ i + j × xDimension]
    P[i,j].z = zSpacing × j

    where 0 <= i < xDimension and 0 <= j < zDimension,
    and P[0,0] is height[0] units above/below the origin of the local
    coordinate system
```

The *set_height* eventIn allows the height MFFloat field to be changed to support animated ElevationGrid nodes.

The *color* field specifies per-vertex or per-quadrilateral colours for the ElevationGrid node depending on the value of *colorPerVertex*. If the *color* field is NULL, the ElevationGrid node is rendered with the overall attributes of the Shape node enclosing the ElevationGrid node (see 4.14, Lighting model).

The *colorPerVertex* field determines whether colours specified in the *color* field are applied to each vertex or each quadrilateral of the ElevationGrid node. If *colorPerVertex* is FALSE and the *color* field is not NULL, the *color* field shall specify a Color node containing at least (*xDimension-1)* × (*zDimension-1)* colours; one for each quadrilateral, ordered as follows:

```
    QuadColor[i,j] = Color[ i + j × (xDimension-1)]

    where 0 <= i < xDimension-1 and 0 <= j < zDimension-1,
    and QuadColor[i,j] is the colour for the quadrilateral defined
        by height[i+j × xDimension], height[(i+1)+j × xDimension],
        height[(i+1)+(j+1)× xDimension] and height[i+(j+1)× xDimension]
```

If *colorPerVertex* is TRUE and the *color* field is not NULL, the *color* field shall specify a Color node containing at least *xDimension* × *zDimension* colours, one for each vertex, ordered as follows:

```
    VertexColor[i,j] = Color[ i + j × xDimension]

    where 0 <= i < xDimension and 0 <= j < zDimension,
    and VertexColor[i,j] is the colour for the vertex defined by
        height[i+j × xDimension]
```

The *normal* field specifies per-vertex or per-quadrilateral normals for the ElevationGrid node. If the *normal* field is NULL, the browser shall automatically generate normals, using the *creaseAngle* field to determine if and how normals are smoothed across the surface (see 4.6.3.5, Crease angle field).

The *normalPerVertex* field determines whether normals are applied to each vertex or each quadrilateral of the ElevationGrid node depending on the value of *normalPerVertex*. If *normalPerVertex* is FALSE and the *normal* node is not NULL, the *normal* field shall specify a Normal node containing at least (*xDimension-1)* × (*zDimension-1)* normals; one for each quadrilateral, ordered as follows:

```
    QuadNormal[i,j] = Normal[ i + j ×(xDimension-1)]

    where 0 <= i < xDimension-1 and 0 <= j < zDimension-1,
    and QuadNormal[i,j] is the normal for the quadrilateral defined
        by height[i+j × xDimension], height[(i+1)+j × xDimension],
        height[(i+1)+(j+1)× xDimension] and height[i+(j+1)× xDimension]
```

If *normalPerVertex* is TRUE and the *normal* field is not NULL, the *normal* field shall specify a Normal node containing at least *xDimension* × *zDimension* normals; one for each vertex, ordered as follows:

```
    VertexNormal[i,j] = Normal[ i + j × xDimension]

    where 0 <= i < xDimension and 0 <= j < zDimension,
    and VertexNormal[i,j] is the normal for the vertex defined
        by height[i+j×xDimension]
```

The *texCoord* field specifies per-vertex texture coordinates for the ElevationGrid node. If *texCoord* is NULL, default texture coordinates are applied to the geometry. The default texture coordinates range from (0,0) at the first vertex to (1,1) at the last vertex. The S texture coordinate is aligned with the positive X-axis, and the T texture coordinate with positive Z-axis. If *texCoord* is not NULL, it shall specify a TextureCoordinate node containing at least (*xDimension)* × (*zDimension)* texture coordinates; one for each vertex, ordered as follows:

```
    VertexTexCoord[i,j] = TextureCoordinate[ i + j × xDimension]

    where 0 <= i < xDimension and 0 <= j < zDimension,
    and VertexTexCoord[i,j] is the texture coordinate for the vertex
        defined by height[i+j × xDimension]
```

The *ccw*, *solid*, and *creaseAngle* fields are described in 4.6.3, Shapes and geometry.

By default, the quadrilaterals are defined with a counterclockwise ordering. Hence, the Y-component of the normal is positive. Setting the *ccw* field to FALSE reverses the normal direction. Backface culling is enabled when the *solid* field is TRUE.

See Figure 6.5 for a depiction of the ElevationGrid node.

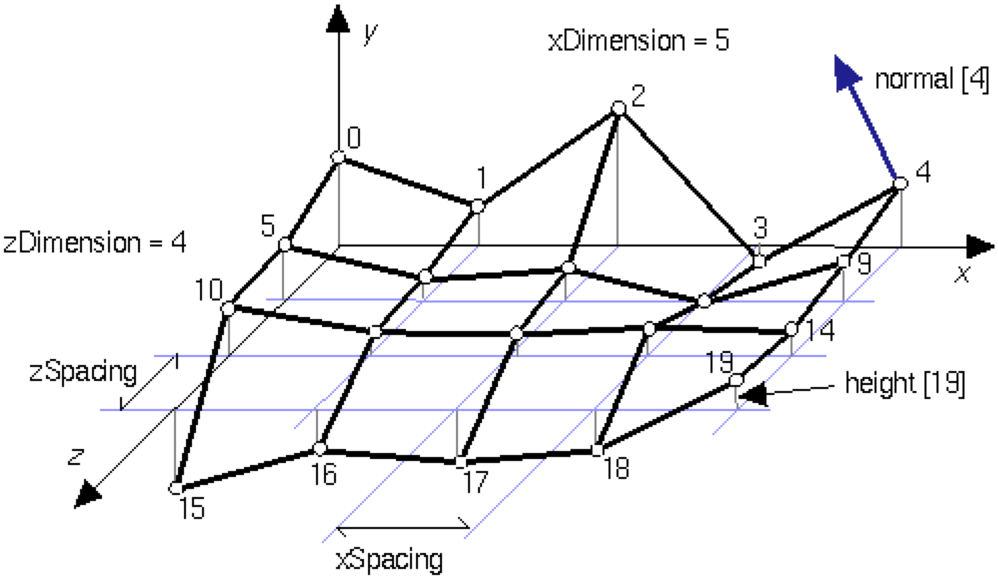


**Figure 6.5 -- ElevationGrid node**

## 6.18 Extrusion

```
Extrusion {
  eventIn MFVec2f     set_crossSection
  eventIn MFRotation  set_orientation
  eventIn MFVec2f     set_scale
  eventIn MFVec3f     set_spine
  field   SFBool      beginCap         TRUE
  field   SFBool      ccw              TRUE
  field   SFBool      convex           TRUE
  field   SFFloat     creaseAngle      0                # [0,∞)
  field   MFVec2f     crossSection     [ 1 1, 1 -1, -1 -1,
                                        -1 1, 1  1 ]    # (-∞,∞)
  field   SFBool      endCap           TRUE
  field   MFRotation  orientation      0 0 1 0          # [-1,1],(- ∞,∞)
  field   MFVec2f     scale            1 1              # (0,∞)
  field   SFBool      solid            TRUE
  field   MFVec3f     spine            [ 0 0 0, 0 1 0 ] # (-∞,∞)
}
```

### 6.18.1 Introduction

The Extrusion node specifies geometric shapes based on a two dimensional cross-section extruded along a three dimensional spine in the local coordinate system. The cross-section can be scaled and rotated at each spine point to produce a wide variety of shapes.

An Extrusion node is defined by:

a. a 2D *crossSection* piecewise linear curve (described as a series of connected vertices);

b. a 3D *spine* piecewise linear curve (also described as a series of connected vertices);

c. a list of 2D *scale* parameters;

d. a list of 3D *orientation* parameters.

### 6.18.2 Algorithmic description

Shapes are constructed as follows. The cross-section curve, which starts as a curve in the Y=0 plane, is first scaled about the origin by the first *scale* parameter (first value scales in X, second value scales in Z). It is then translated by the first spine point and oriented using the first *orientation* parameter (as explained later). The same procedure is followed to place a cross-section at the second spine point, using the second scale and orientation values. Corresponding vertices of the first and second cross-sections are then connected, forming a quadrilateral polygon between each pair of vertices. This same procedure is then repeated for the rest of the spine points, resulting in a surface extrusion along the spine.

The final orientation of each cross-section is computed by first orienting it relative to the spine segments on either side of point at which the cross-section is placed. This is known as the *spine-aligned cross-section plane* (SCP), and is designed to provide a smooth transition from one spine segment to the next (see Figure 6.6). The SCP is then rotated by the corresponding *orientation* value. This rotation is performed relative to the SCP. For example, to impart twist in the cross-section, a rotation about the Y-axis (0 1 0) would be used. Other orientations are valid and rotate the cross-section out of the SCP.

The SCP is computed by first computing its Y-axis and Z-axis, then taking the cross product of these to determine the X-axis. These three axes are then used to determine the rotation value needed to rotate the Y=0 plane to the SCP. This results in a plane that is the approximate tangent of the spine at each point, as shown in Figure 6.6. First the Y-axis is determined, as follows:

Let n be the number of spines and let i be the index variable satisfying 0 <= i < n:

a. *For all points other than the first or last:* The Y-axis for *spine*[i] is found by normalizing the vector defined by (*spine*[i+1] - *spine*[i-1]).

b. *If the spine curve is closed:* The SCP for the first and last points is the same and is found using (*spine*[1] - *spine*[n-2]) to compute the Y-axis.

c. *If the spine curve is not closed:* The Y-axis used for the first point is the vector from *spine*[0] to *spine*[1], and for the last it is the vector from *spine*[*n*-2] to *spine*[*n*-1].

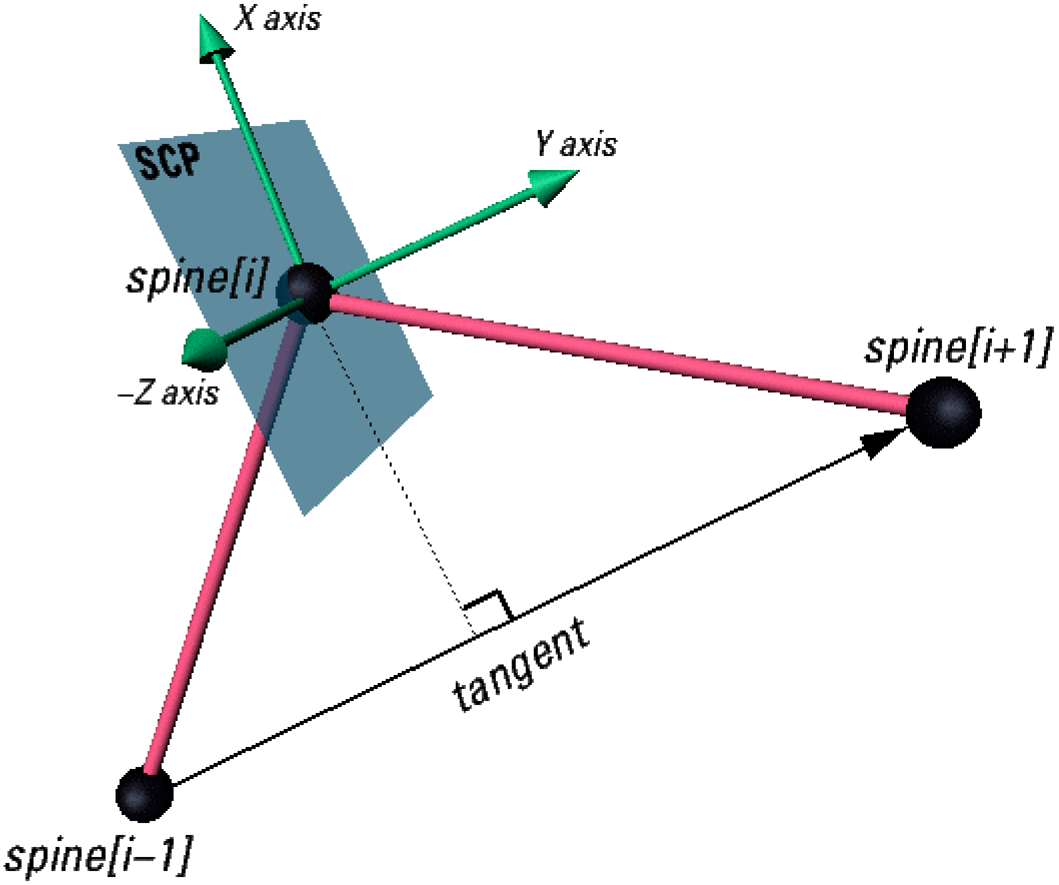


**Figure 6.6 -- Spine-aligned cross-section plane at a spine point.**

The Z-axis is determined as follows:

d. *For all points other than the first or last:* Take the following cross-product:

   `Z = (spine[i+1] - spine[i]) × (spine[i-1] - spine[i])`

e. *If the spine curve is closed:* The SCP for the first and last points is the same and is found by taking the following cross-product:

   `Z = (spine[1] - spine[0]) × (spine[n-2] - spine[0])`

f. *If the spine curve is not closed:* The Z-axis used for the first spine point is the same as the Z-axis for spine[1]. The Z-axis used for the last spine point is the same as the Z-axis for spine[n-2].

g. After determining the Z-axis, its dot product with the Z-axis of the previous spine point is computed. If this value is negative, the Z-axis is flipped (multiplied by -1). In most cases, this prevents small changes in the spine segment angles from flipping the cross-section 180 degrees.

Once the Y- and Z-axes have been computed, the X-axis can be calculated as their cross-product.

### 6.18.3 Special cases

If the number of *scale* or *orientation* values is greater than the number of spine points, the excess values are ignored. If they contain one value, it is applied at all spine points. The results are undefined if the number of scale or orientation values is greater than one but less than the number of spine points. The *scale* values shall be positive.

If the three points used in computing the Z-axis are collinear, the cross-product is zero so the value from the previous point is used instead.

If the Z-axis of the first point is undefined (because the spine is not closed and the first two spine segments are collinear) then the Z-axis for the first spine point with a defined Z-axis is used.

If the entire spine is collinear, the SCP is computed by finding the rotation of a vector along the positive Y-axis (`v1`) to the vector formed by the spine points (`v2`). The Y=0 plane is then rotated by this value.

If two points are coincident, they both have the same SCP. If each point has a different orientation value, then the surface is constructed by connecting edges of the cross-sections as normal. This is useful in creating revolved surfaces.

*Note: combining coincident and non-coincident spine segments, as well as other combinations, can lead to interpenetrating surfaces which the extrusion algorithm makes no attempt to avoid.*

### 6.18.4 Common cases

The following common cases are among the effects which are supported by the Extrusion node:

Surfaces of revolution:

> *If the cross-section is an approximation of a circle and the spine is straight, the Extrusion is equivalent to a surface of revolution, where the* scale *parameters define the size of the cross-section along the spine.*

Uniform extrusions:

> *If the* scale *is (1, 1) and the spine is straight, the cross-section is extruded uniformly without twisting or scaling along the spine. The result is a cylindrical shape with a uniform cross section.*

Bend/twist/taper objects:

> *These shapes are the result of using all fields. The spine curve bends the extruded shape defined by the cross-section, the orientation parameters (given as rotations about the Y-axis) twist it around the spine, and the scale parameters taper it (by scaling about the spine).*

### 6.18.5 Other fields

Extrusion has three *parts*: the sides, the *beginCap* (the surface at the initial end of the spine) and the *endCap* (the surface at the final end of the spine). The caps have an associated SFBool field that indicates whether each exists (TRUE) or doesn't exist (FALSE).

When the *beginCap* or *endCap* fields are specified as TRUE, planar cap surfaces will be generated regardless of whether the *crossSection* is a closed curve. If *crossSection* is not a closed curve, the caps are generated by adding a final point to *crossSection* that is equal to the initial point. An open surface can still have a cap, resulting (for a simple case) in a shape analogous to a soda can sliced in half vertically. These surfaces are generated even if *spine* is also a closed curve. If a field value is FALSE, the corresponding cap is not generated.

Texture coordinates are automatically generated by Extrusion nodes. Textures are mapped so that the coordinates range in the U direction from 0 to 1 along the *crossSection* curve (with 0 corresponding to the first point in *crossSection* and 1 to the last) and in the V direction from 0 to 1 along the *spine* curve (with 0 corresponding to the first listed *spine* point and 1 to the last). If either the *endCap* or *beginCap* exists, the *crossSection* curve is uniformly scaled and translated so that the larger dimension of the cross-section (X or Z) produces texture coordinates that range from 0.0 to 1.0. The *beginCap* and *endCap* textures' S and T directions correspond to the X and Z directions in which the *crossSection* coordinates are defined.

APPENDIX KK

The browser shall automatically generate normals for the Extrusion node,using the *creaseAngle* field to determine if and how normals are smoothed across the surface. Normals for the caps are generated along the Y-axis of the SCP, with the ordering determined by viewing the cross-section from above (looking along the negative Y-axis of the SCP). By default, a *beginCap* with a counterclockwise ordering shall have a normal along the negative Y-axis. An *endCap* with a counterclockwise ordering shall have a normal along the positive Y-axis.

Each quadrilateral making up the sides of the extrusion are ordered from the bottom cross-section (the one at the earlier spine point) to the top. So, one quadrilateral has the points:

```
    spine[0](crossSection[0], crossSection[1])
    spine[1](crossSection[1], crossSection[0])
```

in that order. By default, normals for the sides are generated as described in 4.6.3, Shapes and geometry.

For instance, a circular crossSection with counter-clockwise ordering and the default spine form a cylinder. With *solid* TRUE and *ccw* TRUE, the cylinder is visible from the outside. Changing *ccw* to FALSE makes it visible from the inside.

The *ccw*, *solid*, *convex*, and *creaseAngle* fields are described in 4.6.3, Shapes and geometry.

## 6.19 Fog

```
Fog {
  exposedField SFColor  color           1 1 1      # [0,1]
  exposedField SFString fogType         "LINEAR"
  exposedField SFFloat  visibilityRange 0          # [0,∞)
  eventIn      SFBool   set_bind
  eventOut     SFBool   isBound
}
```

The Fog node provides a way to simulate atmospheric effects by blending objects with the colour specified by the *color* field based on the distances of the various objects from the viewer. The distances are calculated in the coordinate space of the Fog node. The *visibilityRange* specifies the distance in metres (in the local coordinate system) at which objects are totally obscured by the fog. Objects located outside the *visibilityRange* from the viewer are drawn with a constant colour of *color*. Objects very close to the viewer are blended very little with the fog *color*. A *visibilityRange* of 0.0 disables the Fog node. The *visibilityRange* is affected by the scaling transformations of the Fog node's parents; translations and rotations have no affect on *visibilityRange*. Values of the *visibilityRange* field shall be in the range [0,∞).

Since Fog nodes are bindable children nodes (see 4.6.10, Bindable children nodes), a Fog node stack exists, in which the top-most Fog node on the stack is currently active. To push a Fog node onto the top of the stack, a TRUE value is sent to the *set_bind* eventIn. Once active, the Fog node is bound to the browser view. A FALSE value sent to *set_bind*, pops the Fog node from the stack and unbinds it from the browser viewer. More details on the Fog node stack can be found in 4.6.10, Bindable children nodes.

The *fogType* field controls how much of the fog colour is blended with the object as a function of distance. If *fogType* is "LINEAR", the amount of blending is a linear function of the distance, resulting in a depth cueing effect. If *fogType* is "EXPONENTIAL," an exponential increase in blending is used, resulting in a more natural fog appearance.

The effect of fog on lighting calculations is described in 4.14, Lighting model.

# 6.20 FontStyle

```
FontStyle {
  field MFString family        "SERIF"
  field SFBool   horizontal    TRUE
  field MFString justify       "BEGIN"
  field SFString language      ""
  field SFBool   leftToRight   TRUE
  field SFFloat  size          1.0          # (0,∞)
  field SFFloat  spacing       1.0          # [0,∞)
  field SFString style         "PLAIN"
  field SFBool   topToBottom   TRUE
}
```

### 6.20.1 Introduction

The FontStyle node defines the size, family, and style used for Text nodes, as well as the direction of the text strings and any language-specific rendering techniques used for non-English text. See 6.47, Text, for a description of the Text node.

The *size* field specifies the nominal height, in the local coordinate system of the Text node, of glyphs rendered and determines the spacing of adjacent lines of text. Values of the *size* field shall be greater than zero.

The *spacing* field determines the line spacing between adjacent lines of text. The distance between the baseline of each line of text is (*spacing* × *size)* in the appropriate direction (depending on other fields described below). The effects of the *size* and *spacing* field are depicted in Figure 6.7 (*spacing* greater than 1.0). Values of the *spacing* field shall be non-negative.

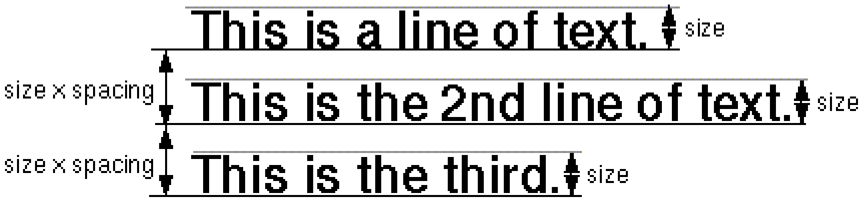


**Figure 6.7 -- Text *size* and *spacing* fields**

### 6.20.2 Font family and style

Font attributes are defined with the *family* and *style* fields. The browser shall map the specified font attributes to an appropriate available font as described below.

The *family* field contains a case-sensitive MFString value that specifies a sequence of font family names in preference order. The browser shall search the MFString value for the first font family name matching a supported font family. If none of the string values matches a supported font family, the default font family **"SERIF"** shall be used. All browsers shall support at least **"SERIF"** (the default) for a serif font such as Times Roman; **"SANS"** for a sans-serif font such as Helvetica; and **"TYPEWRITER"** for a fixed-pitch font such as Courier. An empty *family* value is identical to **["SERIF"]**.

The *style* field specifies a case-sensitive SFString value that may be **"PLAIN"** (the default) for default plain type; **"BOLD"** for boldface type; **"ITALIC"** for italic type; or **"BOLDITALIC"** for bold and italic type. An empty *style* value (**""**) is identical to **"PLAIN"**.

### 6.20.3 Direction and justification

The *horizontal*, *leftToRight*, and *topToBottom* fields indicate the direction of the text. The *horizontal* field indicates whether the text advances horizontally in its major direction (*horizontal* = TRUE, the default) or vertically in its major direction (*horizontal* = FALSE). The *leftToRight* and *topToBottom* fields indicate direction of text advance in the major (characters within a single string) and minor (successive strings) axes of layout. Which field is used for the major direction and which is used for the minor direction is determined by the *horizontal* field.

For horizontal text (*horizontal* = TRUE), characters on each line of text advance in the positive X direction if *leftToRight* is TRUE or in the negative X direction if *leftToRight* is FALSE. Characters are advanced according to their natural advance width. Each line of characters is advanced in the negative Y direction if *topToBottom* is TRUE or in the positive Y direction if *topToBottom* is FALSE. Lines are advanced by the amount of *size* × *spacing*.

For vertical text (*horizontal* = FALSE), characters on each line of text advance in the negative Y direction if *topToBottom* is TRUE or in the positive Y direction if *topToBottom* is FALSE. Characters are advanced according to their natural advance height. Each line of characters is advanced in the positive X direction if *leftToRight* is TRUE or in the negative X direction if *leftToRight* is FALSE. Lines are advanced by the amount of *size* × *spacing*.

The *justify* field determines alignment of the above text layout relative to the origin of the object coordinate system. The *justify* field is an MFString which can contain 2 values. The first value specifies alignment along the major axis and the second value specifies alignment along the minor axis, as determined by the *horizontal* field. An empty *justify* value (**""**) is equivalent to the default value. If the second string, minor alignment, is not specified, minor alignment defaults to the value **"FIRST".** Thus, *justify* values of **""**, **"BEGIN"**, and **["BEGIN" "FIRST"]** are equivalent.

The major alignment is along the X-axis when *horizontal* is TRUE and along the Y-axis when *horizontal is* FALSE. The minor alignment is along the Y-axis when *horizontal* is TRUE and along the X-axis when *horizontal is* FALSE. The possible values for each enumerant of the *justify* field are **"FIRST"**, **"BEGIN"**, **"MIDDLE"**, and **"END"**. For major alignment, each line of text is positioned individually according to the major alignment enumerant. For minor alignment, the block of text representing all lines together is positioned according to the minor alignment enumerant. Tables 6.2-6.5 describe the behaviour in terms of which portion of the text is at the origin

**Table 6.2 -- Major Alignment, *horizontal* = TRUE**

| ***justify* Enumerant** | ***leftToRight* = TRUE** | ***leftToRight* = FALSE** |
|---|---|---|
| FIRST | Left edge of each line | Right edge of each line |
| BEGIN | Left edge of each line | Right edge of each line |
| MIDDLE | Centred about X-axis | Centred about X-axis |
| END | Right edge of each line | Left edge of each line |

**Table 6.3 -- Major Alignment, *horizontal* = FALSE**

| ***justify* Enumerant** | ***topToBottom* = TRUE** | ***topToBottom* = FALSE** |
|---|---|---|
| FIRST | Top edge of each line | Bottom edge of each line |
| BEGIN | Top edge of each line | Bottom edge of each line |
| MIDDLE | Centred about Y-axis | Centre about Y-axis |
| END | Bottom edge of each line | Top edge of each line |

**Table 6.4 -- Minor Alignment, *horizontal* = TRUE**

| ***justify* Enumerant** | ***topToBottom* = TRUE** | ***topToBottom* = FALSE** |
|---|---|---|
| FIRST | Baseline of first line | Baseline of first line |
| BEGIN | Top edge of first line | Bottom edge of first line |
| MIDDLE | Centred about Y-axis | Centred about Y-axis |
| END | Bottom edge of last line | Top edge of last line |

**Table 6.5 -- Minor Alignment, *horizontal* = FALSE**

| ***justify* Enumerant** | ***leftToRight* = TRUE** | ***leftToRight* = FALSE** |
|---|---|---|
| FIRST | Left edge of first line | Right edge of first line |
| BEGIN | Left edge of first line | Right edge of first line |
| MIDDLE | Centred about X-axis | Centred about X-axis |
| END | Right edge of last line | Left edge of last line |

The default minor alignment is **"FIRST"**. This is a special case of minor alignment when *horizontal* is TRUE. Text starts at the baseline at the Y-axis. In all other cases, **"FIRST"** is identical to **"BEGIN"**. In Tables 6.6 and 6.7, each colour-coded cross-hair indicates where the X-axis and Y-axis shall be in relation to the text. Figure 6.8 describes the symbols used in Tables 6.6 and 6.7.

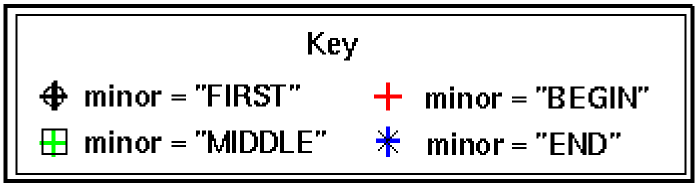
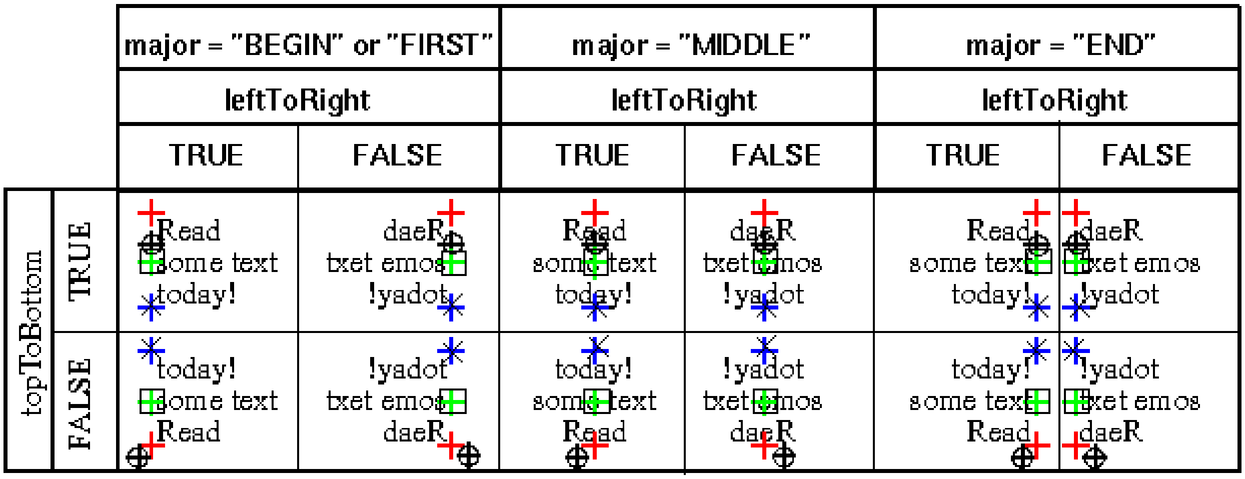


**Figure 6.8 -- Key for Tables 6.6 and 6.7**

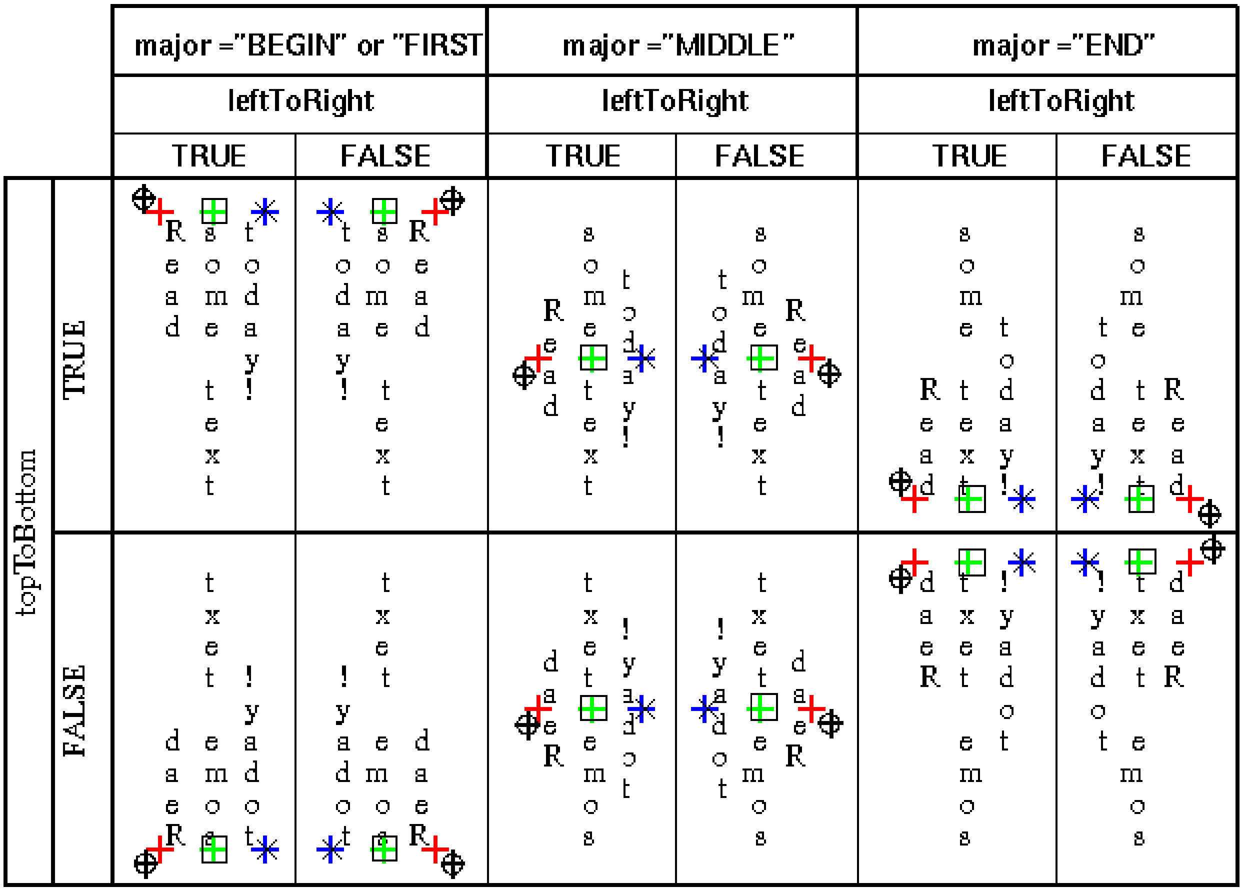
**Table 6.6 --** ***horizontal*** **= TRUE**



| | | major = "BEGIN" or "FIRST" | | major = "MIDDLE" | | major = "END" | |
|---|---|---|---|---|---|---|---|
| | | leftToRight | | leftToRight | | leftToRight | |
| | | TRUE | FALSE | TRUE | FALSE | TRUE | FALSE |
| topToBottom | TRUE | Read<br>some text<br>today! | daeR<br>txet emos<br>!yadot | Read<br>some text<br>today! | daeR<br>txet emos<br>!yadot | Read<br>some text<br>today! | daeR<br>txet emos<br>!yadot |
| | FALSE | today!<br>some text<br>Read | !yadot<br>txet emos<br>daeR | today!<br>some text<br>Read | !yadot<br>txet emos<br>daeR | today!<br>some text<br>Read | !yadot<br>txet emos<br>daeR |

Note: The "FIRST" minor axis marker ⊕ is offset from the "BEGIN" minor axis marker + in cases that they are coincident for presentation purposes only.

**Table 6.7 -- *horizontal = FALSE***



Note: In every case, the "FIRST" minor axis marker ⊕ is coincident with the "BEGIN" minor axis marker + (and is offset for presentation purposes only).

### 6.20.4 Language

The *language* field specifies the context of the language for the text string. Due to the multilingual nature of the ISO/IEC 10646-1:1993, the *language* field is needed to provide a proper language attribute of the text string. The format is based on RFC 1766: language[_territory] 2.[1766]. The value for the language tag is based on ISO 639:1988 (e.g., 'zh' for Chinese, 'jp' for Japanese, and 'sc' for Swedish.) The territory tag is based on ISO 3166:1993 country codes (e.g., 'TW' for Taiwan and 'CN' for China for the 'zh' Chinese language tag). If the *language* field is empty (""), local language bindings are used.

See 2, Normative references, for more information on RFC 1766 (2.[1766]), ISO/IEC 10646:1993 (2.[UTF8]), ISO/IEC 639:1998 (2.[I639]), and ISO 3166:1993 (2.[I3166]).

## 6.21 Group

```
Group {
  eventIn      MFNode  addChildren
  eventIn      MFNode  removeChildren
  exposedField MFNode  children      []
  field        SFVec3f bboxCenter    0 0 0     # (-∞,∞)
  field        SFVec3f bboxSize      -1 -1 -1  # (0,∞) or -1,-1,-1
}
```

A Group node contains children nodes without introducing a new transformation. It is equivalent to a Transform node containing an identity transform.

More details on the *children*, *addChildren*, and *removeChildren* fields and eventIns can be found in 4.6.5, Grouping and children nodes.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Group node's children. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. A default *bboxSize* value, (-1, -1, -1), implies that the bounding box is not specified and, if needed, is calculated by the browser. A description of the *bboxCenter* and *bboxSize* fields is contained in 4.6.4, Bounding boxes.

## 6.22 ImageTexture

```
ImageTexture {
  exposedField MFString url      []
  field        SFBool   repeatS TRUE
  field        SFBool   repeatT TRUE
}
```

The ImageTexture node defines a texture map by specifying an image file and general parameters for mapping to geometry. Texture maps are defined in a 2D coordinate system (s, t) that ranges from [0.0, 1.0] in both directions. The bottom edge of the image corresponds to the S-axis of the texture map, and left edge of the image corresponds to the T-axis of the texture map. The lower-left pixel of the image corresponds to s=0, t=0, and the top-right pixel of the image corresponds to s=1, t=1. These relationships are depicted in Figure 6.9.

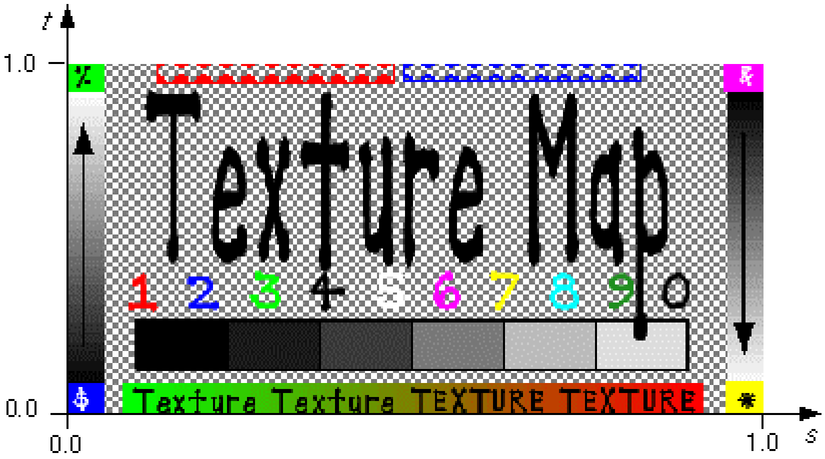


**Figure 6.9 -- Texture map coordinate system**

The texture is read from the URL specified by the *url* field. When the *url* field contains no values ([]), texturing is disabled. Browsers shall support the JPEG (see 2. [JPEG]) and PNG (see 2. [PNG]) image file formats. In addition, browsers may support other image formats (e.g. CGM, 2. [CGM]) which can be rendered into a 2D image. Support for the GIF format (see E. [GIF]) is also recommended (including transparency). Details on the *url* field can be found in 4.5, VRML and the World Wide Web.

See 4.6.11, Texture maps, for a general description of texture maps.

See 4.14, Lighting model, for a description of lighting equations and the interaction between textures, materials, and geometry appearance.

The *repeatS* and *repeatT* fields specify how the texture wraps in the S and T directions. If *repeatS* is TRUE (the default), the texture map is repeated outside the [0.0, 1.0] texture coordinate range in the S direction so that it fills the shape. If *repeatS* is FALSE, the texture coordinates are clamped in the S direction to lie within the [0.0, 1.0] range. The *repeatT* field is analogous to the *repeatS* field.

APPENDIX KK

# 6.23 IndexedFaceSet

```
IndexedFaceSet {
  eventIn       MFInt32 set_colorIndex
  eventIn       MFInt32 set_coordIndex
  eventIn       MFInt32 set_normalIndex
  eventIn       MFInt32 set_texCoordIndex
  exposedField  SFNode  color             NULL
  exposedField  SFNode  coord             NULL
  exposedField  SFNode  normal            NULL
  exposedField  SFNode  texCoord          NULL
  field         SFBool  ccw               TRUE
  field         MFInt32 colorIndex        []          # [-1,∞)
  field         SFBool  colorPerVertex    TRUE
  field         SFBool  convex            TRUE
  field         MFInt32 coordIndex        []          # [-1,∞)
  field         SFFloat creaseAngle       0           # [0,∞)
  field         MFInt32 normalIndex       []          # [-1,∞)
  field         SFBool  normalPerVertex   TRUE
  field         SFBool  solid             TRUE
  field         MFInt32 texCoordIndex     []          # [-1,∞)
}
```

The IndexedFaceSet node represents a 3D shape formed by constructing faces (polygons) from vertices listed in the *coord* field. The *coord* field contains a Coordinate node that defines the 3D vertices referenced by the *coordIndex* field. IndexedFaceSet uses the indices in its *coordIndex* field to specify the polygonal faces by indexing into the coordinates in the Coordinate node. An index of "-1" indicates that the current face has ended and the next one begins. The last face may be (but does not have to be) followed by a "-1" index. If the greatest index in the *coordIndex* field is N, the Coordinate node shall contain N+1 coordinates (indexed as 0 to N). Each face of the IndexedFaceSet shall have:

a. at least three non-coincident vertices;

b. vertices that define a planar polygon;

c. vertices that define a non-self-intersecting polygon.

Otherwise, The results are undefined.

The IndexedFaceSet node is specified in the local coordinate system and is affected by the transformations of its ancestors.

Descriptions of the *coord*, *normal*, and *texCoord* fields are provided in the Coordinate, Normal, and TextureCoordinate nodes, respectively.

Details on lighting equations and the interaction between *color* field, *normal* field, textures, materials, and geometries are provided in 4.14, Lighting model.

If the *color* field is not NULL, it shall contain a Color node whose colours are applied to the vertices or faces of the IndexedFaceSet as follows:

d. If *colorPerVertex* is FALSE, colours are applied to each face, as follows:

   1. If the *colorIndex* field is not empty, then one colour is used for each face of the IndexedFaceSet. There shall be at least as many indices in the *colorIndex* field as there are faces in the IndexedFaceSet. If the greatest index in the *colorIndex* field is N, then there shall be N+1 colours in the Color node. The *colorIndex* field shall not contain any negative entries.

   2. If the *colorIndex* field is empty, then the colours in the Color node are applied to each face of the IndexedFaceSet in order. There shall be at least as many colours in the Color node as there are faces.

e. If *colorPerVertex* is TRUE, colours are applied to each vertex, as follows:

   1. If the *colorIndex* field is not empty, then colours are applied to each vertex of the IndexedFaceSet in exactly the same manner that the *coordIndex* field is used to choose coordinates for each vertex from the Coordinate node. The *colorIndex* field shall contain at least as many indices as the *coordIndex* field, and shall contain end-of-face markers (-1) in exactly the same places as the *coordIndex* field. If the greatest index in the *colorIndex* field is N, then there shall be N+1 colours in the Color node.

   2. If the *colorIndex* field is empty, then the *coordIndex* field is used to choose colours from the Color node. If the greatest index in the *coordIndex* field is N, then there shall be N+1 colours in the Color node.

If the *color* field is NULL, the geometry shall be rendered normally using the Material and texture defined in the Appearance node (see 4.14, Lighting model, for details).

If the *normal* field is not NULL, it shall contain a Normal node whose normals are applied to the vertices or faces of the IndexedFaceSet in a manner exactly equivalent to that described above for applying colours to vertices/faces (where *normalPerVertex* corresponds to *colorPerVertex* and *normalIndex* corresponds to *colorIndex*). If the *normal* field is NULL, the browser shall automatically generate normals, using *creaseAngle* to determine if and how normals are smoothed across shared vertices (see 4.6.3.5, Crease angle field).

If the *texCoord* field is not NULL, it shall contain a TextureCoordinate node. The texture coordinates in that node are applied to the vertices of the IndexedFaceSet as follows:

f. If the *texCoordIndex* field is not empty, then it is used to choose texture coordinates for each vertex of the IndexedFaceSet in exactly the same manner that the *coordIndex* field is used to choose coordinates for each vertex from the Coordinate node. The *texCoordIndex* field shall contain at least as many indices as the *coordIndex* field, and shall contain end-of-face markers (-1) in exactly the same places as the *coordIndex* field. If the greatest index in the *texCoordIndex* field is N, then there shall be N+1 texture coordinates in the TextureCoordinate node.

g. If the *texCoordIndex* field is empty, then the *coordIndex* array is used to choose texture coordinates from the TextureCoordinate node. If the greatest index in the *coordIndex* field is N, then there shall be N+1 texture coordinates in the TextureCoordinate node.

If the *texCoord* field is NULL, a default texture coordinate mapping is calculated using the local coordinate system bounding box of the shape. The longest dimension of the bounding box defines the S coordinates, and the next longest defines the T coordinates. If two or all three dimensions of the bounding box are equal, ties shall be broken by choosing the X, Y, or Z dimension in that order of preference. The value of the S coordinate ranges from 0 to 1, from one end of the bounding box to the other. The T coordinate ranges between 0 and the ratio of the second greatest dimension of the bounding box to the greatest dimension. Figure 6.10 illustrates the default texture coordinates for a simple box shaped IndexedFaceSet with an X dimension twice as large as the Z dimension and four times as large as the Y dimension. Figure 6.11 illustrates the original texture image used on the IndexedFaceSet used in Figure 6.10.

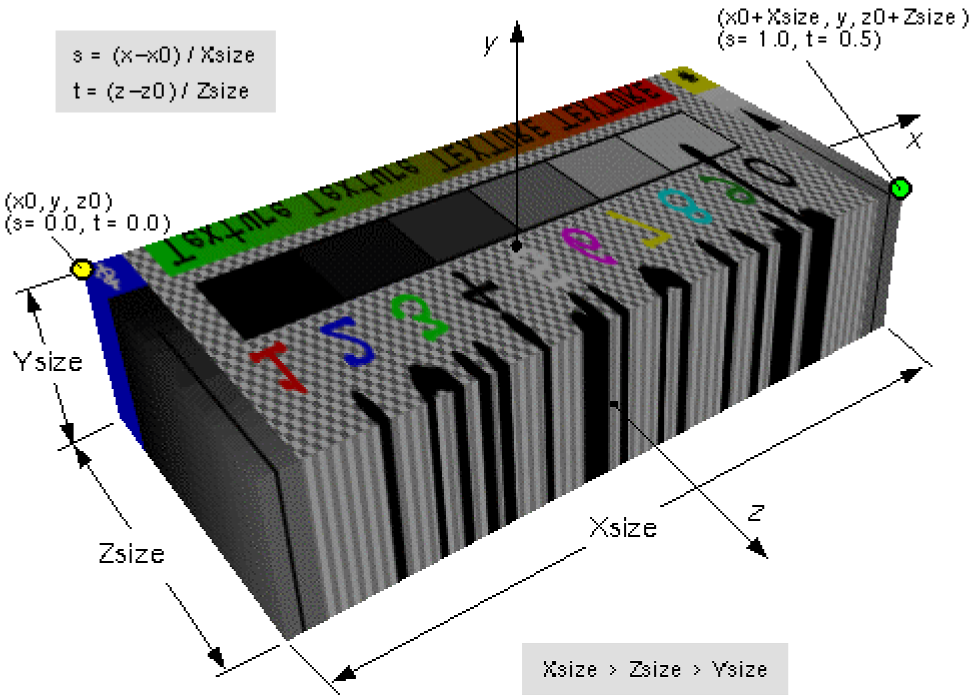


**Figure 6.10 -- IndexedFaceSet texture default mapping**

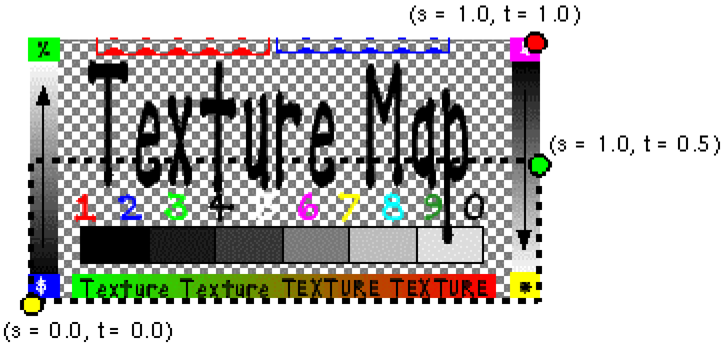


**Figure 6.11 -- ImageTexture for IndexedFaceSet in Figure 6.10**

Subclause 4.6.3, Shapes and geometry, provides a description of the *ccw*, *solid*, *convex*, and *creaseAngle* fields.

# 6.24 IndexedLineSet

```
IndexedLineSet {
  eventIn       MFInt32 set_colorIndex
  eventIn       MFInt32 set_coordIndex
  exposedField  SFNode  color             NULL
  exposedField  SFNode  coord             NULL
  field         MFInt32 colorIndex        []      # [-1,∞)
  field         SFBool  colorPerVertex    TRUE
  field         MFInt32 coordIndex        []      # [-1,∞)
}
```

The IndexedLineSet node represents a 3D geometry formed by constructing polylines from 3D vertices specified in the *coord* field. IndexedLineSet uses the indices in its *coordIndex* field to specify the polylines by connecting vertices from the *coord* field. An index of "-1" indicates that the current polyline has ended and the next one begins. The last polyline may be (but does not have to be) followed by a "-1". IndexedLineSet is specified in the local coordinate system and is affected by the transformations of its ancestors.

The *coord* field specifies the 3D vertices of the line set and contains a Coordinate node.

Lines are not lit, are not texture-mapped, and do not participate in collision detection. The width of lines is implementation dependent and each line segment is solid (i.e., not dashed).

If the *color* field is not NULL, it shall contain a Color node. The colours are applied to the line(s) as follows:

a. If *colorPerVertex* is FALSE:

  1. If the *colorIndex* field is not empty, one colour is used for each polyline of the IndexedLineSet. There shall be at least as many indices in the *colorIndex* field as there are polylines in the IndexedLineSet. If the greatest index in the *colorIndex* field is N, there shall be N+1 colours in the Color node. The *colorIndex* field shall not contain any negative entries.

  2. If the *colorIndex* field is empty, the colours from the Color node are applied to each polyline of the IndexedLineSet in order. There shall be at least as many colours in the Color node as there are polylines.

b. If *colorPerVertex* is TRUE:

  1. If the *colorIndex* field is not empty, colours are applied to each vertex of the IndexedLineSet in exactly the same manner that the *coordIndex* field is used to supply coordinates for each vertex from the Coordinate node. The *colorIndex* field shall contain at least as many indices as the *coordIndex* field and shall contain end-of-polyline markers (-1) in exactly the same places as the *coordIndex* field. If the greatest index in the *colorIndex* field is N, there shall be N+1 colours in the Color node.

  2. If the *colorIndex* field is empty, the *coordIndex* field is used to choose colours from the Color node. If the greatest index in the *coordIndex* field is N, there shall be N+1 colours in the Color node.

If the *color* field is NULL and there is a Material defined for the Appearance affecting this IndexedLineSet, the *emissiveColor* of the Material shall be used to draw the lines. Details on lighting equations as they affect IndexedLineSet nodes are described in 4.14, Lighting model.

APPENDIX KK

## 6.25 Inline

```
Inline {
  exposedField MFString url        []
  field        SFVec3f  bboxCenter 0 0 0     # (-∞,∞)
  field        SFVec3f  bboxSize   -1 -1 -1  # (0,∞) or -1,-1,-1
}
```

The Inline node is a grouping node that reads its children data from a location in the World Wide Web. Exactly when its children are read and displayed is not defined (e.g. reading the children may be delayed until the Inline node's bounding box is visible to the viewer). The *url* field specifies the URL containing the children. An Inline node with an empty URL does nothing.

Each specified URL shall refer to a valid VRML file that contains a list of children nodes, prototypes, and routes at the top level as described in 4.6.5, Grouping and children nodes. The results are undefined if the URL refers to a file that is not VRML or if the VRML file contains non-children nodes at the top level.

If multiple URLs are specified, the browser may display a URL of a lower preference VRML file while it is obtaining, or if it is unable to obtain, the higher preference VRML file. Details on the *url* field and preference order can be found in 4.5, VRML and the World Wide Web.

The results are undefined if the contents of the URL change after it has been loaded.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Inline node's children. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. A default *bboxSize* value, (-1, -1, -1), implies that the bounding box is not specified and if needed shall be calculated by the browser. A description of the *bboxCenter* and *bboxSize* fields is in 4.6.4, Bounding boxes.

## 6.26 LOD

```
LOD {
  exposedField MFNode  level    []
  field        SFVec3f center   0 0 0    # (-∞,∞)
  field        MFFloat range    []       # (0,∞)
}
```

The LOD node specifies various levels of detail or complexity for a given object, and provides hints allowing browsers to automatically choose the appropriate version of the object based on the distance from the user. The *level* field contains a list of nodes that represent the same object or objects at varying levels of detail, ordered from highest level of detail to the lowest level of detail. The *range* field specifies the ideal distances at which to switch between the levels. Subclause 4.6.5, Grouping and children nodes, contains details on the types of nodes that are legal values for *level*.

The *center* field is a translation offset in the local coordinate system that specifies the centre of the LOD node for distance calculations.

The number of nodes in the *level* field shall exceed the number of values in the *range* field by one (i.e., N+1 *level* values for N *range* values). The *range* field contains monotonic increasing values that shall be greater than zero. In order to calculate which level to display, first the distance is calculated from the viewer's location, transformed into the local coordinate system of the LOD node (including any scaling transformations), to the *center* point of the LOD node. Then, the LOD node evaluates the step function *L(d)* to choose a level for a given value of *d* (where *d* is the distance from the viewer position to the centre of the LOD node).

Let *n* ranges, $R_0$, $R_1$, $R_2$, ..., $R_{n-1}$, partition the domain (0, +*infinity*) into *n*+1 subintervals given by (0, $R_0$), [$R_0$, $R_1$)... , [$R_{n-1}$, +*infinity*). Also, let *n* levels $L_0$, $L_1$, $L_2$, ..., $L_{n-1}$ be the values of the step function function *L(d)*. The level node, *L(d)*, for a given distance *d* is defined as follows:

$$\begin{aligned} L(d) &= L_0, && \text{if } d < R_0,\\ &= L_{i+1}, && \text{if } R_i \le d < R_{i+1}, \text{ for } -1 < i < n-1,\\ &= L_{n-1}, && \text{if } d \ge R_{n-1}. \end{aligned}$$

Specifying too few levels will result in the last level being used repeatedly for the lowest levels of detail. If more levels than ranges are specified, the extra levels are ignored. An empty range field is an exception to this rule. This case is a hint to the browser that it may choose a level automatically to maintain a constant display rate. Each value in the *range* field shall be greater than the previous value.

LOD nodes are evaluated top-down in the scene graph. Only the descendants of the currently selected level are rendered. All nodes under an LOD node continue to receive and send events regardless of which LOD node's *level* is active. For example, if an active TimeSensor node is contained within an inactive level of an LOD node, the TimeSensor node sends events regardless of the LOD node's state.

# 6.27 Material

```
Material {
  exposedField SFFloat ambientIntensity  0.2         # [0,1]
  exposedField SFColor diffuseColor      0.8 0.8 0.8 # [0,1]
  exposedField SFColor emissiveColor     0 0 0       # [0,1]
  exposedField SFFloat shininess         0.2         # [0,1]
  exposedField SFColor specularColor     0 0 0       # [0,1]
  exposedField SFFloat transparency      0           # [0,1]
}
```

The Material node specifies surface material properties for associated geometry nodes and is used by the VRML lighting equations during rendering. Subclause 4.14, Lighting model, contains a detailed description of the VRML lighting model equations.

All of the fields in the Material node range from 0.0 to 1.0.

The fields in the Material node determine how light reflects off an object to create colour:

a. The *ambientIntensity* field specifies how much ambient light from light sources this surface shall reflect. Ambient light is omnidirectional and depends only on the number of light sources, not their positions with respect to the surface. Ambient colour is calculated as *ambientIntensity* × *diffuseColor*.

b. The *diffuseColor* field reflects all VRML light sources depending on the angle of the surface with respect to the light source. The more directly the surface faces the light, the more diffuse light reflects.

c. The *emissiveColor* field models "glowing" objects. This can be useful for displaying pre-lit models (where the light energy of the room is computed explicitly), or for displaying scientific data.

d. The *specularColor* and *shininess* fields determine the specular highlights (e.g., the shiny spots on an apple). When the angle from the light to the surface is close to the angle from the surface to the viewer, the *specularColor* is added to the diffuse and ambient colour calculations. Lower shininess values produce soft glows, while higher values result in sharper, smaller highlights.

e. The *transparency* field specifies how "clear" an object is, with 1.0 being completely transparent, and 0.0 completely opaque.

# 6.28 MovieTexture

```
MovieTexture {
  exposedField SFBool   loop             FALSE
  exposedField SFFloat  speed            1.0      # (-∞,∞)
  exposedField SFTime   startTime        0        # (-∞,∞)
  exposedField SFTime   stopTime         0        # (-∞,∞)
  exposedField MFString url              []
  field        SFBool   repeatS          TRUE
  field        SFBool   repeatT          TRUE
  eventOut     SFTime   duration_changed
  eventOut     SFBool   isActive
}
```

The MovieTexture node defines a time dependent texture map (contained in a movie file) and parameters for controlling the movie and the texture mapping. A MovieTexture node can also be used as the source of sound data for a Sound node. In this special case, the MovieTexture node is not used for rendering.

Texture maps are defined in a 2D coordinate system (s, t) that ranges from 0.0 to 1.0 in both directions. The bottom edge of the image corresponds to the S-axis of the texture map, and left edge of the image corresponds to the T-axis of the texture map. The lower-left pixel of the image corresponds to s=0.0, t=0.0, and the top-right pixel of the image corresponds to s=1.0, t=1.0. Figure 6.12 depicts the texture map coordinate system of the MovieTexture.

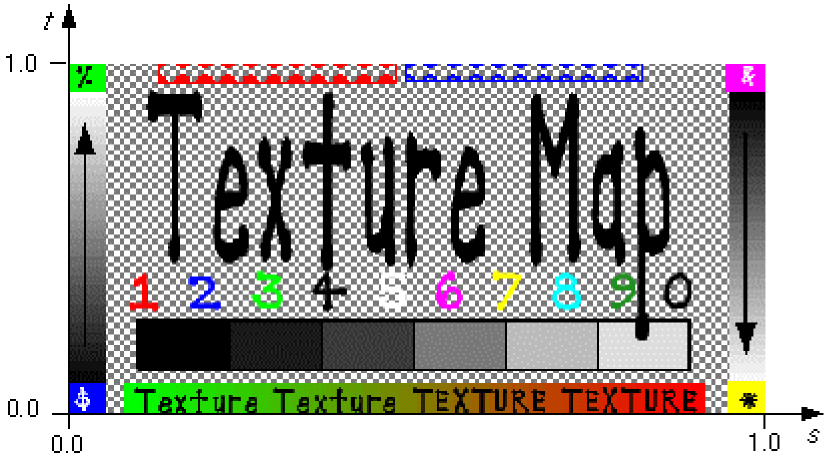


**Figure 6.12 -- MovieTexture node coordinate system**

The *url* field that defines the movie data shall support MPEG1-Systems (audio and video) or MPEG1-Video (video-only) movie file formats 2.[MPEG]. Details on the *url* field can be found in 4.5, VRML and the World Wide Web.

MovieTexture nodes can be referenced by an Appearance node's *texture* field (as a movie texture) and by a Sound node's *source* field (as an audio source only).

See 4.6.11, Texture maps, for a general description of texture maps.

4.14, Lighting model, contains details on lighting equations and the interaction between textures, materials, and geometries.

As soon as the movie is loaded, a *duration_changed* eventOut is sent. This indicates the duration of the movie in seconds. This eventOut value can be read (for instance, by a Script node) to determine the duration of a movie. A value of "-1" implies the movie has not yet loaded or the value is unavailable for some reason.

The *loop*, *startTime*, and *stopTime* exposedFields and the *isActive* eventOut, and their effects on the MovieTexture node, are discussed in detail in the 4.6.9, Time-dependent nodes, section. The cycle of a MovieTexture node is the length of time in seconds for one playing of the movie at the specified *speed*.

The *speed* exposedField indicates how fast the movie shall be played. A *speed* of 2 indicates the movie plays twice as fast. The *duration_changed* output is not affected by the *speed* exposedField. *set_speed* events are ignored while the movie is playing. A negative *speed* implies that the movie will play backwards.

If a MovieTexture node is inactive when the movie is first loaded, frame 0 of the movie texture is displayed if *speed* is non-negative or the last frame of the movie texture is shown if *speed* is negative (see 4.11.3, Discrete and continuous changes). A MovieTexture node shall display frame 0 if *speed* = 0. For positive values of *speed*, an active MovieTexture node displays the frame at movie time *t* as follows (i.e., in the movie's local time system with frame 0 at time 0 with *speed* = 1):

```
    t = (now - startTime) modulo (duration/speed)
```

If *speed* is negative, the MovieTexture node displays the frame at movie time:

```
    t = duration - ((now - startTime) modulo |duration/speed|)
```

When a MovieTexture node becomes inactive, the frame corresponding to the time at which the MovieTexture became inactive will remain as the texture.

## 6.29 NavigationInfo

```
NavigationInfo {
  eventIn      SFBool   set_bind
  exposedField MFFloat  avatarSize      [0.25, 1.6, 0.75] # [0,∞)
  exposedField SFBool   headlight       TRUE
  exposedField SFFloat  speed           1.0               # [0,∞)
  exposedField MFString type            ["WALK", "ANY"]
  exposedField SFFloat  visibilityLimit 0.0               # [0,∞)
  eventOut     SFBool   isBound
}
```

The NavigationInfo node contains information describing the physical characteristics of the viewer's avatar and viewing model. NavigationInfo node is a bindable node (see 4.6.10, Bindable children nodes*)*. Thus, there exists a NavigationInfo node stack in which the top-most NavigationInfo node on the stack is the currently bound NavigationInfo node. The current NavigationInfo node is considered to be a child of the current Viewpoint node regardless of where it is initially located in the VRML file. Whenever the current Viewpoint nodes changes, the current NavigationInfo node shall be re-parented to it by the browser. Whenever the current NavigationInfo node changes, the new NavigationInfo node shall be re-parented to the current Viewpoint node by the browser.

If a TRUE value is sent to the *set_bind* eventIn of a NavigationInfo node, the node is pushed onto the top of the NavigationInfo node stack. When a NavigationInfo node is bound, the browser uses the fields of the NavigationInfo node to set the navigation controls of its user interface and the NavigationInfo node is conceptually re-parented under the currently bound Viewpoint node. All subsequent scaling changes to the current Viewpoint node's coordinate system automatically change aspects (see below) of the NavigationInfo node values used in the browser (e.g., scale changes to any ancestors' transformations). A FALSE value sent to *set_bind* pops the NavigationInfo node from the stack, results in an *isBound* FALSE event, and pops to the next entry in the stack which shall be re-parented to the current Viewpoint node. 4.6.10, Bindable children nodes, has more details on binding stacks.

The *type* field specifies an ordered list of navigation paradigms that specify a combination of navigation types and the initial navigation type. The navigation type of the currently bound NavigationInfo node determines the user interface capabilities of the browser. For example, if the currently bound NavigationInfo node's *type* is "WALK", the browser shall present a WALK navigation user interface paradigm (see below for description of WALK). Browsers shall recognize and support at least the following navigation types: "ANY", "WALK", "EXAMINE", "FLY", and "NONE".

If "ANY" does not appear in the *type* field list of the currently bound NavigationInfo, the browser's navigation user interface shall be restricted to the recognized navigation types specified in the list. In this case, browsers shall not present a user interface that allows the navigation type to be changed to a type not specified in the list. However, if any one of the values in the *type* field are "ANY", the browser may provide any type of navigation interface, and allow the user to change the navigation type dynamically. Furthermore, the first recognized type in the list shall be the initial navigation type presented by the browser's user interface.

ANY navigation specifies that the browser may choose the navigation paradigm that best suits the content and provide a user interface to allow the user to change the navigation paradigm dynamically. The results are undefined if the currently bound NavigationInfo's *type* value is "ANY" and Viewpoint transitions (see 6.53, Viewpoint) are triggered by the Anchor node (see 6.2, Anchor) or the `loadURL()`scripting method (see 4.12.10, Browser script interface).

WALK navigation is used for exploring a virtual world on foot or in a vehicle that rests on or hovers above the ground. It is strongly recommended that WALK navigation define the up vector in the +Y direction and provide some form of terrain following and gravity in order to produce a walking or driving experience. If the bound NavigationInfo's *type* is "WALK", the browser shall strictly support collision detection (see 6.8, Collision).

FLY navigation is similar to WALK except that terrain following and gravity may be disabled or ignored. There shall still be some notion of "up" however. If the bound NavigationInfo's *type* is "FLY", the browser shall strictly support collision detection (see 6.8, Collision).

EXAMINE navigation is used for viewing individual objects and often includes (but does not require) the ability to spin around the object and move the viewer closer or further away.

NONE navigation disables and removes all browser-specific navigation user interface forcing the user to navigate using only mechanisms provided in the scene, such as Anchor nodes or scripts that include `loadURL()`.

If the NavigationInfo type is "WALK", "FLY", "EXAMINE", or "NONE" or a combination of these types (i.e., "ANY" is not in the list), Viewpoint transitions (see 6.53, Viewpoint) triggered by the Anchor node (see 6.2, Anchor) or the `loadURL()`scripting method (see 4.12.10, Browser script interface) shall be implemented as a jump cut from the old Viewpoint to the new Viewpoint with transition effects that shall not trigger events besides the exit and enter events caused by the jump.

Browsers may create browser-specific navigation type extensions. It is recommended that extended *type* names include a unique suffix (e.g., HELICOPTER_mydomain.com) to prevent conflicts. Viewpoint transitions (see 6.53, Viewpoint) triggered by the Anchor node (see 6.2, Anchor) or the `loadURL()`scripting method (see 4.12.10, Browser script interface) are undefined for extended navigation types. If none of the types are recognized by the browser, the default "ANY" is used. These strings values are case sensitive ("any" is not equal to "ANY").

The *speed* field specifies the rate at which the viewer travels through a scene in metres per second. Since browsers may provide mechanisms to travel faster or slower, this field specifies the default, average speed of the viewer when the NavigationInfo node is bound. If the NavigationInfo *type* is EXAMINE, *speed* shall not affect the viewer's rotational speed. Scaling in the transformation hierarchy of the currently bound Viewpoint node (see above) scales the *speed*; parent translation and rotation transformations have no effect on *speed*. Speed shall be non-negative. Zero speed indicates that the avatar's position is stationary, but its orientation and field of view may still change. If the navigation *type* is "NONE", the *speed* field has no effect.

The *avatarSize* field specifies the user's physical dimensions in the world for the purpose of collision detection and terrain following. It is a multi-value field allowing several dimensions to be specified. The first value shall be the allowable distance between the user's position and any collision geometry (as specified by a Collision node ) before a collision is detected. The second shall be the height above the terrain at which the browser shall maintain the viewer. The third shall be the height of the tallest object over which the viewer can move. This allows staircases to be built with dimensions that can be ascended by viewers in all browsers. The transformation hierarchy of the currently bound Viewpoint node scales the *avatarSize*. Translations and rotations have no effect on *avatarSize*.

For purposes of terrain following, the browser maintains a notion of the *down* direction (down vector), since gravity is applied in the direction of the down vector. This down vector shall be along the negative Y-axis in the local coordinate system of the currently bound Viewpoint node (i.e., the accumulation of the Viewpoint node's ancestors' transformations, not including the Viewpoint node's *orientation* field).

Geometry beyond the visibilityLimit may not be rendered. A value of 0.0 indicates an infinite visibility limit. The *visibilityLimit* field is restricted to be greater than or equal to zero.

The *speed*, *avatarSize* and *visibilityLimit* values are all scaled by the transformation being applied to the currently bound Viewpoint node. If there is no currently bound Viewpoint node, the values are interpreted in the world coordinate system. This allows these values to be automatically adjusted when binding to a Viewpoint node that has a scaling transformation applied to it without requiring a new NavigationInfo node to be bound as well. The results are undefined if the scale applied to the Viewpoint node is non-uniform.

The *headlight* field specifies whether a browser shall turn on a headlight. A headlight is a directional light that always points in the direction the user is looking. Setting this field to TRUE allows the browser to provide a headlight, possibly with user interface controls to turn it on and off. Scenes that enlist precomputed lighting (e.g., radiosity solutions) can turn the headlight off. The headlight shall have *intensity* = 1, *color* = (1 1 1), *ambientIntensity* = 0.0, and *direction* = (0 0 -1).

It is recommended that the near clipping plane be set to one-half of the collision radius as specified in the *avatarSize* field (setting the near plane to this value prevents excessive clipping of objects just above the collision volume, and also provides a region inside the collision volume for content authors to include geometry intended to remain fixed relative to the viewer). Such geometry shall not be occluded by geometry outside of the collision volume.

# 6.30 Normal

```
Normal {
  exposedField MFVec3f vector  []   # (-∞,∞)
}
```

This node defines a set of 3D surface normal vectors to be used in the *vector* field of some geometry nodes (e.g., IndexedFaceSet and ElevationGrid). This node contains one multiple-valued field that contains the normal vectors. Normals shall be of unit length.

# 6.31 NormalInterpolator

```
NormalInterpolator {
  eventIn      SFFloat set_fraction       # (-∞,∞)
  exposedField MFFloat key           []   # (-∞,∞)
  exposedField MFVec3f keyValue      []   # (-∞,∞)
  eventOut     MFVec3f value_changed
}
```

The NormalInterpolator node interpolates among a list of normal vector sets specified by the *keyValue* field. The output vector, *value_changed*, shall be a set of normalized vectors.

Values in the *keyValue* field shall be of unit length. The number of normals in the *keyValue* field shall be an integer multiple of the number of keyframes in the *key* field. That integer multiple defines how many normals will be contained in the *value_changed* events.

Normal interpolation shall be performed on the surface of the unit sphere. That is, the output values for a linear interpolation from a point P on the unit sphere to a point Q also on the unit sphere shall lie along the shortest arc (on the unit sphere) connecting points P and Q. Also, equally spaced input fractions shall result in arcs of equal length. The results are undefined if P and Q are diagonally opposite.

A more detailed discussion of interpolators is provided in 4.6.8, Interpolator nodes.

# 6.32 OrientationInterpolator

```
OrientationInterpolator {
  eventIn      SFFloat    set_fraction       # (-∞,∞)
  exposedField MFFloat    key           []   # (-∞,∞)
  exposedField MFRotation keyValue      []   # [-1,1], (-∞,∞)
  eventOut     SFRotation value_changed
}
```

The OrientationInterpolator node interpolates among a list of rotation values specified in the *keyValue* field. These rotations are absolute in object space and therefore are not cumulative. The *keyValue* field shall contain exactly as many rotations as there are keyframes in the *key* field.

An orientation represents the final position of an object after a rotation has been applied. An OrientationInterpolator interpolates between two orientations by computing the shortest path on the unit sphere between the two orientations. The interpolation is linear in arc length along this path. The results are undefined if the two orientations are diagonally opposite.

If two consecutive *keyValue* values exist such that the arc length between them is greater than π, the interpolation will take place on the arc complement. For example, the interpolation between the orientations (0, 1, 0, 0) and (0, 1, 0, 5.0) is equivalent to the rotation between the orientations (0, 1, 0, 2π) and (0, 1, 0, 5.0).

A more detailed discussion of interpolators is contained in 4.6.8, Interpolator nodes.

# 6.33 PixelTexture

```
PixelTexture {
  exposedField SFImage  image      0 0 0    # see 5.5, SFImage
  field        SFBool   repeatS    TRUE
  field        SFBool   repeatT    TRUE
}
```

The PixelTexture node defines a 2D image-based texture map as an explicit array of pixel values (*image* field) and parameters controlling tiling repetition of the texture onto geometry.

Texture maps are defined in a 2D coordinate system (s, t) that ranges from 0.0 to 1.0 in both directions. The bottom edge of the pixel image corresponds to the S-axis of the texture map, and left edge of the pixel image corresponds to the T-axis of the texture map. The lower-left pixel of the pixel image corresponds to s=0.0, t=0.0, and the top-right pixel of the image corresponds to s = 1.0, t = 1.0.

See 4.6.11, Texture maps, for a general description of texture maps. Figure 6.13 depicts an example PixelTexture.

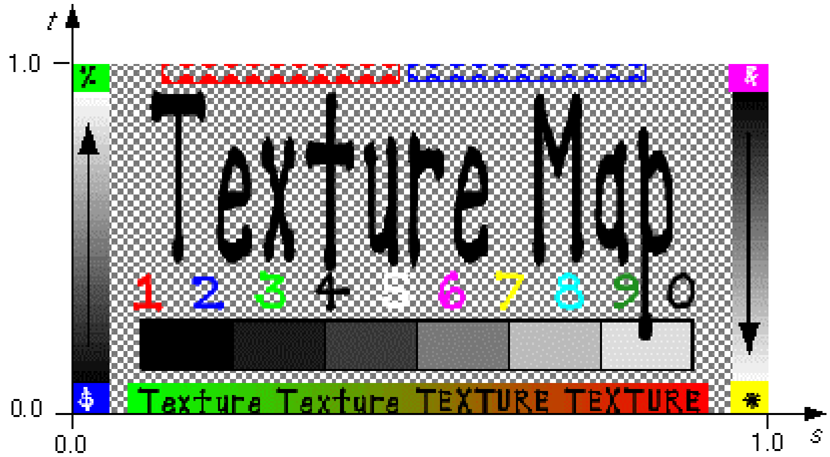


**Figure 6.13 -- PixelTexture node**

See 4.14 ,Lighting model, for a description of how the texture values interact with the appearance of the geometry. 5.5, SFImage, describes the specification of an image.

The *repeatS* and *repeatT* fields specify how the texture wraps in the S and T directions. If *repeatS* is TRUE (the default), the texture map is repeated outside the 0-to-1 texture coordinate range in the S direction so that it fills the shape. If *repeatS* is FALSE, the texture coordinates are clamped in the S direction to lie within the 0.0 to 1.0 range. The *repeatT* field is analogous to the *repeatS* field.

## 6.34 PlaneSensor

```
PlaneSensor {
  exposedField SFBool  autoOffset        TRUE
  exposedField SFBool  enabled           TRUE
  exposedField SFVec2f maxPosition       -1 -1     # (-∞,∞)
  exposedField SFVec2f minPosition       0 0       # (-∞,∞)
  exposedField SFVec3f offset            0 0 0     # (-∞,∞)
  eventOut     SFBool  isActive
  eventOut     SFVec3f trackPoint_changed
  eventOut     SFVec3f translation_changed
}
```

The PlaneSensor node maps pointing device motion into two-dimensional translation in a plane parallel to the Z=0 plane of the local coordinate system. The PlaneSensor node uses the descendent geometry of its parent node to determine whether it is liable to generate events.

The *enabled* exposedField enables and disables the PlaneSensor. If *enabled* is TRUE, the sensor reacts appropriately to user events. If *enabled* is FALSE, the sensor does not track user input or send events. If *enabled* receives a FALSE event and *isActive* is TRUE, the sensor becomes disabled and deactivated, and outputs an *isActive* FALSE event. If *enabled* receives a TRUE event, the sensor is enabled and made ready for user activation.

The PlaneSensor node generates events when the pointing device is activated while the pointer is indicating any descendent geometry nodes of the sensor's parent group. See 4.6.7.5, Activating and manipulating sensors, for details on using the pointing device to activate the PlaneSensor.

Upon activation of the pointing device (e.g., mouse button down) while indicating the sensor's geometry, an *isActive* TRUE event is sent. Pointer motion is mapped into relative translation in the *tracking plane*, (a plane parallel to the sensor's local Z=0 plane and coincident with the initial point of intersection). For each subsequent movement of the bearing, a *translation_changed* event is output which corresponds to the sum of the relative translation from the original intersection point to the intersection point of the new bearing in the plane plus the *offset* value. The sign of the translation is defined by the Z=0 plane of the sensor's coordinate system. *trackPoint_changed* events reflect the unclamped drag position on the surface of this plane. When the pointing device is deactivated and *autoOffset* is TRUE, *offset* is set to the last *translation_changed* value and an *offset_changed* event is generated. More details are provided in 4.6.7.4, Drag sensors.

When the sensor generates an *isActive* TRUE event, it grabs all further motion events from the pointing device until it is deactivated and generates an *isActive* FALSE event. Other pointing-device sensors shall not generate events during this time. Motion of the pointing device while *isActive* is TRUE is referred to as a "drag." If a 2D pointing device is in use, *isActive* events typically reflect the state of the primary button associated with the device (i.e., *isActive* is TRUE when the primary button is pressed, and is FALSE when it is released). If a 3D pointing device (e.g., wand) is in use, *isActive* events typically reflect whether the pointer is within or in contact with the sensor's geometry.

*minPosition* and *maxPosition* may be set to clamp *translation_changed* events to a range of values as measured from the origin of the Z=0 plane. If the X or Y component of *minPosition* is greater than the corresponding component of *maxPosition*, *translation_changed* events are not clamped in that dimension. If the X or Y component of *minPosition* is equal to the corresponding component of *maxPosition*, that component is constrained to the given value. This technique provides a way to implement a line sensor that maps dragging motion into a translation in one dimension.

While the pointing device is activated and moved, *trackPoint_changed* and *translation_changed* events are sent. *trackPoint_changed* events represent the unclamped intersection points on the surface of the tracking plane. If the pointing device is dragged off of the tracking plane while activated (e.g., above horizon line), browsers may interpret this in a variety ways (e.g., clamp all values to the horizon). Each movement of the pointing device, while *isActive* is TRUE, generates *trackPoint_changed* and *translation_changed* events.

Further information about this behaviour can be found in 4.6.7.3, Pointing-device sensors, 4.6.7.4, Drag sensors, and 4.6.7.5, Activating and manipulating sensors.

## 6.35 PointLight

```
PointLight {
  exposedField SFFloat ambientIntensity  0       # [0,1]
  exposedField SFVec3f attenuation       1 0 0   # [0,∞)
  exposedField SFColor color             1 1 1   # [0,1]
  exposedField SFFloat intensity         1       # [0,1]
  exposedField SFVec3f location          0 0 0   # (-∞,∞)
  exposedField SFBool  on                TRUE
  exposedField SFFloat radius            100     # [0,∞)
}
```

The PointLight node specifies a point light source at a 3D location in the local coordinate system. A point light source emits light equally in all directions; that is, it is omnidirectional. PointLight nodes are specified in the local coordinate system and are affected by ancestor transformations.

Subclause 4.6.6, Light sources, contains a detailed description of the *ambientIntensity*, *color*, and *intensity* fields.

A PointLight node illuminates geometry within *radius* metres of its *location*. Both radius and location are affected by ancestors' transformations (scales affect *radius* and transformations affect *location*). The *radius* field shall be greater than or equal to zero.

PointLight node's illumination falls off with distance as specified by three *attenuation* coefficients. The attenuation factor is $1/max(attenuation[0] + attenuation[1] \times r + attenuation[2] \times r^2, 1)$, where *r* is the distance from the light to the surface being illuminated. The default is no attenuation. An *attenuation* value of (0, 0, 0) is identical to (1, 0, 0). Attenuation values shall be greater than or equal to zero. A detailed description of VRML's lighting equations is contained in 4.14, Lighting model.

## 6.36 PointSet

```
PointSet {
  exposedField  SFNode  color      NULL
  exposedField  SFNode  coord      NULL
}
```

The PointSet node specifies a set of 3D points, in the local coordinate system, with associated colours at each point. The *coord* field specifies a Coordinate node (or instance of a Coordinate node). The results are undefined if the *coord* field specifies any other type of node. PointSet uses the coordinates in order. If the *coord* field is NULL, the point set is considered empty.

PointSet nodes are not lit, not texture-mapped, nor do they participate in collision detection. The size of each point is implementation-dependent.

If the *color* field is not NULL, it shall specify a Color node that contains at least the number of points contained in the *coord* node. The results are undefined if the *color* field specifies any other type of node. Colours shall be applied to each point in order. The results are undefined if the number of values in the Color node is less than the number of values specified in the Coordinate node.

If the *color* field is NULL and there is a Material node defined for the Appearance node affecting this PointSet node, the *emissiveColor* of the Material node shall be used to draw the points. More details on lighting equations can be found in 4.14, Lighting model.

## 6.37 PositionInterpolator

```
PositionInterpolator {
  eventIn      SFFloat set_fraction        # (-∞,∞)
  exposedField MFFloat key            []   # (-∞,∞)
  exposedField MFVec3f keyValue       []   # (-∞,∞)
  eventOut     SFVec3f value_changed
}
```

The PositionInterpolator node linearly interpolates among a list of 3D vectors. The *keyValue* field shall contain exactly as many values as in the *key* field.

4.6.8, Interpolator nodes, contains a more detailed discussion of interpolators.

## 6.38 ProximitySensor

```
ProximitySensor {
  exposedField SFVec3f    center      0 0 0   # (-∞,∞)
  exposedField SFVec3f    size        0 0 0   # [0,∞)
  exposedField SFBool     enabled     TRUE
  eventOut     SFBool     isActive
  eventOut     SFVec3f    position_changed
  eventOut     SFRotation orientation_changed
  eventOut     SFTime     enterTime
  eventOut     SFTime     exitTime
}
```

The ProximitySensor node generates events when the viewer enters, exits, and moves within a region in space (defined by a box). A proximity sensor is enabled or disabled by sending it an *enabled* event with a value of TRUE or FALSE. A disabled sensor does not send events.

A ProximitySensor node generates *isActive* TRUE/FALSE events as the viewer enters and exits the rectangular box defined by its *center* and *size* fields. Browsers shall interpolate viewer positions and timestamp the *isActive* events with the exact time the viewer first intersected the proximity region. The *center* field defines the centre point of the proximity region in object space. The *size* field specifies a vector which defines the width (x), height (y), and depth

(z) of the box bounding the region. The components of the *size* field shall be greater than or equal to zero. ProximitySensor nodes are affected by the hierarchical transformations of their parents.

The *enterTime* event is generated whenever the *isActive* TRUE event is generated (user enters the box), and *exitTime* events are generated whenever an *isActive* FALSE event is generated (user exits the box).

The *position_changed* and *orientation_changed* eventOuts send events whenever the user is contained within the proximity region and the position and orientation of the viewer changes with respect to the ProximitySensor node's coordinate system including enter and exit times. The viewer movement may be a result of a variety of circumstances resulting from browser navigation, ProximitySensor node's coordinate system changes, or bound Viewpoint node's position or orientation changes.

Each ProximitySensor node behaves independently of all other ProximitySensor nodes. Every enabled ProximitySensor node that is affected by the viewer's movement receives and sends events, possibly resulting in multiple ProximitySensor nodes receiving and sending events simultaneously. Unlike TouchSensor nodes, there is no notion of a ProximitySensor node lower in the scene graph "grabbing" events.

Instanced (DEF/USE) ProximitySensor nodes use the union of all the boxes to check for enter and exit. A multiply instanced ProximitySensor node will detect enter and exit for all instances of the box and send enter/exit events appropriately. However, the results are undefined if the any of the boxes of a multiply instanced ProximitySensor node overlap.

A ProximitySensor node that surrounds the entire world has an *enterTime* equal to the time that the world was entered and can be used to start up animations or behaviours as soon as a world is loaded. A ProximitySensor node with a box containing zero volume (i.e., any *size* field element of 0.0) cannot generate events. This is equivalent to setting the *enabled* field to FALSE.

A ProximitySensor read from a VRML file shall generate *isActive* TRUE, *position_changed*, *orientation_changed* and *enterTime* events if the sensor is enabled and the viewer is inside the proximity region. A ProximitySensor inserted into the transformation hierarchy shall generate *isActive* TRUE, *position_changed*, *orientation_changed* and *enterTime* events if the sensor is enabled and the viewer is inside the proximity region. A ProximitySensor removed from the transformation hierarchy shall generate *isActive* FALSE, *position_changed*, *orientation_changed* and *exitTime* events if the sensor is enabled and the viewer is inside the proximity region.

## 6.39 ScalarInterpolator

```
ScalarInterpolator {
  eventIn      SFFloat set_fraction          # (-∞,∞)
  exposedField MFFloat key           []      # (-∞,∞)
  exposedField MFFloat keyValue      []      # (-∞,∞)
  eventOut     SFFloat value_changed
}
```

This node linearly interpolates among a list of SFFloat values. This interpolator is appropriate for any parameter defined using a single floating point value. Examples include width, radius, and intensity fields. The *keyValue* field shall contain exactly as many numbers as there are keyframes in the *key* field.

A more detailed discussion of interpolators is available in 4.6.8, Interpolator nodes.

APPENDIX KK

# 6.40 Script

```
Script {
  exposedField MFString url           []
  field        SFBool   directOutput  FALSE
  field        SFBool   mustEvaluate  FALSE
  # And any number of:
  eventIn      eventType eventName
  field        fieldType fieldName initialValue
  eventOut     eventType eventName
}
```

The Script node is used to program behaviour in a scene. Script nodes typically

a. signify a change or user action;

b. receive events from other nodes;

c. contain a program module that performs some computation;

d. effect change somewhere else in the scene by sending events.

Each Script node has associated programming language code, referenced by the *url* field, that is executed to carry out the Script node's function. That code is referred to as the "script" in the rest of this description. Details on the *url* field can be found in 4.5, VRML and the World Wide Web.

Browsers are not required to support any specific language. Detailed information on scripting languages is described in 4.12, Scripting. Browsers supporting a scripting language for which a language binding is specified shall adhere to that language binding.

Sometime before a script receives the first event it shall be initialized (any language-dependent or user-defined `initialize()` is performed). The script is able to receive and process events that are sent to it. Each event that can be received shall be declared in the Script node using the same syntax as is used in a prototype definition:

```
eventIn type name
```

The *type* can be any of the standard VRML fields (as defined in 5, Field and event reference). *Name* shall be an identifier that is unique for this Script node.

The Script node is able to generate events in response to the incoming events. Each event that may be generated shall be declared in the Script node using the following syntax:

```
eventOut type name
```

With the exception of the *url* field, exposedFields are not allowed in Script nodes.

If the Script node's *mustEvaluate* field is FALSE, the browser may delay sending input events to the script until its outputs are needed by the browser. If the *mustEvaluate* field is TRUE, the browser shall send input events to the script as soon as possible, regardless of whether the outputs are needed. The *mustEvaluate* field shall be set to TRUE only if the Script node has effects that are not known to the browser (such as sending information across the network). Otherwise, poor performance may result.

Once the script has access to a VRML node (via an SFNode or MFNode value either in one of the Script node's fields or passed in as an eventIn), the script is able to read the contents of that node's exposed fields. If the Script node's *directOutput* field is TRUE, the script may also send events directly to any node to which it has access, and may dynamically establish or break routes. If *directOutput* is FALSE (the default), the script may only affect the rest of the world via events sent through its eventOuts. The results are undefined if *directOutput* is FALSE and the script sends events directly to a node to which it has access.

A script is able to communicate directly with the VRML browser to get information such as the current time and the current world URL. This is strictly defined by the API for the specific scripting language being used.

The location of the Script node in the scene graph has no affect on its operation. For example, if a parent of a Script node is a Switch node with *whichChoice* set to "-1" (i.e., ignore its children), the Script node continues to operate as specified (i.e., it receives and sends events).

## 6.41 Shape

```
Shape {
  exposedField SFNode appearance NULL
  exposedField SFNode geometry   NULL
}
```

The Shape node has two fields, *appearance* and *geometry,* which are used to create rendered objects in the world. The *appearance* field contains an Appearance node that specifies the visual attributes (e.g., material and texture) to be applied to the geometry. The *geometry* field contains a geometry node. The specified geometry node is rendered with the specified appearance nodes applied. See 4.6.3, Shapes and geometry, and 6.3, Appearance, for more information.

4.14, Lighting model, contains details of the VRML lighting model and the interaction between Appearance nodes and geometry nodes.

If the *geometry* field is NULL, the object is not drawn.

## 6.42 Sound

```
Sound {
  exposedField SFVec3f  direction     0 0 1   # (-∞,∞)
  exposedField SFFloat  intensity     1       # [0,1]
  exposedField SFVec3f  location      0 0 0   # (-∞,∞)
  exposedField SFFloat  maxBack       10      # [0,∞)
  exposedField SFFloat  maxFront      10      # [0,∞)
  exposedField SFFloat  minBack       1       # [0,∞)
  exposedField SFFloat  minFront      1       # [0,∞)
  exposedField SFFloat  priority      0       # [0,1]
  exposedField SFNode   source        NULL
  field        SFBool   spatialize    TRUE
}
```

The Sound node specifies the spatial presentation of a sound in a VRML scene. The sound is located at a point in the local coordinate system and emits sound in an elliptical pattern (defined by two ellipsoids). The ellipsoids are oriented in a direction specified by the *direction* field. The shape of the ellipsoids may be modified to provide more or less directional focus from the location of the sound.

The *source* field specifies the sound source for the Sound node. If the *source* field is not specified, the Sound node will not emit audio. The *source* field shall specify either an AudioClip node or a MovieTexture node. If a MovieTexture node is specified as the sound source, the MovieTexture shall refer to a movie format that supports sound (e.g., MPEG1-Systems, see 2.[MPEG]).

The *intensity* field adjusts the loudness (decibels) of the sound emitted by the Sound node (note: this is different from the traditional definition of intensity with respect to sound; see E.[SNDA]). The *intensity* field has a value that ranges from 0.0 to 1.0 and specifies a factor which shall be used to scale the normalized sample data of the sound source during playback. A Sound node with an intensity of 1.0 shall emit audio at its maximum loudness (before attenuation), and a Sound node with an intensity of 0.0 shall emit no audio. Between these values, the loudness should increase linearly from a -20 dB change approaching an *intensity* of 0.0 to a 0 dB change at an *intensity* of 1.0.

The *priority* field provides a hint for the browser to choose which sounds to play when there are more active Sound nodes than can be played at once due to either limited system resources or system load. 7.3.4, Sound priority, attenuation, and spatialization, describes a recommended algorithm for determining which sounds to play under such circumstances. The *priority* field ranges from 0.0 to 1.0, with 1.0 being the highest priority and 0.0 the lowest priority.

The *location* field determines the location of the sound emitter in the local coordinate system. A Sound node's output is audible only if it is part of the traversed scene. Sound nodes that are descended from LOD, Switch, or any grouping or prototype node that disables traversal (i.e., drawing) of its children are not audible unless they are traversed. If a Sound node is disabled by a Switch or LOD node, and later it becomes part of the traversal again, the sound shall resume where it would have been had it been playing continuously.

The Sound node has an inner ellipsoid that defines a volume of space in which the maximum level of the sound is audible. Within this ellipsoid, the normalized sample data is scaled by the *intensity* field and there is no attenuation. The inner ellipsoid is defined by extending the *direction* vector through the *location*. The *minBack* and *minFront* fields specify distances behind and in front of the *location* along the *direction* vector respectively. The inner ellipsoid has one of its foci at *location* (the second focus is implicit) and intersects the *direction* vector at *minBack* and *minFront*.

The Sound node has an outer ellipsoid that defines a volume of space that bounds the audibility of the sound. No sound can be heard outside of this outer ellipsoid. The outer ellipsoid is defined by extending the *direction* vector through the *location*. The *maxBack* and *maxFront* fields specify distances behind and in front of the *location* along the *direction* vector respectively. The outer ellipsoid has one of its foci at *location* (the second focus is implicit) and intersects the *direction* vector at *maxBack* and *maxFront*.

The *minFront*, *maxFront*, *minBack*, and *maxBack* fields are defined in local coordinates, and shall be greater than or equal to zero. The *minBack* field shall be less than or equal to *maxBack*, and *minFront* shall be less than or equal to *maxFront*. The ellipsoid parameters are specified in the local coordinate system but the ellipsoids' geometry is affected by ancestors' transformations.

Between the two ellipsoids, there shall be a linear attenuation ramp in loudness, from 0 dB at the minimum ellipsoid to -20 dB at the maximum ellipsoid:

```
attenuation = -20 × (d’ / d")
```

where d' is the distance along the location-to-viewer vector, measured from the transformed minimum ellipsoid boundary to the viewer, and d" is the distance along the location-to-viewer vector from the transformed minimum ellipsoid boundary to the transformed maximum ellipsoid boundary (see Figure 6.14).

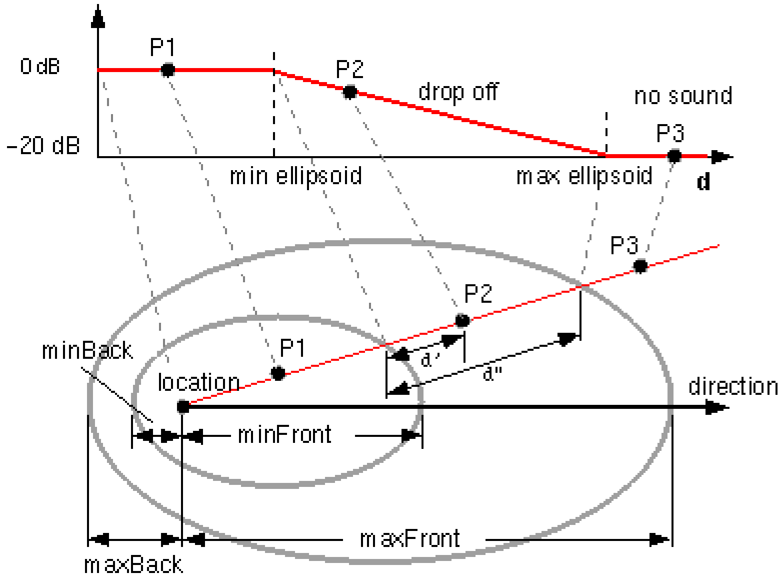


**Figure 6.14 -- Sound node geometry**

The *spatialize* field specifies if the sound is perceived as being directionally located relative to the viewer. If the *spatialize* field is TRUE and the viewer is located between the transformed inner and outer ellipsoids, the viewer's direction and the relative location of the Sound node should be taken into account during playback. Details outlining the minimum required spatialization functionality can be found in 7.3.4, Sound priority, attenuation, and spatialization. If the *spatialize* field is FALSE, then directional effects are ignored, but the ellipsoid dimensions and *intensity* will still affect the loudness of the sound. If the sound source is multi-channel (e.g., stereo), then the source should retain its channel separation during playback.

## 6.43 Sphere

```
Sphere {
  field SFFloat radius  1    # (0,∞)
}
```

The Sphere node specifies a sphere centred at (0, 0, 0) in the local coordinate system. The *radius* field specifies the radius of the sphere and shall be greater than zero. Figure 6.15 depicts the fields of the Sphere node.

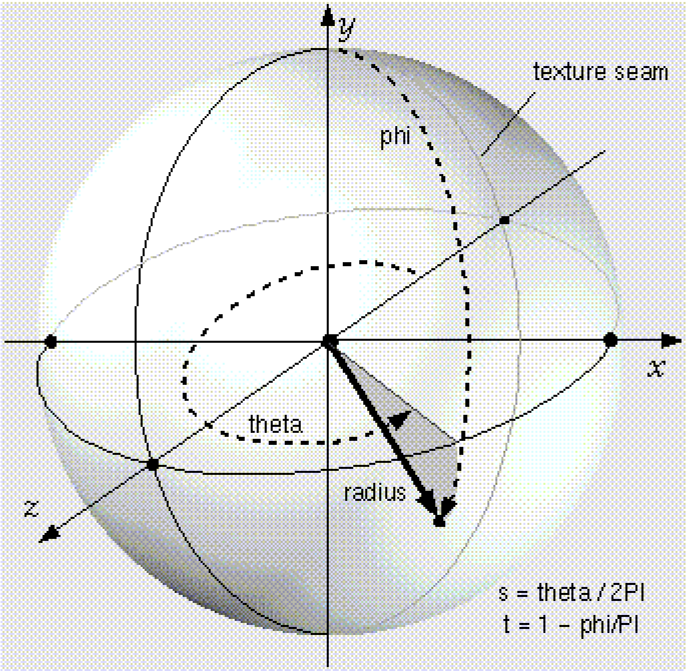


**Figure 6.15 -- Sphere node**

When a texture is applied to a sphere, the texture covers the entire surface, wrapping counterclockwise from the back of the sphere (i.e., longitudinal arc intersecting the -Z-axis) when viewed from the top of the sphere. The texture has a seam at the back where the X=0 plane intersects the sphere and Z values are negative. TextureTransform affects the texture coordinates of the Sphere.

The Sphere node's geometry requires outside faces only. When viewed from the inside the results are undefined.

## 6.44 SphereSensor

```
SphereSensor {
  exposedField SFBool     autoOffset        TRUE
  exposedField SFBool     enabled           TRUE
  exposedField SFRotation offset            0 1 0 0  # [-1,1], (-∞,∞)
  eventOut     SFBool     isActive
  eventOut     SFRotation rotation_changed
  eventOut     SFVec3f    trackPoint_changed
}
```

The SphereSensor node maps pointing device motion into spherical rotation about the origin of the local coordinate system. The SphereSensor node uses the descendent geometry of its parent node to determine whether it is liable to generate events.

The *enabled* exposed field enables and disables the SphereSensor node. If *enabled* is TRUE, the sensor reacts appropriately to user events. If *enabled* is FALSE, the sensor does not track user input or send events. If *enabled* receives a FALSE event and *isActive* is TRUE, the sensor becomes disabled and deactivated, and outputs an *isActive* FALSE event. If *enabled* receives a TRUE event the sensor is enabled and ready for user activation.

The SphereSensor node generates events when the pointing device is activated while the pointer is indicating any descendent geometry nodes of the sensor's parent group. See 4.6.7.5, Activating and manipulating sensors, for details on using the pointing device to activate the SphereSensor.

Upon activation of the pointing device (e.g., mouse button down) over the sensor's geometry, an *isActive* TRUE event is sent. The vector defined by the initial point of intersection on the SphereSensor's geometry and the local origin determines the radius of the sphere that is used to map subsequent pointing device motion while dragging. The virtual sphere defined by this radius and the local origin at the time of activation is used to interpret subsequent pointing device motion and is not affected by any changes to the sensor's coordinate system while the sensor is active. For each position of the bearing, a *rotation_changed* event is sent which corresponds to the sum of the relative rotation from the original intersection point plus the *offset* value. *trackPoint_changed* events reflect the unclamped drag position on the surface of this sphere. When the pointing device is deactivated and *autoOffset* is TRUE, *offset* is set to the last *rotation_changed* value and an *offset_changed* event is generated. See 4.6.7.4, Drag sensors, for more details.

When the sensor generates an *isActive* TRUE event, it grabs all further motion events from the pointing device until it is released and generates an *isActive* FALSE event (other pointing-device sensors shall not generate events during this time). Motion of the pointing device while *isActive* is TRUE is termed a "drag". If a 2D pointing device is in use, *isActive* events will typically reflect the state of the primary button associated with the device (i.e., *isActive* is TRUE when the primary button is pressed and FALSE when it is released). If a 3D pointing device (e.g., wand) is in use, *isActive* events will typically reflect whether the pointer is within (or in contact with) the sensor's geometry.

While the pointing device is activated, *trackPoint_changed* and *rotation_changed* events are output. *trackPoint_changed* events represent the unclamped intersection points on the surface of the invisible sphere. If the pointing device is dragged off the sphere while activated, browsers may interpret this in a variety of ways (e.g., clamp all values to the sphere or continue to rotate as the point is dragged away from the sphere). Each movement of the pointing device while *isActive* is TRUE generates *trackPoint_changed* and *rotation_changed* events.

Further information about this behaviour can be found in 4.6.7.3, Pointing-device sensors, 4.6.7.4, Drag sensors, and 4.6.7.5, Activating and manipulating sensors.

## 6.45 SpotLight

```
SpotLight {
  exposedField SFFloat ambientIntensity  0         # [0,1]
  exposedField SFVec3f attenuation       1 0 0     # [0,∞)
  exposedField SFFloat beamWidth         1.570796  # (0,π/2]
  exposedField SFColor color             1 1 1     # [0,1]
  exposedField SFFloat cutOffAngle       0.785398  # (0,π/2]
  exposedField SFVec3f direction         0 0 -1    # (-∞,∞)
  exposedField SFFloat intensity         1         # [0,1]
  exposedField SFVec3f location          0 0 0     # (-∞,∞)
  exposedField SFBool  on                TRUE
  exposedField SFFloat radius            100       # [0,∞)
}
```

The SpotLight node defines a light source that emits light from a specific point along a specific direction vector and constrained within a solid angle. Spotlights may illuminate geometry nodes that respond to light sources and intersect the solid angle defined by the SpotLight. Spotlight nodes are specified in the local coordinate system and are affected by ancestors' transformations.

A detailed description of *ambientIntensity*, *color*, *intensity*, and VRML's lighting equations is provided in 4.6.6, Light sources. More information on lighting concepts can be found in 4.14, Lighting model, including a detailed description of the VRML lighting equations.

The *location* field specifies a translation offset of the centre point of the light source from the light's local coordinate system origin. This point is the apex of the solid angle which bounds light emission from the given light source. The *direction* field specifies the direction vector of the light's central axis defined in the local coordinate system.

The *on* field specifies whether the light source emits light. If *on* is TRUE, the light source is emitting light and may illuminate geometry in the scene. If *on* is FALSE, the light source does not emit light and does not illuminate any geometry.

The *radius* field specifies the radial extent of the solid angle and the maximum distance from *location* that may be illuminated by the light source. The light source does not emit light outside this radius. The *radius* shall be greater than or equal to zero.

Both radius and location are affected by ancestors' transformations (scales affect *radius* and transformations affect *location*).

The *cutOffAngle* field specifies the outer bound of the solid angle. The light source does not emit light outside of this solid angle. The *beamWidth* field specifies an inner solid angle in which the light source emits light at uniform full intensity. The light source's emission intensity drops off from the inner solid angle (*beamWidth*) to the outer solid angle (*cutOffAngle*) as described in the following equations:

```
angle = the angle between the Spotlight’s direction vector
        and the vector from the Spotlight location to the point
        to be illuminated

if (angle >= cutOffAngle):
    multiplier = 0
else if (angle <= beamWidth):
    multiplier = 1
else:
    multiplier = (angle - cutOffAngle) / (beamWidth - cutOffAngle)

intensity(angle) = SpotLight.intensity × multiplier
```

If the *beamWidth* is greater than the *cutOffAngle*, *beamWidth* is defined to be equal to the *cutOffAngle* and the light source emits full intensity within the entire solid angle defined by *cutOffAngle*. Both *beamWidth* and *cutOffAngle* shall be greater than 0.0 and less than or equal to π/2. Figure 6.16 depicts the *beamWidth*, *cutOffAngle*, *direction*, *location*, and *radius* fields of the SpotLight node.

SpotLight illumination falls off with distance as specified by three *attenuation* coefficients. The attenuation factor is 1/*max(attenuation[0]* + *attenuation[1]* × *r* + *attenuation[2]* × $r^2$ , 1*)*, where *r* is the distance from the light to the surface being illuminated. The default is no attenuation. An *attenuation* value of (0, 0, 0) is identical to (1, 0, 0). Attenuation values shall be greater than or equal to zero. A detailed description of VRML's lighting equations is contained in 4.14, Lighting model.

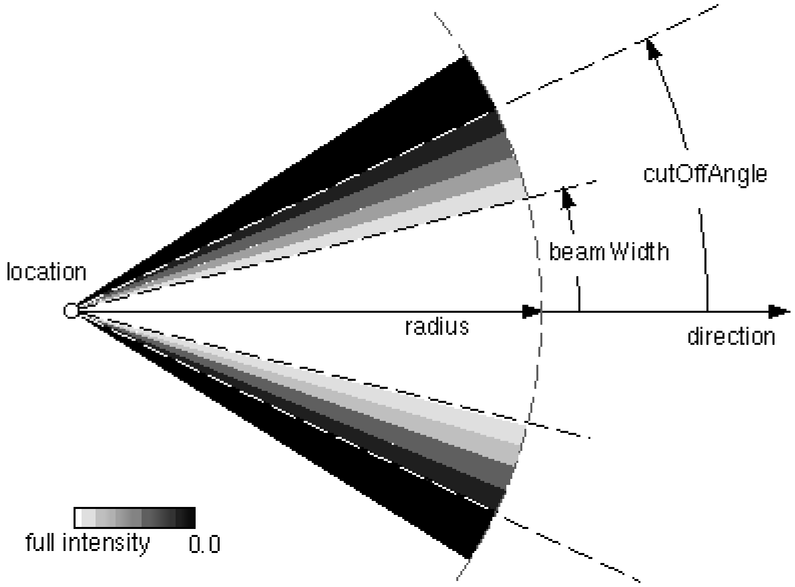


**Figure 6.16 -- SpotLight node**

# 6.46 Switch

```
Switch {
  exposedField    MFNode  choice      []
  exposedField    SFInt32 whichChoice -1    # [-1,∞)
}
```

The Switch grouping node traverses zero or one of the nodes specified in the *choice* field.

4.6.5, Grouping and children nodes, describes details on the types of nodes that are legal values for *choice*.

The *whichChoice* field specifies the index of the child to traverse, with the first child having index 0. If *whichChoice* is less than zero or greater than the number of nodes in the *choice* field, nothing is chosen.

All nodes under a Switch continue to receive and send events regardless of the value of *whichChoice*. For example, if an active TimeSensor is contained within an inactive choice of an Switch, the TimeSensor sends events regardless of the Switch's state.

# 6.47 Text

```
Text {
  exposedField  MFString string    []
  exposedField  SFNode   fontStyle NULL
  exposedField  MFFloat  length    []       # [0,∞)
  exposedField  SFFloat  maxExtent 0.0      # [0,∞)
}
```

## 6.47.1 Introduction

The Text node specifies a two-sided, flat text string object positioned in the Z=0 plane of the local coordinate system based on values defined in the fontStyle field (see 6.20, FontStyle). Text nodes may contain multiple text strings specified using the UTF-8 encoding as specified by ISO 10646-1:1993 (see 2.[UTF8]). The text strings are stored in the order in which the text mode characters are to be produced as defined by the parameters in the FontStyle node.

The text strings are contained in the *string* field. The *fontStyle* field contains one FontStyle node that specifies the font size, font family and style, direction of the text strings, and any specific language rendering techniques used for the text.

The *maxExtent* field limits and compresses all of the text strings if the length of the maximum string is longer than the maximum extent, as measured in the local coordinate system. If the text string with the maximum length is shorter than the *maxExtent*, then there is no compressing. The maximum extent is measured horizontally for horizontal text (FontStyle node: *horizontal*=TRUE) and vertically for vertical text (FontStyle node: *horizontal*=FALSE). The *maxExtent* field shall be greater than or equal to zero.

The *length* field contains an MFFloat value that specifies the length of each text string in the local coordinate system. If the string is too short, it is stretched (either by scaling the text or by adding space between the characters). If the string is too long, it is compressed (either by scaling the text or by subtracting space between the characters). If a length value is missing (for example, if there are four strings but only three length values), the missing values are considered to be 0. The *length* field shall be greater than or equal to zero.

Specifying a value of 0 for both the *maxExtent* and *length* fields indicates that the string may be any length.

## 6.47.2 ISO 10646-1:1993 Character Encodings

Characters in ISO 10646 (see 2.[UTF8]) are encoded in multiple octets. Code space is divided into four units, as follows:

```
+-------------+------------+-----------+------------+
| Group-octet | Plane-octet | Row-octet | Cell-octet |
+-------------+------------+-----------+------------+
```

ISO 10646-1:1993 allows two basic forms for characters:

a. UCS-2 (Universal Coded Character Set-2). This form is also known as the Basic Multilingual Plane (BMP). Characters are encoded in the lower two octets (row and cell).

b. UCS-4 (Universal Coded Character Set-4). Characters are encoded in the full four octets.

In addition, three transformation formats (UCS Transformation Format or UTF) are accepted: UTF-7, UTF-8, and UTF-16. Each represents the nature of the transformation: 7-bit, 8-bit, or 16-bit. UTF-7 and UTF-16 are referenced in 2.[UTF8].

UTF-8 maintains transparency for all ASCII code values (0...127). It allows ASCII text (0x0..0x7F) to appear without any changes and encodes all characters from 0x80.. 0x7FFFFFFF into a series of six or fewer bytes.

If the most significant bit of the first character is 0, the remaining seven bits are interpreted as an ASCII character. Otherwise, the number of leading 1 bits indicates the number of bytes following. There is always a zero bit between the count bits and any data.

The first byte is one of the following. The X indicates bits available to encode the character:

```
 0XXXXXXX only one byte   0..0x7F (ASCII)
 110XXXXX two bytes       Maximum character value is 0x7FF
 1110XXXX three bytes     Maximum character value is 0xFFFF
 11110XXX four bytes      Maximum character value is 0x1FFFFF
 111110XX five bytes      Maximum character value is 0x3FFFFFF
 1111110X six bytes       Maximum character value is 0x7FFFFFFF
```

All following bytes have the format 10XXXXXX.

As a two byte example, the symbol for a register trade mark is &REG; or 174 in ISO Latin-1 (see 2.[I8859]). It is encoded as 0x00AE in UCS-2 of ISO 10646. In UTF-8, it has the following two byte encoding: 0xC2, 0xAE.

### 6.47.3 Appearance

Textures are applied to text as follows. The texture origin is at the origin of the first string, as determined by the justification. The texture is scaled equally in both S and T dimensions, with the font height representing 1 unit. S increases to the right, and T increases up.

4.14, Lighting model, has details on VRML lighting equations and how Appearance, Material and textures interact with lighting.

The Text node does not participate in collision detection.

## 6.48 TextureCoordinate

```
TextureCoordinate {
  exposedField MFVec2f point  []      # (-∞,∞)
}
```

The TextureCoordinate node specifies a set of 2D texture coordinates used by vertex-based geometry nodes (e.g., IndexedFaceSet and ElevationGrid) to map textures to vertices. Textures are two dimensional colour functions that, given an *(s, t)* coordinate, return a colour value *colour(s, t)*. Texture map values (ImageTexture, MovieTexture, and PixelTexture) range from [0.0, 1.0] along the S-axis and T-axis. However, TextureCoordinate values, specified by the *point* field, may be in the range (-∞,∞). Texture coordinates identify a location (and thus a colour value) in the texture map. The horizontal coordinate *s* is specified first, followed by the vertical coordinate *t*.

If the texture map is repeated in a given direction (S-axis or T-axis), a texture coordinate C (s or t) is mapped into a texture map that has N pixels in the given direction as follows:

```
   Texture map location = (C - floor(C)) × N
```

If the texture map is not repeated, the texture coordinates are clamped to the 0.0 to 1.0 range as follows:

APPENDIX KK

```
    Texture map location = N,     if C > 1.0,
                         = 0.0,   if C < 0.0,
                         = C × N, if 0.0 <= C <= 1.0.
```

Details on repeating textures are specific to texture map node types described in 6.22, ImageTexture, 6.28, MovieTexture, and 6.33, PixelTexture.

## 6.49 TextureTransform

```
TextureTransform {
  exposedField SFVec2f center      0 0     # (-∞,∞)
  exposedField SFFloat rotation    0       # (-∞,∞)
  exposedField SFVec2f scale       1 1     # (-∞,∞)
  exposedField SFVec2f translation 0 0     # (-∞,∞)
}
```

The TextureTransform node defines a 2D transformation that is applied to texture coordinates (see 6.48, TextureCoordinate). This node affects the way textures coordinates are applied to the geometric surface. The transformation consists of (in order):

a. a translation;

b. a rotation about the centre point;

c. a non-uniform scale about the centre point.

These parameters support changes to the size, orientation, and position of textures on shapes. Note that these operations appear reversed when viewed on the surface of geometry. For example, a *scale* value of (2 2) will scale the texture coordinates and have the net effect of shrinking the texture size by a factor of 2 (texture coordinates are twice as large and thus cause the texture to repeat). A translation of (0.5 0.0) translates the texture coordinates +.5 units along the S-axis and has the net effect of translating the texture -0.5 along the S-axis on the geometry's surface. A rotation of π/2 of the texture coordinates results in a -π/2 rotation of the texture on the geometry.

The *center* field specifies a translation offset in texture coordinate space about which the *rotation* and *scale* fields are applied. The *scale* field specifies a scaling factor in S and T of the texture coordinates about the *center* point. *scale* values shall be in the range (-∞,∞). The *rotation* field specifies a rotation in radians of the texture coordinates about the *center* point after the scale has been applied. A positive rotation value makes the texture coordinates rotate counterclockwise about the centre, thereby rotating the appearance of the texture itself clockwise. The *translation* field specifies a translation of the texture coordinates.

In matrix transformation notation, where *Tc* is the untransformed texture coordinate, *Tc'* is the transformed texture coordinate, *C* (*center*), *T* (*translation*), *R* (*rotation*), and *S* (*scale*) are the intermediate transformation matrices,

```
    Tc' = -C × S × R × C × T × Tc
```

Note that this transformation order is the reverse of the Transform node transformation order since the texture coordinates, not the texture, are being transformed (i.e., the texture coordinate system).

# 6.50 TimeSensor

```
TimeSensor {
  exposedField SFTime   cycleInterval 1       # (0,∞)
  exposedField SFBool   enabled       TRUE
  exposedField SFBool   loop          FALSE
  exposedField SFTime   startTime     0       # (-∞,∞)
  exposedField SFTime   stopTime      0       # (-∞,∞)
  eventOut     SFTime   cycleTime
  eventOut     SFFloat  fraction_changed      # [0, 1]
  eventOut     SFBool   isActive
  eventOut     SFTime   time
}
```

TimeSensor nodes generate events as time passes. TimeSensor nodes can be used for many purposes including:

a. driving continuous simulations and animations;

b. controlling periodic activities (*e.g.*, one per minute);

c. initiating single occurrence events such as an alarm clock.

The TimeSensor node contains two discrete eventOuts: *isActive* and *cycleTime*. The *isActive* eventOut sends TRUE when the TimeSensor node begins running, and FALSE when it stops running. The *cycleTime* eventOut sends a time event at *startTime* and at the beginning of each new cycle (useful for synchronization with other time-based objects). The remaining eventOuts generate continuous events. The *fraction_changed* eventOut, an SFFloat in the closed interval [0,1], sends the completed fraction of the current cycle. The *time* eventOut sends the absolute time for a given *simulation tick.*

If the *enabled* exposedField is TRUE, the TimeSensor node is enabled and may be running. If a *set_enabled* FALSE event is received while the TimeSensor node is running, the sensor performs the following actions:

d. evaluates and sends all relevant outputs;

e. sends a FALSE value for *isActive*;

f. disables itself.

Events on the exposedFields of the TimeSensor node (e.g., *set_startTime)* are processed and their corresponding eventOuts (e.g., *startTime_changed)* are sent regardless of the state of the *enabled* field. The remaining discussion assumes *enabled* is TRUE.

The *loop*, *startTime*, and *stopTime* exposedFields and the *isActive* eventOut and their effects on the TimeSensor node are discussed in detail in 4.6.9, Time-dependent nodes. The "cycle" of a TimeSensor node lasts for *cycleInterval* seconds. The value of *cycleInterval* shall be greater than zero.

A *cycleTime* eventOut can be used for synchronization purposes such as sound with animation. The value of a *cycleTime* eventOut will be equal to the time at the beginning of the current cycle. A *cycleTime* eventOut is generated at the beginning of every cycle, including the cycle starting at *startTime*. The first *cycleTime* eventOut for a TimeSensor node can be used as an alarm (single pulse at a specified time).

When a TimeSensor node becomes active, it generates an *isActive* = TRUE event and begins generating *time*, *fraction_changed*, and *cycleTime* events which may be routed to other nodes to drive animation or simulated behaviours. The behaviour at read time is described below. The *time* event sends the absolute time for a given tick of the TimeSensor node (time fields and events represent the number of seconds since midnight GMT January 1, 1970).

*fraction_changed* events output a floating point value in the closed interval [0, 1]. At *startTime* the value of *fraction_changed* is 0. After *startTime*, the value of *fraction_changed* in any cycle will progress through the range (0.0, 1.0]. At *startTime* + N × *cycleInterval*, for N = 1, 2, ..., that is, at the end of every cycle, the value of *fraction_changed* is 1.

Let *now* represent the time at the current simulation tick. Then the *time* and *fraction_changed* eventOuts can then be computed as:

```
time = now
temp = (now - startTime) / cycleInterval
f    = fractionalPart(temp)
if (f == 0.0 && now > startTime) fraction_changed = 1.0
else fraction_changed = f
```

where `fractionalPart(x)` is a function that returns the fractional part, (that is, the digits to the right of the decimal point), of a nonnegative floating point number.

A TimeSensor node can be set up to be active at read time by specifying *loop* TRUE (not the default) and *stopTime* less than or equal to *startTime* (satisfied by the default values). The *time* events output absolute times for each tick of the TimeSensor node simulation. The *time* events shall start at the first simulation tick greater than or equal to *startTime*. *time* events end at *stopTime*, or at *startTime* + *N* × *cycleInterval* for some positive integer value of *N*, or loop forever depending on the values of the other fields. An active TimeSensor node shall stop at the first simulation tick when *now* >= *stopTime* > *startTime*.

No guarantees are made with respect to how often a TimeSensor node generates time events, but a TimeSensor node shall generate events at least at every simulation tick. TimeSensor nodes are guaranteed to generate final *time* and *fraction_changed* events. If loop is FALSE at the end of the *N*th cycleInterval and was TRUE at *startTime* + *M* × *cycleInterval* for all *0* < *M* < *N*, the final *time* event will be generated with a value of (*startTime* + *N* × *cycleInterval*) or *stopTime* (if *stopTime* > *startTime),* whichever value is less. If *loop* is TRUE at the completion of every cycle, the final event is generated as evaluated at *stopTime* (if *stopTime* > *startTime)* or never.

An active TimeSensor node ignores *set_cycleInterval* and *set_startTime* events. An active TimeSensor node also ignores *set_stopTime* events for *set_stopTime* less than or equal to *startTime*. For example, if a *set_startTime* event is received while a TimeSensor node is active, that *set_startTime* event is ignored (the *startTime* field is not changed, and a *startTime_changed* eventOut is not generated). If an active TimeSensor node receives a *set_stopTime* event that is less than the current time, and greater than *startTime*, it behaves as if the *stopTime* requested is the current time and sends the final events based on the current time (note that *stopTime* is set as specified in the eventIn).

A TimeSensor read from a VRML file shall generate *isActive* TRUE, *time* and *fraction_changed* events if the sensor is enabled and all conditions for a TimeSensor to be active are met.

# 6.51 TouchSensor

```
TouchSensor {
  exposedField SFBool  enabled TRUE
  eventOut     SFVec3f hitNormal_changed
  eventOut     SFVec3f hitPoint_changed
  eventOut     SFVec2f hitTexCoord_changed
  eventOut     SFBool  isActive
  eventOut     SFBool  isOver
  eventOut     SFTime  touchTime
}
```

A TouchSensor node tracks the location and state of the pointing device and detects when the user points at geometry contained by the TouchSensor node's parent group. A TouchSensor node can be enabled or disabled by sending it an *enabled* event with a value of TRUE or FALSE. If the TouchSensor node is disabled, it does not track user input or send events.

The TouchSensor generates events when the pointing device points toward any geometry nodes that are descendants of the TouchSensor's parent group. See 4.6.7.5, Activating and manipulating sensors, for more details on using the pointing device to activate the TouchSensor.

The *isOver* eventOut reflects the state of the pointing device with regard to whether it is pointing towards the TouchSensor node's geometry or not. When the pointing device changes state from a position such that its bearing does not intersect any of the TouchSensor node's geometry to one in which it does intersect geometry, an *isOver* TRUE event is generated. When the pointing device moves from a position such that its bearing intersects geometry to one in which it no longer intersects the geometry, or some other geometry is obstructing the TouchSensor node's geometry, an *isOver* FALSE event is generated. These events are generated only when the pointing device has moved and changed `over' state. Events are not generated if the geometry itself is animating and moving underneath the pointing device.

As the user moves the bearing over the TouchSensor node's geometry, the point of intersection (if any) between the bearing and the geometry is determined. Each movement of the pointing device, while *isOver* is TRUE, generates *hitPoint_changed*, *hitNormal_changed* and *hitTexCoord_changed* events. *hitPoint_changed* events contain the 3D point on the surface of the underlying geometry, given in the TouchSensor node's coordinate system. *hitNormal_changed* events contain the surface normal vector at the *hitPoint*. *hitTexCoord_changed* events contain the texture coordinates of that surface at the *hitPoint*. The values of *hitTexCoord_changed* and *hitNormal_changed* events are computed as appropriate for the associated shape.

If *isOver* is TRUE, the user may activate the pointing device to cause the TouchSensor node to generate *isActive* events (e.g., by pressing the primary mouse button). When the TouchSensor node generates an *isActive* TRUE event, it grabs all further motion events from the pointing device until it is released and generates an *isActive* FALSE event (other pointing-device sensors will not generate events during this time). Motion of the pointing device while *isActive* is TRUE is termed a "drag." If a 2D pointing device is in use, *isActive* events reflect the state of the primary button associated with the device (i.e., *isActive* is TRUE when the primary button is pressed and FALSE when it is released). If a 3D pointing device is in use, *isActive* events will typically reflect whether the pointing device is within (or in contact with) the TouchSensor node's geometry.

The eventOut field *touchTime* is generated when all three of the following conditions are true:

a. The pointing device was pointing towards the geometry when it was initially activated (*isActive* is TRUE).

b. The pointing device is currently pointing towards the geometry (*isOver* is TRUE).

c. The pointing device is deactivated (*isActive* FALSE event is also generated).

More information about this behaviour is described in 4.6.7.3, Pointing-device sensors, 4.6.7.4, Drag sensors, and 4.6.7.5, Activating and manipulating sensors.

# 6.52 Transform

```
Transform {
  eventIn      MFNode      addChildren
  eventIn      MFNode      removeChildren
  exposedField SFVec3f     center           0 0 0    # (-∞,∞)
  exposedField MFNode      children         []
  exposedField SFRotation  rotation         0 0 1 0  # [-1,1], (-∞,∞)
  exposedField SFVec3f     scale            1 1 1    # (0,∞)
  exposedField SFRotation  scaleOrientation 0 0 1 0  # [-1,1], (-∞,∞)
  exposedField SFVec3f     translation      0 0 0    # (-∞,∞)
  field        SFVec3f     bboxCenter       0 0 0    # (-∞,∞)
  field        SFVec3f     bboxSize         -1 -1 -1 # (0,∞) or -1,-1,-1
}
```

The Transform node is a grouping node that defines a coordinate system for its children that is relative to the coordinate systems of its ancestors. See 4.4.4, Transformation hierarchy, and 4.4.5, Standard units and coordinate system, for a description of coordinate systems and transformations.

4.6.5, Grouping and children nodes, provides a description of the *children*, *addChildren*, and *removeChildren* fields and eventIns.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the children of the Transform node. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. A default *bboxSize* value, (-1, -1, -1), implies that the bounding box is not specified and, if needed, shall be calculated by the browser. The bounding box shall be large enough at all times to enclose the union of the group's children's bounding boxes; it shall not include any transformations performed by the group itself (i.e., the bounding box is defined in the local coordinate system of the children). The results are undefined if the specified bounding box is smaller than the true bounding box of the group. A description of the *bboxCenter* and *bboxSize* fields is provided in 4.6.4, Bounding boxes.

The *translation*, *rotation*, *scale*, *scaleOrientation* and *center* fields define a geometric 3D transformation consisting of (in order):

a. a (possibly) non-uniform scale about an arbitrary point;

b. a rotation about an arbitrary point and axis;

c. a translation.

The *center* field specifies a translation offset from the origin of the local coordinate system (0,0,0). The *rotation* field specifies a rotation of the coordinate system. The *scale* field specifies a non-uniform scale of the coordinate system. *scale* values shall be greater than zero. The *scaleOrientation* specifies a rotation of the coordinate system before the scale (to specify scales in arbitrary orientations). The *scaleOrientation* applies only to the scale operation. The *translation* field specifies a translation to the coordinate system.

Given a 3-dimensional point **P** and Transform node, **P** is transformed into point **P'** in its parent's coordinate system by a series of intermediate transformations. In matrix transformation notation, where C (*center*), SR (*scaleOrientation*), T (*translation*), R (*rotation*), and S (*scale*) are the equivalent transformation matrices,

```
    P′ = T × C × R × SR × S × -SR × -C × P
```

The following Transform node:

```
Transform {
    center           C
    rotation         R
    scale            S
    scaleOrientation SR
    translation      T
    children         [...]
}
```

is equivalent to the nested sequence of:

```
Transform {
  translation T
  children Transform {
    translation C
    children Transform {
      rotation R
      children Transform {
        rotation SR
        children Transform {
          scale S
          children Transform {
            rotation -SR
            children Transform {
              translation -C
              children [...]
}}}}}}}
```

# 6.53 Viewpoint

```
Viewpoint {
  eventIn      SFBool     set_bind
  exposedField SFFloat    fieldOfView    0.785398  # (0,π)
  exposedField SFBool     jump           TRUE
  exposedField SFRotation orientation    0 0 1 0   # [-1,1], (-∞,∞)
  exposedField SFVec3f    position       0 0 10    # (-∞,∞)
  field        SFString   description    ""
  eventOut     SFTime     bindTime
  eventOut     SFBool     isBound
}
```

The Viewpoint node defines a specific location in the local coordinate system from which the user may view the scene. Viewpoint nodes are bindable children nodes (see 4.6.10, Bindable children nodes) and thus there exists a Viewpoint node stack in the browser in which the top-most Viewpoint node on the stack is the currently active Viewpoint node. If a TRUE value is sent to the *set_bind* eventIn of a Viewpoint node, it is moved to the top of the Viewpoint node stack and activated. When a Viewpoint node is at the top of the stack, the user's view is conceptually re-parented as a child of the Viewpoint node. All subsequent changes to the Viewpoint node's coordinate system change the user's view (e.g., changes to any ancestor transformation nodes or to the Viewpoint node's *position* or *orientation* fields). Sending a *set_bind* FALSE event removes the Viewpoint node from the stack and produces *isBound* FALSE and *bindTime* events. If the popped Viewpoint node is at the top of the viewpoint stack, the user's view is re-parented to the next entry in the stack. More details on binding stacks can be found in 4.6.10, Bindable children nodes. When a Viewpoint node is moved to the top of the stack, the existing top of stack Viewpoint node sends an *isBound* FALSE event and is pushed down the stack.

An author can automatically move the user's view through the world by binding the user to a Viewpoint node and then animating either the Viewpoint node or the transformations above it. Browsers shall allow the user view to be navigated relative to the coordinate system defined by the Viewpoint node (and the transformations above it) even if the Viewpoint node or its ancestors' transformations are being animated.

The *bindTime* eventOut sends the time at which the Viewpoint node is bound or unbound. This can happen:

a. during loading;

b. when a *set_bind* event is sent to the Viewpoint node;

c. when the browser binds to the Viewpoint node through its user interface described below.

The *position* and *orientation* fields of the Viewpoint node specify relative locations in the local coordinate system. *Position* is relative to the coordinate system's origin (0,0,0), while *orientation* specifies a rotation relative to the default orientation. In the default position and orientation, the viewer is on the Z-axis looking down the -Z-axis toward the origin with +X to the right and +Y straight up. Viewpoint nodes are affected by the transformation hierarchy.

Navigation types (see 6.29, NavigationInfo) that require a definition of a down vector (e.g., terrain following) shall use the negative Y-axis of the coordinate system of the currently bound Viewpoint node. Likewise, navigation types that require a definition of an up vector shall use the positive Y-axis of the coordinate system of the currently bound Viewpoint node. The *orientation* field of the Viewpoint node does not affect the definition of the down or up vectors. This allows the author to separate the viewing direction from the gravity direction.

The *jump* field specifies whether the user's view "jumps" to the position and orientation of a bound Viewpoint node or remains unchanged. This jump is instantaneous and discontinuous in that no collisions are performed and no ProximitySensor nodes are checked in between the starting and ending jump points. If the user's position before the jump is inside a ProximitySensor the *exitTime* of that sensor shall send the same timestamp as the bind eventIn. Similarly, if the user's position after the jump is inside a ProximitySensor the *enterTime* of that sensor shall send the same timestamp as the bind eventIn. Regardless of the value of *jump* at bind time, the relative viewing transformation between the user's view and the current Viewpoint node shall be stored with the current Viewpoint node for later use when *un-jumping* (i.e., popping the Viewpoint node binding stack from a Viewpoint node with *jump* TRUE*)*. The following summarizes the bind stack rules (see 4.6.10, Bindable children nodes) with additional rules regarding Viewpoint nodes (displayed in boldface type):

d. During read, the first encountered Viewpoint node is bound by pushing it to the top of the Viewpoint node stack. If a Viewpoint node name is specified in the URL that is being read, this named Viewpoint node is considered to be the first encountered Viewpoint node. Nodes contained within Inline nodes, within the strings passed to the Browser.createVrmlFromString() method, or within files passed to the Browser.createVrmlFromURL() method (see 4.12.10, Browser script interface) are not candidates for the first encountered Viewpoint node. The first node within a prototype instance is a valid candidate for the first encountered Viewpoint node. The first encountered Viewpoint node sends an *isBound* TRUE event.

e. When a *set_bind* TRUE event is received by a Viewpoint node,

   1. If it is not on the top of the stack: **The relative transformation from the current top of stack Viewpoint node to the user's view is stored with the current top of stack Viewpoint node.** The current top of stack node sends an *isBound* FALSE event. The new node is moved to the top of the stack and becomes the currently bound Viewpoint node. The new Viewpoint node (top of stack) sends an *isBound* TRUE event. **If *jump* is TRUE for the new Viewpoint node, the user's view is**

APPENDIX KK

**instantaneously "jumped" to match the values in the *position* and *orientation* fields of the new Viewpoint node.**

2. If the node is already at the top of the stack, this event has no affect.

f. When a *set_bind* FALSE event is received by a Viewpoint node in the stack, it is removed from the stack. If it was on the top of the stack,

1. it sends an *isBound* FALSE event,

2. the next node in the stack becomes the currently bound Viewpoint node (i.e., pop) and issues an *isBound* TRUE event,

3. **if its *jump* field value is TRUE, the user's view is instantaneously "jumped" to the *position* and *orientation* of the next Viewpoint node in the stack with the stored relative transformation of this next Viewpoint node applied.**

g. If a *set_bind* FALSE event is received by a node not in the stack, the event is ignored and *isBound* events are not sent.

h. When a node replaces another node at the top of the stack, the *isBound* TRUE and FALSE events from the two nodes are sent simultaneously (i.e., with identical timestamps).

i. If a bound node is deleted, it behaves as if it received a *set_bind* FALSE event (see c.).

The *jump* field may change after a Viewpoint node is bound. The rules described above still apply. If *jump* was TRUE when the Viewpoint node is bound, but changed to FALSE before the *set_bind* FALSE is sent, the Viewpoint node does not *un-jump* during unbind. If *jump* was FALSE when the Viewpoint node is bound, but changed to TRUE before the *set_bind* FALSE is sent, the Viewpoint node does perform the *un-jump* during unbind.

Note that there are two other mechanisms that result in the binding of a new Viewpoint:

j. An Anchor node's *url* field specifies a "#ViewpointName".

k. A script invokes the `loadURL()` method and the URL argument specifies a "#ViewpointName".

Both of these mechanisms override the *jump* field value of the specified Viewpoint node (#ViewpointName) and assume that *jump* is TRUE when binding to the new Viewpoint. The behaviour of the viewer transition to the newly bound Viewpoint depends on the currently bound NavigationInfo node's *type* field value (see 6.29, NavigationInfo).

The *fieldOfView* field specifies a preferred minimum viewing angle from this viewpoint in radians. A small field of view roughly corresponds to a telephoto lens; a large field of view roughly corresponds to a wide-angle lens. The field of view shall be greater than zero and smaller than π. The value of *fieldOfView* represents the minimum viewing angle in any direction axis perpendicular to the view. For example, a browser with a rectangular viewing projection shall have the following relationship:

$$\frac{\texttt{display width}}{\texttt{display height}} = \frac{\texttt{tan}(\texttt{FOV}_{\texttt{horizontal}}/2)}{\texttt{tan}(\texttt{FOV}_{\texttt{vertical}}/2)}$$

where the smaller of display width or display height determines which angle equals the *fieldOfView* (the larger angle is computed using the relationship described above). The larger angle shall not exceed π and may force the smaller angle to be less than *fieldOfView* in order to sustain the aspect ratio.

The *description* field specifies a textual description of the Viewpoint node. This may be used by browser-specific user interfaces. If a Viewpoint's *description* field is empty it is recommended that the browser not present this Viewpoint in its browser-specific user interface.

APPENDIX KK

The URL syntax "`.../scene.wrl#ViewpointName`" specifies the user's initial view when loading "scene.wrl" to be the first Viewpoint node in the VRML file that appears as **`DEF ViewpointName Viewpoint {...}`**. This overrides the first Viewpoint node in the VRML file as the initial user view, and a *set_bind* TRUE message is sent to the Viewpoint node named "ViewpointName". If the Viewpoint node named "ViewpointName" is not found, the browser shall use the first Viewpoint node in the VRML file (i.e. the normal default behaviour). The URL syntax "`#ViewpointName`" (i.e. no file name) specifies a viewpoint within the existing VRML file. If this URL is loaded (e.g. Anchor node's *url* field or `loadURL()` method is invoked by a Script node), the Viewpoint node named "ViewpointName" is bound (a *set_bind* TRUE event is sent to this Viewpoint node).

The results are undefined if a Viewpoint node is bound and is the child of an LOD, Switch, or any node or prototype that disables its children. If a Viewpoint node is bound that results in collision with geometry, the browser shall perform its self-defined navigation adjustments as if the user navigated to this point (see 6.8, Collision).

## 6.54 VisibilitySensor

```
VisibilitySensor {
  exposedField SFVec3f center   0 0 0      # (-∞,∞)
  exposedField SFBool  enabled  TRUE
  exposedField SFVec3f size     0 0 0      # [0,∞)
  eventOut     SFTime  enterTime
  eventOut     SFTime  exitTime
  eventOut     SFBool  isActive
}
```

The VisibilitySensor node detects visibility changes of a rectangular box as the user navigates the world. VisibilitySensor is typically used to detect when the user can see a specific object or region in the scene in order to activate or deactivate some behaviour or animation. The purpose is often to attract the attention of the user or to improve performance.

The *enabled* field enables and disables the VisibilitySensor node. If *enabled* is set to FALSE, the VisibilitySensor node does not send events. If *enabled* is TRUE, the VisibilitySensor node detects changes to the visibility status of the box specified and sends events through the *isActive* eventOut. A TRUE event is output to *isActive* when any portion of the box impacts the rendered view. A FALSE event is sent when the box has no effect on the view. Browsers shall guarantee that, if *isActive* is FALSE, the box has absolutely no effect on the rendered view. Browsers may err liberally when *isActive* is TRUE. For example, the box may affect the rendering.

The exposed fields *center* and *size* specify the object space location of the box centre and the extents of the box (i.e., width, height, and depth). The VisibilitySensor node's box is affected by hierarchical transformations of its parents. The components of the *size* field shall be greater than or equal to zero.

The *enterTime* event is generated whenever the *isActive* TRUE event is generated, and *exitTime* events are generated whenever *isActive* FALSE events are generated. A VisibilitySensor read from a VRML file shall generate *isActive* TRUE and *enterTime* events if the sensor is enabled and the visibility box is visible. A VisibilitySensor inserted into the transformation hierarchy shall generate *isActive* TRUE and *enterTime* events if the sensor is enabled and the visibility box is visible. A VisibilitySensor removed from the transformation hierarchy shall generate *isActive* FALSE and *exitTime* events if the sensor is enabled and the visibility box is visible.

Each VisibilitySensor node behaves independently of all other VisibilitySensor nodes. Every enabled VisibilitySensor node that is affected by the user's movement receives and sends events, possibly resulting in multiple VisibilitySensor nodes receiving and sending events simultaneously. Unlike TouchSensor nodes, there is no notion of a VisibilitySensor node lower in the scene graph "grabbing" events. Multiply instanced VisibilitySensor nodes (i.e., DEF/USE) use the union of all the boxes defined by their instances. An instanced VisibilitySensor node shall detect visibility changes for all instances of the box and send events appropriately.

## 6.55 WorldInfo

```
WorldInfo {
  field MFString info  []
  field SFString title ""
}
```

The WorldInfo node contains information about the world. This node is strictly for documentation purposes and has no effect on the visual appearance or behaviour of the world. The *title* field is intended to store the name or title of the world so that browsers can present this to the user (perhaps in the window border). Any other information about the world can be stored in the *info* field, such as author information, copyright, and usage instructions.

APPENDIX KK

# 7 Conformance and minimum support requirements

## 7.1 Introduction

### 7.1.1 Table of contents

### 7.1.2 Objectives

This clause addresses conformance of VRML files, VRML generators and VRML browsers.

The primary objectives of the specifications in this clause are:

a. to promote interoperability by eliminating arbitrary subsets of, or extensions to, ISO/IEC 14772;

b. to promote uniformity in the development of conformance tests;

c. to promote consistent results across VRML browsers;

d. to facilitate automated test generation.

### 7.1.3 Scope

Conformance is defined for VRML files and for VRML browsers. For VRML generators, conformance guidelines are presented for enhancing the likelihood of successful interoperability.

A concept of *base profile conformance* is defined to ensure interoperability of VRML generators and VRML browsers. Base profile conformance is based on a set of limits and minimal requirements. Base profile conformance is intended to provide a functional level of reasonable utility for VRML generators while limiting the complexity and resource requirements of VRML browsers. Base profile conformance may not be adequate for all uses of VRML.

This clause addresses the VRML data stream and implementation requirements. Implementation requirements include the latitude allowed for VRML generators and VRML browsers. This clause does not directly address the environmental, performance, or resource requirements of the generator or browser.

This clause does not define the application requirements or dictate application functional content within a VRML file.

The scope of this clause is limited to rules for the open interchange of VRML content.

# 7.2 Conformance

## 7.2.1 Conformance of VRML files

A VRML file is *syntactically correct* according to ISO/IEC 14772 if the following conditions are met:

a. The VRML file contains as its first element a VRML header comment (see 4.2.2, Header).

b. All entities contained therein match the functional specification of the corresponding entities of ISO/IEC 14772-1. The VRML file shall obey the relationships defined in the formal grammar and all other syntactic requirements.

c. The sequence of entities in the VRML file obeys the relationships specified in ISO/IEC 14772-1 producing the structure specified in ISO/IEC 14772-1.

d. All field values in the VRML file obey the relationships specified in ISO/IEC 14772-1 producing the structure specified in ISO/IEC 14772-1.

e. No nodes appear in the VRML file other than those specified in ISO/IEC 14772-1 unless required for the encoding technique or those defined by the PROTO or EXTERNPROTO entities.

f. The VRML file is encoded according to the rules of ISO/IEC 14772.

g. It does not contain behaviour described as undefined elsewhere in this specification.

A VRML file conforms to the *base profile* if:

h. It is syntactically correct.

i. It meets the restrictions of Table 7.1.

## 7.2.2 Conformance of VRML generators

A VRML generator is conforming to this part of ISO/IEC 14772 if all VRML files that are generated are syntactically correct.

A VRML generator conforms to the base profile if it can be configured such that all VRML files generated conform to the base profile.

APPENDIX KK

### 7.2.3 Conformance of VRML browsers

A VRML browser conforms to the base profile if:

a. It is able to read any VRML file that conforms to the base profile.

b. It presents the graphical and audio characteristics of the VRML nodes in any VRML file that conforms to the base profile, within the latitude defined in this clause.

c. It correctly handles user interaction and generation of events as specified in ISO/IEC 14772, within the latitude defined in this clause.

d. It satisfies the requirements of 7.3.2, Minimum support requirements for browsers, as enumerated in Table 7.1.

## 7.3 Minimum support requirements

### 7.3.1 Minimum support requirements for generators

There is no minimum complexity which is required of (or appropriate for) VRML generators. Any compliant set of nodes of arbitrary complexity may be generated, as appropriate to represent application content.

### 7.3.2 Minimum support requirements for browsers

This subclause defines the minimum complexity which shall be supported by a VRML browser. Browser implementations may choose to support greater limits but may not reduce the limits described in Table 7.1. When the VRML file contains nodes which exceed the limits implemented by the browser, the results are undefined. Where latitude is specified in Table 7.1 for a particular node, full support is required for other aspects of that node.

### 7.3.3 VRML requirements for conforming to the base profile

In the following table, the first column defines the item for which conformance is being defined. In some cases, general limits are defined but are later overridden in specific cases by more restrictive limits. The second column defines the requirements for a VRML file conforming to the base profile; if a VRML file contains any items that exceed these limits, it may not be possible for a VRML browser conforming to the base profile to successfully parse that VRML file. The third column defines the minimum complexity for a VRML scene that a VRML browser conforming to the base profile shall be able to present to the user. The word "ignore" in the minimum browser support column refers only to the display of the item; in particular, *set_* events to ignored exposedFields must still generate corresponding *_changed* events.

**Table 7.1 -- Specifications for VRML browsers conforming to the base profile**

| *Item* | **VRML File Limit** | **Minimum Browser Support** |
|---|---|---|
| All groups | 500 children. | 500 children. Ignore *bboxCenter* and *bboxSize*. |
| All interpolators | 1000 key-value pairs. | 1000 key-value pairs. |
| All lights | 8 simultaneous lights. | 8 simultaneous lights. |
| Names for DEF/PROTO/field | 50 utf8 octets. | 50 utf8 octets. |
| All *url* fields | 10 URLs. | 10 URLs. URN's ignored.<br>Support `http', `file', and `ftp' protocols.<br>Support relative URLs where relevant. |
| PROTO/ EXTERNPROTO | 30 fields, 30 eventIns, 30 eventOuts, 30 exposedFields. | 30 fields, 30 eventIns, 30 eventOuts, 30 exposedFields. |
| EXTERNPROTO | n/a | URL references VRML files conforming to the base profile |
| PROTO definition nesting depth | 5 levels. | 5 levels. |
| SFBool | No restrictions. | Full support. |
| SFColor | No restrictions. | Full support. |
| SFFloat | No restrictions. | Full support. |
| SFImage | 256 width. 256 height. | 256 width. 256 height. |
| SFInt32 | No restrictions. | Full support. |
| SFNode | No restrictions. | Full support. |
| SFRotation | No restrictions. | Full support. |
| SFString | 30,000 utf8 octets. | 30,000 utf8 octets. |
| SFTime | No restrictions. | Full support. |
| SFVec2f | 15,000 values. | 15,000 values. |
| SFVec3f | 15,000 values. | 15,000 values. |

APPENDIX KK

| | | |
|---|---|---|
| MFColor | 15,000 values. | 15,000 values. |
| MFFloat | 1,000 values. | 1,000 values. |
| MFInt32 | 20,000 values. | 20,000 values. |
| MFNode | 500 values. | 500 values. |
| MFRotation | 1,000 values. | 1,000 values. |
| MFString | 30,000 utf8 octets per string, 10 strings. | 30,000 utf8 octets per string, 10 strings. |
| MFTime | 1,000 values. | 1,000 values. |
| MFVec2f | 15,000 values. | 15,000 values. |
| MFVec3f | 15,000 values. | 15,000 values. |
| Anchor | No restrictions. | Ignore *parameter*. Ignore *description.* |
| Appearance | No restrictions. | Full support. |
| AudioClip | 30 second uncompressed PCM WAV. | 30 second uncompressed PCM WAV. Ignore *description*. |
| Background | No restrictions. | One *skyColor*, one *groundColor*, panorama images as per ImageTexture. |
| Billboard | Restrictions as for all groups. | Full support except as for all groups. |
| Box | No restrictions. | Full support. |
| Collision | Restrictions as for all groups. | Full support except as for all groups. Any navigation behaviour acceptable when collision occurs. |
| Color | 15,000 colours. | 15,000 colours. |
| ColorInterpolator | Restrictions as for all interpolators. | Full support except as for all interpolators. |
| Cone | No restrictions. | Full support. |
| Coordinate | 15,000 points. | 15,000 points. |
| CoordinateInterpolator | 15,000 coordinates per | 15,000 coordinates per *keyValue*. Support as for all |

| | | |
|---|---|---|
| | *keyValue*. Restrictions as for all interpolators. | interpolators. |
| Cylinder | No restrictions. | Full support. |
| CylinderSensor | No restrictions. | Full support. |
| DirectionalLight | No restrictions. | Not scoped by parent Group or Transform. |
| ElevationGrid | 16,000 heights. | 16,000 heights. |
| Extrusion | (#*crossSection* points)*(#*spine* points) <= 2,500. | (#*crossSection* points)*(#*spine* points) <= 2,500. |
| Fog | No restrictions. | "EXPONENTIAL" treated as "LINEAR" |
| FontStyle | No restrictions. | If the values of the text aspects character set, *family*, *style* cannot be simultaneously supported, the order of precedence shall be: 1) character set 2) *family* 3) *style*. Browser must display all characters in ISO 8859-1 character set 2.[I8859]. |
| Group | Restrictions as for all groups. | Full support except as for all groups. |
| ImageTexture | JPEG and PNG format. Restrictions as for PixelTexture. | JPEG and PNG format. Support as for PixelTexture. |
| IndexedFaceSet | 10 vertices per face. 5000 faces. Less than 15,000 indices. | 10 vertices per face. 5000 faces. 15,000 indices in any index field. |
| IndexedLineSet | 15,000 total vertices. 15,000 indices in any index field. | 15,000 total vertices. 15,000 indices in any index field. |
| Inline | No restrictions. | Full support except as for all groups.<br>*url* references VRML files conforming to the base profile |
| LOD | Restrictions as for all groups. | At least first 4 *level*/*range* combinations interpreted, and support as for all groups. Implementations may disregard *level* distances. |
| Material | No restrictions. | Ignore ambient intensity. Ignore specular colour. Ignore emissive colour. One-bit transparency; transparency values >= 0.5 transparent. |

APPENDIX KK

| | | |
|---|---|---|
| MovieTexture | MPEG1-Systems and MPEG1-Video formats. | MPEG1-Systems and MPEG1-Video formats. Display one active movie texture. Ignore *speed* field. |
| NavigationInfo | No restrictions. | Ignore *avatarSize*. Ignore *visibilityLimit*. |
| Normal | 15,000 normals | 15,000 normals |
| NormalInterpolator | 15,000 normals per *keyValue*. Restrictions as for all interpolators. | 15,000 normals per *keyValue*. Support as for all interpolators. |
| OrientationInterpolator | Restrictions as for all interpolators. | Full support except as for all interpolators. |
| PixelTexture | 256 width. 256 height. | 256 width. 256 height. Display fully transparent and fully opaque pixels. |
| PlaneSensor | No restrictions. | Full support. |
| PointLight | No restrictions. | Ignore *radius*. Linear attenuation. |
| PointSet | 5000 points. | 5000 points. |
| PositionInterpolator | Restrictions as for all interpolators. | Full support except as for all interpolators. |
| ProximitySensor | No restrictions. | Full support. |
| ScalarInterpolator | Restrictions as for all interpolators. | Full support except as for all interpolators. |
| Script | 25 eventIns. 25 eventOuts. 25 fields. | 25 eventIns. 25 eventOuts. 25 fields.<br>No scripting language support required. |
| Shape | No restrictions. | Full support. |
| Sound | No restrictions. | 2 active sounds. Linear distance attenuation. No spatialization. See 7.3.4. |
| Sphere | No restrictions. | Full support. |
| SphereSensor | No restrictions. | Full support. |
| SpotLight | No restriction | Ignore *beamWidth*. Ignore *radius*. Linear attenuation. |
| Switch | Restrictions as for all groups. | Full support except as for all groups. |

| | | |
|---|---|---|
| Text | 100 characters per string. 100 strings. | 100 characters per string. 100 strings. |
| TextureCoordinate | 15,000 coordinates. | 15,000 coordinates. |
| TextureTransform | No restrictions. | Full support. |
| TimeSensor | No restrictions. | Ignored if *cycleInterval* < 0.01 second. |
| TouchSensor | No restrictions. | Full support. |
| Transform | Restrictions as for all groups. | Full support except as for all groups. |
| Viewpoint | No restrictions. | Ignore *fieldOfView*. Ignore *description*. |
| VisibilitySensor | No restrictions. | Always visible. |
| WorldInfo | No restrictions. | Ignored. |

## 7.3.4 Sound priority, attenuation, and spatialization

### 7.3.4.1 Sound priority

If the browser does not have the resources to play all of the currently active sounds, it is recommended that the browser sort the active sounds into an ordered list using the following sort keys in the order specified:

a. decreasing *priority;*

b. for sounds with *priority* > 0.5, increasing (now-*startTime*);

c. decreasing *intensity* at viewer location (*intensity* &times; intensity attenuation);

where *priority* is the *priority* field of the Sound node, now represents the current time, *startTime* is the *startTime* field of the audio source node specified in the *source* field, and intensity attenuation refers to the intensity multiplier derived from the linear decibel attenuation ramp between inner and outer ellipsoids.

It is important that sort key 2 be used for the high priority (event and cue) sounds so that new cues will be heard even when the browser is "full" of currently active high priority sounds. Sort key 2 should not be used for normal priority sounds, so selection among them will be based on sort key 3 (intensity at the location of the viewer).

The browser shall play as many sounds from the beginning of this sorted list as it can given available resources and allowable latency between rendering. On most systems, the resources available for MIDI streams are different from those for playing sampled sounds, thus it may be beneficial to maintain a separate list to handle MIDI data.

### 7.3.4.2 Sound attenuation and spatialization

In order to create a linear decrease in loudness as the viewer moves from the inner to the outer ellipsoid of the sound, the attenuation must be based on a linear decibel ramp. To make the falloff consistent across browsers, the decibel ramp is to vary from 0 dB at the minimum ellipsoid to -20 dB at the outer ellipsoid. Sound nodes with an outer ellipsoid that is ten times larger than the minimum will display the inverse square intensity dropoff that approximates sound attenuation in an anechoic environment.

Browsers may support spatial localization of sounds whose *spatialize* field is TRUE as well as their underlying sound libraries will allow. Browsers shall at least support stereo panning of non-MIDI sounds based on the angle between the viewer and the source. This angle is obtained by projecting the Sound *location* (in global space) onto the XZ plane of the viewer. Determine the angle between the Z-axis and the vector from the viewer to the transformed *location*, and assign a pan value in the range [0.0, 1.0] as depicted in Figure 7.1. Given this pan value, left and right channel levels can be obtained using the following equations:

$$\text{leftPanFactor} = 1 - \text{pan}^2$$

$$\text{rightPanFactor} = 1 - (1 - \text{pan})^2$$

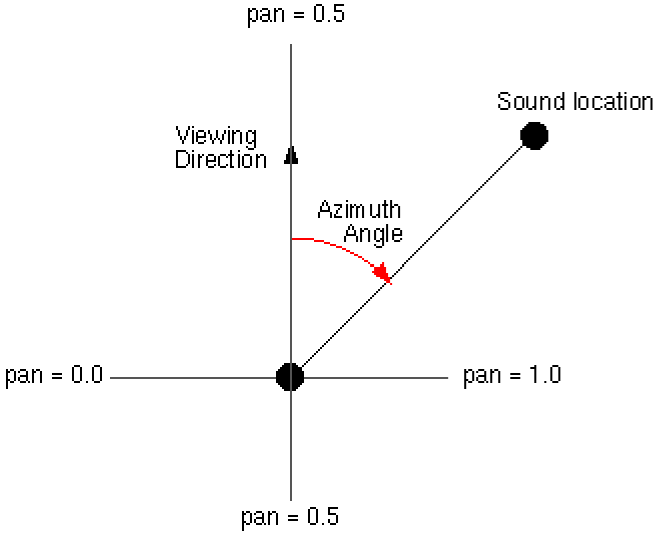


**Figure 7.1: Stereo Panning**

Using this technique, the loudness of the sound is modified by the *intensity* field value, then distance attenuation to obtain the unspatialized audio output. The values in the unspatialized audio output are then scaled by leftPanFactor and rightPanFactor to determine the final left and right output signals. The use of more sophisticated localization techniques is encouraged, but not required (see E.[SNDB]).

# Annex A
**(normative)**

# Grammar definition

## A.1 Table of contents and introduction

### A.1.1 Table of contents

This annex provides a detailed description of the grammar for each syntactic element in this part of ISO/IEC 14772. The following table of contents lists the topics in this clause:

### A.1.2 Introduction

It is not possible to parse VRML files using a context-free grammar. Semantic knowledge of the names and types of fields, eventIns, and eventOuts for each node type (either built-in or user-defined using **PROTO** or **EXTERNPROTO**) shall be used during parsing so that the parser knows which field type is being parsed.

The '#' (0x23) character begins a comment wherever it appears outside of the first line of the VRML file or quoted SFString or MFString fields. The '#' character and all characters until the next line terminator comprise the comment and are treated as whitespace.

The carriage return (0x0d), linefeed (0x0a), space (0x20), tab (0x09), and comma (0x2c) characters are whitespace characters wherever they appear outside of quoted SFString or MFString fields. Any number of whitespace characters and comments may be used to separate the syntactic entities of a VRML file. All reserved keywords are displayed in boldface type.

Any characters (including linefeed and '#') may appear within the quotes of SFString and MFString fields. A double quote character within a string shall be preceded with a backslash (e.g, "Each double quotes character \" shall have a backslash."). A backslash character within a string shall be preceded with a backslash forming two backslashes (e.g., "One backslash \\ character").

Clause 6, Nodes reference, contains a description of the allowed fields, eventIns and eventOuts for all pre-defined node types. The *double*, *float*, and *int32* symbols are expressed using Perl regular expression syntax; see E.[PERL] for details. The *IdFirstChar*, *IdRestChars*, and *string* symbols have not been formally specified; Clause 5, Fields and events reference, contains a more complete description of their syntax.

The following conventions are used in the semi-formal grammar specified in this clause:

APPENDIX KK

a. Keywords and terminal symbols which appear literally in the VRML file, are specified in **bold**.

b. Nonterminal symbols used in the grammar are specified in *italic*.

c. Production rules begin with a nonterminal symbol and the sequence of characters "::=", and end with a semi-colon (";").

d. Alternation for production rules is specified using the vertical-bar symbol ("|").

Table A.1 contains the complete list of lexical elements for the grammar in this part of ISO/IEC 14772.

**Table A.1 -- VRML lexical elements**

| **Keywords** | **Terminal symbols** | **Other symbols** |
|---|---|---|
| **DEF**<br>**EXTERNPROTO**<br>**FALSE**<br>**IS**<br>**NULL**<br>**PROTO**<br>**ROUTE**<br>**TO**<br>**TRUE**<br>**USE**<br>**eventIn**<br>**eventOut**<br>**exposedField**<br>**field** | period (**.**)<br>open brace (**{**)<br>close brace (**}**)<br>open bracket (**[**)<br>close bracket (**]**) | *Id*<br>*double*<br>*fieldType*<br>*float*<br>*int32*<br>*string* |

Terminal symbols and the string symbol may be separated by one or more whitespace characters. Keywords and the *Id*, *fieldType*, *float*, *int32*, and *double* symbols shall be separated by one or more whitespace characters.

# A.2 General

*vrmlScene* ::=
*statements* ;

*statements* ::=
*statement* |
*statement statements* |
*empty* ;

*statement* ::=
*nodeStatement* |
*protoStatement* |
*routeStatement* ;

*nodeStatement* ::=
*node* |
**DEF** *nodeNameId node* |
**USE** *nodeNameId* ;

*rootNodeStatement ::=*
*node* | **DEF** *nodeNameId node ;*

*protoStatement ::=*
*proto* |
*externproto ;*

*protoStatements ::=*
*protoStatement* |
*protoStatement protoStatements* |
*empty ;*

*proto ::=*
**PROTO** *nodeTypeId* **[** *interfaceDeclarations* **] {** *protoBody* **}** *;*

*protoBody ::=*
*protoStatements rootNodeStatement statements ;*

*interfaceDeclarations ::=*
*interfaceDeclaration* |
*interfaceDeclaration interfaceDeclarations* |
*empty ;*

*restrictedInterfaceDeclaration ::=*
**eventIn** *fieldType eventInId* |
**eventOut** *fieldType eventOutId* |
**field** *fieldType fieldId fieldValue ;*

*interfaceDeclaration ::=*
*restrictedInterfaceDeclaration* |
**exposedField** *fieldType fieldId fieldValue ;*

*externproto ::=*
**EXTERNPROTO** *nodeTypeId* **[** *externInterfaceDeclarations* **]** *URLList ;*

*externInterfaceDeclarations ::=*
*externInterfaceDeclaration* |
*externInterfaceDeclaration externInterfaceDeclarations* |
*empty ;*

*externInterfaceDeclaration ::=*
**eventIn** *fieldType eventInId* |
**eventOut** *fieldType eventOutId* |
**field** *fieldType fieldId* |
**exposedField** *fieldType fieldId ;*

*routeStatement ::=*
**ROUTE** *nodeNameId* **.** *eventOutId* **TO** *nodeNameId* **.** *eventInId ;*

*URLList ::=*
*mfstringValue ;*

*empty ::=*
*;*

APPENDIX KK

# A.3 Nodes

*node ::=*
*nodeTypeId* **{** *nodeBody* **}** |
**Script {** *scriptBody* **}** *;*

*nodeBody ::=*
*nodeBodyElement* |
*nodeBodyElement nodeBody* |
*empty ;*

*scriptBody ::=*
*scriptBodyElement* |
*scriptBodyElement scriptBody* |
*empty ;*

*scriptBodyElement ::=*
*nodeBodyElement* |
*restrictedInterfaceDeclaration* |
**eventIn** *fieldType eventInId* **IS** *eventInId* |
**eventOut** *fieldType eventOutId* **IS** *eventOutId* |
**field** *fieldType fieldId* **IS** *fieldId ;*

*nodeBodyElement ::=*
*fieldId fieldValue* |
*fieldId* **IS** *fieldId* |
*eventInId* **IS** *eventInId* |
*eventOutId* **IS** *eventOutId* |
*routeStatement* |
*protoStatement ;*

*nodeNameId ::=*
*Id ;*

*nodeTypeId ::=*
*Id ;*

*fieldId ::=*
*Id ;*

*eventInId ::=*
*Id ;*

*eventOutId ::=*
*Id ;*

*Id ::=*
*IdFirstChar* |
*IdFirstChar IdRestChars ;*

*IdFirstChar ::=*
Any ISO-10646 character encoded using UTF-8 except: 0x30-0x39, 0x0-0x20, 0x22, 0x23, 0x27, 0x2b, 0x2c, 0x2d, 0x2e, 0x5b, 0x5c, 0x5d, 0x7b, 0x7d, 0x7f ;

*IdRestChars ::=*
Any number of ISO-10646 characters except: 0x0-0x20, 0x22, 0x23, 0x27, 0x2c, 0x2e, 0x5b, 0x5c, 0x5d, 0x7b, 0x7d, 0x7f ;

# A.4 Fields

*fieldType ::=*
**MFColor** |
**MFFloat** |
**MFInt32** |
**MFNode** |
**MFRotation** |
**MFString** |
**MFTime** |
**MFVec2f** |
**MFVec3f** |
**SFBool** |
**SFColor** |
**SFFloat** |
**SFImage** |
**SFInt32** |
**SFNode** |
**SFRotation** |
**SFString** |
**SFTime** |
**SFVec2f** |
***SFVec3f*** *;*

*fieldValue ::=*
*sfboolValue* |
*sfcolorValue* |
*sffloatValue* |
*sfimageValue* |
*sfint32Value* |
*sfnodeValue* |
*sfrotationValue* |
*sfstringValue* |
*sftimeValue* |
*sfvec2fValue* |
*sfvec3fValue* |
*mfcolorValue* |
*mffloatValue* |
*mfint32Value* |
*mfnodeValue* |
*mfrotationValue* |
*mfstringValue* |
*mftimeValue* |
*mfvec2fValue* |
*mfvec3fValue ;*

*sfboolValue ::=*
**TRUE** |
**FALSE** ;

*sfcolorValue ::=*
*float float float ;*

*sffloatValue ::=*
*float ;*

APPENDIX KK

*float ::=*
([+/-]?((([0-9]+(\.)?)|([0-9]*\.[0-9]+))([eE][+\-]?[0-9]+)?)).

*sfimageValue ::=*
*int32 int32 int32 ...*

*sfint32Value ::=*
*int32 ;*

*int32 ::=*
([+\-]?(([0-9]+)|(0[xX][0-9a-fA-F]+)))

*sfnodeValue ::=*
*nodeStatement* |
**NULL** *;*

*sfrotationValue ::=*
*float float float float ;*

*sfstringValue ::=*
*string ;*

*string ::=*
".*" ... double-quotes must be \", backslashes must be \\...

*sftimeValue ::=*
*double ;*

*double ::=*
([+/-]?((([0-9]+(\.)?)|([0-9]*\.[0-9]+))([eE][+\-]?[0-9]+)?))

*mftimeValue ::=*
*sftimeValue* |
**[ ]** |
**[** *sftimeValues* **]** *;*

*sftimeValues ::=*
*sftimeValue* |
*sftimeValue sftimeValues ;*

*sfvec2fValue ::=*
*float float ;*

*sfvec3fValue ::=*
*float float float ;*

*mfcolorValue ::=*
*sfcolorValue* |
**[ ]** |
**[** *sfcolorValues* **]** *;*

sfcolorValues *::=*

sfcolorValue |

sfcolorValue sfcolorValues *;*

mffloatValue *::=*

sffloatValue |

***[ ]*** |

***[*** sffloatValues ***]*** *;*

**APPENDIX KK**

```
sffloatValues ::=
    sffloatValue |
    sffloatValue sffloatValues ;
mfint32Value ::=
    sfint32Value |
    [ ] |
    [ sfint32Values ] ;
sfint32Values ::=
    sfint32Value |
    sfint32Value sfint32Values ;
mfnodeValue ::=
    nodeStatement |
    [ ] |
    [ nodeStatements ] ;
nodeStatements ::=
    nodeStatement |
    nodeStatement nodeStatements ;
mfrotationValue ::=
    sfrotationValue |
    [ ] |
    [ sfrotationValues ] ;
sfrotationValues ::=
    sfrotationValue |
    sfrotationValue sfrotationValues ;
mfstringValue ::=
    sfstringValue |
    [ ] |
    [ sfstringValues ] ;
sfstringValues ::=
    sfstringValue |
    sfstringValue sfstringValues ;
mfvec2fValue ::=
    sfvec2fValue |
    [ ] |
    [ sfvec2fValues] ;
sfvec2fValues ::=
    sfvec2fValue |
    sfvec2fValue sfvec2fValues ;
mfvec3fValue ::=
    sfvec3fValue |
    [ ] |
    [ sfvec3fValues ] ;
sfvec3fValues ::=

    sfvec3fValue |

    sfvec3fValue sfvec3fValues ;
```

APPENDIX KK

# Annex B
**(normative)**

# Java platform scripting reference

## B.1 Introduction

This annex describes the Java platform classes and methods that enable Script nodes (see 6.40, Script) to interact with VRML scenes. See 4.12, Scripting, for a general description of scripting languages in ISO/IEC 14772. Note that support for the Java platform is not required by ISO/IEC 14772, but any access of the Java platform from within VRML Script nodes shall conform with the requirements specified in this annex.

## B.2 Platform

The Javatm platform is an object-oriented, hardware and operating system independent, multi-threaded, general-purpose application environment developed by Sun Microsystems, Inc. The Java platform consists of the language, the virtual machine, and a set of core class libraries. A conforming Java platform implements all three components according to their specifications. See 2.[JAVA] for a description of the language, the virtual machine, and the three core classes java.lang, java.util, and java.io. The other core class libraries, which are not used in this annex, are described in E.[JAPI].

## B.3 Supported protocol in the script node's *url* field

### B.3.1 *url* field

The *url* field of the Script node may contain URL references to Java bytecode as illustrated below:

```
Script {
  url "http://foo.co.jp/Example.class"
  eventIn SFBool start
}
```

### B.3.2 File extension

The file extension for Java bytecode is `.class`.

### B.3.3 MIME type

The MIME type for Java bytecode is defined as follows:

```
application/x-java
```

APPENDIX KK

# B.4 EventIn handling

## B.4.1 Description

Events to the Script node are passed to the corresponding Java platform method (processEvents() or processEvent()) in the script. The script is specified in the *url* field of the Script node.

For a Java bytecode file specified in the *url* field, the following three conditions hold:

a. it shall contain the class definition whose name is exactly the same as the body of the file name

b. it shall be a subclass of the Script class (see B.9.2.3, vrml.node package)

c. it shall be declared as a "public" class

For example, the following Script node has one eventIn whose name is *start*.

```
Script {
  url "http://foo.co.jp/Example1.class"
  eventIn SFBool start
}
```

This node points to the script file Example1.class. Its source (Example1.java) looks like this:

```
import vrml.*;
import vrml.field.*;
import vrml.node.*;

public class Example1 extends Script {
    ...
    // This method is called when any event is received
    public void processEvent(Event e){
       // ... perform some operation ...
    }
}
```

In the above example, when the *start* eventIn is sent the processEvent() method receives the eventIn and is executed.

## B.4.2 Parameter passing with Event objects

When a Script node receives an eventIn, a processEvent() or processEvents() method in the file specified in the url field of the Script node is called, which receives the eventIn as a Java platform object (Event object, see B.4.3, processEvents() and processEvent() methods).

The Event object has three fields of information associated with it: name, value, and timestamp, whose values are passed by the eventIn. These can be retrieved using the corresponding method on the Event object.

```
public class Event implements Cloneable {
   public String getName();
   public ConstField getValue();
   public double getTimeStamp();
   // other methods ...
}
```

Suppose that the eventIn type is SFXXX and eventIn name is eventInYYY, then

APPENDIX KK

a. getName() shall return the string "eventInYYY "

b. getValue() shall return ConstField containing the value of the eventIn

c. getTimeStamp() shall return a double (in seconds) containing the timestamp when the eventIn occurred (see 4.11, Time)

In the example below, the eventIn name is *start* and the eventIn value is cast to ConstSFBool. Also, the timestamp for the time when the eventIn occurred is available as a double. These are passed as an Event object to the processEvent() method:

```
public void processEvent(Event e){
    if(e.getName().equals("start")){
          ConstSFBool v = (ConstSFBool)e.getValue();
          if(v.getValue()==true){
               // ... perform some operation with e.getTimeStamp()...
          }
     }
}
```

## B.4.3 processEvents() and processEvent() methods

### B.4.3.1 processEvents() method

Authors can define a processEvents() method within a class that is called when the script receives some set of events. The prototype of the processEvents() method is **`public void processEvents(int count, Event events[]);`**

*count* indicates the number of events delivered. *events* is the array of events delivered. Its default behaviour is to iterate over each event, calling processEvent() on each one as follows:

```
public void processEvents(int count, Event events[])
{
    for (int i = 0; i < count; i++){
        processEvent(events[i]);
    }
}
```

Although authors might change this operation by giving a user-defined processEvents() method, in most cases, they only change the processEvent() method and the eventsProcessed() method as described below.

When multiple eventIns are routed from a single node to a single Script node and eventIns which have the same timestamp are received, processEvents() receives multiple events as an event array. Otherwise, each incoming event invokes separate processEvents().

For example, the processEvents() method receives two events in the following case, when the TouchSensor is activated:

```
Transform {
  children [
    DEF TS TouchSensor {}
    Shape { geometry Cone {} }
  ]
}
DEF SC Script {
  url "Example.class"
  eventIn SFBool isActive
  eventIn SFTime touchTime
```

```
    }
    ROUTE TS.isActive  TO SC.isActive
    ROUTE TS.touchTime TO SC.touchTime
```

### B.4.3.2 processEvent() method

Authors can define a processEvent() method within a class. The prototype of the processEvent() is **`public void processEvent(Event event);`**

Its default behaviour is no operation.

## B.4.4 eventsProcessed() method

Authors may define an eventsProcessed() method within a class that is called after some set of events has been received. This allows Script nodes that do not rely on the ordering of events received to generate fewer events than an equivalent Script node that generates events whenever events are received (see B.4.3.1, processEvents() method).

The prototype of the eventsProcessed() method is **`public void eventsProcessed();`**

Its default behaviour is no operation.

## B.4.5 shutdown() method

Authors may define a shutdown() method within the Script class that is called when the corresponding Script node is deleted or the world containing the Script node is unloaded or replaced by another world (see 4.12.3, Initialize() and shutdown()).

The prototype of the shutdown() method is **`public void shutdown();`**

Its default behaviour is no operation.

## B.4.6 initialize() method

Authors may define an initialize() method within the Script class that is called before the browser presents the world to the user and before any events are processed by any nodes in the same VRML file as the Script node containing this script (see 4.12.3, Initialize() and shutdown()). The various methods of the Script class such as getEventIn(), getEventOut(), getExposedField(), and getField() are not guaranteed to return correct values before the initialize() method has been executed. The initialize() method is called once during the life of the Script object.

The prototype of the initialize() method is **`public void initialize();`**

Its default behaviour is no operation. See Example2.java in B.5.1 for an example of a user-specified initialize() method.

# B.5 Accessing fields and events

## B.5.1 Accessing fields, eventIns and eventOuts of the script

The fields, eventIns, and eventOuts of a Script node are accessible from its corresponding Script class. Each field defined in the Script node is available to the Script class by using its name. Its value can be read-from or written-into. This value is persistent across function calls. EventOuts defined in the Script node can be read. EventIns defined in the Script node can be written to.

Accessing the fields of the Script node can be done by using the following three types of Script class methods:

a. **`Field getField(String fieldName)`**
is the method to get the reference to the Script node's field whose name is *fieldName*. The return value can be converted to the appropriate subclass of the Field class, (see B.6.2, Field class and ConstField class).

b. **`Field getEventOut(String eventOutName)`**
is the method to get the reference to the Script node's eventOut whose name is *eventOutName*. The return value can be converted to the appropriate subclass of the Field class, (see B.6.2, Field class and ConstField class).

c. **`Field getEventIn(String eventInName)`**
is the method to get the reference to the Script node's eventIn whose name is *eventInName*. The return value can be converted to the appropriate subclass of the Field class, (see B.6.2, Field class and ConstField class). EventIn is a write-only field. When the getValue() method is invoked on a Field object obtained by the getEventIn() method, the return value is unspecified.

When the setValue(), set1Value(), addValue(), insertValue(), delete() or clear() methods are invoked on a Field object obtained by the **getField()** method, the new value is stored in the corresponding VRML node's field (see also B.6.2, Field class and ConstField class, and B.6.3, Array handling). In the case of the set1Value(), addValue(), insertValue() or delete() methods, all elements of the VRML node's field are retrieved, then the value specified as an argument is set, added, inserted, deleted (as appropriate) to/from the elements, and then stored as the elements in the corresponding VRML node's field. In the case of the clear() method, all elements of a VRML node's field are cleared (see the definition of the clear() method).

When the setValue(), set1Value(), addValue(), insertValue(), delete() or clear() methods are invoked on a Field object obtained by the **getEventOut()** method, the call generates an eventOut in the VRML scene (see also B.6.2, Field class and ConstField class, and B.6.3, Array handling). The effect of this eventOut is specified by the associated Route(s) in the VRML scene. In the case of the set1Value(), addValue(), insertValue() or delete() methods, all elements of the VRML node's eventOut are retrieved, then the value specified as an argument is set, added, inserted or deleted (as appropriate) to/from the elements, then stored as the elements in the corresponding VRML node's eventOut, and then the eventOut is sent. In the case of the clear() method, all elements of VRML node's eventOut are cleared and an eventOut with zero elements is sent (see the definition of the clear() method).

When the setValue() or clear() methods are invoked on a Field object obtained by the **getEventIn()** method, the call generates an eventIn to the Script node. When the set1Value(), addValue(), insertValue() or delete() methods are invoked on a Field object obtained by the getEventIn() method, the exception (InvalidFieldChangeException) is thrown.

For example, the following Script node (Example2) defines an eventIn *start*, a field *state*, and an eventOut *on*. The method initialize() is invoked before any events are received, and the method processEvent() is invoked when *start* receives an event:

```
Script {
```

APPENDIX KK

```
    url      "Example2.class"
    eventIn  SFBool start
    field    SFBool state TRUE
    eventOut SFBool on
  }
```

Example2.java:

```
// Example2 toggles a persistent field variable "state" in the VRML
// Script node each time an eventIn "start" is received, then sets
// eventOut "on" equal to the value of "state"
import vrml.*;
import vrml.field.*;
import vrml.node.*;

public class Example2 extends Script {
    private SFBool state; // field
    private SFBool on;    // eventOut

    public void initialize(){
        state = (SFBool) getField("state");
        on = (SFBool) getEventOut("on");
    }

    public void processEvent(Event e){
        if(state.getValue()==true){
            on.setValue(false); // set false to eventOut ’on’
            state.setValue(false);
        }
        else {
            on.setValue(true);  // set true to eventOut ’on’
            state.setValue(true);
        }
    }
}
```

## B.5.2 Accessing fields, eventIns and eventOuts of other nodes

If a script program has an access to a node, any eventIn, eventOut or exposedField of that node is accessible by using the getEventIn(), getEventOut() or getExposedField() method defined in the node's class (see B.6.4, Node class).

The typical way for a Script node to have an access to another VRML node is to have an SFNode field which provides a reference to the other node. The following Example3 shows how this is done:

```
DEF SomeNode Transform {}
Script {
  field SFNode node USE SomeNode # SomeNode is a Transform node
  eventIn SFVec3f pos            # new value to be inserted in
                                 #   SomeNode’s translation field
  url "Example3.class"
}
```

Example3.java:

```
import vrml.*;
import vrml.field.*;
import vrml.node.*;

public class Example3 extends Script {
    private SFNode node;   // field
    private SFVec3f trans; // translation field captured from remote
                           // Transform node

    public void initialize(){
        node = (SFNode) getField("node");
    }

    public void processEvent(Event e){
        // get the reference to the 'translation' field of the Transform node
        trans = (SFVec3f)((Node) node.getValue()).getExposedField("translation");
        // reset translation to value given in Event e, which is eventIn pos
        // in the VRML Script node.
        trans.setValue((ConstSFVec3f)e.getValue());
    }
}
```

### B.5.3 Sending eventOuts or eventIns

Assume that the thread which executes processEvent() (or processEvents()) is called 'main' thread and any other thread spawned by the Script, except for the 'main' thread, is called 'sub' thread. Sending eventOuts/eventIns in the 'main' thread follows the model described in 4.10.3, Execution model, and sending eventOuts/eventIns in any 'sub' thread follows the model described in 4.12.6, Asynchronous scripts.

**In the 'main' thread:** Calling one of the setValue(), set1Value, addValue(), insertValue(), clear() or delete() methods on an eventOut/eventIn sends that event at that time. Calling the methods multiple times during one execution of the thread still only sends one event which corresponds to the first call of the method. All other calls are ignored. The event is assigned the same timestamp as the initial event which caused the main thread to execute.

**In the 'sub' thread:** Calling one of the setValue(), set1Value, addValue(), insertValue(), clear() or delete() method on an eventOut/eventIn sends that event at that time. Calling the methods multiple times during one execution of the thread sends one event per call of the method. The browser assigns the timestamp to the event.

**Note:** sending eventIns is ordinarily performed by the VRML scene, not by Java platform scripts. Exceptions are possible as specified in B.5.1, Accessing fields, eventIns and eventOuts of the script.

## B.6 Exposed classes and methods for nodes and fields

### B.6.1 Introduction

Java platform classes for VRML are defined in the packages: *vrml, vrml.node* and *vrml.field*.

The Field class extends the Java platform's Object class by default; thus, Field has the full functionality of the Object class, including the getClass() method. The rest of the package defines a "Const" read-only class for each VRML field type, with a getValue() method for each class; and another read/write class for each VRML field type, with both getValue() and setValue() methods for each class. A getValue() method converts a VRML type value into a Java platform type value. A setValue() method converts a Java platform type value into a VRML type value and sets it to the VRML field.

Some methods are listed as "throws exception," meaning that errors are possible. It may be necessary to write exception handlers (using the Java platform's **catch()** method) when those methods are used. Any method not listed as "throws exception" is guaranteed to generate no exceptions. Each method that throws an exception includes a prototype showing which exception(s) can be thrown.

### B.6.2 Field class and ConstField class

All VRML data types have equivalent Java platform classes. The Field class is the root of all field types.

```
    public abstract class Field implements Cloneable {
        // methods
    }
```

This class has two types of subclasses: read-only classes and read/write classes

a. **Read-only classes**
These classes support the getValue() method. Some classes support additional convenience methods to get value(s) from the object.

ConstSFBool, ConstSFColor, ConstMFColor, ConstSFFloat, ConstMFFloat, ConstSFImage, ConstSFInt32, ConstMFInt32, ConstSFNode, ConstMFNode, ConstSFRotation, ConstMFRotation, ConstSFString, ConstMFString, ConstSFVec2f, ConstMFVec2f, ConstSFVec3f, ConstMFVec3f, ConstSFTime, ConstMFTime

b. **Read/write classes**
These classes support both getValue() and setValue() methods. If the class name is prefixed with **MF** (meaning that it is a multiple valued field class), the class also supports the set1Value(), addValue() and insertValue() methods. Some classes support additional convenience methods to get and set value(s) from the object.

SFBool, SFColor, MFColor, SFFloat, MFFloat, SFImage, SFInt32, MFInt32, SFNode, MFNode, SFRotation, MFRotation, SFString, MFString, SFVec2f, MFVec2f, SFVec3f, MFVec3f, SFTime, MFTime

The VRML Field class and its subclasses have several methods to get and set value(s): getSize(), getValue(), get1Value(), setValue(), set1Value(), addValue(), insertValue(), clear(), delete() and toString(). In these methods, getSize(), get1Value(), set1Value(), addValue(), insertValue(), clear() and delete() are only available for multiple value field classes (MF classes).

c. **`getSize()`**
is the method to return the number of elements of each multiple value field class (MF class).

d. **`getValue()`**
is the method to convert a VRML type value into a Java platform type value and return it.

e. **`get1Value(int index)`**
is the method to convert a single VRML type value (*index*-th element of an array) and return it as a single Java platform type value. The index of the first element is 0. Attempting to get an element beyond the length of the element array throws an exception (ArrayIndexOutOfBoundsException).

f. **`setValue(value)`**
is the method to convert a Java platform type *value* into a VRML type value and copy it to the target object.

g. **`set1Value(int index, value)`**
is the method to convert from a Java platform type *value* to a VRML type value and copy it to the *index*-th element of the target object. The index of the first element is 0. Attempting to set an element beyond the length of the element array throws an exception (ArrayIndexOutOfBoundsException).

h. **`addValue(value)`**
is the method to convert from a Java platform type *value* to a VRML type value and append it to the target object, thus adding an element.

i. **`insertValue(int index, value)`**
is the method to convert from a Java platform type *value* to a VRML type value and insert it as a new element at the *index*-th position, thus adding an element. The index of the first element is 0. Attempting to insert the element beyond the length of the element array throws an exception (ArrayIndexOutOfBoundsException).

j. **`clear()`**
is the method to clear all elements in the target object so that it has no more elements in it.

k. **`delete(int index)`**
is the method to delete the *index*-th element from the target object, thus decreasing the length of the element array by one. The index of the first element is 0. Attempting to delete the element beyond the length of the element array throws an exception (ArrayIndexOutOfBoundsException).

l. **`toString()`**
is the method to return a String containing the VRML utf8 encoded value (or values) of the equivalent of the field. In the case of the SFNode(ConstSFNode) and MFNode (ConstMFNode),

- **SFNode(ConstSFNode)**: the method returns the VRML utf8 string that, if parsed as the value of an SFNode field, would produce this node. If the browser is unable to reproduce this node, the name of the node followed by the open brace and close brace shall be returned. Additional information may be included as one or more VRML comment strings.

- **MFNode(ConstMFNode)**: the method returns the VRML utf8 string that, if parsed as the value of a MFNode field, would produce this array of nodes. If the browser is unable to reproduce this node, the name of the nodes followed by the open brace and close brace shall be returned. Additional information may be included as one or more VRML comment strings

See also B.5.1, Accessing fields, eventIns and eventOuts of the Script, B.6.3, Array handling, B.6.4, Node class, and B.9.2.1, vrml package, for each class' methods definition.

## B.6.3 Array handling

### B.6.3.1 Format

Some constructors and other methods of the field classes take an array as an argument.

a. **A single-dimensional array**

Some constructors and other methods of the following classes take a single-dimensional array as an argument. The array is treated as follows:

1. **ConstSFColor, ConstMFColor, SFColor and MFColor**

```
float colors[]
```

colors[] consists of a set of three float-values (representing red, green and blue).

2. **ConstSFRotation, ConstMFRotation, SFRotation and MFRotation**

```
float rotations[]
```

rotations[] consists of a set of four float-values (representing axisX, axisY, axisZ and angle).

3. **ConstSFVec2f, ConstMFVec2f, SFVec2f and MFVec2f**

```
float vec2s[]
```

vec2s[] consists of a set of two float-values (representing x and y).

4. **ConstSFVec3f, ConstMFVec3f, SFVec3f and MFVec3f**

```
float vec3s[]
```

APPENDIX KK

vec3s[] consists of a set of three float-values (representing x, y and z).

5. **ConstSFImage and SFImage**

```
byte pixels[]
```

pixels[] consists of 2-dimensional pixel image. The ordering of the individual components for an individual pixel within the array of bytes will be as follows:

| # Comp. | byte[i] | byte[i + 1] | byte[i + 2] | byte[i + 3] |
|---|---|---|---|---|
| 1 | intensity1 | intensity2 | intensity3 | intensity4 |
| 2 | intensity1 | alpha1 | intensity2 | alpha2 |
| 3 | red1 | green1 | blue1 | red2 |
| 4 | red1 | green1 | blue1 | alpha1 |

The order of pixels in the array are to follow that defined in 5.5, SFImage. byte 0 is pixel 0, starting from the bottom left corner.

b. **A single integer and a single-dimensional array**

Some constructors and other methods take a single integer value (called *size*) and a single-dimensional array as arguments: for example, MFFloat(int size, float values[]). The *size* parameter specifies the number of valid elements in the array, from 0-th element to (*size* - 1)-th element, all other values are ignored. This means that the method may be passed an array of length *size* or larger. The same rule for a single-dimensional array is applied to the valid elements.

c. **An array of arrays**

Some constructors and other methods alternatively take an array of arrays as an argument. The array is treated as follows:

1. **ConstMFColor and MFColor**

```
float colors[][]
```

colors[][] consists of an array of sets of three float-values (representing red, green and blue).

2. **ConstMFRotation and MFRotation**

```
float rotations[][]
```

rotations[][] consists of an array of sets of four float-values (representing axisX, axisY, axisZ and angle).

3. **ConstMFVec2f and MFVec2f**

```
float vec2s[][]
```

vec2s[][] consists of an array of sets of two float-values (representing x and y).

4. **ConstMFVec3f and MFVec3f**

```
float vec3s[][]
```

vec3s[][] consists of an array of sets of three float-values (representing x, y and z).

### B.6.3.2 Constructors and methods

The following describes how arrays are interpreted in detail for each constructor and method.

Microsoft et al. Exhibit 1005

Suppose *NA* represents the number of elements in the array specified as an argument of some constructors and other methods, and *NT* represents the number of elements which the target object requires or has. For example, if the target object is SFColor, it requires exactly 3 float values.

In the following description, suppose SF* represents subclasses of Field class, ConstSF* represents subclasses of ConstField class, MF* represents subclasses of MField class and ConstMF* represents subclasses of ConstMField class.

a. **A single-dimensional array**

In the following description, if the target object is:

- ConstSFColor and SFColor, *NT* is exactly 3
- ConstMFColor and MFColor, *NT* is a multiple of 3, and *NA* is rounded down to a multiple of 3
- ConstSFRotation and SFRotation, *NT* is exactly 4
- ConstMFRotation and MFRotation, *NT* is a multiple of 4, and *NA* is rounded down to a multiple of 4
- ConstSFVec2f and SFVec2f, *NT* is exactly 2
- ConstMFVec2f and MFVec2f, *NT* is a multiple of 2, and *NA* is rounded down to a multiple of 2
- ConstSFVec3f and SFVec3f, *NT* is exactly 3
- ConstMFVec3f and MFVec3f, *NT* is a multiple of 3, and *NA* is rounded down to a multiple of 3
- ConstSFImage and SFImage, *NT* is exactly *width**height**components* (*width*, *height* and number of *components* in the image, see 5.5, SFImage)

1. **For ConstSF* objects and SF* objects**

   For all constructors and methods which take a single-dimensional array as an argument, the following rules are applied. *NA* shall be larger than or equal to *NT*. If *NA* is larger than *NT*, the elements from the 0-th to the (*NT* - 1)-th element are used and remaining elements are ignored. Otherwise, an exception(ArrayIndexOutOfBoundsException) is thrown.

   For example, when the array is used as an argument of the setValue() for SFColor, the array shall contain at least 3 float values. If the array contains more than 3 float values, the first 3 values are used.

2. **For ConstMF* objects and MF* objects**

- **For constructor.**

  The same rule for ConstSF* and SF* objects is applied.

  For example, when the array is used as an argument of the constructor for MFColor, the array shall contain at least 3 float values. If the array contains 3N, 3N +1 or 3N + 2 float values, the first 3N values are used.

- **For setValue() method.**

  If *NT* is smaller than or equal to *NA*, *NT* is increased to *NA* and then all elements of the array are copied into the target object. If *NT* is larger than *NA*, *NT* is decreased to *NA* and then all elements of the array are copied into the target object.

- **For getValue() method.**

  If *NT* is smaller than or equal to *NA*, all elements of the target object are copied into the first *NT* elemets of the array. If *NT* is larger than *NA*, an exception (ArrayIndexOutOfBoundsException) is thrown.

- **For set1Value() method.**

  The target element (the *index*-th element) is treated as an SF* object. So the same rule for ConstSF* and SF* objects is applied.

- **For get1Value() method.**

  The target element (the *index*-th element) is treated as an SF* (or ConstSF*) object. So the same rule for ConstSF* and SF* objects is applied.

- **For addValue() and insertValue() method.**

  The corresponding SF* object is created using the argument, and then added to the target object or inserted into the target object.

b. **A single integer and a single-dimensional array**

For all constructors and methods which take a single integer value (called *size*) and a single-dimensional array as arguments, for example, MFFloat(int size, float values[]), the following rule is applied.

The *size* parameter specifies the number of valid elements in the array from the 0-th element to the (*size* - 1)-th element; all other values are ignored. This means that the method may be passed an array of length *size* or larger.

The valid elements are copied to a new array and the rules for a single-dimensional array are applied to the new array for all methods.

c. **An array of arrays**

This argument is used only for MF* objects and ConstMF* objects. In the following case, suppose *NA* is the number of arrays (for example float f[4][3], *NA* is 4) specified as an argument of some constructors and other methods and *NT* is the return value of getSize() method of each object.

- **For constructor.**
  The object which has *NA* elements is created.

- **For setValue() method.**
  If *NT* is smaller than or equal to *NA*, *NT* is increased to *NA* and then all elements of the array are copied into the target object. If *NT* is larger than *NA*, *NT* is decreased to *NA* and then all elements of the array are copied into the target object.

- **For getValue() method.**
  If *NT* is smaller than or equal to *NA*, all elements of the target object are copied into the array. If *NT* is larger than *NA*, an exception(ArrayIndexOutOfBoundsException) is thrown.

## B.6.4 Node class

The *Node* class has several methods:

a. **`String getType()`**
is the method to return the type of the node.

b. **`ConstField getEventOut(String eventOutName)`**
is the method to get the reference to the node's eventOut whose name is *eventOutName*. The return value can be converted to the appropriate subclass of the Field class, (see B.6.2, Field class and ConstField class).

c. **`Field getEventIn(String eventInName)`**
is the method to get the reference to the node's eventIn whose name is *eventInName*. The return value can be converted to the appropriate subclass of the Field class, (see B.6.2, Field class and ConstField class). EventIn is a write-only field. When the getValue() method is invoked on a Field object obtained by the getEventIn() method, the return value is unspecified.

d. **`Field getExposedField(String exposedFieldName)`**
is the method to get the reference to the node's exposedField whose name is *exposedFieldName*. The return value can be converted to the appropriate subclass of the Field class, (see B.6.2, Field class and ConstField class).

e. **`Browser getBrowser()`**
is the method to get the browser object that this node is contained in (see B.6.5, Browser class).

f. **`String toString()`**
is the same as the toString() method of SFNode (ConstSFNode).

When the setValue(), set1Value(), addValue(), insertValue(), delete() or clear() methods are invoked on a Field object obtained by the **getExposedField()** method, the call generates an eventOut in the VRML scene (see also B.6.2, Field class and ConstField class, and B.6.3, Array handling). The effect of this eventOut is specified by the associated Route(s) in the VRML scene. In the case of the set1Value(), addValue(), insertValue() or delete() methods, all elements of the VRML node's exposedField are retrieved, then the value specified as an argument is set, added, inserted or deleted (as appropriate) to/from the elements, then stored as the elements in the corresponding VRML node's exposedField, and then the eventOut is sent. In the case of the clear() method, all elements of VRML node's exposedField are cleared and an eventOut with zero elements is sent (see the definition of the clear() method).

When the setValue() or clear() methods are invoked on a Field object obtained by the **getEventIn()** method, the call generates an eventIn in the VRML scene. When the set1Value(), addValue(), insertValue() or delete() methods are invoked on the Field object, an exception (InvalidFieldChangeException) is thrown.

## B.6.5 Browser class

This section lists the public Java platform interfaces to the *Browser* class, which allows scripts to get and set browser information. For descriptions of the following methods, see 4.12.10, Browser script interface. Table B.1 lists the Browser class methods.

**Table B.1 -- Browser class methods**

| Return value | Method name |
|---|---|
| String | **getName**() |
| String | **getVersion**() |
| float | **getCurrentSpeed**() |
| float | **getCurrentFrameRate**() |
| String | **getWorldURL**() |
| void | **replaceWorld**(BaseNode[] nodes) |
| BaseNode[] | **createVrmlFromString**(String vrmlSyntax) |
| void | **createVrmlFromURL**(String[] url, BaseNode node, String event) |
| void | **addRoute**(BaseNode fromNode, String fromEventOut, BaseNode toNode, String toEventIn) |
| void | **deleteRoute**(BaseNode fromNode, String fromEventOut, BaseNode toNode, String toEventIn) |
| void | **loadURL**(String[] url, String[] parameter) |
| void | **setDescription**(String description) |

See B.9.2.1, vrml package, for each method's definition.

Table B.2 contains conversions from the types used in Browser class to Java platform types.

**Table B.2 -- VRML and Java platform types**

| VRML type | Java platform type |
|---|---|
| SFString | String |
| SFFloat | float |
| MFString | String[] |
| MFNode | BaseNode[] |

When a relative URL is specified as an argument of the loadURL() and createVrmlFromURL() method, the path is relative to the script file containing these methods (see 4.5.3, Relative URLs).

### B.6.6 User-defined classes and packages

The Java platform classes defined by a user can be used in the Java program. They are first searched from the directories specified in the CLASSPATH environment variable and then the directory where the Java program's class file is placed.

If the Java platform class is in a package, this package is searched from the directories specified in the CLASSPATH environment variable and then the directory where the Java program's class file is placed.

### B.6.7 Standard Java platform packages

Java programs have access to the full set of classes available in `java.*`. All parts of the Java platform are required to work as "normal" for the Java platform. So all methods specified in this annex are required to be thread-safe. The security model is browser specific.

## B.7 Exceptions

Java platform methods may throw the following exceptions:

a. **InvalidFieldException**
is thrown at the time getField() is executed and the field name is invalid.

b. **InvalidEventInException**
is thrown at the time getEventIn() is executed and the eventIn name is invalid.

c. **InvalidEventOutException**
is thrown at the time getEventOut() is executed and the eventOut name is invalid.

d. **InvalidExposedFieldException**
is thrown at the time getExposedField() is executed and the exposedField name is invalid.

e. **InvalidVRMLSyntaxException**
is thrown at the time createVrmlFromString(), createVrmlFromURL() or loadURL() is executed and the vrml syntax is invalid.

f. **InvalidRouteException**
is thrown at the time addRoute() or deleteRoute() is executed and one or more of the arguments is invalid.

g. **InvalidFieldChangeException**
may be thrown as a result of all sorts of illegal field changes, for example:

   1. Adding a node from one World as the child of a node in another World.

   2. Creating a circularity in a scene graph.

   3. Setting an invalid string on enumerated fields, such as the fogType field of the Fog node.

4. Calling the set1Value(), addValue() or delete() on a Field object obtained by the getEventIn() method.

h. **ArrayIndexOutOfBoundsException**
is generated at the time getValue(), set1Value(), insertValue() or delete() is executed and the index is out of bound (see B.6.2, Field class and ConstField class). This is the standard exception defined in the Java platform Array class.

i. **IllegalArgumentException**
is generated at the time loadURL() or createVrmlFromURL() is executed and an error is occurred before retrieving the content of the url (see B.6.5, Browser class). This is the standard exception defined by the Java platform.

If exceptions are not caught by authors, a browser's behaviour is unspecified (see B.10, Example of exception class).

# B.8 Examples

The following is an example of a Script node which determines whether a given color contains a lot of red. The Script node exposes a field, an eventIn, and an eventOut:

```
Script {
  field    SFColor currentColor 0 0 0
  eventIn  SFColor colorIn
  eventOut SFBool  isRed
  url      "Example4.class"
}
```

The following is the source code for the Example4.java file that gets called every time an eventIn is routed to the above Script node:

Example4.java:

```
import vrml.*;
import vrml.field.*;
import vrml.node.*;

public class Example4 extends Script {
    // Declare field(s)
    private SFColor currentColor;

    // Declare eventOut
    private SFBool isRed;

    // buffer for  SFColor.getValue().
    private float colorBuff[] = new float[3];

    public void initialize(){
       currentColor = (SFColor) getField("currentColor");
       isRed = (SFBool) getEventOut("isRed");
    }

    public void processEvent(Event e){
        // This method is called when a colorIn event is received
        currentColor.setValue((ConstSFColor)e.getValue());
    }
```

```
        public void eventsProcessed(){
            currentColor.getValue(colorBuff);
            if (colorBuff[0] >= 0.5) // if red is at or above 50%
                isRed.setValue(true);
        }
    }
```

Details on when the methods defined in Example4.java are called may be found in 4.10.3, Execution model.

### Example5: createVrmlFromUrl()

```
    Script {
      url "Example5.class"
      field   MFString target_url "foo.wrl"
      eventIn MFNode   nodesLoaded
      eventIn SFBool   trigger_event
    }
```

Example5.java:

```
  import vrml.*;
  import vrml.field.*;
  import vrml.node.*;

  public class Example5 extends Script {
      private MFString target_url; // field
      private Browser browser;

      public void initialize(){
          target_url = (MFString)getField("target_url");
          browser = this.getBrowser();
      }

      public void processEvent(Event e){
          if(e.getName().equals("trigger_event")){
               // do something and then fetch values
               String[] urls;
               urls = new String[target_url.getSize()];
               target_url.getValue(urls);
               browser.createVrmlFromURL(urls, this, "nodesLoaded");
          }
          if(e.getName().equals("nodesLoaded")){
              // do something
          }
      }
  }
```

### Example6: addRoute()

```
    DEF TS TouchSensor {}
    Script {
      url     "Example6.class"
      field   SFNode fromNode USE TS
      eventIn SFBool clicked
      eventIn SFBool trigger_event
    }
```

Example6.java:

```
import vrml.*;
import vrml.field.*;
import vrml.node.*;

public class Example6 extends Script {

    private SFNode fromNode;
    private Browser browser;

    public void initialize(){
        fromNode = (SFNode) getField("fromNode");
        browser = this.getBrowser();
    }

    public void processEvent(Event e){
        if(e.getName().equals("trigger_event")){
            // do something and then add routing
            browser.addRoute(fromNode.getValue(), "isActive", this, "clicked");
        }
        if(e.getName().equals("clicked")){
            // do something
        }
    }
}
```

# B.9 Class definitions

## B.9.1 Class hierarchy

The classes are divided into three packages: vrml, vrml.field and vrml.node.

```
java.lang.Object
      |
      +- vrml.Event
      +- vrml.Browser
      +- vrml.Field
      |       +- vrml.field.SFBool
      |       +- vrml.field.SFColor
      |       +- vrml.field.SFFloat
      |       +- vrml.field.SFImage
      |       +- vrml.field.SFInt32
      |       +- vrml.field.SFNode
      |       +- vrml.field.SFRotation
      |       +- vrml.field.SFString
      |       +- vrml.field.SFTime
      |       +- vrml.field.SFVec2f
      |       +- vrml.field.SFVec3f
      |       |
      |       +- vrml.MField
      |       |       +- vrml.field.MFColor
      |       |       +- vrml.field.MFFloat
      |       |       +- vrml.field.MFInt32
      |       |       +- vrml.field.MFNode
```

```
     |       |       +- vrml.field.MFRotation
     |       |       +- vrml.field.MFString
     |       |       +- vrml.field.MFTime
     |       |       +- vrml.field.MFVec2f
     |       |       +- vrml.field.MFVec3f
     |       |
     |       +- vrml.ConstField
     |               +- vrml.field.ConstSFBool
     |               +- vrml.field.ConstSFColor
     |               +- vrml.field.ConstSFFloat
     |               +- vrml.field.ConstSFImage
     |               +- vrml.field.ConstSFInt32
     |               +- vrml.field.ConstSFNode
     |               +- vrml.field.ConstSFRotation
     |               +- vrml.field.ConstSFString
     |               +- vrml.field.ConstSFTime
     |               +- vrml.field.ConstSFVec2f
     |               +- vrml.field.ConstSFVec3f
     |               |
     |               +- vrml.ConstMField
     |                       +- vrml.field.ConstMFColor
     |                       +- vrml.field.ConstMFFloat
     |                       +- vrml.field.ConstMFInt32
     |                       +- vrml.field.ConstMFNode
     |                       +- vrml.field.ConstMFRotation
     |                       +- vrml.field.ConstMFString
     |                       +- vrml.field.ConstMFTime
     |                       +- vrml.field.ConstMFVec2f
     |                       +- vrml.field.ConstMFVec3f
     |
     +- vrml.BaseNode
             +- vrml.node.Node
             +- vrml.node.Script

java.lang.Exception
     |
     +- java.lang.RuntimeException
     |       +- java.lang.IllegalArgumentException
     |               +- vrml.InvalidEventInException
     |               +- vrml.InvalidEventOutException
     |               +- vrml.InvalidExposedFieldException
     |               +- vrml.InvalidFieldChangeException
     |               +- vrml.InvalidFieldException
     |               +- vrml.InvalidRouteException
     |
     +- vrml.InvalidVRMLSyntaxException
```

APPENDIX KK

## B.9.2 VRML packages

### B.9.2.1 vrml package

```
package vrml;

public class Event implements Cloneable
{
   public String getName();
   public double getTimeStamp();
   public ConstField getValue();
   public Object clone();

   public String toString();   // This overrides a method in Object
}

public class Browser
{
   private Browser();
   public String toString();   // This overrides a method in Object

   // Browser interface
   public String getName();
   public String getVersion();

   public float getCurrentSpeed();

   public float getCurrentFrameRate();

   public String getWorldURL();
   public void replaceWorld(BaseNode[] nodes);

   public BaseNode[] createVrmlFromString(String vrmlSyntax)
     throws InvalidVRMLSyntaxException;

   public void createVrmlFromURL(String[] url, BaseNode node, String event)
     throws InvalidVRMLSyntaxException;

   public void addRoute(BaseNode fromNode, String fromEventOut,
                        BaseNode toNode, String toEventIn);

   public void deleteRoute(BaseNode fromNode, String fromEventOut,
                           BaseNode toNode, String toEventIn);

   public void loadURL(String[] url, String[] parameter)
     throws InvalidVRMLSyntaxException;

   public void setDescription(String description);
}

public abstract class Field implements Cloneable
{
   public Object clone();
}
```

```
public abstract class MField extends Field
{
   public abstract int getSize();
   public abstract void clear();
   public abstract void delete(int index);
}

public abstract class ConstField extends Field
{
}

public abstract class ConstMField extends ConstField
{
   public abstract int getSize();
}

//
// This is the general BaseNode class
//
public abstract class BaseNode
{
   // Returns the type of the node.  If the node is a prototype
   // it returns the name of the prototype.
   public String getType();

   // Get the Browser that this node is contained in.
   public Browser getBrowser();
}
```

### B.9.2.2 vrml.field package

```
package vrml.field;

public class SFBool extends Field
{
   public SFBool();
   public SFBool(boolean value);

   public boolean getValue();

   public void setValue(boolean b);
   public void setValue(ConstSFBool b);
   public void setValue(SFBool b);

   public String toString();   // This overrides a method in Object
}

public class SFColor extends Field
{
   public SFColor();
   public SFColor(float red, float green, float blue);

   public void getValue(float colors[]);
   public float getRed();
   public float getGreen();
   public float getBlue();

   public void setValue(float colors[]);
   public void setValue(float red, float green, float blue);
   public void setValue(ConstSFColor color);
   public void setValue(SFColor color);

   public String toString();   // This overrides a method in Object
}

public class SFFloat extends Field
{
   public SFFloat();
   public SFFloat(float f);

   public float getValue();

   public void setValue(float f);
   public void setValue(ConstSFFloat f);
   public void setValue(SFFloat f);

   public String toString();   // This overrides a method in Object
}

public class SFImage extends Field
{
   public SFImage();
   public SFImage(int width, int height, int components, byte pixels[]);

   public int getWidth();
   public int getHeight();
   public int getComponents();
   public void getPixels(byte pixels[]);

   public void setValue(int width, int height, int components,
                        byte pixels[]);
   public void setValue(ConstSFImage image);
   public void setValue(SFImage image);

   public String toString();   // This overrides a method in Object
}

public class SFInt32 extends Field
{
   public SFInt32();
   public SFInt32(int value);

   public int getValue();

   public void setValue(int i);
   public void setValue(ConstSFInt32 i);
   public void setValue(SFInt32 i);

   public String toString();   // This overrides a method in Object
}
```

```
public class SFNode extends Field
{
   public SFNode();
   public SFNode(BaseNode node);

   public BaseNode getValue();

   public void setValue(BaseNode node);
   public void setValue(ConstSFNode node);
   public void setValue(SFNode node);

   public String toString();   // This overrides a method in Object
}

public class SFRotation extends Field
{
   public SFRotation();
   public SFRotation(float axisX, float axisY, float axisZ, float angle);

   public void getValue(float rotations[]);

   public void setValue(float rotations[]);
   public void setValue(float axisX, float axisY, float axisZ,
                        float angle);
   public void setValue(ConstSFRotation rotation);
   public void setValue(SFRotation rotation);

   public String toString();   // This overrides a method in Object
}

public class SFString extends Field
{
   public SFString();
   public SFString(String s);

   public String getValue();

   public void setValue(String s);
   public void setValue(ConstSFString s);
   public void setValue(SFString s);

   public String toString();   // This overrides a method in Object
}

public class SFTime extends Field
{
   public SFTime();
   public SFTime(double time);

   public double getValue();

   public void setValue(double time);
   public void setValue(ConstSFTime time);
   public void setValue(SFTime time);

   public String toString();   // This overrides a method in Object
}
```

```
public class SFVec2f extends Field
{
   public SFVec2f();
   public SFVec2f(float x, float y);

   public void getValue(float vec2s[]);
   public float getX();
   public float getY();

   public void setValue(float vec2s[]);
   public void setValue(float x, float y);
   public void setValue(ConstSFVec2f vec);
   public void setValue(SFVec2f vec);

   public String toString();   // This overrides a method in Object
}

public class SFVec3f extends Field
{
   public SFVec3f();
   public SFVec3f(float x, float y, float z);

   public void getValue(float vec3s[]);
   public float getX();
   public float getY();
   public float getZ();

   public void setValue(float vec3s[]);
   public void setValue(float x, float y, float z);
   public void setValue(ConstSFVec3f vec);
   public void setValue(SFVec3f vec);

   public String toString();   // This overrides a method in Object
}

public class MFColor extends MField
{
   public MFColor();
   public MFColor(float colors[][]);
   public MFColor(float colors[]);
   public MFColor(int size, float colors[]);

   public void getValue(float colors[][]);
   public void getValue(float colors[]);

   public void get1Value(int index, float colors[]);
   public void get1Value(int index, SFColor color);

   public void setValue(float colors[][]);
   public void setValue(float colors[]);
   public void setValue(int size, float colors[]);
   /****************************************************
    color[0] ... color[size - 1] are used as color data
    in the way that color[0], color[1], and color[2]
    represent the first color. The number of colors
    is defined as "size / 3".
    ***************************************************/
   public void setValue(MFColor colors);
   public void setValue(ConstMFColor colors);

   public void set1Value(int index, ConstSFColor color);
   public void set1Value(int index, SFColor color);
   public void set1Value(int index, float red, float green, float blue);

   public void addValue(ConstSFColor color);
   public void addValue(SFColor color);
   public void addValue(float red, float green, float blue);

   public void insertValue(int index, ConstSFColor color);
   public void insertValue(int index, SFColor color);
   public void insertValue(int index, float red, float green, float blue);

   public String toString();   // This overrides a method in Object
}


public class MFFloat extends MField
{
   public MFFloat();
   public MFFloat(int size, float values[]);
   public MFFloat(float values[]);

   public void getValue(float values[]);

   public float get1Value(int index);

   public void setValue(float values[]);
   public void setValue(int size, float values[]);
   public void setValue(MFFloat value);
   public void setValue(ConstMFFloat value);

   public void set1Value(int index, float f);
   public void set1Value(int index, ConstSFFloat f);
   public void set1Value(int index, SFFloat f);

   public void addValue(float f);
   public void addValue(ConstSFFloat f);
   public void addValue(SFFloat f);

   public void insertValue(int index, float f);
   public void insertValue(int index, ConstSFFloat f);
   public void insertValue(int index, SFFloat f);

   public String toString();   // This overrides a method in Object
}

public class MFInt32 extends MField
{
   public MFInt32();
   public MFInt32(int size, int values[]);
   public MFInt32(int values[]);

   public void getValue(int values[]);
   public int get1Value(int index);

   public void setValue(int values[]);
   public void setValue(int size, int values[]);
   public void setValue(MFInt32 value);
   public void setValue(ConstMFInt32 value);

   public void set1Value(int index, int i);
   public void set1Value(int index, ConstSFInt32 i);
   public void set1Value(int index, SFInt32 i);

   public void addValue(int i);
   public void addValue(ConstSFInt32 i);
   public void addValue(SFInt32 i);

   public void insertValue(int index, int i);
   public void insertValue(int index, ConstSFInt32 i);
   public void insertValue(int index, SFInt32 i);

   public String toString();   // This overrides a method in Object
}

public class MFNode extends MField
{
   public MFNode();
   public MFNode(int size, BaseNode node[]);
   public MFNode(BaseNode node[]);

   public void getValue(BaseNode node[]);

   public BaseNode get1Value(int index);

   public void setValue(BaseNode node[]);
   public void setValue(int size, BaseNode node[]);
   public void setValue(MFNode node);
   public void setValue(ConstMFNode node);

   public void set1Value(int index, BaseNode node);
   public void set1Value(int index, ConstSFNode node);
   public void set1Value(int index, SFNode node);

   public void addValue(BaseNode node);
   public void addValue(ConstSFNode node);
   public void addValue(SFNode node);

   public void insertValue(int index, BaseNode node);
   public void insertValue(int index, ConstSFNode node);
   public void insertValue(int index, SFNode node);

   public String toString();   // This overrides a method in Object
}

public class MFRotation extends MField
{
   public MFRotation();
   public MFRotation(float rotations[][]);
   public MFRotation(float rotations[]);
   public MFRotation(int size, float rotations[]);

   public void getValue(float rotations[][]);
   public void getValue(float rotations[]);

   public void get1Value(int index, float rotations[]);
   public void get1Value(int index, SFRotation rotation);

   public void setValue(float rotations[][]);
   public void setValue(float rotations[]);
   public void setValue(int size, float rotations[]);
   public void setValue(MFRotation rotations);
   public void setValue(ConstMFRotation rotations);

   public void set1Value(int index, ConstSFRotation rotation);
   public void set1Value(int index, SFRotation rotation);
   public void set1Value(int index, float axisX, float axisY, float axisZ,
float angle);

   public void addValue(ConstSFRotation rotation);
   public void addValue(SFRotation rotation);
   public void addValue(float axisX, float axisY, float axisZ, float angle);

   public void insertValue(int index, ConstSFRotation rotation);
   public void insertValue(int index, SFRotation rotation);
   public void insertValue(int index, float axisX, float axisY, float axisZ,
                           float angle);

   public String toString();   // This overrides a method in Object
}

public class MFString extends MField
{
   public MFString();
   public MFString(int size, String s[]);
   public MFString(String s[]);

   public void getValue(String s[]);

   public String get1Value(int index);

   public void setValue(String s[]);
   public void setValue(int size, String s[]);
   public void setValue(MFString s);
   public void setValue(ConstMFString s);

   public void set1Value(int index, String s);
   public void set1Value(int index, ConstSFString s);
   public void set1Value(int index, SFString s);

   public void addValue(String s);
   public void addValue(ConstSFString s);
   public void addValue(SFString s);

   public void insertValue(int index, String s);
   public void insertValue(int index, ConstSFString s);
   public void insertValue(int index, SFString s);

   public String toString();   // This overrides a method in Object
}

public class MFTime extends MField
{
   public MFTime();
   public MFTime(int size, double times[]);
   public MFTime(double times[]);

   public void getValue(double times[]);

   public double get1Value(int index);

   public void setValue(double times[]);
   public void setValue(int size, double times[]);
   public void setValue(MFTime times);
   public void setValue(ConstMFTime times);

   public void set1Value(int index, double time);
   public void set1Value(int index, ConstSFTime time);
   public void set1Value(int index, SFTime time);

   public void addValue(double time);
   public void addValue(ConstSFTime time);
   public void addValue(SFTime time);

   public void insertValue(int index, double time);
   public void insertValue(int index, ConstSFTime time);
   public void insertValue(int index, SFTime time);

   public String toString();   // This overrides a method in Object
}

public class MFVec2f extends MField
{
   public MFVec2f();
   public MFVec2f(float vec2s[][]);
   public MFVec2f(float vec2s[]);
   public MFVec2f(int size, float vec2s[]);

   public void getValue(float vec2s[][]);
   public void getValue(float vec2s[]);

   public void get1Value(int index, float vec2s[]);
   public void get1Value(int index, SFVec2f vec);

   public void setValue(float vec2s[][]);
   public void setValue(float vec2s[]);
   public void setValue(int size, float vec2s[]);
   public void setValue(MFVec2f vecs);
   public void setValue(ConstMFVec2f vecs);

   public void set1Value(int index, float x, float y);
   public void set1Value(int index, ConstSFVec2f vec);
   public void set1Value(int index, SFVec2f vec);
   public void addValue(float x, float y);
   public void addValue(ConstSFVec2f vec);
   public void addValue(SFVec2f vec);

   public void insertValue(int index, float x, float y);
   public void insertValue(int index, ConstSFVec2f vec);
   public void insertValue(int index, SFVec2f vec);

   public String toString();   // This overrides a method in Object
}

public class MFVec3f extends MField
{
   public MFVec3f();
   public MFVec3f(float vec3s[][]);
   public MFVec3f(float vec3s[]);
   public MFVec3f(int size, float vec3s[]);

   public void getValue(float vec3s[][]);
   public void getValue(float vec3s[]);

   public void get1Value(int index, float vec3s[]);
   public void get1Value(int index, SFVec3f vec);

   public void setValue(float vec3s[][]);
   public void setValue(float vec3s[]);
   public void setValue(int size, float vec3s[]);
   public void setValue(MFVec3f vecs);
   public void setValue(ConstMFVec3f vecs);

   public void set1Value(int index, float x, float y, float z);
   public void set1Value(int index, ConstSFVec3f vec);
   public void set1Value(int index, SFVec3f vec);

   public void addValue(float x, float y, float z);
   public void addValue(ConstSFVec3f vec);
   public void addValue(SFVec3f vec);

   public void insertValue(int index, float x, float y, float z);
   public void insertValue(int index, ConstSFVec3f vec);
   public void insertValue(int index, SFVec3f vec);

   public String toString();   // This overrides a method in Object
}

public class ConstSFBool extends ConstField
{
   public ConstSFBool(boolean value);

   public boolean getValue();

   public String toString();   // This overrides a method in Object
}
```

APPENDIX KK

```
public class ConstSFColor extends ConstField
{
   public ConstSFColor(float red, float green, float blue);

   public void getValue(float colors[]);
   public float getRed();
   public float getGreen();
   public float getBlue();

   public String toString();   // This overrides a method in Object
}

public class ConstSFFloat extends ConstField
{
   public ConstSFFloat(float value);

   public float getValue();

   public String toString();   // This overrides a method in Object
}

public class ConstSFImage extends ConstField
{
   public ConstSFImage(int width, int height, int components, byte pixels[]);

   public int getWidth();
   public int getHeight();
   public int getComponents();
   public void getPixels(byte pixels[]);

   public String toString();   // This overrides a method in Object
}

public class ConstSFInt32 extends ConstField
{
   public ConstSFInt32(int value);

   public int getValue();

   public String toString();   // This overrides a method in Object
}

public class ConstSFNode extends ConstField
{
   public ConstSFNode(BaseNode node);

   public BaseNode getValue();

   public String toString();   // This overrides a method in Object
}

public class ConstSFRotation extends ConstField
{
   public ConstSFRotation(float axisX, float axisY, float axisZ, float angle);

   public void getValue(float rotations[]);
```

Microsoft et al. Exhibit 1005

APPENDIX KK

```
   public String toString();   // This overrides a method in Object
}

public class ConstSFString extends ConstField
{
   public ConstSFString(String value);

   public String getValue();

   public String toString();   // This overrides a method in Object
}

public class ConstSFTime extends ConstField
{
   public ConstSFTime(double time);

   public double getValue();

   public String toString();   // This overrides a method in Object
}

public class ConstSFVec2f extends ConstField
{
   public ConstSFVec2f(float x, float y);

   public void getValue(float vec2s[]);
   public float getX();
   public float getY();

   public String toString();   // This overrides a method in Object
}

public class ConstSFVec3f extends ConstField
{
   public ConstSFVec3f(float x, float y, float z);

   public void getValue(float vec3s[]);
   public float getX();
   public float getY();
   public float getZ();

   public String toString();   // This overrides a method in Object
}

public class ConstMFColor extends ConstMField
{
   public ConstMFColor(float colors[][]);
   public ConstMFColor(float colors[]);
   public ConstMFColor(int size, float colors[]);

   public void getValue(float colors[][]);
   public void getValue(float colors[]);

   public void get1Value(int index, float colors[]);
   public void get1Value(int index, SFColor color);

   public String toString();   // This overrides a method in Object
```

```
}

public class ConstMFFloat extends ConstMField
{
   public ConstMFFloat(int size, float values[]);
   public ConstMFFloat(float values[]);

   public void getValue(float values[]);

   public float get1Value(int index);

   public String toString();   // This overrides a method in Object
}

public class ConstMFInt32 extends ConstMField
{
   public ConstMFInt32(int size, int values[]);
   public ConstMFInt32(int values[]);

   public void getValue(int values[]);

   public int get1Value(int index);

   public String toString();   // This overrides a method in Object
}

public class ConstMFNode extends ConstMField
{
   public ConstMFNode(int size, BaseNode node[]);
   public ConstMFNode(BaseNode node[]);

   public void getValue(BaseNode node[]);

   public BaseNode get1Value(int index);

   public String toString();   // This overrides a method in Object
}

public class ConstMFRotation extends ConstMField
{
   public ConstMFRotation(float rotations[][]);
   public ConstMFRotation(float rotations[]);
   public ConstMFRotation(int size, float rotations[]);

   public void getValue(float rotations[][]);
   public void getValue(float rotations[]);

   public void get1Value(int index, float rotations[]);
   public void get1Value(int index, SFRotation rotation);

   public String toString();   // This overrides a method in Object
}

public class ConstMFString extends ConstMField
{
   public ConstMFString(int size, String s[]);
   public ConstMFString(String s[]);
   public void getValue(String values[]);

   public String get1Value(int index);

   public String toString();   // This overrides a method in Object
}

public class ConstMFTime extends ConstMField
{
   public ConstMFTime(int size, double times[]);
   public ConstMFTime(double times[]);

   public void getValue(double times[]);

   public double get1Value(int index);

   public String toString();   // This overrides a method in Object
}

public class ConstMFVec2f extends ConstMField
{
   public ConstMFVec2f(float vec2s[][]);
   public ConstMFVec2f(float vec2s[]);
   public ConstMFVec2f(int size, float vec2s[]);

   public void getValue(float vec2s[][]);
   public void getValue(float vec2s[]);

   public void get1Value(int index, float vec2s[]);
   public void get1Value(int index, SFVec2f vec);

   public String toString();   // This overrides a method in Object
}

public class ConstMFVec3f extends ConstMField
{
   public ConstMFVec3f(float vec3s[][]);
   public ConstMFVec3f(float vec3s[]);
   public ConstMFVec3f(int size, float vec3s[]);

   public void getValue(float vec3s[][]);
   public void getValue(float vec3s[]);

   public void get1Value(int index, float vec3s[]);
   public void get1Value(int index, SFVec3f vec);

   public String toString();   // This overrides a method in Object
}
```

APPENDIX KK

### B.9.2.3 vrml.node package

```
package vrml.node;

//
// This is the general Node class
//
public abstract class Node extends BaseNode
{
   // Get an EventIn by name. Return value is write-only.
   //   Throws an InvalidEventInException if eventInName isn't a valid
   //   eventIn name for a node of this type.
   public final Field getEventIn(String eventInName);

   // Get an EventOut by name. Return value is read-only.
   //   Throws an InvalidEventOutException if eventOutName isn't a valid
   //   eventOut name for a node of this type.
   public final ConstField getEventOut(String eventOutName);

   // Get an exposed field by name.
   //    Throws  an  InvalidExposedFieldException  if  exposedFieldName  isn't  a
valid
   //   exposedField name for a node of this type.
   public final Field getExposedField(String exposedFieldName);

   public String toString();   // This overrides a method in Object
}

//
// This is the general Script class, to be subclassed by all scripts.
// Note that the provided methods allow the script author to explicitly
// throw tailored exceptions in case something goes wrong in the
// script.
//
public abstract class Script extends BaseNode
{
   // This method is called before any event is generated
   public void initialize();

   // Get a Field by name.
   //   Throws an InvalidFieldException if fieldName isn't a valid
   //   field name for a node of this type.
   protected final Field getField(String fieldName);

   // Get an EventOut by name.
   //   Throws an InvalidEventOutException if eventOutName isn't a valid
   //   eventOut name for a node of this type.
   protected final Field getEventOut(String eventOutName);

   // Get an EventIn by name.
   //   Throws an InvalidEventInException if eventInName isn't a valid
   //   eventIn name for a node of this type.
   protected final Field getEventIn(String eventInName);

   // processEvents() is called automatically when the script receives
   //   some set of events. It shall not be called directly except by its
   //   subclass.
   //   count indicates the number of events delivered.
   public void processEvents(int count, Event events[]);

   // processEvent() is called automatically when the script receives
   // an event.
   public void processEvent(Event event);

   // eventsProcessed() is called after every invocation of processEvents().
   public void eventsProcessed()

   // shutdown() is called when this Script node is deleted.
   public void shutdown();

   public String toString();   // This overrides a method in Object
}
```

## B.10 Example of exception class

public class **InvalidEventInException** extends IllegalArgumentException

```
{
   /**
    * Constructs an InvalidEventInException with no detail message.
    */
   public InvalidEventInException(){
      super();
   }
   /**
    * Constructs an InvalidEventInException with the specified detail
    * message.
    * A detail message is a String that describes this particular exception.
    * @param s the detail message
    */
   public InvalidEventInException(String s){
      super(s);
   }
}

public class InvalidEventOutException extends IllegalArgumentException
{
   public InvalidEventOutException(){
      super();
   }
   public InvalidEventOutException(String s){
      super(s);
   }
}
```

```
public class InvalidExposedFieldException extends IllegalArgumentException
{
   public InvalidExposedFieldException(){
      super();
   }
   public InvalidExposedFieldException(String s){
      super(s);
   }
}

public class InvalidFieldChangeException extends IllegalArgumentException
{
   public InvalidFieldChangeException(){
      super();
   }
   public InvalidFieldChangeException(String s){
      super(s);
   }
}

public class InvalidFieldException extends IllegalArgumentException
{
   public InvalidFieldException(){
      super();
   }
   public InvalidFieldException(String s){
      super(s);
   }
}

public class InvalidRouteException extends IllegalArgumentException
{
   public InvalidRouteException(){
      super();
   }
   public InvalidRouteException(String s){
      super(s);
   }
}

public class InvalidVRMLSyntaxException extends Exception
{
   public InvalidVRMLSyntaxException(){
      super();
   }
   public InvalidVRMLSyntaxException(String s){
      super(s);
   }

   public String getMessage();  // This overrides a method in Exception
}
```

APPENDIX KK

# Annex C

**(normative)**

# ECMAScript scripting reference

## C.1 Introduction and table of contents

This annex describes the ECMAScript programming language that enables Script nodes (see 6.40, Script) to interact with VRML scenes. See 4.12, Scripting, for a general description of scripting languages in ISO/IEC 14772. Note that support for the ECMAScript is not required by ISO/IEC 14772, but any access of ECMAScript from within VRML Script nodes shall conform with the requirements specified in this annex.

# C.2 Language

ECMAScript is a general purpose, cross-platform programming language that can be used with ISO/IEC 14772 to provide scripting of events, objects, and actions. ECMAScript is fully described in 2.[ESCR]. Prior to standardization as ECMA-262, ECMAScript was known as Netscape JavaScript. Several syntactic entities in this annex reflect this origin.

# C.3 Supported protocol in the Script node's *url* field

## C.3.1 *url* field

The *url* field of the Script node may contain URL references to ECMAScript code as illustrated below:

```
Script { url "http://foo.com/myScript.js" }
```

The javascript: protocol allows the script to be placed inline as follows:

```
Script { url "javascript: function foo( ) { ... }" }
```

Browsers supporting the ECMAScript scripting language shall support the javascript: protocol as well as the the other required protocols (see 7, Conformance and minimum support requirements).

The *url* field may contain multiple URL's referencing either a remote file or in-line code as shown in the following example:

```
Script {
  url [ "http://foo.com/myScript.js",
  "javascript: function foo( ) { ... }" ]
}
```

## C.3.2 File extension

The file extension for ECMAScript source code is '.js', unless a protocol returning mime types is used (such as HTTP). In that case, any suffix is allowed as long as the proper mime type is returned (see C.3.3, Mime type).

## C.3.3 MIME type

The MIME type for ECMAScript source code is defined as follows:

```
application/x-javascript
```

# C.4 eventIn handling

## C.4.1 Receiving eventIns

Events sent to the Script node are passed to the corresponding ECMAScript function in the script. The script is specified in the *url* field of the Script node. The function's name is the same as the eventIn and is passed two arguments, the event value and its timestamp (see C.4.2, Parameter passing and the eventIn function). If there is no corresponding ECMAScript function in the script, the browser's behaviour is undefined.

For example, the following Script node has one eventIn field whose name is *start*:

```
Script {
  eventIn SFBool start
  url "javascript: function start(value, timestamp) { ... }"
}
```

In the above example, when the *start* eventIn is sent, the start( ) function is executed.

## C.4.2 Parameter passing and the eventIn function

When a Script node receives an eventIn, a corresponding function in the file specified in the *url* field of the Script node is called. This function has two arguments. The value of the eventIn is passed as the first argument and the timestamp of the eventIn is passed as the second argument. The type of the value is the same as the type of the eventIn and the type of the timestamp is SFTime. C.6.1, VRML field to ECMAScript variable conversion, provides a description of how VRML types appear in ECMAScript. The values of the parameters have no visibility outside the function.

## C.4.3 eventsProcessed( ) function

Authors may define an eventsProcessed() function that is called after some set of events has been received. This allows Script nodes that do not rely on the ordering of events received to generate fewer events than an equivalent Script node that generates events whenever events are received (see C.4.1, Receiving eventIns).

The eventsProcessed( ) function takes no parameters. Events generated from it are given the timestamp of the last event processed.

## C.4.4 initialize( ) function

Authors may define a function named initialize( ) which is invoked before the browser presents the world to the user and before any events are processed by any nodes in the same VRML file as the Script node containing this script (see 4.12.3, Initialize() and shutdown()).

The initialize( ) function has no parameters. Events generated from initialize( ) are given the timestamp of when the Script node was loaded.

## C.4.5 shutdown( ) function

Authors may define a function named shutdown( ) which is invoked when the corresponding Script node is deleted or when the world containing the Script node is unloaded or replaced by another world (see 4.12.3, Initialize() and shutdown()).

APPENDIX KK

The shutdown( ) function has no parameters. Events generated from shutdown( ) are given the timestamp of when the Script node was deleted.

# C.5 Accessing fields and events

## C.5.1 Accessing fields and eventOuts of the Script

The fields and eventOuts of a Script node are accessible from its ECMAScript functions. As in all other nodes, the fields are accessible only within the Script. The eventIns are not accessible. The Script node's eventIns can be routed to and its eventOuts can be routed from. Another Script node with a reference to this node can access its eventIns and eventOuts as for any other node.

A field defined in a Script node is available to the script by using its name. Its value can be read or written. This value is persistent across function calls. EventOuts defined in the script node can also be read. The value is the last value assigned.

## C.5.2 Accessing fields and eventOuts of other nodes

The script can access any exposedField, eventIn or eventOut of any node to which it has access:

```
DEF SomeNode Transform { }
Script {
  field SFNode node USE SomeNode
  eventIn SFVec3f pos
  directOutput TRUE
  url "javascript:
    function pos(value) {
      node.set_translation = value;
    }"
}
```

This example sends a set_translation eventIn to the Transform node. An eventIn on a passed node can appear only on the left side of the assignment. An eventOut in the passed node can appear only on the right side, which reads the last value sent out. Fields in the passed node cannot be accessed. However, exposedFields can either send an event to the "*set_...*" eventIn or read the current value of the "*..._changed*" eventOut. This follows the routing model of the rest of ISO/IEC 14772.

Events generated by setting an eventIn on a node are sent at the completion of the currently executing function. The eventIn shall be assigned a value of the same datatype; no partial assignments are allowed. For example, it is not possible to assign the red value of an SFColor eventIn. Since eventIns are strictly write-only, the remainder of the partial assignment would have invalid field values. Assigning to the eventIn field multiple times during one execution of the function still only sends one event and that event is the last value assigned.

## C.5.3 Sending eventOuts

Assigning to an eventOut of a Script node, or a component of an eventOut (i.e. MF eventOut or a property of an SF eventOut), sends an event to that eventOut. Events are sent at the end of script execution. An eventOut may be assigned a value multiple times within the script, but the value sent shall be the last value assigned to the eventOut. If the value of individual components of an eventOut are changed, the last value given to each component shall be sent. Components that are not changed in the script, send their initial value determined at the beginning of the script

Microsoft et al. Exhibit 1005

execution. For example, the following script segment produces an eventOut value of (4, 3, 1) for the eventOut SFVec3f *foo_changed* with an initial value of (6, 6, 6):

```
a = foo_changed; // copy by reference a(6,6,6)
a.x = 5;         // foo_changed(5,6,6)
a.z = 1;         // foo_changed(5,6,1)
b = foo_changed; // copy by reference b(5,6,1)
b.x = 4;         // foo_changed(4,6,1)
c = a;           // copy by reference c(4,6,1)
c.y = 3;         // foo_changed(4,3,1))
```

# C.6 ECMAScript objects

## C.6.1 Notational conventions

Since ECMAScript is an untyped language it has no language constructs to describe the types of parameters passed to, or values returned from, functions. Therefore this annex uses a notational convention to describe these types. Parameters passed are preceded by their type, and the type of any return value precedes the function name. Normally these types correspond to VRML field types, so those names are used. In the case of no return value, the identifier *void* is used. In the case of a ECMAScript numeric value or numeric array return, the identifier *numeric* or *numeric[ ]* is used. In the case of a string return, the identifier *String* is used.

## C.6.2 VRML field to ECMAScript variable conversion

ECMAScript native datatypes consist of boolean, numeric and string. The language is not typed, so datatypes are implicit upon assignment. The VRML SFBool is mapped to the ECMAScript boolean. In addition to the ECMAScript *true* and *false* constants, the VRML TRUE and FALSE values may be used. The VRML SFInt32, SFFloat and SFTime fields are mapped to the numeric datatype and will be maintained in double precision accuracy. These types are passed by value in function calls. All other VRML fields are mapped to ECMAScript objects. ECMAScript objects are passed by reference.

The ECMAScript boolean, numeric and string are automatically converted to other datatypes when needed. See 2.[ESCR] for more details.

In ECMAScript, assigning a new value to a variable gives the variable the datatype of the new value, in addition to the value. Scalar values (boolean and numeric) are assigned by copying the value. Other objects are assigned by reference.

When assignments are made to eventOuts and fields, the values are converted to the VRML field type. Values assigned are always copied. This contrasts with normal assignment in ECMAScript where all assignments except for scalar are performed by reference.

For eventOut objects, assignment copies the value to the eventOut, which will be sent upon completion of the current function. Assigning an eventOut to an internal variable copies by reference. Subsequent assignments to that internal variable will behave like assignments to the eventOut (i.e., an event will be sent at the end of the function). Field objects behave identically to eventOut objects, except that no event is sent upon completion of the function.

Assigning an element of an MF object to an internal variable creates a reference to that element. The type shall be the corresponding SF object type. If the MF object is an eventOut and an assignment is made to the internal variable, an event will be sent at the end of the function. Assigning an SF object to an element of an MF object which is a field or eventOut (which shall be of the corresponding type) copies the value of the SF object into the MF object element. If the MF object is an eventOut an event will be sent at the end of the function.

## C.6.3 Browser object

This subclause lists the class static functions available in the *Browser* object which allow scripts to get and set browser information. Descriptions of the functions are provided in 4.12.10, Browser script interface. The syntax for a call is:

```
mymfnode = Browser.createVrmlFromString('Sphere {}');
```

Table C.1 describes the Browser object's functions, parameters, and return values.

**Table C.1 -- Browser object functions**

| Return value | Function |
|---|---|
| String | **getName**( ) |
| String | **getVersion**( ) |
| numeric | **getCurrentSpeed**( ) |
| numeric | **getCurrentFrameRate**( ) |
| String | **getWorldURL**( ) |
| void | **replaceWorld**( MFNode nodes ) |
| MFNode | **createVrmlFromString**( String vrmlSyntax ) |
| void | **createVrmlFromURL**( MFString url, Node node, String event ) |
| void | **addRoute**( SFNode fromNode, String fromEventOut,<br>SFNode toNode, String toEventIn) |
| void | **deleteRoute**( SFNode fromNode, String fromEventOut,<br>SFNode toNode, String toEventIn ) |
| void | **loadURL**( MFString url, MFString parameter ) |
| void | **setDescription**( String description ) |

## C.6.4 SFColor object

### C.6.4.1 Description

The SFColor object corresponds to a VRML SFColor field. All properties are accessed using the syntax *sfColorObjectName.<property>*, where *sfColorObjectName* is an instance of an SFColor object. The properties may also be accessed by the indices [0] for red, [1] for green and [2] blue. All functions are invoked using the syntax *sfColorObjectName.method(<argument-list>)*, where *sfColorObjectName* is an instance of an SFColor object.

### C.6.4.2 Instance creation function

*sfColorObjectName* = new SFColor(float *r*, float *g*, float *b)*

where

*r*, *g*, and *b* are the red, green, and blue values of the colour. Missing values will be filled by 0.0.

### C.6.4.3 Properties

The properties of the SFColor object are described in Table C.2.

**Table C.2 -- SFColor properties**

| Property | Description |
|---|---|
| numeric *r* | red component of the colour |
| numeric *g* | green component of the colour |
| numeric *b* | blue component of the colour |

### C.6.4.4 Functions

The functions of the SFColor object are described in Table C.3.

**Table C.3 -- SFColor functions**

| Function | Description |
|---|---|
| void setHSV(float *h*, float *s*, float *v*) | Sets the value of the colour by specifying the values of *hue*, *saturation*, and *value*. |
| numeric[3] getHSV( ) | Returns the value of the colour in a 3 element numeric array, with *hue* at index 0, *saturation* at index 1, and *value* at index 2. |
| String toString( ) | Returns a String containing the ISO/IEC 14772 UTF-8 encoded value of *r*, *g* and *b*. |

## C.6.5 SFImage object

### C.6.5.1 Description

The SFImage object corresponds to a VRML SFImage field.

### C.6.5.2 Instance creation function

*sfImageObjectName* = new SFImage(numeric *x*, numeric *y*, numeric *comp*, MFInt32 *array)*

where

*x* is the x-dimension of the image. *y* is the y-dimension of the image. *comp* is the number of components of the image (1 for greyscale, 2 for greyscale+alpha, 3 for rgb, 4 for rgb+alpha). *Array* contains the *x* × *y* values for the pixels of the image. The format of each pixel is an SFImage as in the PixelTexture node.

### C.6.5.3 Properties

The properties of the SFImage object are listed in Table C.4.

**Table C.4 -- SFImage properties**

| **Property** | **Description** |
|---|---|
| numeric *x* | x dimension of the image |
| numeric *y* | y dimension of the image |
| numeric *comp* | *number of components of the image:*<br>*1: greyscale*<br>*2: greyscale + alpha*<br>*3: rgb*<br>*4: rgb + alpha* |
| MFInt32 *array* | image data |

### C.6.5.4 Functions

The function of the SFImage object is described in Table C.5.

**Table C.5 -- SFImage function**

| **Function** | **Description** |
|---|---|
| String toString( ) | Returns a String containing the ISO/IEC 14772 UTF-8 encoded value of x, y, comp and array. |

## C.6.6 SFNode object

### C.6.6.1 Description

The SFNode object corresponds to a VRML SFNode field.

### C.6.6.2 Instance creation function

*sfNodeObjectName* = new SFNode(String *vrmlstring)*

where

*vrmlstring* is an ISO 10646 string containing a legal VRML string as described in 4.12.10.9, MFNode createVrmlFromString( SFString vrmlSyntax ). If the string produces other than one top-level node, the results are undefined. The string may contain any number of ROUTE's, PROTO's, and EXTERNPROTO's in accordance with 4.12.10.9, MFNode createVrmlFromString( SFString vrmlSyntax ).

### C.6.6.3 Properties

Each node may assign values to its eventIns and obtain the last output values of its eventOuts using the *sfNodeObjectName.eventName* syntax.

### C.6.6.4 functions

The function of the SFNode object is described in Table C.6.

**Table C.6 -- SFNode function**

| Function | Description |
|---|---|
| String toString( ) | Returns the VRML UTF-8 string that, if parsed as the value of an SFNode field, would produce this node. If the browser is unable to reproduce this node, the name of the node followed by the open brace and close brace shall be returned. Additional information may be included as one or more VRML comment strings. |

## C.6.7 SFRotation object

### C.6.7.1 Description

The SFRotation object corresponds to a VRML SFRotation field. It has four numeric properties: x, y, z (the axis of rotation) and angle. These may also be addressed by indices [0] through [3].

### C.6.7.2 Instance creation functions

*sfRotationObjectName* = new SFRotation(numeric *x*, numeric *y*, numeric *z*, numeric *angle)*

where

*x*, *y*, and *z* are the axis of the rotation. *angle* is the angle of the rotation (in radians). Missing values default to 0.0, except *y*, which defaults to 1.0.

*sfRotationObjectName* = new SFRotation(SFVec3f *axis,* numeric *angle*)

where

*axis* is the axis of rotation. *angle* is the angle of the rotation (in radians)

*sfRotationObjectName* = new SFRotation(SFVec3f *fromVector,* SFVec3f *toVector*) where

*fromVector* and *toVector* are normalized and the rotation value that would rotate from the *fromVector* to the *toVector* is stored in the object.

### C.6.7.3 Properties

The properties of the SFRotation object are described in Table C.7.

**Table C.7 -- SFRotation properties**

| Property | Description |
|---|---|
| numeric *x* | first value of the axis vector |
| numeric *y* | second value of the axis vector |
| numeric *z* | third value of the axis vector |
| numeric *angle* | the angle of the rotation (in radians) |

### C.6.7.4 Functions

The functions of the SFRotation object are described in Table C.8.

**Table C.8 -- SFRotation functions**

| Function | Description |
|---|---|
| SFVec3f getAxis( ) | Returns the axis of rotation. |
| SFRotation inverse( ) | Returns the inverse of this object's rotation. |
| SFRotation multiply(SFRotation *rot*) | Returns the object multiplied by the passed value. |
| SFVec3f multVec(SFVec3f *vec*) | Returns the value of *vec* multiplied by the matrix corresponding to this object's rotation. |
| void setAxis(SFVec3f *vec*) | Sets the axis of rotation to the value passed in *vec*. |
| SFRotation slerp(SFRotation *dest*, numeric *t*) | Returns the value of the spherical linear interpolation between this object's rotation and *dest* at value 0 <= *t* <= 1. For *t* = 0, the value is this object's rotation. For *t* = 1, the value is *dest*. |
| String toString( ) | Returns a String containing the ISO/IEC 14772 UTF-8 encoded value of x, y, z, and angle. |

APPENDIX KK

# C.6.8 SFVec2f object

### C.6.8.1 Description

The SFVec2f object corresponds to a VRML SFVec2f field. Each component of the vector can be accessed using the *x* and *y* properties or using C-style array dereferencing (i.e., *sfVec2fObjectName[0]* or *sfVec2fObjectName[1]).*

### C.6.8.2 Instance creation function

*sfVec2fObjectName* = new SFVec2f(numeric *x*, numeric *y)*

Missing values default to 0.0.

### C.6.8.3 Properties

The properties of the SFVec2f object are described in Table C.9.

**Table C.9 -- SFVec2f properties**

| Property | Description |
|---|---|
| numeric *x* | First value of the vector. |
| numeric *y* | Second value of the vector. |

### C.6.8.4 Functions

The functions of the SFVec2f object are described in Table C.10.

**Table C.10 -- SFVec2f functions**

| Function | Description |
|---|---|
| SFVec2f add(SFVec2f *vec*) | Returns the value of the passed value added, component-wise, to the object. |
| SFVec2f divide(numeric *n*) | Returns the value of the object divided by the passed value. |
| numeric dot(SFVec2f *vec*) | Returns the dot product of this vector and the passed value. |
| numeric length( ) | Returns the geometric length of this vector. |
| SFVec2f multiply(numeric *n*) | Returns the value of the object multiplied by the passed value. |
| SFVec2f normalize( ) | Returns the object converted to unit length . |
| SFVec2f subtract(SFVec2f *vec*) | *Returns the value of the passed value subtracted, component-wise, from the object.* |
| String toString( ) | Returns a String containing the ISO/IEC 14772 UTF-8 encoded value of x and y. |

## C.6.9 SFVec3f object

### C.6.9.1 Description

The SFVec3f object corresponds to a VRML SFVec3f field. Each component of the vector can be accessed using the x, y, and z properties or using C-style array dereferencing (i.e., *sfVec3fObjectName[0]*, *sfVec3fObjectName[1]* or *sfVec3fObjectName[2]).*

### C.6.9.2 Instance creation function

*sfVec3fObjectName* = new SFVec3f(numeric *x,* numeric *y,* numeric *z)*

Missing values default to 0.0.

### C.6.9.3 Properties

The properties of the SFVec3f object are described in Table C.11.

**Table C.11 -- SFVec2f properties**

| **Property** | **Description** |
|---|---|
| numeric *x* | First value of the vector. |
| numeric *y* | Second value of the vector. |
| numeric *z* | Third value of the vector. |

### C.6.9.4 Functions

The functions of the SFVec3f object are described in Table C.12.

**Table C.12 -- SFVec3f functions**

| **Function** | **Description** |
|---|---|
| SFVec3f add(SFVec3f *vec*) | Returns the value of the passed value added, component-wise, to the object. |
| SFVec3f cross(SFVec3f *vec*) | Returns the cross product of the object and the passed value. |
| SFVec3f divide(numeric *n*) | Returns the value of the object divided by the passed value. |
| numeric dot(SFVec3f *vec*) | Returns the dot product of this vector and the passed value. |
| numeric length( ) | Returns the geometric length of this vector. |

APPENDIX KK

| | |
|---|---|
| SFVec3f multiply(*numeric n*) | Returns the value of the object multiplied by the passed value. |
| SFVec3f negate( ) | Returns the value of the component-wise negation of the object. |
| SFVec3f normalize( ) | Returns the object converted to unit length . |
| SFVec3f subtract(SFVec3f *vec*) | Returns the value of the passed value subtracted, component-wise, from the object. |
| String toString( ) | Returns a String containing the ISO/IEC 14772 UTF-8 encoded value of x, y, and z. |

## C.6.10 MFColor object

### C.6.10.1 Description

The MFColor object corresponds to a VRML MFColor field. It is used to store a one-dimensional array of SFColor objects. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfColorObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= length results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to SFColor (0, 0, 0).

### C.6.10.2 Instance creation function

*mfColorObjectName* = new MFColor(SFColor *c1*, SFColor *c2, ...)*

The creation function shall initialize the array using 0 or more SFColor-valued expressions passed as parameters.

### C.6.10.3 Property

The property of the MFColor object is described in Table C.13.

**Table C.13 -- MFColor properties**

| **Property** | **Description** |
|---|---|
| numeric *length* | property for getting/setting the number of elements in the array. |

### C.6.10.4 Function

The single function of the MFColor object is described in Table C.14.

**Table C.14 -- MFColor functions**

| **Function** | **Description** |
|---|---|

APPENDIX KK

| | |
|---|---|
| String toString( ) | Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFColor array. |

## C.6.11 MFFloat object

### C.6.11.1 Description

The MFFloat object corresponds to a VRML MFFloat field. It is used to store a one-dimensional array of SFFloat values. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfFloatObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to 0.0.

### C.6.11.2 Instance creation function

*mfFloatObjectName* = new MFFloat(numeric *n1*, numeric *n2, ...)*

where

The creation function shall initialize the array using 0 or more numeric-valued expressions passed as parameters.

### C.6.11.3 Property

The property of the MFFloat object is described in Table C.15.

**Table C.15 -- MFFloat properties**

| Property | Description |
|---|---|
| numeric *length* | property for getting/setting the number of elements in the array. |

### C.6.11.4 Function

The single function of the MFFloat object is described in Table C.16.

**Table C.16 -- MFFloat function**

| Function | Description |
|---|---|
| String toString( ) | Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFFloat array. |

## C.6.12 MFInt32 object

### C.6.12.1 Description

The MFInt32 object corresponds to a VRML MFInt32 field. It is used to store a one-dimensional array of SFInt32 values. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfInt32ObjectName*[*index]*, where *index* is an integer-valued expression with 0 <= *index* < length and length is

Microsoft et al. Exhibit 1005

the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to 0.

### C.6.12.2 Instance creation function

*mfInt32ObjectName* = new MFInt32(numeric *n1*, numeric *n2, ...)*

where

The creation function shall initialize the array using 0 or more integer-valued expressions passed as parameters.

### C.6.12.3 Property

The property of the MFInt32 object is described in Table C.17.

**Table C.17 -- MFInt32 property**

| Property | Description |
|---|---|
| numeric *length* | property for getting/setting the number of elements in the array. |

### C.6.12.4 Function

The single function of the MFInt32 object is described in Table C.18.

**Table C.18 -- MFInt32 function**

| Function | Description |
|---|---|
| String toString( ) | Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFInt32 array. |

## C.6.13 MFNode object

### C.6.13.1 Description

The MFNode object corresponds to a VRML MFNode field. It is used to store a one-dimensional array of SFNode objects. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfNodeObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to NULL.

### C.6.13.2 Instance creation function

*mfNodeObjectName* = new MFNode(SFNode *n1*, SFNode *n2, ...)*

where

The creation function shall initialize the array using 0 or more SFNode-valued expressions passed as parameters.

### C.6.13.3 Property

The property of the MFNode object is described in Table C.19.

**Table C.19 -- MFNode property**

| Property | Description |
|---|---|
| numeric *length* | property for getting/setting the number of elements in the array. |

### C.6.13.4 Function

The single function of the MFNode object is described in Table C.20.

**Table C.20 -- MFNode function**

| Function | Description |
|---|---|
| String toS tring( ) | Returns the VRML UTF-8 string that, if parsed as the value of a MFNode field, would produce this array of nodes. If the browser is unable to reproduce this node, the name of the node followed by the open brace and close brace shall be returned. Additional information may be included as one or more VRML comment strings |

## C.6.14 MFRotation object

### C.6.14.1 Description

The MFRotation object corresponds to a VRML MFRotation field. It is used to store a one-dimensional array of SFRotation objects. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfRotationObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to SFRotation (0, 0, 1, 0).

### C.6.14.2 Instance creation function

*mfRotationObjectName* = new MFRotation(SFRotation *r1,* SFRotation *r2, ...)*

where

The creation function shall initialize the array using 0 or more SFRotation-valued expressions passed as parameters.

### C.6.14.3 Property

The property of the MFRotation object is described in Table C.21.

**Table C.21 -- MFRotation property**

| Property | Description |
|---|---|

| | |
|---|---|
| numeric *length* | property for getting/setting the number of elements in the array. |

### C.6.14.4 Function

The single function of the MFRotation object is described in Table C.22.

**Table C.22 -- MFRotation function**

| Function | Description |
|---|---|
| String toString( ) | Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFRotation array. |

## C.6.15 MFString object

### C.6.15.1 Description

The MFString object corresponds to a VRML 2.0 MFString field. It is used to store a one-dimensional array of String objects. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfStringObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to the empty string.

### C.6.15.2 Instance creation function

*mfStringObjectName* = new MFString(String *s1*, String *s2, ...)*

where

The creation function shall initialize the array using 0 or more String-valued expressions passed as parameters.

### C.6.15.3 Property

The property of the MFString object is described in Table C.23.

**Table C.23 -- MFString property**

| Property | Description |
|---|---|
| numeric *length* | property for getting/setting the number of elements in the array. |

### C.6.15.4 Function

The single function of the MFString object is described in Table C.24.

**Table C.24 -- MFString function**

| Function | Description |
|---|---|

| | |
|---|---|
| String toString( ) | Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFString array. |

## C.6.16 MFTime object

### C.6.16.1 Description

The MFTime object corresponds to a VRML MFTime field. It is used to store a one-dimensional array of SFTime values. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfTimeObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to 0.0.

### C.6.16.2 Instance creation function

*mfTimeObjectName* = new MFTime(numeric *n1,* numeric *n2, ...)*

The creation function shall initialize the array using 0 or more numeric-valued expressions passed as parameters.

### C.6.16.3 Property

The property of the MFTime object is described in Table C.25.

**Table C.25 -- MFTime property**

| Property | Description |
|---|---|
| numeric *length* | property for getting/setting the number of elements in the array. |

### C.6.16.4 Function

The function of the MFTime object is described in Table C.26.

**Table C.26 -- MFTime function**

| Function | Description |
|---|---|
| String toString( ) | Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFTime array. |

## C.6.17 MFVec2f object

### C.6.17.1 Description

The MFVec2f object corresponds to a VRML MFVec2f field. It is used to store a one-dimensional array of SFVec2f objects. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfVec2fObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to SFVec2f (0, 0).

### C.6.17.2 Instance creation function

*mfVec2fObjectName* = new MFVec2f(SFVec2f *v1*, SFVec2f *v2, ...)*

The creation function shall initialize the array using 0 or more SFVec2f-valued expressions passed as parameters.

### C.6.17.3 Property

The property of the MFVec2f object is described in Table C.27.

**Table C.27 -- MFVec2f property**

| Property | Description |
|---|---|
| numeric *length* | property for getting/setting the number of elements in the array. |

### C.6.17.4 Function

The single function of the MFVec2f object is described in Table C.28.

**Table C.28 -- MFVec2f function**

| Function | Description |
|---|---|
| String toString( ) | Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFVec2f array. |

## C.6.18 MFVec3f object

### C.6.18.1 Description

The MFVec3f object corresponds to a VRML MFVec3f field. It is used to store a one-dimensional array of SFVec3f objects. Individual elements of the array can be referenced using the standard C-style dereferencing operator (e.g., *mfVec3fObjectName*[*index*], where *index* is an integer-valued expression with 0 <= *index* < length and length is the number of elements in the array). Assigning to an element with *index* >= *length* results in the array being dynamically expanded to contain length elements. All elements not explicitly initialized are set to SFVec3f (0, 0, 0).

### C.6.18.2 Instance creation function

*mfVec3fObjectName* = new MFVec3f(SFVec3f *v1*, SFVec3f *v2,...)*

where

The creation function shall initialize the array using 0 or more SFVec3f-valued expressions passed as parameters.

### C.6.18.3 Property

The property of the MFVec3f object is described in Table C.29.

**Table C.29 -- MFVec3f property**

| Property | Description |
|---|---|
| numeric *length* | property for getting/setting the number of elements in the array. |

### C.6.18.4 Function

The single function of the MFVec3f object is described in Table C.30.

**Table C.30 -- MFVec3f function**

| Function | Description |
|---|---|
| String toString( ) | Returns a String containing the ISO/IEC 14772 utf 8 encoded value of the MFVec3f array. |

## C.6.19 VrmlMatrix object

### C.6.19.1 Description

The VrmlMatrix object provides many useful functions for performing manipulations on 4x4 matrices. Each of element of the matrix can be accessed using C-style array dereferencing (i.e., vrmlMatrixObjectName[0][1] is the element in row 0, column 1). The results of dereferencing a VrmlMatrix object using a single index (i.e., vrmlMatrixObjectName[0]) are undefined. The translation elements are in the fourth row. For example, vrmlMatrixObjectName[3][0] is the X offset.

### C.6.19.2 Instance creation functions

*VrmlMatrixObjectName* = new VrmlMatrix(
numeric *f11*, numeric *f12*, numeric *f13*, numeric *f14*,
numeric *f21*, numeric *f22*, numeric *f23*, numeric *f24*,
numeric *f31*, numeric *f32*, numeric *f33*, numeric *f34*,
numeric *f41*, numeric *f42*, numeric *f43*, numeric *f44)*

A new matrix initialized with the values in *f11* through *f44* is created and returned. The translation values will be *f41, f42*, and *f43*.

*VrmlMatrixObjectName* = new VrmlMatrix( )

A new matrix initialized with the identity matrix is created and returned.

### C.6.19.3 Properties

The VRMLMatrix object has no properties.

### C.6.19.4 Functions

The functions of the VRMLMatrix object are listed in Table C.31.

**Table C.31 -- VRMLMatrix functions**

| Function | Description |
|---|---|
| void setTransform(SFVec3f *translation*,<br>SFRotation *rotation*,<br>SFVec3f *scale*,<br>SFRotation *scaleOrientation*,<br>SFVec3f *center*) | Sets the VrmlMatrix to the passed values. Any of the rightmost parameters may be omitted. The function has 0 to 5 parameters. For example, specifying 0 parameters results in an identity matrix while specifying 1 parameter results in a translation and specifying 2 parameters results in a translation and a rotation. Any unspecified parameter is set to its default as specified for the Transform node. |
| void getTransform(SFVec3f *translation*,<br>SFRotation *rotation*,<br>SFVec3f *scale*) | Decomposes the VrmlMatrix and returns the components in the passed *translation*, *rotation*, and *scale* objects. The types of these passed objects is the same as the first three arguments to **setTransform**. If any passed object is not sent, or if the null object is sent for any value, that value is not returned. Any projection or shear information in the matrix is ignored. |
| VrmlMatrix inverse( ) | Returns a VrmlMatrix whose value is the inverse of this object. |
| VrmlMatrix transpose( ) | Returns a VrmlMatrix whose value is the transpose of this object. |
| VrmlMatrix multLeft(VrmlMatrix *matrix*) | Returns a VrmlMatrix whose value is the object multiplied by the passed *matrix* on the left. |
| VrmlMatrix multRight(VrmlMatrix *matrix*) | Returns a VrmlMatrix whose value is the object multiplied by the passed *matrix* on the right. |
| SfVec3f multVecMatrix(SFVec3f *vec*) | Returns an SFVec3f whose value is the object multiplied by the passed row vector. |
| SFVec3f multMatrixVec(SFVec3f *vec*) | Returns an SFVec3f whose value is the object multiplied by the passed column vector. |
| String toString( ) | Returns a String containing the values of the VrmlMatrix. |

# C.7 Examples

The following is an example of a Script node which determines whether a given colour contains a lot of red. The Script node exposes a Color field, an eventIn, and an eventOut:

```
DEF Example_1 Script {
        field    SFColor currentColor 0 0 0
    eventIn  SFColor colorIn
```

APPENDIX KK

```
    eventOut SFBool  isRed

    url "javascript:
        function colorIn(newColor, ts) {
            // This function is called when a colorIn event is received
            currentColor = newColor;
        }

        function eventsProcessed( ) {
            if (currentColor[0] >= 0.5)
                // if red is at or above 50%
                isRed = true;
        }"
}
```

Details on when the functions defined in Example_1 Script are called are provided in 4.12.2, Script execution.

The following example illustrate use of the createVrmlFromURL( ) function:

```
 DEF Example_2 Script {
    field   SFNode myself USE Example_2
    field   SFNode root USE ROOT_TRANSFORM
    field   MFString url "foo.wrl"
    eventIn MFNode   nodesLoaded
    eventIn SFBool   trigger_event

    url "javascript:
        function trigger_event(value, ts){
            // do something and then fetch values
            Browser.createVRMLFromURL(url, myself, 'nodesLoaded');
        }

        function nodesLoaded(value, timestamp){
            if (value.length > 5) {
                 // do something more than 5 nodes in this MFNode...
            }
            root.addChildren = value;
        }"
}
```

The following example illustrates use of the addRoute( ) function:

```
DEF Sensor TouchSensor {}

DEF Baa Script {
    field   SFNode myself USE Baa
    field   SFNode fromNode USE Sensor
    eventIn SFBool clicked
    eventIn SFBool trigger_event

    url "javascript:
        function trigger_event(eventIn_value){
            // do something and then add routing
            Browser.addRoute(fromNode, 'isActive', myself, 'clicked');
        }

        function clicked(value){
            // do something
        }"
}
```

The following example illustrates assigning with references and assigning by copying:

```
Script {
    eventIn  SFBool  eI
    eventOut SFVec3f eO
    field    MFVec3f f []

    url "javascript:
        function eI( ) {
            eO = new SFVec3f(0,1,2);  // ’eO’ contains the value
                                      // (0,1,2) which will be sent
                                      // out when the function
                                      // is complete.
            a = eO;                   // ’a’ references the eventOut
                                      // ’e0’
            b = a;                    // ’a’ and ’b’ now both reference
                                      // ’e0’
            a.x = 3;                  // ’e0’ will send (3,1,2) at the
                                      // end of the function
            f[1] = a;                 // ’f[1]’ contains the value
                                      // (3,1,2).
            c = f[1];                 // ’c’ reference the field
                                      // element f[1]
            f[1].y = 4;               // ’f[1]’ and ’c’ both contain
                                      // (3,4,2)
        }"
}
```

The following example illustrates uses of the fields and functions of SFVec3f and MFVec3f:

```
DEF SCR-VEC3F Script {
    eventIn SFTime touched1
    eventIn SFTime touched2
    eventIn SFTime touched3
    eventIn SFTime touched4
    eventOut SFVec3f new_translation
    field SFInt32 count 1
    field MFVec3f verts  []

    url "javascript:
        function initialize( ) {
            verts[0] = new SFVec3f(0, 0, 0);
            verts[1] = new SFVec3f(1, 1.732, 0);
            verts[2] = new SFVec3f(2, 0, 0);
            verts[3] = new SFVec3f(1, 0.577, 1.732);
        }

        function touched1 (value) {
            new_translation = verts[count]; // move sphere around tetra
            count++;
            if (count >= verts.length) count = 1;
        }

        function touched2 (value) {
```

APPENDIX KK

```
            var tVec;
            tVec = new_translation.divide(2); // Zeno’s paradox to origin
            new_translation = new_translation.subtract(tVec);
        }

        function touched4 (value) {
            new_translation = new_translation.negate( );
        }

        function touched3 (value) {
            var a;
            a = verts[1].length( );
            a = verts[3].dot(verts[2].cross(verts[1]));
            a = verts[1].x;
            new_translation = verts[2].normalize( );
            new_translation = new_translation.add(new_translation);
        }"
}
```

# Annex D
(informative)

# Examples

## D.1 Introduction and table of contents

This annex provides a variety of VRML examples.

## D.2 Simple example

This example contains a simple scene defining a view of a red sphere and a blue box, lit by a directional light:

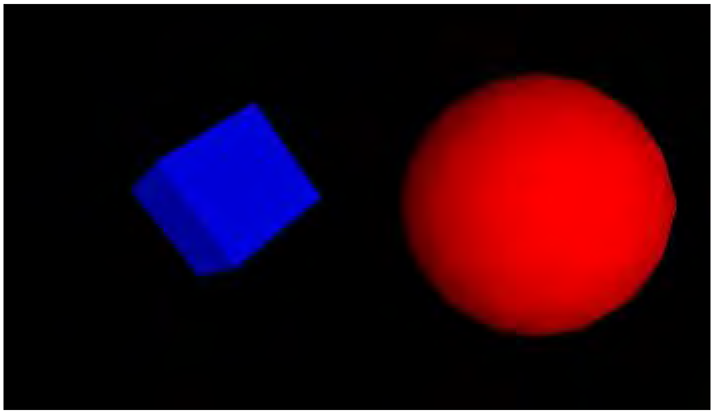


**Figure D.1: Red sphere meets blue box**

```
#VRML V2.0 utf8
Transform {
  children [
    NavigationInfo { headlight FALSE } # We'll add our own light

    DirectionalLight {        # First child
        direction 0 0 -1      # Light illuminating the scene
    }

    Transform {               # Second child - a red sphere
      translation 3 0 1
      children [
        Shape {
          geometry Sphere { radius 2.3 }
          appearance Appearance {
            material Material { diffuseColor 1 0 0 }   # Red
         }
        }
      ]
    }

    Transform {               # Third child - a blue box
      translation -2.4 .2 1
      rotation     0 1 1  .9
      children [
        Shape {
          geometry Box {}
          appearance Appearance {
            material Material { diffuseColor 0 0 1 }  # Blue
         }
        }
      ]
    }

  ] # end of children for world
}
```

Click here to view this example in a VRML browser.

## D.3 Instancing (sharing)

Reading the following file results in three spheres being drawn. The first sphere defines a unit sphere at the origin named "Joe", the second sphere defines a smaller sphere translated along the +x axis, the third sphere is a reference to the second sphere and is translated along the -x axis. If any changes occur to the second sphere (e.g. radius changes), then the third sphere, will change too:

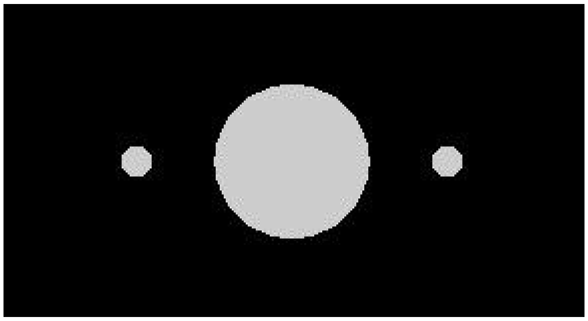


**Figure D.2: Instancing**

```
#VRML V2.0 utf8
Transform {
  children [
    DEF Joe Shape { geometry Sphere {} }
    Transform {
      translation 2 0 0
      children    DEF Joe Shape { geometry Sphere { radius .2 } }
    }
    Transform {
      translation -2 0 0
      children    USE Joe
    }

  ]
}
```

Click here to view this example in a VRML browser. (Note that the spheres are unlit because no appearance was specified.)

## D.4 Prototype example

A simple table with variable colours for the legs and top might be prototyped as:

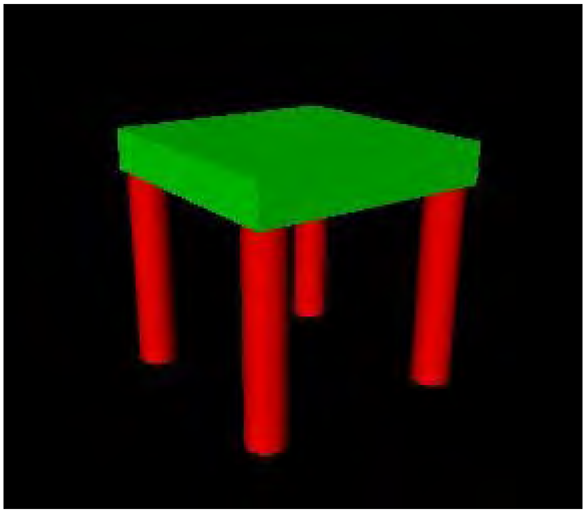


Figure D.3: Prototype

```
#VRML V2.0 utf8
PROTO TwoColorTable [ field SFColor legColor  .8 .4 .7
                      field SFColor topColor .6 .6 .1 ]
{
  Transform {
    children [
      Transform {   # table top
       translation 0 0.6 0
        children
          Shape {
            appearance Appearance {
              material Material { diffuseColor IS topColor }
            }
            geometry Box { size 1.2 0.2 1.2 }
          }
      }

      Transform {   # first table leg
       translation -.5 0 -.5
        children
          DEF Leg Shape {
            appearance Appearance {
              material Material { diffuseColor IS legColor }
            }
            geometry Cylinder { height 1 radius .1 }
          }
      }
      Transform {   # another table leg
       translation .5 0 -.5
        children USE Leg
      }
      Transform {   # another table leg
       translation -.5 0 .5
        children USE Leg
      }
      Transform {   # another table leg
       translation .5 0 .5
        children USE Leg
      }
    ] # End of root Transform's children
  } # End of root Transform
} # End of prototype

# The prototype is now defined. Although it contains a
# number of nodes, only the legColor and topColor fields
# are public. Instead of using the default legColor and
# topColor, this instance of the table has red legs and
# a green top:


TwoColorTable {
  legColor 1 0 0 topColor 0 1 0
}
NavigationInfo { type "EXAMINE" }      # Use the Examine viewer
```

Click here to view this example in a VRML browser.

## D.5 Scripting example

This Script node decides whether or not to open a bank vault given openVault and combinationEntered messages. To do this, it remembers whether or not the correct combination has been entered. The Script node combined with a Sphere, a TouchSensor and a Sound node to show how is works. When the pointing device is over the sphere, the *combinationEntered* eventIn of the Script is sent. Then, when the Sphere is touched (typically when the mouse button is pressed) the Script is sent the *openVault* eventIn. This generates the *vaultUnlocked* eventOut which starts a 'click' sound. Here is the example:

```
#VRML V2.0 utf8

DEF OpenVault Script {
    # Declarations of what's in this Script node:
    eventIn SFTime openVault
    eventIn SFBool combinationEntered
    eventOut SFTime vaultUnlocked
    field SFBool unlocked FALSE

    # Implementation of the logic:
    url "javascript:
        function combinationEntered(value) { unlocked = value; }
        function openVault(value) {
        if (unlocked) vaultUnlocked = value;
    }"
}

Shape {
    appearance Appearance {
        material Material { diffuseColor 1 0 0 }
    }
    geometry Sphere { }
}

Sound {
    source     DEF Click AudioClip {
             url       "click.wav"
             stopTime 1
    }

    minFront   1000
    maxFront   1000
    minBack    1000
    maxBack    1000
}


DEF TS TouchSensor { }

ROUTE TS.isOver TO OpenVault.combinationEntered
ROUTE TS.touchTime TO OpenVault.openVault
ROUTE OpenVault.vaultUnlocked TO Click.startTime
```

Note that the *openVault* eventIn and the *vaultUnlocked* eventOut are of type SFTime, which allows them to be wired directly to a TouchSensor or TimeSensor.

Click here to view this example in a VRML browser.

## D.6 Geometric properties

The following IndexedFaceSet (contained in a Shape node) uses all four of the geometric property nodes to specify vertex coordinates, colours per vertex, normals per vertex, and texture coordinates per vertex (note that the material sets the overall transparency):

```
#VRML V2.0 utf8

Shape {
    geometry IndexedFaceSet {
        coordIndex [ 0, 1, 3, -1, 0, 2, 3, -1 ]
        coord Coordinate {
            point [ 0 0 0, 1 0 0, 1 0 -1, 0.5 1 0 ]
        }
        color Color {
            color [ 0.2 0.7 0.8, 0.5 0 0, 0.1 0.8 0.1, 0 0 0.7 ]
        }
        normal Normal {
            vector [ 0 0 1, 0 0 1, 0 0 1, 0 0 1 ]
        }
        texCoord TextureCoordinate {
            point [ 0 0, 1 0, 1 0.4, 1 1 ]
        }
    }
    appearance Appearance {
        material Material { transparency 0.5 }
        texture  PixelTexture {
            image 2 2 1 0xFF 0x80 0x80 0xFF
        }
    }
}
```

Click here to view this example in a VRML browser.

## D.7 Prototypes and alternate representations

VRML 2.0 has the capability to define new nodes. The following is an example of a new node RefractiveMaterial. This node behaves as a Material node with an added field, *indexOfRefraction*. The list of URLs for the EXTERNPROTO are searched in order. If the browser recognizes the URN,

   **`urn:inet:foo.com:types:RefractiveMaterial`**,

it may treat it as a native type (or load the implementation). Otherwise, the URL,

   **`http://www.myCompany.com/vrmlNodes/RefractiveMaterial.wrl`**,

is used as a backup to ensure that the node is supported on any browsers. See below for the PROTO implementation that treats RefractiveMaterial as a Material (and ignores the *refractiveIndex* field).

```
#VRML V2.0 utf8

# external protype definition
EXTERNPROTO RefractiveMaterial [
    exposedField SFFloat ambientIntensity
    exposedField SFColor diffuseColor
    exposedField SFColor specularColor
    exposedField SFColor emissiveColor
    exposedField SFFloat shininess
    exposedField SFFloat transparency
    exposedField SFFloat indexOfRefraction  ]
[
  "urn:inet:foo.com:types:RefractiveMaterial",
  "http://www.myCompany.com/vrmlNodes/RefractiveMaterial.wrl",
  "refractivematerial.wrl",
]

Shape {
    geometry Sphere { }
    appearance Appearance {
        # Instance of a RefractiveMaterial
        material RefractiveMaterial {
            ambientIntensity  0.2
            diffuseColor      1 0 0
            indexOfRefraction 0.3
        }
    }
}
```

APPENDIX KK

The URL **`http://www.myCompany.com/vrmlNodes/RefractiveMaterial.wrl`** contains the following:

```
#VRML V2.0 utf8

PROTO RefractiveMaterial [            # prototype definition
    exposedField SFFloat ambientIntensity  0
    exposedField SFColor diffuseColor      0.5 0.5 0.5
    exposedField SFColor specularColor     0 0 0
    exposedField SFColor emissiveColor     0 0 0
    exposedField SFFloat shininess         0
    exposedField SFFloat transparency      0
    exposedField SFFloat indexOfRefraction 0.1 ]
{
    Material {
        ambientIntensity IS ambientIntensity
        diffuseColor     IS diffuseColor
        specularColor    IS specularColor
        emissiveColor    IS emissiveColor
        shininess        IS shininess
        transparency     IS transparency
    }
}
```

Note that the name of the new node type, **RefractiveMaterial**, is not used by the browser to decide if the node is native or not; the URL/URN names determine the node's implementation.

Click here to view this example in a VRML browser.

## D.8 Anchor

The *target* parameter can be used by the anchor node to send a request to load a URL into another frame:

```
Anchor {
  url "http://somehost/somefile.html"
  parameter [ "target=name_of_frame" ]
  children Shape { geometry Cylinder {} }
}
```

An Anchor may be used to bind the viewer to a particular *viewpoint* in a virtual world by specifying a URL ending with *#viewpointName*, where *viewpointName* is the DEF name of a viewpoint defined in the world. For example:

```
Anchor {
  url "http://www.school.edu/vrml/someScene.wrl#OverView"
  children Shape { geometry Box {} }
}
```

specifies an anchor that puts the viewer in the *someScene* world bound to the viewpoint named *OverView* when the box is chosen (note that *OverView* is the DEF name of the viewpoint, not the value of the viewpoint's description field).

If no world is specified, the current scene is implied. For example:

```
Anchor {
  url "#Doorway"
  children Shape { geometry Sphere {} }
}
```

binds the user's view to the viewpoint with the DEF name *Doorway* in the current scene.

## D.9 Directional light

A directional light source illuminates only the objects in its enclosing grouping node. The light illuminates everything within this coordinate system including the objects that precede it in the scene graph as shown below:

```
#VRML V2.0 utf8

Group {
    children [
       DEF UnlitShapeOne Transform {
            translation -3 0 0

           children Shape {
               appearance DEF App Appearance {
                   material Material {
                       diffuseColor 0.8 0.4 0.2
                   }
               }
               geometry Box { }
           }
        }

       DEF LitParent Group {
           children [
               DEF LitShapeOne Transform {
                    translation 0 2 0

                   children Shape {
                       appearance USE App
                       geometry Sphere { }
                   }
               }

               # lights the shapes under LitParent
               DirectionalLight { }
               DEF LitShapeTwo Transform {
                    translation 0 -2 0

                   children Shape {
                       appearance USE App
                       geometry Cylinder { }
                    }
                }
           ]
       }

       DEF UnlitShapeTwo Transform {
            translation 3 0 0

           children Shape {
               appearance USE App
               geometry Cone { }
            }
       }
    ]
}
```

Click here to view this example in a VRML browser.

# D.10 PointSet

This simple example defines a PointSet composed of 3 points. The first point is red (1 0 0), the second point is green (0 1 0), and the third point is blue (0 0 1). The second PointSet instances the Coordinate node defined in the first PointSet, but defines different colours:

```
#VRML V2.0 utf8

Shape {
    geometry PointSet {
        coord DEF mypts Coordinate {
            point [ 0 0 0, 2 2 2, 3 3 3 ]
        }
        color Color { color [ 1 0 0, 0 1 0, 0 0 1 ] }
    }
}

Transform {
    translation 2 0 0

    children Shape {
        geometry PointSet {
           coord USE mypts
           color Color { color [ .5 .5 0, 0 .5 .5, 1 1 1 ] }
        }
    }
}
```

Click here to view this example in a VRML browser.

# D.11 Level of detail

The LOD node is typically used for switching between different versions of geometry at specified distances from the viewer. However, if the range field is left at its default value, the browser selects the most appropriate child from the list given. It can make this selection based on performance or perceived importance of the object. Children should be listed with most detailed version first just as for the normal case. This "performance LOD" feature can be combined with the normal LOD function to give the browser a selection of children from which to choose at each distance.

In this example, the browser is free to choose either a detailed or a less-detailed version of the object when the viewer is closer than 10 meters (as measured in the coordinate space of the LOD). The browser should display the less detailed version of the object if the viewer is between 10 and 50 meters and should display nothing at all if the viewer is farther than 50 meters. Browsers should try to honor the hints given by authors, and authors should try to give browsers as much freedom as they can to choose levels of detail based on performance.

```
#VRML V2.0 utf8

LOD {
    range [ 10, 50 ]
    level [
        LOD {
            level [
                Shape { geometry Sphere { } }
                DEF LoRes Shape { geometry Box { } }
            ]
        }
        USE LoRes,
        Shape { } # Display nothing
    ]
}
```

For best results, ranges should be specified only where necessary and LOD nodes should be nested with and without ranges.

Click here to view this example in a VRML browser.

# D.12 Color interpolator

This example interpolates from red to green to blue in a 10 second cycle:

```
#VRML V2.0 utf8

DEF myColor ColorInterpolator {
    key        [   0.0,    0.5,    1.0 ]
    keyValue   [ 1 0 0,  0 1 0,  0 0 1 ] # red, green, blue
}

DEF myClock TimeSensor {
    cycleInterval 10.0      # 10 second animation
    loop          TRUE      # infinitely cycling animation
}

Shape {
    appearance Appearance {
        material DEF myMaterial Material { }
    }
    geometry Sphere { }
}

ROUTE myClock.fraction_changed TO myColor.set_fraction
ROUTE myColor.value_changed TO myMaterial.set_diffuseColor
```

Click here to view this example in a VRML browser.

# D.13 TimeSensor

## D.13.1 Introduction

The TimeSensor is very flexible. The following are some of the many ways in which it can be used:

e. a TimeSensor can be triggered to run continuously by setting *cycleInterval* > 0, and *loop* = TRUE, and then routing a time output from another node that triggers the loop (*e.g.*,, the *touchTime* eventOut of a TouchSensor can be routed to the TimeSensor's *startTime* to start the TimeSensor running).

f. a TimeSensor can be made to run continuously upon reading by setting *cycleInterval* > 0, *startTime* > 0, *stopTime* = 0, and *loop* = TRUE.

## D.13.2 Click to animate

The first example animates a box when the user clicks on it:

```
#VRML V2.0 utf8

DEF XForm Transform {
    children [
        Shape {
            appearance Appearance {
                material Material { diffuseColor 1 0 0 }
            }
            geometry Box {}
        }
        DEF Clicker TouchSensor {}

        # Run once for 2 sec.
        DEF TimeSource TimeSensor { cycleInterval 2.0 }

        # Animate one full turn about Y axis:
        DEF Animation OrientationInterpolator {
            key      [ 0,      .33,       .66,        1.0 ]
            keyValue [ 0 1 0 0, 0 1 0 2.1, 0 1 0 4.2, 0 1 0 0 ]
        }
    ]
}
ROUTE Clicker.touchTime TO TimeSource.startTime
ROUTE TimeSource.fraction_changed TO Animation.set_fraction
ROUTE Animation.value_changed TO Xform.rotation
```

Click here to view this example in a VRML browser.

APPENDIX KK

### D.13.3 Alarm clock

The second example plays chimes once an hour:

```
#VRML V2.0 utf8

Group {
    children [
        DEF Hour TimeSensor {
            loop          TRUE
            cycleInterval 3600.0 # 60*60 seconds == 1 hour
        }

        Sound {
            source DEF Sounder AudioClip {
                url "click.wav" }
            }
        }
    ]
}

ROUTE Hour.cycleTime TO Sounder.startTime
```

Click here to view this example in a VRML browser.

## D.14 Shuttles and pendulums

Shuttles and pendulums are great building blocks for composing interesting animations. This shuttle translates its children back and forth along the X axis, from -1 to 1 (by default). The *distance* field can be used to change this default. The pendulum rotates its children about the Z axis, from 0 to 3.14159 radians and back again (by default). The *maxAngle* field can be used to change this default.

```
#VRML V2.0 utf8

PROTO Shuttle [
    field        SFTime  rate      1
    field        SFFloat distance  1
    field        MFNode  children  [ ]
    exposedField SFTime  startTime 0
    exposedField SFTime  stopTime  0
    field        SFBool  loop      TRUE
] {
    DEF F Transform { children IS children }
    DEF T TimeSensor {
        cycleInterval IS rate
        startTime IS startTime
        stopTime IS stopTime
        loop IS loop
    }

    DEF S Script {
        field    SFFloat    distance IS distance
```

APPENDIX KK

```
        eventOut MFVec3f    position

        url "javascript:
            function initialize() {
                // constructor:setup interpolator,
                pos1 = new SFVec3f(-distance, 0, 0);
                pos2 = new SFVec3f(distance, 0, 0);
                position = new MFVec3f(pos1, pos2, pos1);
            }",
    }

    DEF I PositionInterpolator {
        key [ 0, 0.5, 1 ]
        keyValue [ -1 0 0, 1 0 0, -1 0 0 ]
    }

    ROUTE T.fraction_changed TO I.set_fraction
    ROUTE I.value_changed TO F.set_translation
    ROUTE S.position TO I.set_keyValue
}

PROTO Pendulum [
    field        SFTime  rate      1
    field        SFFloat maxAngle  3.14159
    field        MFNode  children  [ ]
    exposedField SFTime  startTime 0
    exposedField SFTime  stopTime  0
    field        SFBool  loop      TRUE
] {
    DEF F Transform { children IS children }
    DEF T TimeSensor {
        cycleInterval IS rate
        startTime IS startTime
        stopTime IS stopTime
        loop IS loop
    }
    DEF S Script {
        field    SFFloat    maxAngle IS maxAngle
        eventOut MFRotation rotation

        url "javascript:
            function initialize() {
                // constructor:setup interpolator,
                rot1 = new SFRotation(0, 0, 1, 0);
                rot2 = new SFRotation(0, 0, 1, maxAngle/2);
                rot3 = new SFRotation(0, 0, 1, maxAngle);
                rotation = new MFRotation(rot1, rot2, rot3,
                                          rot2, rot1);
            }",
    }
    DEF I OrientationInterpolator {
        key [ 0, 0.25, 0.5, 0.75, 1 ]
        keyValue [ 0 0 1 0,
                   0 0 1 1.57,
                   0 0 1 3.14,
                   0 0 1 1.57,
                   0 0 1 0 ]
    }

    ROUTE T.fraction_changed TO I.set_fraction
    ROUTE I.value_changed TO F.set_rotation
    ROUTE S.rotation TO I.set_keyValue
}

Transform {
    translation -3 0 0
    children Pendulum {
        rate 3
        maxAngle 6.28
        children Shape { geometry Cylinder { height 5 } }
    }
}

Transform {
    translation 3 0 0
    children Shuttle {
        rate 2
        children Shape { geometry Sphere { } }
    }
}
```

Click here to view this example in a VRML browser.

These nodes can be used to do a continuous animation when *loop* is TRUE. When *loop* is FALSE they can perform a single cycle under control of the *startTime* and *stopTime* fields. The *rate* field controls the speed of the animation. The *children* field holds the children to be animated.

# D.15 Robot

This example is a simple implementation of a robot. This robot has very simple body parts: a cube for his head, a sphere for his body and cylinders for arms (he hovers so he has no feet!). He is something of a sentry--he walks forward and walks back across a path. He does this whenever the viewer is near. This makes use of the Shuttle and Pendulum of D.14.

```
#VRML V2.0 utf8

EXTERNPROTO Shuttle [
    field        SFTime  rate
    field        SFFloat distance
    field        MFNode  children
    exposedField SFTime  startTime
    exposedField SFTime  stopTime
    field        SFBool  loop
]
"exampleD.14.wrl#Shuttle"

EXTERNPROTO Pendulum [
    field        SFTime  rate
    field        SFFloat maxAngle
    field        MFNode  children
    exposedField SFTime  startTime
    exposedField SFTime  stopTime
    field        SFBool  loop
]
"exampleD.14.wrl#Pendulum"

Viewpoint {
    position 0 0 150
}

DEF Near ProximitySensor { size 200 200 200 }

DEF Walk Shuttle {
    stopTime 1
    rate 10
    distance 20

    children [
        # The Robot
        Transform {
            rotation 0 1 0 1.57

            children [
                Shape {
                    appearance DEF A Appearance {
                        material Material {
                            diffuseColor 0 0.5 0.7
                        }
                    }
                    geometry Box { } # head
                }
                Transform {
                    scale 1 5 1
                    translation 0 -5 0
                    children Shape {
                        appearance USE A
                        geometry Sphere { }
                    } # body
                }
                Transform {
                    rotation 0 1 0 1.57
                    translation 1.5 0 0

                    children DEF Arm Pendulum {
                        stopTime 1
                        rate 1
                        maxAngle 0.52 # 30 degrees

                        children [
                            Transform {
                                translation 0 -3 0

                                children Shape {
                                    appearance USE A
                                    geometry Cylinder {
                                        height 4
                                        radius 0.5
                                    }
                                }
                            }
                        ]
                    }
                }

                # duplicate arm on other side and flip so
                # it swings in opposition
                Transform {
                    rotation 0 -1 0 1.57
                    translation -1.5 0 0
                    children USE Arm
                }
            ]
        }
    ]
}

ROUTE Near.enterTime TO Walk.startTime
ROUTE Near.enterTime TO Arm.startTime
ROUTE Near.exitTime TO Walk.stopTime
ROUTE Near.exitTime TO Arm.stopTime
```

Click here to view this example in a VRML browser.

Move closer to the robot to start the animation.

## D.16 Chopper

This example of a helicopter demonstrates how to do simple animation triggered by a TouchSensor. It uses an EXTERNPROTO to include a Rotor node from the Internet which does the actual animation.

```
#VRML V2.0 utf8

EXTERNPROTO Rotor [
    field        SFTime  rate
    field        MFNode  children
    exposedField SFTime  startTime
    exposedField SFTime  stopTime
]
"rotor.wrl"

PROTO Chopper [
    field SFTime rotorSpeed 1
] {
    Group {
        children [
            DEF Touch TouchSensor { } # Gotta get touch events
            Inline { url "chopperbody.wrl" }
            DEF Top Rotor {
                # initially, the rotor should not spin
                stopTime 1
                rate IS rotorSpeed
                children Inline { url "chopperrotor.wrl" }
            }
        ]
    }

    DEF RotorScript Script {
        eventIn  SFTime startOrStopEngine
        eventOut SFTime startEngine
        eventOut SFTime stopEngine
        field    SFBool engineStarted FALSE

        url "javascript:
            function startOrStopEngine(value) {
                // start or stop engine:
                if (!engineStarted) {
                    startEngine = value;
                    engineStarted = TRUE;
                }
                else {
                    stopEngine = value;
                    engineStarted = FALSE;
                }
            }"
    }

    ROUTE Touch.touchTime TO RotorScript.startOrStopEngine
    ROUTE RotorScript.startEngine TO Top.startTime
    ROUTE RotorScript.stopEngine TO Top.stopTime
}

Viewpoint { position 0 0 5 }
DEF MyScene Group {
    children DEF MikesChopper Chopper { }
}
```

Click here to view this example in a VRML browser.

# D.17 Guided tour

VRML provides control of the viewer’s camera through use of a script. This is useful for things such as guided tours, merry-go-round rides, and transportation devices such as buses and elevators. These next two examples show a couple of ways to use this feature.

This example is a simple guided tour through the world. Upon entry, a guide orb hovers in front of the viewer. Click on this and a tour through the world begins. The orb follows the user around on his tour. A ProximitySensor ensures that the tour is started only if the user is close to the initial starting point. Note that this is done without scripts thanks to the *touchTime* output of the TouchSensor.

APPENDIX KK

```
#VRML V2.0 utf8

Group {
    children [
        Transform {
            translation 0 -1 0

            children Shape {
                appearance Appearance {
                    material Material { }
                }
                geometry Box { size 30 0.2 30 }
            }
        }
        Transform {
            translation -1 0 0

            children Shape {
                appearance Appearance {
                    material Material {
                        diffuseColor 0.5 0.8 0
                    }
                }
                geometry Cone { }
            }
        }
        Transform {
            translation 1 0 0

            children Shape {
                appearance Appearance {
                    material Material {
                        diffuseColor 0 0.2 0.7
                    }
                }
                geometry Cylinder { }
            }
        }

        DEF GuideTransform Transform {
            children [
                DEF TourGuide Viewpoint { jump FALSE },
                DEF ProxSensor ProximitySensor { size 50 50 50 }
                DEF StartTour TouchSensor { },
                Transform {
                    translation 0.6 0.4 8

                    children Shape {
                        appearance Appearance {
                            material Material {
                                diffuseColor 1 0.6 0
                            }
                        }
                        geometry Sphere { radius 0.2 }
                    } # the guide orb
                }
            ]
```

APPENDIX KK

```
        }
    ]
}

DEF GuidePI PositionInterpolator {
    key [ 0, 0.2, 0.3, 0.5, 0.6, 0.8, 0.9, 1 ]
    keyValue [ 0 0 0, 0 0 -5,
               2 0 -5, 2 6 -15
               -4 6 -15, -4 0 -5,
               0 0 -5, 0 0 0
    ]
}

DEF GuideRI OrientationInterpolator {
    key [ 0, 0.2, 0.3, 0.5, 0.6, 0.8, 0.9, 1 ]
    keyValue [ 0 1 0 0, 0 1 0 0,
               0 1 0 1.2, 0 1 0 3,
               0 1 0 3.5, 0 1 0 5,
               0 1 0 0, 0 1 0 0,
    ]
}

DEF TS TimeSensor { cycleInterval 30 } # 60 second tour

ROUTE ProxSensor.isActive TO StartTour.set_enabled
ROUTE StartTour.touchTime TO TS.startTime
ROUTE TS.isActive TO TourGuide.set_bind
ROUTE TS.fraction_changed TO GuidePI.set_fraction
ROUTE TS.fraction_changed TO GuideRI.set_fraction
ROUTE GuidePI.value_changed TO GuideTransform.set_translation
ROUTE GuideRI.value_changed TO GuideTransform.set_rotation
```

Click here to view this example in a VRML browser.

## D.18 Elevator

This is another example of animating the camera by depicting an elevator to ease access to a multi-storey building. For this example, a 2 storey building is shown and it is assumed that the elevator is already at the ground floor. To go up, the user just steps onto the elevator platform. A ProximitySensor fires and starts the elevator up automatically. Additional features such as call buttons for outside the elevator, elevator doors, and floor selector buttons could be added to make the elevator easier to use.

```
#VRML V2.0 utf8

Transform {
    translation 0 0 -3.5

    children Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0 1 0
            }
```

Microsoft et al. Exhibit 1005

```
        }
        geometry Cone { }
    }
}

Transform {
    translation 0 4 -3.5

    children Shape {
        appearance Appearance {
            material Material {
                diffuseColor 1 0 0
            }
        }
        geometry Cone { }
    }
}

Transform {
    translation 0 8 -3.5

    children Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0 0 1
            }
        }
        geometry Cone { }
    }
}

Group {
    children [
        DEF ETransform Transform {
            children [
                DEF EViewpoint Viewpoint { jump FALSE }
                DEF EProximity ProximitySensor { size 2 5 5 }
                Transform {
                    translation 0 -1 0

                    children Shape {
                        appearance Appearance {
                            material Material { }
                        }
                        geometry Box { size 2 0.2 5 }
                    }
                }
            ]
        }
    ]
}

DEF ElevatorPI PositionInterpolator {
    key [ 0, 1 ]
    keyValue [ 0 0 0, 0 8 0 ] # a floor is 4 meters high
}
DEF TS TimeSensor { cycleInterval 10 } # 10 second travel time
```

```
ROUTE EProximity.enterTime TO TS.startTime
ROUTE TS.isActive TO EViewpoint.set_bind
ROUTE TS.fraction_changed TO ElevatorPI.set_fraction
ROUTE ElevatorPI.value_changed TO ETransform.set_translation
```

Click here to view this example in a VRML browser.

## D.19

This example illustrates the execution model example described in 4.10.3, Execution model.

```
#VRML V2.0 utf8
DEF TS TouchSensor { }
DEF Script1 Script {
    eventIn  SFTime touchTime
    eventOut SFBool toScript2
    eventOut SFBool toScript3
    eventOut SFString string
    url "javascript:
        function touchTime() {
            toScript2 = TRUE;
        }
        function eventsProcessed() {
            string = ’Script1.eventsProcessed’;
            toScript3 = TRUE;
        }"
}
DEF Script2 Script {
   eventIn  SFBool fromScript1
   eventOut SFBool toScript4
    eventOut SFString string
    url "javascript:
        function fromScript1() {
        }
        function eventsProcessed() {
            string = ’Script2.eventsProcessed’;
            toScript4 = TRUE;
        }"
}
DEF Script3 Script {
   eventIn  SFBool fromScript1
   eventOut SFBool toScript5
   eventOut SFBool toScript6
   eventOut SFString string
    url "javascript:
        function fromScript1() {
            toScript5 = TRUE;
        }
        function eventsProcessed() {
            string = ’Script3.eventsProcessed’;
            toScript6 = TRUE;
        }"
}
```

```
DEF Script4 Script {
    eventIn SFBool fromScript2
    url "javascript:
        function fromScript2() {
        }"
}
DEF Script5 Script {
    eventIn SFBool fromScript3
    url "javascript:
        function fromScript3() {
        }"
}
DEF Script6 Script {
    eventIn  SFBool fromScript3
    eventOut SFBool toScript7
    eventOut SFString string
    url "javascript:
        function fromScript3() {
            toScript7 = TRUE;
        }
        function eventsProcessed() {
            string = ’Script6.eventsProcessed’;
        }"
}
DEF Script7 Script {
    eventIn  SFBool fromScript6
    url "javascript:
        function fromScript6() {
        }"
}
ROUTE TS.touchTime TO Script1.touchTime
ROUTE Script1.toScript2 TO Script2.fromScript1
ROUTE Script1.toScript3 TO Script3.fromScript1
ROUTE Script2.toScript4 TO Script4.fromScript2
ROUTE Script3.toScript5 TO Script5.fromScript3
ROUTE Script3.toScript6 TO Script6.fromScript3
ROUTE Script6.toScript7 TO Script7.fromScript6

# Display the results
DEF Collector Script {
    eventOut   MFString string
    eventIn SFString fromString
    url "javascript:
        function initialize() { string[0] = ’Event Sequence:’; }
        function fromString(s) {
            i = string.length;
            string[i] = ’    ’+i+’) ’+s+’ occurred’;
        }"
}
Transform {
    translation 0 2 0
    children Shape {
        appearance Appearance {
            material Material { diffuseColor 0 0.6 0 }
        }
        geometry Sphere { }
    }
```

```
}
Shape { geometry DEF Result Text { } }
Viewpoint { position 7 -1 18 }
ROUTE Script1.string TO Collector.fromString
ROUTE Script2.string TO Collector.fromString
ROUTE Script3.string TO Collector.fromString
ROUTE Script6.string TO Collector.fromString
ROUTE Collector.string TO Result.string
```

Click here to view this example in a VRML browser.

Clicking on the green sphere should display a text string for each eventsProcessed event. The two possible correct displays for this example are:

```
Event Sequence:
  1) Script1.eventsProcessed occurred
  2) Script2.eventsProcessed occurred
  3) Script3.eventsProcessed occurred
  4) Script6.eventsProcessed occurred
```

or

```
Event Sequence:
  1) Script2.eventsProcessed occurred
  2) Script1.eventsProcessed occurred
  3) Script3.eventsProcessed occurred
  4) Script6.eventsProcessed occurred
```

# Annex E
## (informative)
# Bibliography

This annex contains the informative references in this part of ISO/IEC 14772. These are references to unofficial standards or documents. All official standards are referenced in 2, Normative references.

| Identifier | Reference |
|---|---|
| **DATA** | "The Data: URL scheme," IETF Internet Draft working document.<br>http://ds.internic.net/internet-drafts/draft-masinter-url-data-03.txt |
| **FOLE** | Foley, van Dam, Feiner and Hughes, Computer Graphics Principles and Practice, 2nd Edition, Addison Wesley, Reading, MA, 1990.<br>http://www.awl.com |
| **GIF** | "GIF™ - Graphics Interchange Format™" - A standard defining a mechanism for the storage and transmission of raster-based graphics information, Version 89a, CompuServe.<br>http://www.w3.org/pub/WWW/Graphics/GIF/spec-gif89a.txt |
| **JAPI** | "The Java™ Application Programming Interface, Volume 1 Core Packages" by James Gosling, Frank Yellin and The Java Team, Addison Wesley, Reading Massachusetts, 1996, ISBN 0-201-63453-8.<br>http://java.sun.com/docs/books/apis/index.html<br>"The Java™ Application Programming Interface, Volume 2 Window Toolkit and Applets" by James Gosling, Frank Yellin and The Java Team, Addison Wesley, Reading Massachusetts, 1996, ISBN 0-201-63459-7.<br>http://java.sun.com/docs/books/apis/index.html |
| **MIME** | "The Model Primary Content Type for Multipurpose Internet Mail Extensions," IETF Internet standards track protocol.<br>http://ds.internic.net/rfc/rfc2077.txt |
| **OPEN** | "The OpenGL Graphics System: A Specification (Version 1.1)," Silicon Graphics, Inc., 1995.<br>http://www.sgi.com/Technology/openGL/glspec1.1/glspec.html |
| **PERL** | "Programming Perl" by Larry Wall, Tom Christiansen and Randal L. Schwartz, O'Reilly & Associates, Sebastapol, CA, 1996.<br>http://www.oreilly.com/ |
| **SNDA** | "Fundamentals of Computer Music", Dodge & Jerse, Shirmer Books, New York, 1985, pp 20-21. |
| **SNDB** | Spatial Audio Work in the Multimedia Computing Group, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA.<br>http://www.cc.gatech.edu/gvu/multimedia/spatsound/spatsound.html |

| | |
|---|---|
| **URN** | "Universal Resource Name," IETF Internet standards track protocol.<br>http://ds.internic.net/rfc/rfc2141.txt,<br>http://services.bunyip.com:8000/research/ietf/urn-ietf/ |
| **WAV** | "Waveform Audio File Format, Multimedia Programming Interface and Data Specification v1.0", Issued by IBM & Microsoft, 1991.<br>ftp://ftp.cwi.nl/pub/audio/RIFF-format,<br>http://keck.ucsf.edu/~jwright/RIFF-format.html,<br>http://www.seanet.com/HTML/Users/matts/riffmci/riffmci.htm |

# Annex F

**(informative)**

# Recommendations for non-normative extensions

## F.1 Introduction

This annex describes recommended practice for non-normative extensions to ISO/IEC 14772.

## F.2 URNs

URNs are location-independent pointers to a file or to different representations of the same content. In most ways, URNs can be used like URLs except that, when fetched, a smart browser should fetch them from the closest source. URN resolution over the Internet has not yet been standardized. However, URNs may be used now as persistent unique identifiers for referenced entities such as files, EXTERNPROTOs, and textures. General information on URNs is available at E.[URN].

URNs may be assigned by anyone with a domain name. For example, if the company Foo owns foo.com, it may allocate URNs that begin with "urn:inet:foo.com:". An example of such usage is

> "urn:inet:foo.com:texture:wood001".

See the draft specification referenced in E.[URN] for a description of the legal URN syntax.

To reference a texture, EXTERNPROTO, or other file by a URN, the URN is included in the *url* field of another node. For example:

```
    ImageTexture {
        url [ "http://www.foo.com/textures/woodblock_floor.gif",
              "urn:inet:foo.com:textures:wood001" ]
    }
```

specifies a URL file as the first choice and a URN as the second choice.

APPENDIX KK

# F.3 Browser extensions

Browsers that wish to add functionality beyond the capabilities of ISO/IEC 14772 can do so by creating prototypes or external prototypes. If the new node cannot be expressed using the prototyping mechanism (i.e., it cannot be expressed in the form of a VRML scene graph), it can be defined as an external prototype with a unique URN specification. Authors who use the extended functionality may provide multiple, alternative URLs or URNs to represent content to ensure it is viewable on all browsers.

For example, suppose a browser wants to create a native Torus geometry node implementation:

```
EXTERNPROTO Torus [ field SFFloat bigR, field SFFloat smallR ]
["urn:inet:browser.com:library:Torus",
 "http://.../proto_torus.wrl" ]
```

This browser will recognize the URN and use the URN resource's own private implementation of the Torus node. Other browsers may not recognize the URN, and skip to the next entry in the URL list and search for the specified prototype file. If no URLs are found, the Torus is assumed to be an empty node.

The prototype name "Torus" in the above example has no meaning whatsoever. The URN/URL uniquely and precisely defines the name/location of the node implementation. The prototype name is strictly a convention chosen by the author and shall not be interpreted in any semantic manner. The following example uses both "Ring" and "Donut" to name the torus node. However, the URN/URL pair "`urn:browser.com:library:Torus, http://.../proto_torus.wrl`" specifies the actual definitions of the Torus node:

```
#VRML V2.0 utf8
EXTERNPROTO Ring [field SFFloat bigR, field SFFloat smallR ]
  ["urn:browser.com:library:Torus", "http://.../proto_torus.wrl" ]
EXTERNPROTO Donut [field SFFloat bigR, field SFFloat smallR ]
  ["urn:browser.com:library:Torus", "http://.../proto_torus.wrl" ]

Transform { ... children Shape { geometry Ring { } } }
Transform { ... children Shape { geometry Donut { } } }
```