

Figure 8-9 *SKIP_ROWS, *SKIP_PIXELS, and *ROW_LENGTH Parameters

Often a particular machine's hardware is optimized for moving pixel data to and from memory, if the data is saved in memory with a particular byte alignment. For example, in a machine with 32-bit words, hardware can often retrieve data much faster if it's initially aligned on a 32-bit boundary, which typically has an address that is a multiple of 4. Likewise, 64-bit architectures might work better when the data is aligned to 8-byte boundaries. On some machines, however, byte alignment makes no difference.

As an example, suppose your machine works better with pixel data aligned to a 4-byte boundary. Images are most efficiently saved by forcing the data for each row of the image to begin on a 4-byte boundary. If the image is 5 pixels wide and each pixel consists of 1 byte each of red, green, and blue information, a row requires $5 \times 3 = 15$ bytes of data. Maximum display efficiency can be achieved if the first row, and each successive row, begins on a 4-byte boundary, so there is 1 byte of waste in the memory storage for each row. If your data is stored like this, set the *ALIGNMENT parameter appropriately (to 4, in this case).

If *ALIGNMENT is set to 1, the next available byte is used. If it's 2, a byte is skipped if necessary at the end of each row so that the first byte of the next row has an address that's a multiple of 2. In the case of bitmaps (or 1-bit images) where a single bit is saved for each pixel, the same byte alignment works, although you have to count individual bits. For example, if you're saving a single bit per pixel, the row length is 75, and the alignment is 4, then each row requires $75/8$, or $9\ 3/8$ bytes. Since 12 is the smallest multiple of 4 that is bigger than $9\ 3/8$, 12 bytes of memory are used for each row. If the alignment is 1, then 10 bytes are used for each row, as $9\ 3/8$ is rounded

up to the next byte. (There is a simple use of `glPixelStorei()` in Example 8-4 on page 306.)

Pixel-Transfer Operations

As image data is transferred from memory into the framebuffer, or from the framebuffer into memory, OpenGL can perform several operations on it. For example, the ranges of components can be altered—normally, the red component is between 0.0 and 1.0, but you might prefer to keep it in some other range; or perhaps the data you’re using from a different graphics system stores the red component in a different range. You can even create maps to perform arbitrary conversion of color indices or color components during pixel transfer. Conversions such as these performed during the transfer of pixels to and from the framebuffer are called pixel-transfer operations. They’re controlled with the `glPixelTransfer*()` and `glPixelMap*()` commands.

Be aware that although the color, depth, and stencil buffers have many similarities, they don’t behave identically, and a few of the modes have special cases for special buffers. All the mode details are covered in this section and the sections that follow, including all the special cases.

Some of the pixel-transfer function characteristics are set with `glPixelTransfer*()`. The other characteristics are specified with `glPixelMap*()`, which is described in the next section.

```
void glPixelTransfer{if}(GLenum pname, TYPE param);
```

Sets pixel-transfer modes that affect the operation of `glDrawPixels()`, `glReadPixels()`, `glCopyPixels()`, `glTexImage1D()`, `glTexImage2D()`, `glCopyTexImage1D()`, `glCopyTexImage2D()`, `glTexSubImage1D()`, `glTexSubImage2D()`, `glCopyTexSubImage1D()`, `glCopyTexSubImage2D()`, and `glGetTexImage()`. The parameter *pname* must be one of those listed in the first column of Table 8-4, and its value, *param*, must be in the valid range shown.

Parameter Name	Type	Initial Value	Valid Range
GL_MAP_COLOR	GLboolean	FALSE	TRUE/FALSE
GL_MAP_STENCIL	GLboolean	FALSE	TRUE/FALSE

Table 8-4 `glPixelTransfer*()` Parameters

Parameter Name	Type	Initial Value	Valid Range
GL_INDEX_SHIFT	GLint	0	(-∞, ∞)
GL_INDEX_OFFSET	GLint	0	(-∞, ∞)
GL_RED_SCALE	GLfloat	1.0	(-∞, ∞)
GL_GREEN_SCALE	GLfloat	1.0	(-∞, ∞)
GL_BLUE_SCALE	GLfloat	1.0	(-∞, ∞)
GL_ALPHA_SCALE	GLfloat	1.0	(-∞, ∞)
GL_DEPTH_SCALE	GLfloat	1.0	(-∞, ∞)
GL_RED_BIAS	GLfloat	0	(-∞, ∞)
GL_GREEN_BIAS	GLfloat	0	(-∞, ∞)
GL_BLUE_BIAS	GLfloat	0	(-∞, ∞)
GL_ALPHA_BIAS	GLfloat	0	(-∞, ∞)
GL_DEPTH_BIAS	GLfloat	0	(-∞, ∞)

Table 8-4 glPixelTransfer*() Parameters (continued)

If the `GL_MAP_COLOR` or `GL_MAP_STENCIL` parameter is `TRUE`, then mapping is enabled. See the next subsection to learn how the mapping is done and how to change the contents of the maps. All the other parameters directly affect the pixel component values.

A scale and bias can be applied to the red, green, blue, alpha, and depth components. For example, you may wish to scale red, green, and blue components that were read from the framebuffer before converting them to a luminance format in processor memory. Luminance is computed as the sum of the red, green, and blue components, so if you use the default value for `GL_RED_SCALE`, `GL_GREEN_SCALE` and `GL_BLUE_SCALE`, the components all contribute equally to the final intensity or luminance value. If you want to convert RGB to luminance, according to the NTSC standard, you set `GL_RED_SCALE` to `.30`, `GL_GREEN_SCALE` to `.59`, and `GL_BLUE_SCALE` to `.11`.

Indices (color and stencil) can also be transformed. In the case of indices a shift and offset are applied. This is useful if you need to control which portion of the color table is used during rendering.

Pixel Mapping

All the color components, color indices, and stencil indices can be modified by means of a table lookup before they are placed in screen memory. The command for controlling this mapping is `glPixelMap*()`.

```
void glPixelMap{ui us fv}(GLenum map, GLint mapsize,  
                        const TYPE *values);
```

Loads the pixel map indicated by *map* with *mapsize* entries, whose values are pointed to by *values*. Table 8-5 lists the map names and values; the default sizes are all 1 and the default values are all 0. Each map's size must be a power of 2.

Map Name	Address	Value
GL_PIXEL_MAP_I_TO_I	color index	color index
GL_PIXEL_MAP_S_TO_S	stencil index	stencil index
GL_PIXEL_MAP_I_TO_R	color index	R
GL_PIXEL_MAP_I_TO_G	color index	G
GL_PIXEL_MAP_I_TO_B	color index	B
GL_PIXEL_MAP_I_TO_A	color index	A
GL_PIXEL_MAP_R_TO_R	R	R
GL_PIXEL_MAP_G_TO_G	G	G
GL_PIXEL_MAP_B_TO_B	B	B
GL_PIXEL_MAP_A_TO_A	A	A

Table 8-5 `glPixelMap*()` Parameter Names and Values

The maximum size of the maps is machine-dependent. You can find the sizes of the pixel maps supported on your machine with `glGetIntegerv()`. Use the query argument `GL_MAX_PIXEL_MAP_TABLE` to obtain the maximum size for all the pixel map tables, and use `GL_PIXEL_MAP_*_TO_*_SIZE` to obtain the current size of the specified map. The six maps whose address is a color index or stencil index must always be sized to an integral power of 2. The four RGBA maps can be any size from 1 through `GL_MAX_PIXEL_MAP_TABLE`.

To understand how a table works, consider a simple example. Suppose that you want to create a 256-entry table that maps color indices to color indices using `GL_PIXEL_MAP_I_TO_I`. You create a table with an entry for each of the values between 0 and 255 and initialize the table with `glPixelMap*()`. Assume you're using the table for thresholding and want to map indices below 101 (indices 0 to 100) to 0, and all indices 101 and above to 255. In this case, your table consists of 101 0s and 155 255s. The pixel map is enabled using the routine `glPixelTransfer*()` to set the parameter `GL_MAP_COLOR` to `TRUE`. Once the pixel map is loaded and enabled, incoming color indices below 101 come out as 0, and incoming pixels between 101 and 255 are mapped to 255. If the incoming pixel is larger than 255, it's first masked by 255, throwing out all the bits above the eighth, and the resulting masked value is looked up in the table. If the incoming index is a floating-point value (say 88.14585), it's rounded to the nearest integer value (giving 88), and that number is looked up in the table (giving 0).

Using pixel maps, you can also map stencil indices or convert color indices to RGB. (See "Reading and Drawing Pixel Rectangles" on page 309 for information about the conversion of indices.)

Magnifying, Reducing, or Flipping an Image

After the pixel-storage modes and pixel-transfer operations are applied, images and bitmaps are rasterized. Normally, each pixel in an image is written to a single pixel on the screen. However, you can arbitrarily magnify, reduce, or even flip (reflect) an image by using `glPixelZoom()`.

```
void glPixelZoom(GLfloat zoomx, GLfloat zoomy);
```

Sets the magnification or reduction factors for pixel-write operations (`glDrawPixels()` or `glCopyPixels()`), in the x - and y -dimensions. By default, $zoom_x$ and $zoom_y$ are 1.0. If they're both 2.0, each image pixel is drawn to 4 screen pixels. Note that fractional magnification or reduction factors are allowed, as are negative factors. Negative zoom factors reflect the resulting image about the current raster position.

During rasterization, each image pixel is treated as a $zoom_x \times zoom_y$ rectangle, and fragments are generated for all the pixels whose centers lie within the rectangle. More specifically, let (x_{rp}, y_{rp}) be the current raster position. If a particular group of elements (index or components) is the n th in a row and belongs to the m th column, consider the region in window coordinates bounded by the rectangle with corners at

$(x_{rp} + zoom_x * n, y_{rp} + zoom_y * m)$ and $(x_{rp} + zoom_x(n+1), y_{rp} + zoom_y(m+1))$

Any fragments whose centers lie inside this rectangle (or on its bottom or left boundaries) are produced in correspondence with this particular group of elements.

A negative zoom can be useful for flipping an image. OpenGL describes images from the bottom row of pixels to the top (and from left to right). If you have a “top to bottom” image, such as a frame of video, you may want to use `glPixelZoom(1.0, -1.0)` to make the image right side up for OpenGL. Be sure that you reposition the current raster position appropriately, if needed.

Example 8-4 shows the use of `glPixelZoom()`. A checkerboard image is initially drawn in the lower-left corner of the window. Pressing a mouse button and moving the mouse uses `glCopyPixels()` to copy the lower-left corner of the window to the current cursor location. (If you copy the image onto itself, it looks wacky!) The copied image is zoomed, but initially it is zoomed by the default value of 1.0, so you won’t notice. The ‘z’ and ‘Z’ keys increase and decrease the zoom factors by 0.5. Any window damage causes the contents of the window to be redrawn. Pressing the ‘r’ key resets the image and the zoom factors.

Example 8-4 Drawing, Copying, and Zooming Pixel Data: `image.c`

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>

#define checkImageWidth 64
#define checkImageHeight 64
GLubyte checkImage[checkImageHeight][checkImageWidth][3];

static GLdouble zoomFactor = 1.0;
static GLint height;

void makeCheckImage(void)
{
    int i, j, c;

    for (i = 0; i < checkImageHeight; i++) {
        for (j = 0; j < checkImageWidth; j++) {
            c = (((i&0x8)==0)^(j&0x8)==0) * 255;
            checkImage[i][j][0] = (GLubyte) c;
        }
    }
}
```

```

        checkImage[i][j][1] = (GLubyte) c;
        checkImage[i][j][2] = (GLubyte) c;
    }
}

void init(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_FLAT);
    makeCheckImage();
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glRasterPos2i(0, 0);
    glDrawPixels(checkImageWidth, checkImageHeight, GL_RGB,
                GL_UNSIGNED_BYTE, checkImage);
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    height = (GLint) h;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble) w, 0.0, (GLdouble) h);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void motion(int x, int y)
{
    static GLint screeny;

    screeny = height - (GLint) y;
    glRasterPos2i (x, screeny);
    glPixelZoom (zoomFactor, zoomFactor);
    glCopyPixels (0, 0, checkImageWidth, checkImageHeight,
                GL_COLOR);
    glPixelZoom (1.0, 1.0);
    glFlush ();
}

```

```

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 'r':
        case 'R':
            zoomFactor = 1.0;
            glutPostRedisplay();
            printf ("zoomFactor reset to 1.0\n");
            break;
        case 'z':
            zoomFactor += 0.5;
            if (zoomFactor >= 3.0)
                zoomFactor = 3.0;
            printf ("zoomFactor is now %4.1f\n", zoomFactor);
            break;
        case 'Z':
            zoomFactor -= 0.5;
            if (zoomFactor <= 0.5)
                zoomFactor = 0.5;
            printf ("zoomFactor is now %4.1f\n", zoomFactor);
            break;
        case 27:
            exit(0);
            break;
        default:
            break;
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(250, 250);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMotionFunc(motion);
    glutMainLoop();
    return 0;
}

```

Reading and Drawing Pixel Rectangles

This section describes the reading and drawing processes in detail. The pixel conversions performed when going from framebuffer to memory (reading) are similar but not identical to the conversions performed when going in the opposite direction (drawing), as explained in the following sections. You may wish to skip this section the first time through, especially if you do not plan to use the pixel-transfer operations right away.

The Pixel Rectangle Drawing Process

Figure 8-10 and the following list describe the operation of drawing pixels into the framebuffer.

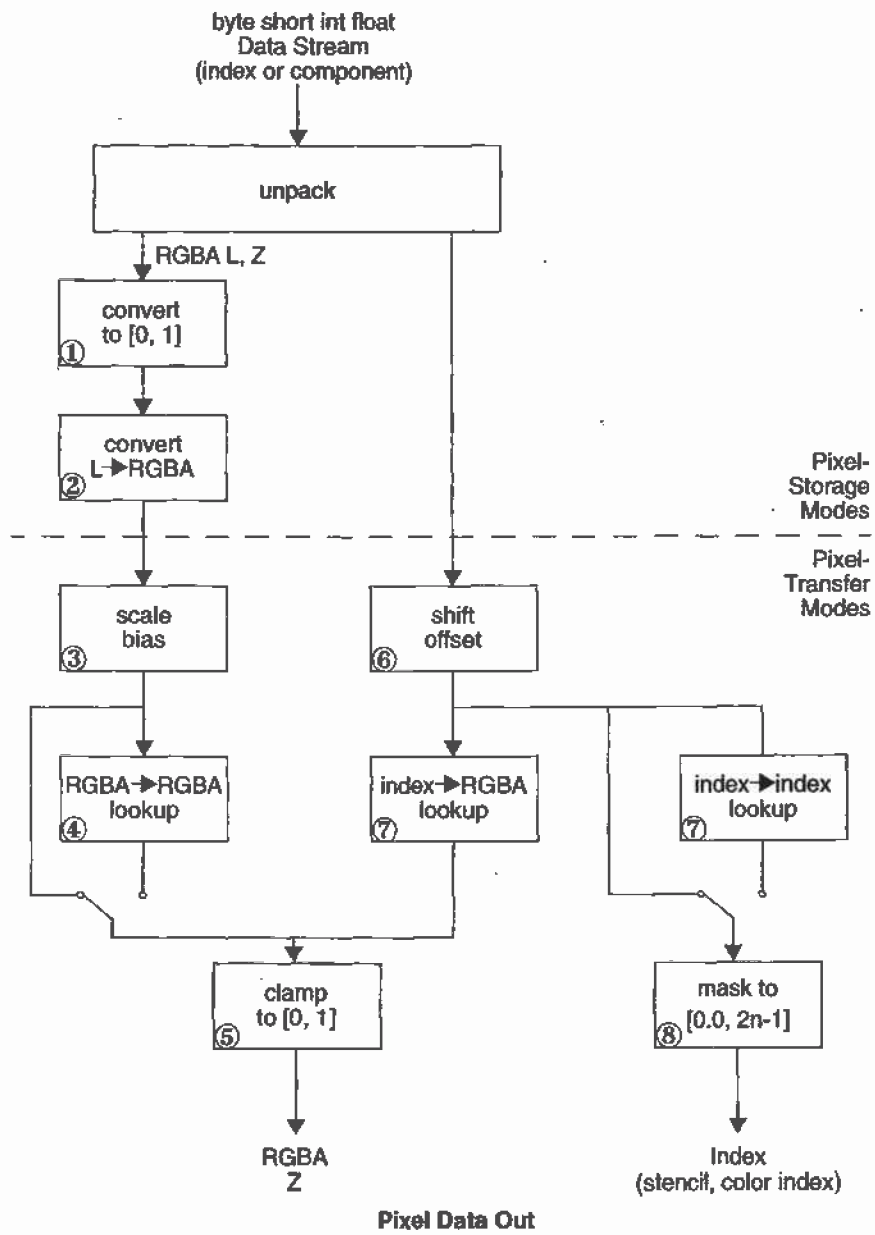


Figure 8-10 Drawing Pixels with `glDrawPixels()`

-
1. If the pixels aren't indices (that is, the format isn't `GL_COLOR_INDEX` or `GL_STENCIL_INDEX`), the first step is to convert the components to floating-point format if necessary. (See Table 4-1 on page 164 for the details of the conversion.)
 2. If the format is `GL_LUMINANCE` or `GL_LUMINANCE_ALPHA`, the luminance element is converted into R, G, and B, by using the luminance value for each of the R, G, and B components. In `GL_LUMINANCE_ALPHA` format, the alpha value becomes the A value. If `GL_LUMINANCE` is specified, the A value is set to 1.0.
 3. Each component (R, G, B, A, or depth) is multiplied by the appropriate scale, and the appropriate bias is added. For example, the R component is multiplied by the value corresponding to `GL_RED_SCALE` and added to the value corresponding to `GL_RED_BIAS`.
 4. If `GL_MAP_COLOR` is true, each of the R, G, B, and A components is clamped to the range [0.0,1.0], multiplied by an integer one less than the table size, truncated, and looked up in the table. (See "Tips for Improving Pixel Drawing Rates" on page 314 for more details.)
 5. Next, the R, G, B, and A components are clamped to [0.0,1.0], if they weren't already, and converted to fixed-point with as many bits to the left of the binary point as there are in the corresponding framebuffer component.
 6. If you're working with index values (stencil or color indices), then the values are first converted to fixed-point (if they were initially floating-point numbers) with some unspecified bits to the right of the binary point. Indices that were initially fixed-point remain so, and any bits to the right of the binary point are set to zero.

The resulting index value is then shifted right or left by the absolute value of `GL_INDEX_SHIFT` bits; the value is shifted left if `GL_INDEX_SHIFT > 0` and right otherwise. Finally, `GL_INDEX_OFFSET` is added to the index.

7. The next step with indices depends on whether you're using RGBA mode or color-index mode. In RGBA mode, a color index is converted to RGBA using the color components specified by `GL_PIXEL_MAP_I_TO_R`, `GL_PIXEL_MAP_I_TO_G`, `GL_PIXEL_MAP_I_TO_B`, and `GL_PIXEL_MAP_I_TO_A`. (See "Pixel Mapping" on page 304 for details.) Otherwise, if `GL_MAP_COLOR` is `GL_TRUE`, a color index is looked up through the table `GL_PIXEL_MAP_I_TO_I`. (If `GL_MAP_COLOR` is `GL_FALSE`, the index is unchanged.) If the image is made up of stencil indices rather than

color indices, and if `GL_MAP_STENCIL` is `GL_TRUE`, the index is looked up in the table corresponding to `GL_PIXEL_MAP_S_TO_S`. If `GL_MAP_STENCIL` is `FALSE`, the stencil index is unchanged.

8. Finally, if the indices haven't been converted to RGBA, the indices are then masked to the number of bits of either the color-index or stencil buffer, whichever is appropriate.

The Pixel Rectangle Reading Process

Many of the conversions done during the pixel rectangle drawing process are also done during the pixel rectangle reading process. The pixel reading process is shown in Figure 8-11 and described in the following list.

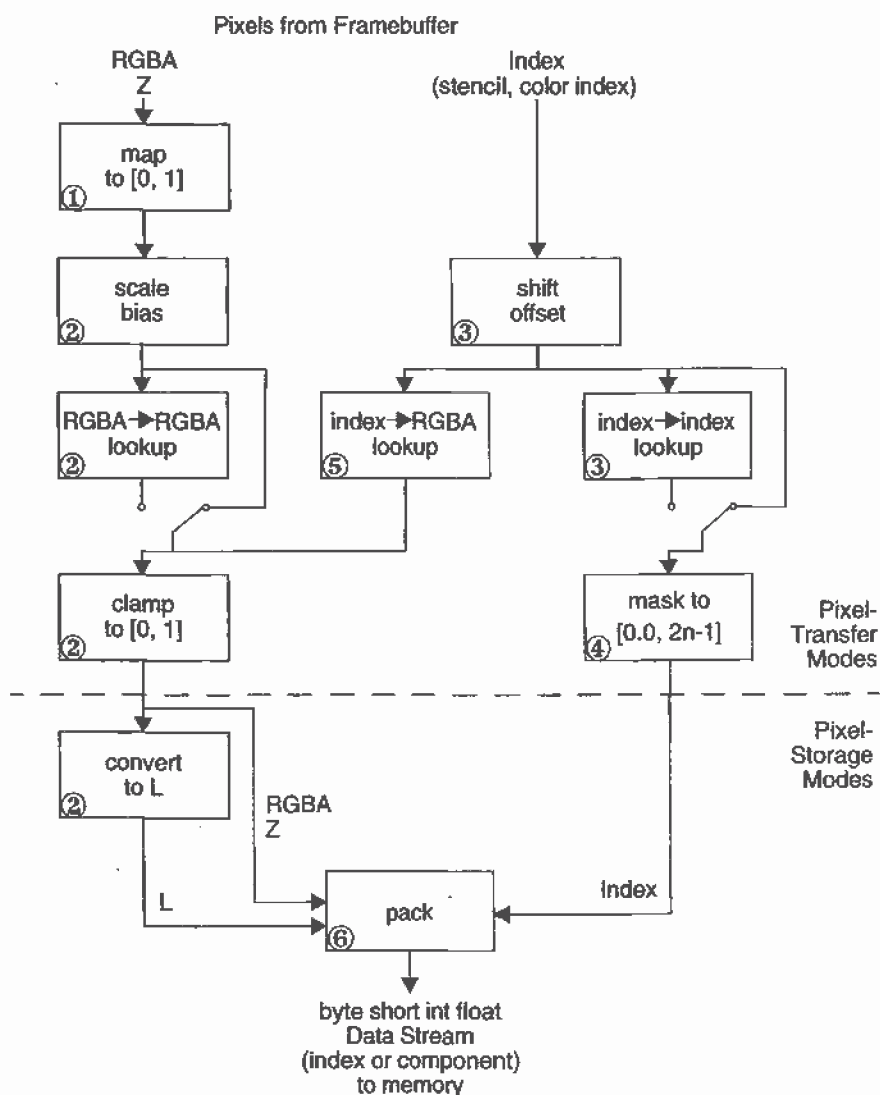


Figure 8-11 Reading Pixels with `glReadPixels()`

1. If the pixels to be read aren't indices (that is, the format isn't `GL_COLOR_INDEX` or `GL_STENCIL_INDEX`), the components are mapped to `[0.0,1.0]`—that is, in exactly the opposite way that they are when written.
2. Next, the scales and biases are applied to each component. If `GL_MAP_COLOR` is `GL_TRUE`, they're mapped and again clamped to `[0.0,1.0]`. If luminance is desired instead of RGB, the R, G, and B components are added ($L = R + G + B$).

3. If the pixels are indices (color or stencil), they're shifted, offset, and, if `GL_MAP_COLOR` is `GL_TRUE`, also mapped.
4. If the storage format is either `GL_COLOR_INDEX` or `GL_STENCIL_INDEX`, the pixel indices are masked to the number of bits of the storage type (1, 8, 16, or 32) and packed into memory as previously described.
5. If the storage format is one of the component kind (such as luminance or RGB), the pixels are always mapped by the index-to-RGBA maps. Then, they're treated as though they had been RGBA pixels in the first place (including potential conversion to luminance).
6. Finally, for both index and component data, the results are packed into memory according to the `GL_PACK*` modes set with `glPixelStore*()`.

The scaling, bias, shift, and offset values are the same as those used when drawing pixels, so if you're both reading and drawing pixels, be sure to reset these components to the appropriate values before doing a read or a draw. Similarly, the various maps must be properly reset if you intend to use maps for both reading and drawing.

Note: It might seem that luminance is handled incorrectly in both the reading and drawing operations. For example, luminance is not usually equally dependent on the R, G, and B components as it may be assumed from both Figure 8-10 and Figure 8-11. If you wanted your luminance to be calculated such that the R component contributed 30 percent, the G 59 percent, and the B 11 percent, you can set `GL_RED_SCALE` to `.30`, `GL_RED_BIAS` to `0.0`, and so on. The computed L is then $.30R + .59G + .11B$.

Tips for Improving Pixel Drawing Rates

As you can see, OpenGL has a rich set of features for reading, drawing and manipulating pixel data. Although these features are often very useful, they can also decrease performance. Here are some tips for improving pixel draw rates.

- For best performance, set all pixel-transfer parameters to their default values, and set pixel zoom to `(1.0,1.0)`.

-
- A series of fragment operations is applied to pixels as they are drawn into the framebuffer. (See “Testing and Operating on Fragments” on page 382.) For optimum performance disable all fragment operations.
 - While performing pixel operations, disable other costly states, such as texturing and lighting.
 - If you use an image format and type that matches the framebuffer, you can reduce the amount of work that the OpenGL implementation has to do. For example, if you are writing images to an RGB framebuffer with 8 bits per component, call `glDrawPixels()` with *format* set to RGB and *type* set to UNSIGNED_BYTE.
 - For some implementations, unsigned image formats are faster to use than signed image formats.
 - It is usually faster to draw a large pixel rectangle than to draw several small ones, since the cost of transferring the pixel data can be amortized over many pixels.
 - If possible, reduce the amount of data that needs to be copied by using small data types (for example, use GL_UNSIGNED_BYTE) and fewer components (for example, use format GL_LUMINANCE_ALPHA).
 - Pixel-transfer operations, including pixel mapping and values for scale, bias, offset, and shift other than the defaults, may decrease performance.

Texture Mapping



Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Understand what texture mapping can add to your scene
- Specify a texture image
- Control how a texture image is filtered as it's applied to a fragment
- Create and manage texture images in texture objects and, if available, control a high-performance working set of those texture objects
- Specify how the color values in the image combine with those of the fragment to which it's being applied
- Supply texture coordinates to indicate how the texture image should be aligned to the objects in your scene
- Use automatic texture coordinate generation to produce effects like contour maps and environment maps

So far, every geometric primitive has been drawn as either a solid color or smoothly shaded between the colors at its vertices—that is, they've been drawn without texture mapping. If you want to draw a large brick wall without texture mapping, for example, each brick must be drawn as a separate polygon. Without texturing, a large flat wall—which is really a single rectangle—might require thousands of individual bricks, and even then the bricks may appear too smooth and regular to be realistic.

Texture mapping allows you to glue an image of a brick wall (obtained, perhaps, by scanning in a photograph of a real wall) to a polygon and to draw the entire wall as a single polygon. Texture mapping ensures that all the right things happen as the polygon is transformed and rendered. For example, when the wall is viewed in perspective, the bricks may appear smaller as the wall gets farther from the viewpoint. Other uses for texture mapping include depicting vegetation on large polygons representing the ground in flight simulation; wallpaper patterns; and textures that make polygons look like natural substances such as marble, wood, or cloth. The possibilities are endless. Although it's most natural to think of applying textures to polygons, textures can be applied to all primitives—points, lines, polygons, bitmaps, and images. Plates 6, 8, 18–21, 24–27, and 29–31 all demonstrate the use of textures.

Because there are so many possibilities, texture mapping is a fairly large, complex subject, and you must make several programming choices when using it. For instance, you can map textures to surfaces made of a set of polygons or to curved surfaces, and you can repeat a texture in one or both directions to cover the surface. A texture can even be one-dimensional. In addition, you can automatically map a texture onto an object in such a way that the texture indicates contours or other properties of the item being viewed. Shiny objects can be textured so that they appear to be in the center of a room or other environment, reflecting the surroundings off their surfaces. Finally, a texture can be applied to a surface in different ways. It can be painted on directly (like a decal placed on a surface), used to modulate the color the surface would have been painted otherwise, or used to blend a texture color with the surface color. If this is your first exposure to texture mapping, you might find that the discussion in this chapter moves fairly quickly. As an additional reference, you might look at the chapter on texture mapping in *Fundamentals of Three-Dimensional Computer Graphics* by Alan Watt (Reading, MA: Addison-Wesley Publishing Company, 1990).

Textures are simply rectangular arrays of data—for example, color data, luminance data, or color and alpha data. The individual values in a texture array are often called *texels*. What makes texture mapping tricky is that a

rectangular texture can be mapped to nonrectangular regions, and this must be done in a reasonable way.

Figure 9-1 illustrates the texture-mapping process. The left side of the figure represents the entire texture, and the black outline represents a quadrilateral shape whose corners are mapped to those spots on the texture. When the quadrilateral is displayed on the screen, it might be distorted by applying various transformations—rotations, translations, scaling, and projections. The right side of the figure shows how the texture-mapped quadrilateral might appear on your screen after these transformations. (Note that this quadrilateral is concave and might not be rendered correctly by OpenGL without prior tessellation. See Chapter 11 for more information about tessellating polygons.)

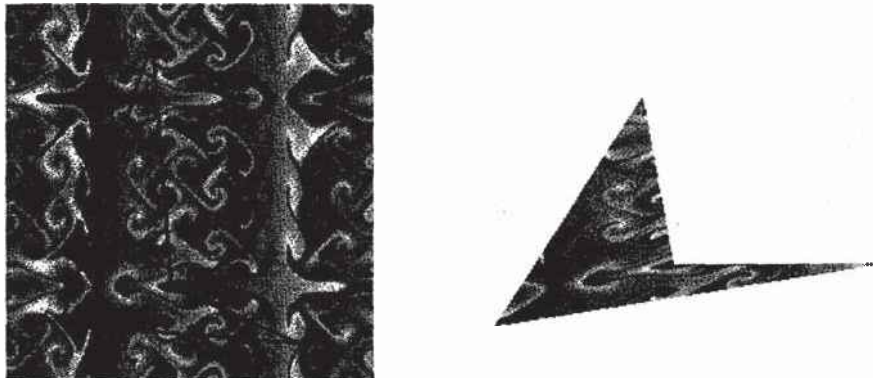


Figure 9-1 Texture-Mapping Process

Notice how the texture is distorted to match the distortion of the quadrilateral. In this case, it's stretched in the x direction and compressed in the y direction; there's a bit of rotation and shearing going on as well. Depending on the texture size, the quadrilateral's distortion, and the size of the screen image, some of the texels might be mapped to more than one fragment, and some fragments might be covered by multiple texels. Since the texture is made up of discrete texels (in this case, 256×256 of them), filtering operations must be performed to map texels to fragments. For example, if many texels correspond to a fragment, they're averaged down to fit; if texel boundaries fall across fragment boundaries, a weighted average of the applicable texels is performed. Because of these calculations, texturing is computationally expensive, which is why many specialized graphics systems include hardware support for texture mapping.

An application may establish texture objects, with each texture object representing a single texture (and possible associated mipmaps). Some

implementations of OpenGL can support a special *working set* of texture objects that have better performance than texture objects outside the working set. These high-performance texture objects are said to be *resident* and may have special hardware and/or software acceleration available. You may use OpenGL to create and delete texture objects and to determine which textures constitute your working set.

This chapter covers the OpenGL's texture-mapping facility in the following major sections.

- “An Overview and an Example” on page 321 gives a brief, broad look at the steps required to perform texture mapping. It also presents a relatively simple example of texture mapping.
- “Specifying the Texture” on page 326 explains how to specify one- or two-dimensional textures. It also discusses how to use a texture's borders, how to supply a series of related textures of different sizes, and how to control the filtering methods used to determine how an applied texture is mapped to screen coordinates.
- “Filtering” on page 344 details how textures are either magnified or minified as they are applied to the pixels of polygons. Minification using special mipmap textures is also explained.
- “Texture Objects” on page 346 describes how to put texture images into objects so that you can control several textures at one time. With texture objects, you may be able to create a working set of high-performance textures, which are said to be resident. You may also prioritize texture objects to increase or decrease the likelihood that a texture object is resident.
- “Texture Functions” on page 354 discusses the methods used for painting a texture onto a surface. You can choose to have the texture color values replace those that would be used if texturing wasn't in effect, or you can have the final color be a combination of the two.
- “Assigning Texture Coordinates” on page 357 describes how to compute and assign appropriate texture coordinates to the vertices of an object. It also explains how to control the behavior of coordinates that lie outside the default range—that is, how to repeat or clamp textures across a surface.
- “Automatic Texture-Coordinate Generation” on page 364 shows how to have OpenGL automatically generate texture coordinates so that you can achieve such effects as contour and environment maps.

-
- “Advanced Features” on page 371 explains how to manipulate the texture matrix stack and how to use the q texture coordinate.

Version 1.1 of OpenGL introduces several new texture-mapping operations:

- Thirty-eight additional internal texture image formats
- Texture proxy, to query whether there are enough resources to accommodate a given texture image
- Texture subimage, to replace all or part of an existing texture image rather than completely deleting and creating a texture to achieve the same effect
- Specifying texture data from framebuffer memory (as well as from processor memory)
- Texture objects, including resident textures and prioritizing

If you try to use one of these texture-mapping operations and can't find it, check the version number of your implementation of OpenGL to see if it actually supports it. (See “Which Version Am I Using?” on page 503.)

An Overview and an Example

This section gives an overview of the steps necessary to perform texture mapping. It also presents a relatively simple texture-mapping program. Of course, you know that texture mapping can be a very involved process.

Steps in Texture Mapping

To use texture mapping, you perform these steps.

1. Create a texture object and specify a texture for that object.
2. Indicate how the texture is to be applied to each pixel.
3. Enable texture mapping.
4. Draw the scene, supplying both texture and geometric coordinates.

Keep in mind that texture mapping works only in RGBA mode. Texture mapping results in color-index mode are undefined.

Create a Texture Object and Specify a Texture for That Object

A texture is usually thought of as being two-dimensional, like most images, but it can also be one-dimensional. The data describing a texture may consist of one, two, three, or four elements per texel, representing anything from a modulation constant to an (R, G, B, A) quadruple.

In Example 9-1, which is very simple, a single texture object is created to maintain a single two-dimensional texture. This example does not find out how much memory is available. Since only one texture is created, there is no attempt to prioritize or otherwise manage a working set of texture objects. Other advanced techniques, such as texture borders or mipmaps, are not used in this simple example.

Indicate How the Texture Is to Be Applied to Each Pixel

You can choose any of four possible functions for computing the final RGBA value from the fragment color and the texture-image data. One possibility is simply to use the texture color as the final color; this is the *decal* mode, in which the texture is painted on top of the fragment, just as a decal would be applied. (Example 9-1 uses decal mode.) The *replace* mode, a variant of the decal mode, is a second method. Another method is to use the texture to *modulate*, or scale, the fragment's color; this technique is useful for combining the effects of lighting with texturing. Finally, a constant color can be blended with that of the fragment, based on the texture value.

Enable Texture Mapping

You need to enable texturing before drawing your scene. Texturing is enabled or disabled using `glEnable()` or `glDisable()` with the symbolic constant `GL_TEXTURE_1D` or `GL_TEXTURE_2D` for one- or two-dimensional texturing, respectively. (If both are enabled, `GL_TEXTURE_2D` is the one that is used.)

Draw the Scene, Supplying Both Texture and Geometric Coordinates

You need to indicate how the texture should be aligned relative to the fragments to which it's to be applied before it's "glued on." That is, you need to specify both texture coordinates and geometric coordinates as you specify the objects in your scene. For a two-dimensional texture map, for example, the texture coordinates range from 0.0 to 1.0 in both directions, but the coordinates of the items being textured can be anything. For the brick-wall example, if the wall is square and meant to represent one copy of

the texture, the code would probably assign texture coordinates (0, 0), (1, 0), (1, 1), and (0, 1) to the four corners of the wall. If the wall is large, you might want to paint several copies of the texture map on it. If you do so, the texture map must be designed so that the bricks on the left edge match up nicely with the bricks on the right edge, and similarly for the bricks on the top and those on the bottom.

You must also indicate how texture coordinates outside the range [0.0,1.0] should be treated. Do the textures repeat to cover the object, or are they clamped to a boundary value?

A Sample Program

One of the problems with showing sample programs to illustrate texture mapping is that interesting textures are large. Typically, textures are read from an image file, since specifying a texture programmatically could take hundreds of lines of code. In Example 9-1, the texture—which consists of alternating white and black squares, like a checkerboard—is generated by the program. The program applies this texture to two squares, which are then rendered in perspective, one of them facing the viewer squarely and the other tilting back at 45 degrees, as shown in Figure 9-2. In object coordinates, both squares are the same size.

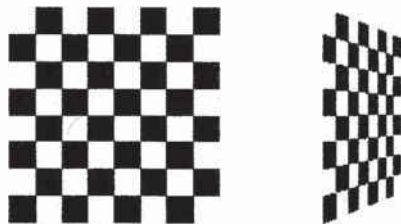


Figure 9-2 Texture-Mapped Squares

Example 9-1 Texture-Mapped Checkerboard: checker.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>

/* Create checkerboard texture */
#define checkImageWidth 64
#define checkImageHeight 64
```

```

static GLubyte checkImage[checkImageHeight][checkImageWidth][4];

static GLuint texName;

void makeCheckImage(void)
{
    int i, j, c;

    for (i = 0; i < checkImageHeight; i++) {
        for (j = 0; j < checkImageWidth; j++) {
            c = (((i&0x8)==0)^((j&0x8)==0))*255;
            checkImage[i][j][0] = (GLubyte) c;
            checkImage[i][j][1] = (GLubyte) c;
            checkImage[i][j][2] = (GLubyte) c;
            checkImage[i][j][3] = (GLubyte) 255;
        }
    }
}

void init(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_FLAT);
    glEnable (GL_DEPTH_TEST);

    makeCheckImage();
    glPixelStorei (GL_UNPACK_ALIGNMENT, 1);

    glGenTextures (1, &texName);
    glBindTexture (GL_TEXTURE_2D, texName);

    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);
    glTexImage2D (GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,
                checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                checkImage);
}

void display(void)
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable (GL_TEXTURE_2D);
    glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
}

```

```

glBindTexture(GL_TEXTURE_2D, texName);
glBegin(GL_QUADS);
glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 1.0, 0.0);
glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
glTexCoord2f(1.0, 0.0); glVertex3f(0.0, -1.0, 0.0);

glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
glTexCoord2f(0.0, 1.0); glVertex3f(1.0, 1.0, 0.0);
glTexCoord2f(1.0, 1.0); glVertex3f(2.41421, 1.0, -1.41421);
glTexCoord2f(1.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
glEnd();
glFlush();
glDisable(GL_TEXTURE_2D);
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.6);
}

void keyboard (unsigned char key, int x, int y)
{
    switch (key) {
        case 27:
            exit(0);
            break;
        default:
            break;
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(250, 250);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
}

```



```

    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}

```

The checkerboard texture is generated in the routine `makeCheckImage()`, and all the texture-mapping initialization occurs in the routine `init()`. `glGenTextures()` and `glBindTexture()` name and create a texture object for a texture image. (See “Texture Objects” on page 346.) The single, full-resolution texture map is specified by `glTexImage2D()`, whose parameters indicate the size of the image, type of the image, location of the image, and other properties of it. (See “Specifying the Texture” on page 326 for more information about `glTexImage2D()`.)

The four calls to `glTexParameter*()` specify how the texture is to be wrapped and how the colors are to be filtered if there isn’t an exact match between pixels in the texture and pixels on the screen. (See “Repeating and Clamping Textures” on page 360 and “Filtering” on page 344.)

In `display()`, `glEnable()` turns on texturing. `glTexEnv*()` sets the drawing mode to `GL_DECAL` so that the textured polygons are drawn using the colors from the texture map (rather than taking into account what color the polygons would have been drawn without the texture).

Then, two polygons are drawn. Note that texture coordinates are specified along with vertex coordinates. The `glTexCoord*()` command behaves similarly to the `glNormal()` command. `glTexCoord*()` sets the current texture coordinates; any subsequent vertex command has those texture coordinates associated with it until `glTexCoord*()` is called again.

Note: The checkerboard image on the tilted polygon might look wrong when you compile and run it on your machine—for example, it might look like two triangles with different projections of the checkerboard image on them. If so, try setting the parameter `GL_PERSPECTIVE_CORRECTION_HINT` to `GL_NICEST` and running the example again. To do this, use `glHint()`.

Specifying the Texture

The command `glTexImage2D()` defines a two-dimensional texture. It takes several arguments, which are described briefly here and in more detail in the subsections that follow. The related command for one-dimensional

textures, `glTexImage1D()`, is described in “One-Dimensional Textures” on page 335.

```
void glTexImage2D(GLenum target, GLint level, GLint internalFormat,
                  GLsizei width, GLsizei height, GLint border,
                  GLenum format, GLenum type,
                  const GLvoid *pixels);
```

Defines a two-dimensional texture. The *target* parameter is set to either the constant `GL_TEXTURE_2D` or `GL_PROXY_TEXTURE_2D`. You use the *level* parameter if you're supplying multiple resolutions of the texture map; with only one resolution, *level* should be 0. (See “Multiple Levels of Detail” on page 338 for more information about using multiple resolutions.)

The next parameter, *internalFormat*, indicates which of the R, G, B, and A components or luminance or intensity values are selected for use in describing the texels of an image. The value of *internalFormat* is an integer from 1 to 4, or one of thirty-eight symbolic constants. The thirty-eight symbolic constants that are also legal values for *internalFormat* are `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_INTENSITY`, `GL_INTENSITY4`, `GL_INTENSITY8`, `GL_INTENSITY12`, `GL_INTENSITY16`, `GL_RGB`, `GL_R3_G3_B2`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGBA8`, `GL_RGBA12`, and `GL_RGBA16`. (See “Texture Functions” on page 354 for a discussion of how these selected components are applied.)

If *internalFormat* is one of the thirty-eight symbolic constants, then you are asking for specific components and perhaps the resolution of those components. For example, if *internalFormat* is `GL_R3_G3_B2`, you are asking that texels be 3 bits of red, 3 bits of green, and 2 bits of blue, but OpenGL is not guaranteed to deliver this. OpenGL is only obligated to choose an internal representation that closely approximates what is requested, but an exact match is usually not required. By definition, `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA`, `GL_RGB`, and `GL_RGBA` are lenient, because they do not ask for a specific resolution. (For compatibility with the OpenGL release 1.0, the numeric values 1, 2, 3,

and 4; for *internalFormat*, are equivalent to the symbolic constants `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA`, `GL_RGB`, and `GL_RGBA`, respectively.)

The *width* and *height* parameters give the dimensions of the texture image; *border* indicates the width of the border, which is either zero (no border) or one. (See “Using a Texture’s Borders” on page 337.) Both *width* and *height* must have the form $2^m + 2b$, where m is a nonnegative integer (which can have a different value for *width* than for *height*) and b is the value of *border*. The maximum size of a texture map depends on the implementation of OpenGL, but it must be at least 64×64 (or 66×66 with borders).

The *format* and *type* parameters describe the format and data type of the texture image data. They have the same meaning as they do for `glDrawPixels()`. (See “Imaging Pipeline” on page 296.) In fact, texture data is in the same format as the data used by `glDrawPixels()`, so the settings of `glPixelStore*()` and `glPixelTransfer*()` are applied. (In Example 9-1, the call

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

is made because the data in the example isn’t padded at the end of each texel row.) The *format* parameter can be `GL_COLOR_INDEX`, `GL_RGB`, `GL_RGBA`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_LUMINANCE`, or `GL_LUMINANCE_ALPHA`—that is, the same formats available for `glDrawPixels()` with the exceptions of `GL_STENCIL_INDEX` and `GL_DEPTH_COMPONENT`.

Similarly, the *type* parameter can be `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_FLOAT`, or `GL_BITMAP`.

Finally, *pixels* contains the texture-image data. This data describes the texture image itself as well as its border.

The internal format of a texture image may affect the performance of texture operations. For example, some implementations perform texturing with `GL_RGBA` faster than `GL_RGB`, because the color components align the processor memory better. Since this varies, you should check specific information about your implementation of OpenGL.

The internal format of a texture image also may control how much memory a texture image consumes. For example, a texture of internal format

GL_RGBA8 uses 32 bits per texel, while a texture of internal format GL_R3_G3_B2 only uses 8 bits per texel. Of course, there is a corresponding trade-off between memory consumption and color resolution.

Note: Although texture-mapping results in color-index mode are undefined, you can still specify a texture with a GL_COLOR_INDEX image. In that case, pixel-transfer operations are applied to convert the indices to RGBA values by table lookup before they're used to form the texture image.

The number of texels for both the width and height of a texture image, not including the optional border, must be a power of 2. If your original image does not have dimensions that fit that limitation, you can use the OpenGL Utility Library routine `gluScaleImage()` to alter the size of your textures.

```
int gluScaleImage(GLenum format, GLint widthin, GLint heightin,
                 GLenum typein, const void *datain, GLint widthout,
                 GLint heightout, GLenum typeout, void *dataout);
```

Scales an image using the appropriate pixel-storage modes to unpack the data from *datain*. The *format*, *typein*, and *typeout* parameters can refer to any of the formats or data types supported by `glDrawPixels()`. The image is scaled using linear interpolation and box filtering (from the size indicated by *widthin* and *heightin* to *widthout* and *heightout*), and the resulting image is written to *dataout*, using the pixel GL_PACK* storage modes. The caller of `gluScaleImage()` must allocate sufficient space for the output buffer. A value of 0 is returned on success, and a GLU error code is returned on failure.

The framebuffer itself can also be used as a source for texture data. `glCopyTexImage2D()` reads a rectangle of pixels from the framebuffer and uses it for a new texture.

```
void glCopyTexImage2D(GLenum target, GLint level,
                    GLint internalFormat,
                    GLint x, GLint y, GLsizei width, GLsizei height,
                    GLint border);
```

Creates a two-dimensional texture, using framebuffer data to define the texels. The pixels are read from the current GL_READ_BUFFER and are processed exactly as if `glCopyPixels()` had been called but stopped before final conversion. The settings of `glPixelTransfer*()` are applied.

The *target* parameter must be set to the constant `GL_TEXTURE_2D`. The *level*, *internalFormat*, and *border* parameters have the same effects that they have for `glTexImage2D()`. The texture array is taken from a screen-aligned pixel rectangle with the lower-left corner at coordinates specified by the *(x, y)* parameters. The *width* and *height* parameters specify the size of this pixel rectangle. Both *width* and *height* must have the form 2^m+2b , where *m* is a nonnegative integer (which can have a different value for *width* than for *height*) and *b* is the value of *border*.

The next sections give more detail about texturing, including the use of the *target*, *border*, and *level* parameters. The *target* parameter can be used to accurately query the size of a texture (by creating a texture proxy with `glTexImage*D()`) and whether a texture possibly can be used within the texture resources of an OpenGL implementation. Redefining a portion of a texture is described in “Replacing All or Part of a Texture Image” on page 332. One-dimensional textures are discussed in “One-Dimensional Textures” on page 335. The texture border, which has its size controlled by the *border* parameter, is detailed in “Using a Texture’s Borders” on page 337. The *level* parameter is used to specify textures of different resolutions and is incorporated into the special technique of *mipmapping*, which is explained in “Multiple Levels of Detail” on page 338. Mipmapping requires understanding how to filter textures as they’re applied; filtering is the subject of “Filtering” on page 344.

Texture Proxy

To an OpenGL programmer who uses textures, size is important. Texture resources are typically limited and vary among OpenGL implementations. There is a special texture proxy target to evaluate whether sufficient resources are available.

`glGetIntegerv(GL_MAX_TEXTURE_SIZE,...)` tells you the largest dimension (width or height, without borders) of a texture image, typically the size of the largest square texture supported. However, `GL_MAX_TEXTURE_SIZE` does not consider the effect of the internal format of a texture. A texture image that stores texels using the `GL_RGBA16` internal format may be using 64 bits per texel, so its image may have to be 16 times smaller than an image with the `GL_LUMINANCE4` internal format. (Also, images requiring borders or mipmaps may further reduce the amount of available memory.)

A special place holder, or *proxy*, for a texture image allows the program to query more accurately whether OpenGL can accommodate a texture of a

desired internal format. To use the proxy to query OpenGL, call `glTexImage2D()` with a *target* parameter of `GL_PROXY_TEXTURE_2D` and the given *level*, *internalFormat*, *width*, *height*, *border*, *format*, and *type*. (For one-dimensional textures, use corresponding 1D routines and symbolic constants.) For a proxy, you should pass `NULL` as the pointer for the *pixels* array.

To find out whether there are enough resources available for your texture, after the texture proxy has been created, query the texture state variables with `glGetTexLevelParameter*()`. If there aren't enough resources to accommodate the texture proxy, the texture state variables for width, height, border width, and component resolutions are set to 0.

```
void glGetTexLevelParameteri(GLenum target, GLint level,
                             GLenum pname, TYPE *params);
```

Returns in *params* texture parameter values for a specific level of detail, specified as *level*. *target* defines the target texture and is one of `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_PROXY_TEXTURE_1D`, or `GL_PROXY_TEXTURE_2D`. Accepted values for *pname* are `GL_TEXTURE_WIDTH`, `GL_TEXTURE_HEIGHT`, `GL_TEXTURE_BORDER`, `GL_TEXTURE_INTERNAL_FORMAT`, `GL_TEXTURE_RED_SIZE`, `GL_TEXTURE_GREEN_SIZE`, `GL_TEXTURE_BLUE_SIZE`, `GL_TEXTURE_ALPHA_SIZE`, `GL_TEXTURE_LUMINANCE_SIZE`, or `GL_TEXTURE_INTENSITY_SIZE`.

`GL_TEXTURE_COMPONENTS` is also accepted for *pname*, but only for backward compatibility with OpenGL Release 1.0—`GL_TEXTURE_INTERNAL_FORMAT` is the recommended symbolic constant for Release 1.1.

Example 9-2 demonstrates how to use the texture proxy to find out if there are enough resources to create a 64×64 texel texture with RGBA components with 8 bits of resolution. If this succeeds, then `glGetTexLevelParameteriv()` stores the internal format (in this case, `GL_RGBA8`) into the variable *format*.

Example 9-2 Querying Texture Resources with a Texture Proxy

```
GLint format;

glTexImage2D(GL_PROXY_TEXTURE_2D, 0, GL_RGBA8,
             64, 64, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D, 0,
                        GL_TEXTURE_INTERNAL_FORMAT, &format);
```

Note: There is one major limitation about texture proxies: The texture proxy tells you if there is space for your texture, but only if all texture resources are available (in other words, if it's the only texture in town). If other textures are using resources, then the texture proxy query may respond affirmatively, but there may not be enough space to make your texture resident (that is, part of a possibly high-performance working set of textures). (See "Texture Objects" on page 346 for more information about managing resident textures.)

Replacing All or Part of a Texture Image

Creating a texture may be more computationally expensive than modifying an existing one. In OpenGL Release 1.1, there are new routines to replace all or part of a texture image with new information. This can be helpful for certain applications, such as using real-time, captured video images as texture images. For that application, it makes sense to create a single texture and use `glTexSubImage2D()` to repeatedly replace the texture data with new video images. Also, there are no size restrictions for `glTexSubImage2D()` that force the height or width to be a power of two. This is helpful for processing video images, which generally do not have sizes that are powers of two.

```
void glTexSubImage2D(GLenum target, GLint level, GLint xoffset,
                    GLint yoffset, GLsizei width, GLsizei height,
                    GLenum format, GLenum type, const GLvoid *pixels);
```

Defines a two-dimensional texture image that replaces all or part of a contiguous subregion (in 2D; it's simply a rectangle) of the current, existing two-dimensional texture image. The *target* parameter must be set to `GL_TEXTURE_2D`.

The *level*, *format*, and *type* parameters are similar to the ones used for `glTexImage2D()`. *level* is the mipmap level-of-detail number. It is not an error to specify a width or height of zero, but the subimage will have no effect. *format* and *type* describe the format and data type of the texture image data. The subimage is also affected by modes set by `glPixelStore*()` and `glPixelTransfer*()`.

pixels contains the texture data for the subimage. *width* and *height* are the dimensions of the subregion that is replacing all or part of the current texture image. *xoffset* and *yoffset* specify the texel offset in the *x* and *y*

directions (with (0, 0) at the lower-left corner of the texture) and specify where to put the subimage within the existing texture array. This region may not include any texels outside the range of the originally defined texture array.

In Example 9-3, some of the code from Example 9-1 has been modified so that pressing the 's' key drops a smaller checkered subimage into the existing image. (The resulting texture is shown in Figure 9-3.) Pressing the 'r' key restores the original image. Example 9-3 shows the two routines, `makeCheckImages()` and `keyboard()`, that have been substantially changed. (See "Texture Objects" on page 346 for more information about `glBindTexture()`.)

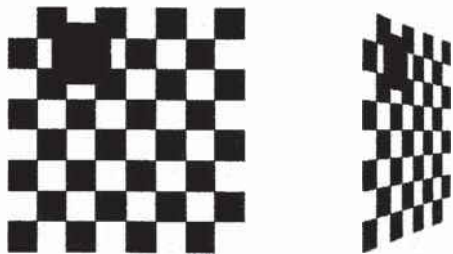


Figure 9-3 Texture with Subimage Added

Example 9-3 Replacing a Texture Subimage: `texsub.c`

```
/* Create checkerboard textures */
#define checkImageWidth 64
#define checkImageHeight 64
#define subImageWidth 16
#define subImageHeight 16
static GLubyte checkImage[checkImageHeight][checkImageWidth][4];
static GLubyte subImage[subImageHeight][subImageWidth][4];

void makeCheckImages(void)
{
    int i, j, c;

    for (i = 0; i < checkImageHeight; i++) {
        for (j = 0; j < checkImageWidth; j++) {
            c = (((i&0x8)==0)^(j&0x8)==0)*255;
            checkImage[i][j][0] = (GLubyte) c;
            checkImage[i][j][1] = (GLubyte) c;
            checkImage[i][j][2] = (GLubyte) c;
            checkImage[i][j][3] = (GLubyte) 255;
        }
    }
}
```



```

    }
    for (i = 0; i < subImageHeight; i++) {
        for (j = 0; j < subImageWidth; j++) {
            c = (((i&0x4)==0)^(j&0x4)==0)*255;
            subImage[i][j][0] = (GLubyte) c;
            subImage[i][j][1] = (GLubyte) 0;
            subImage[i][j][2] = (GLubyte) 0;
            subImage[i][j][3] = (GLubyte) 255;
        }
    }
}

void keyboard (unsigned char key, int x, int y)
{
    switch (key) {
        case 's':
        case 'S':
            glBindTexture(GL_TEXTURE_2D, texName);
            glTexSubImage2D(GL_TEXTURE_2D, 0, 12, 44,
                           subImageWidth, subImageHeight, GL_RGBA,
                           GL_UNSIGNED_BYTE, subImage);

            glutPostRedisplay();
            break;
        case 'r':
        case 'R':
            glBindTexture(GL_TEXTURE_2D, texName);
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,
                        checkImageWidth, checkImageHeight, 0,
                        GL_RGBA, GL_UNSIGNED_BYTE, checkImage);

            glutPostRedisplay();
            break;
        case 27:
            exit(0);
            break;
        default:
            break;
    }
}

```

Once again, the framebuffer itself can be used as a source for texture data; this time, a texture subimage. `glCopyTexSubImage2D()` reads a rectangle of pixels from the framebuffer and replaces a portion of an existing texture array. (`glCopyTexSubImage2D()` is kind of a cross between `glCopyTexImage2D()` and `glTexSubImage2D()`.)

```
void glCopyTexSubImage2D(GLenum target, GLint level,
                        GLint xoffset, GLint yoffset, GLint x, GLint y,
                        GLsizei width, GLsizei height);
```

Uses image data from the framebuffer to replace all or part of a contiguous subregion of the current, existing two-dimensional texture image. The pixels are read from the current `GL_READ_BUFFER` and are processed exactly as if `glCopyPixels()` had been called, stopping before final conversion. The settings of `glPixelStore*()` and `glPixelTransfer*()` are applied.

The *target* parameter must be set to `GL_TEXTURE_2D`. *level* is the mipmap level-of-detail number. *xoffset* and *yoffset* specify the texel offset in the *x* and *y* directions (with (0, 0) at the lower-left corner of the texture) and specify where to put the subimage within the existing texture array. The subimage texture array is taken from a screen-aligned pixel rectangle with the lower-left corner at coordinates specified by the (*x*, *y*) parameters. The *width* and *height* parameters specify the size of this subimage rectangle.

One-Dimensional Textures

Sometimes a one-dimensional texture is sufficient—for example, if you're drawing textured bands where all the variation is in one direction. A one-dimensional texture behaves like a two-dimensional one with *height* = 1, and without borders along the top and bottom. All the two-dimensional texture and subtexture definition routines have corresponding one-dimensional routines. To create a simple one-dimensional texture, use `glTexImage1D()`.

```
void glTexImage1D(GLenum target, GLint level, GLint internalFormat,
                 GLsizei width, GLint border, GLenum format,
                 GLenum type, const GLvoid *pixels);
```

Defines a one-dimensional texture. All the parameters have the same meanings as for `glTexImage2D()`, except that the image is now a one-dimensional array of texels. As before, the value of *width* is 2^m (or 2^m+2 , if there's a border), where *m* is a nonnegative integer. You can supply mipmaps, proxies (set *target* to `GL_PROXY_TEXTURE_1D`), and the same filtering options are available as well.

For a sample program that uses a one-dimensional texture map, see Example 9-6 on page 365.

To replace all or some of the texels of a one-dimensional texture, use `glTexSubImage1D()`.

```
void glTexSubImage1D(GLenum target, GLint level, GLint xoffset,
                    GLsizei width, GLenum format,
                    GLenum type, const GLvoid *pixels);
```

Defines a one-dimensional texture array that replaces all or part of a contiguous subregion (in 1D, a row) of the current, existing one-dimensional texture image. The *target* parameter must be set to `GL_TEXTURE_1D`.

The *level*, *format*, and *type* parameters are similar to the ones used for `glTexImage1D()`. *level* is the mipmap level-of-detail number. *format* and *type* describe the format and data type of the texture image data. The subimage is also affected by modes set by `glPixelStore*()` or `glPixelTransfer*()`.

pixels contains the texture data for the subimage. *width* is the number of texels that replace part or all of the current texture image. *xoffset* specifies the texel offset for where to put the subimage within the existing texture array.

To use the framebuffer as the source of a new or replacement for an old one-dimensional texture, use either `glCopyTexImage1D()` or `glCopyTexSubImage1D()`.

```
void glCopyTexImage1D(GLenum target, GLint level,
                     GLint internalFormat, GLint x, GLint y,
                     GLsizei width, GLint border);
```

Creates a one-dimensional texture, using framebuffer data to define the texels. The pixels are read from the current `GL_READ_BUFFER` and are processed exactly as if `glCopyPixels()` had been called but stopped before final conversion. The settings of `glPixelStore*()` and `glPixelTransfer*()` are applied.

The *target* parameter must be set to the constant `GL_TEXTURE_1D`. The *level*, *internalFormat*, and *border* parameters have the same effects that they have for `glCopyTexImage2D()`. The texture array is taken from a row of pixels with the lower-left corner at coordinates specified by the *(x, y)* parameters. The *width* parameter specifies the number of pixels in this row. The value of *width* is 2^m (or 2^m+2 if there's a border), where *m* is a nonnegative integer.

```
void glCopyTexSubImage1D(GLenum target, GLint level, GLint xoffset,
                          GLint x, GLint y, GLsizei width);
```

Uses image data from the framebuffer to replace all or part of a contiguous subregion of the current, existing one-dimensional texture image. The pixels are read from the current `GL_READ_BUFFER` and are processed exactly as if `glCopyPixels()` had been called but stopped before final conversion. The settings of `glPixelStore*()` and `glPixelTransfer*()` are applied.

The *target* parameter must be set to `GL_TEXTURE_1D`. *level* is the mipmap level-of-detail number. *xoffset* specifies the texel offset and specifies where to put the subimage within the existing texture array. The subimage texture array is taken from a row of pixels with the lower-left corner at coordinates specified by the *(x, y)* parameters. The *width* parameter specifies the number of pixels in this row.

Using a Texture's Borders

Advanced

If you need to apply a larger texture map than your implementation of OpenGL allows, you can, with a little care, effectively make larger textures by tiling with several different textures. For example, if you need a texture twice as large as the maximum allowed size mapped to a square, draw the square as four subsquares, and load a different texture before drawing each piece.

Since only a single texture map is available at one time, this approach might lead to problems at the edges of the textures, especially if some form of linear filtering is enabled. The texture value to be used for pixels at the edges must be averaged with something beyond the edge, which, ideally, should come from the adjacent texture map. If you define a border for each texture whose texel values are equal to the values of the texels on the edge of the adjacent texture map, then the correct behavior results when linear filtering takes place.

To do this correctly, notice that each map can have eight neighbors—one adjacent to each edge, and one touching each corner. The values of the texels in the corner of the border need to correspond with the texels in the texture maps that touch the corners. If your texture is an edge or corner of the whole tiling, you need to decide what values would be reasonable to put in the borders. The easiest reasonable thing to do is to copy the value of the



adjacent texel in the texture map. Remember that the border values need to be supplied at the same time as the texture-image data, so you need to figure this out ahead of time.

A texture's border color is also used if the texture is applied in such a way that it only partially covers a primitive. (See "Repeating and Clamping Textures" on page 360 for more information about this situation.)

Multiple Levels of Detail



Advanced

Textured objects can be viewed, like any other objects in a scene, at different distances from the viewpoint. In a dynamic scene, as a textured object moves farther from the viewpoint, the texture map must decrease in size along with the size of the projected image. To accomplish this, OpenGL has to filter the texture map down to an appropriate size for mapping onto the object, without introducing visually disturbing artifacts. For example, to render a brick wall, you may use a large (say 128×128 texel) texture image when it is close to the viewer. But if the wall is moved farther away from the viewer until it appears on the screen as a single pixel, then the filtered textures may appear to change abruptly at certain transition points.

To avoid such artifacts, you can specify a series of prefiltered texture maps of decreasing resolutions, called *mipmaps*, as shown in Figure 9-4. The term *mipmap* was coined by Lance Williams, when he introduced the idea in his paper, "Pyramidal Parametrics" (SIGGRAPH 1983 Proceedings). *Mip* stands for the Latin *multum in parvo*, meaning "many things in a small place." Mipmapping uses some clever methods to pack image data into memory.

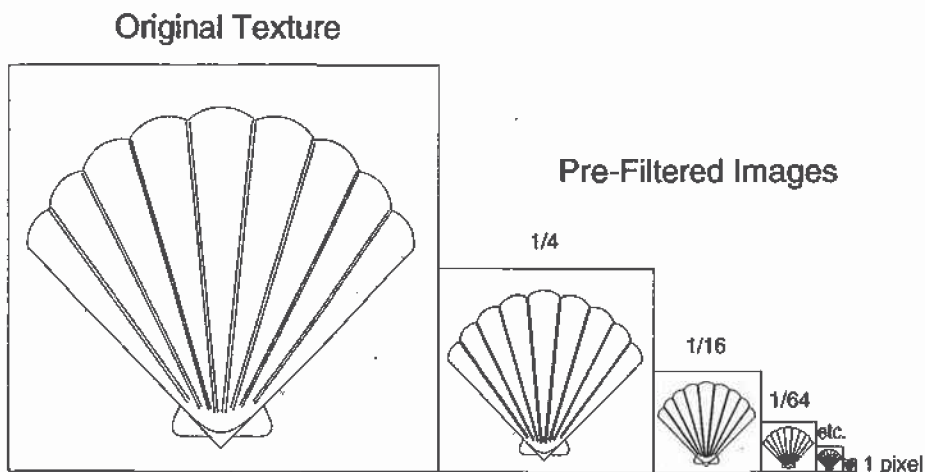


Figure 9-4 Mipmaps

When using mipmapping, OpenGL automatically determines which texture map to use based on the size (in pixels) of the object being mapped. With this approach, the level of detail in the texture map is appropriate for the image that's drawn on the screen—as the image of the object gets smaller, the size of the texture map decreases. Mipmapping requires some extra computation and texture storage area; however, when it's not used, textures that are mapped onto smaller objects might shimmer and flash as the objects move.

To use mipmapping, you must provide all sizes of your texture in powers of 2 between the largest size and a 1×1 map. For example, if your highest-resolution map is 64×16, you must also provide maps of size 32×8, 16×4, 8×2, 4×1, 2×1, and 1×1. The smaller maps are typically filtered and averaged-down versions of the largest map in which each texel in a smaller texture is an average of the corresponding four texels in the larger texture. (Since OpenGL doesn't require any particular method for calculating the smaller maps, the differently sized textures could be totally unrelated. In practice, unrelated textures would make the transitions between mipmaps extremely noticeable.)

To specify these textures, call `glTexImage2D()` once for each resolution of the texture map, with different values for the *level*, *width*, *height*, and *image* parameters. Starting with zero, *level* identifies which texture in the series is specified; with the previous example, the largest texture of size 64×16 would be declared with *level* = 0, the 32×8 texture with *level* = 1, and so on. In addition, for the mipmapped textures to take effect, you need to choose one of the appropriate filtering methods described in the next section.

Example 9-4 illustrates the use of a series of six texture maps decreasing in size from 32×32 to 1×1. This program draws a rectangle that extends from the foreground far back in the distance, eventually disappearing at a point, as shown in Plate 20. Note that the texture coordinates range from 0.0 to 8.0 so 64 copies of the texture map are required to tile the rectangle, eight in each direction. To illustrate how one texture map succeeds another, each map has a different color.

Example 9-4 Mipmap Textures: `mipmap.c`

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>

GLubyte mipmapImage32[32][32][4];
GLubyte mipmapImage16[16][16][4];
GLubyte mipmapImage8[8][8][4];
GLubyte mipmapImage4[4][4][4];
GLubyte mipmapImage2[2][2][4];
GLubyte mipmapImage1[1][1][4];

static GLuint texName;

void makeImages(void)
{
    int i, j;

    for (i = 0; i < 32; i++) {
        for (j = 0; j < 32; j++) {
            mipmapImage32[i][j][0] = 255;
            mipmapImage32[i][j][1] = 255;
            mipmapImage32[i][j][2] = 0;
            mipmapImage32[i][j][3] = 255;
        }
    }
    for (i = 0; i < 16; i++) {
        for (j = 0; j < 16; j++) {
            mipmapImage16[i][j][0] = 255;
            mipmapImage16[i][j][1] = 0;
            mipmapImage16[i][j][2] = 255;
            mipmapImage16[i][j][3] = 255;
        }
    }
    for (i = 0; i < 8; i++) {
        for (j = 0; j < 8; j++) {
```

```

        mipmapImage8[i][j][0] = 255;
        mipmapImage8[i][j][1] = 0;
        mipmapImage8[i][j][2] = 0;
        mipmapImage8[i][j][3] = 255;
    }
}
for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
        mipmapImage4[i][j][0] = 0;
        mipmapImage4[i][j][1] = 255;
        mipmapImage4[i][j][2] = 0;
        mipmapImage4[i][j][3] = 255;
    }
}
for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
        mipmapImage2[i][j][0] = 0;
        mipmapImage2[i][j][1] = 0;
        mipmapImage2[i][j][2] = 255;
        mipmapImage2[i][j][3] = 255;
    }
}
mipmapImage1[0][0][0] = 255;
mipmapImage1[0][0][1] = 255;
mipmapImage1[0][0][2] = 255;
mipmapImage1[0][0][3] = 255;
}

void init(void)
{
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);

    glTranslatef(0.0, 0.0, -3.6);
    makeImages();
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    glGenTextures(1, &texName);
    glBindTexture(GL_TEXTURE_2D, texName);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST_MIPMAP_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 32, 32, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage32);
}

```



```

    glTexImage2D(GL_TEXTURE_2D, 1, GL_RGBA, 16, 16, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage16);
    glTexImage2D(GL_TEXTURE_2D, 2, GL_RGBA, 8, 8, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage8);
    glTexImage2D(GL_TEXTURE_2D, 3, GL_RGBA, 4, 4, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage4);
    glTexImage2D(GL_TEXTURE_2D, 4, GL_RGBA, 2, 2, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage2);
    glTexImage2D(GL_TEXTURE_2D, 5, GL_RGBA, 1, 1, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage1);

    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glEnable(GL_TEXTURE_2D);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindTexture(GL_TEXTURE_2D, texName);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
    glTexCoord2f(0.0, 8.0); glVertex3f(-2.0, 1.0, 0.0);
    glTexCoord2f(8.0, 8.0); glVertex3f(2000.0, 1.0, -6000.0);
    glTexCoord2f(8.0, 0.0); glVertex3f(2000.0, -1.0, -6000.0);
    glEnd();
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (GLfloat)w/(GLfloat)h, 1.0, 3000.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void keyboard (unsigned char key, int x, int y)
{
    switch (key) {
        case 27:
            exit(0);
            break;
        default:
            break;
    }
}

```

```

}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(50, 50);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}

```

Example 9-4 illustrates mipmapping by making each mipmap a different color so that it's obvious when one map is replaced by another. In a real situation, you define mipmaps so that the transition is as smooth as possible. Thus, the maps of lower resolution are usually filtered versions of an original, high-resolution map. The construction of a series of such mipmaps is a software process, and thus isn't part of OpenGL, which is simply a rendering library. However, since mipmap construction is such an important operation, however, the OpenGL Utility Library contains two routines that aid in the manipulation of images to be used as mipmapped textures.

Assuming you have constructed the level 0, or highest-resolution map, the routines `gluBuild1DMipmaps()` and `gluBuild2DMipmaps()` construct and define the pyramid of mipmaps down to a resolution of 1×1 (or 1, for one-dimensional texture maps). If your original image has dimensions that are not exact powers of 2, `gluBuild*DMipmaps()` helpfully scales the image to the nearest power of 2.

```

int gluBuild1DMipmaps(GLenum target, GLint components, GLint width,
                     GLenum format, GLenum type, void *data);
int gluBuild2DMipmaps(GLenum target, GLint components, GLint width,
                     GLint height, GLenum format, GLenum type,
                     void *data);

```

Constructs a series of mipmaps and calls `glTexImage*D()` to load the images. The parameters for *target*, *components*, *width*, *height*, *format*, *type*, and *data* are exactly the same as those for `glTexImage1D()` and `glTexImage2D()`. A value of 0 is returned if all the mipmaps are constructed successfully; otherwise, a GLU error code is returned.

Filtering

Texture maps are square or rectangular, but after being mapped to a polygon or surface and transformed into screen coordinates, the individual texels of a texture rarely correspond to individual pixels of the final screen image. Depending on the transformations used and the texture mapping applied, a single pixel on the screen can correspond to anything from a tiny portion of a texel (magnification) to a large collection of texels (minification), as shown in Figure 9-5. In either case, it's unclear exactly which texel values should be used and how they should be averaged or interpolated. Consequently, OpenGL allows you to specify any of several filtering options to determine these calculations. The options provide different trade-offs between speed and image quality. Also, you can specify independently the filtering methods for magnification and minification.

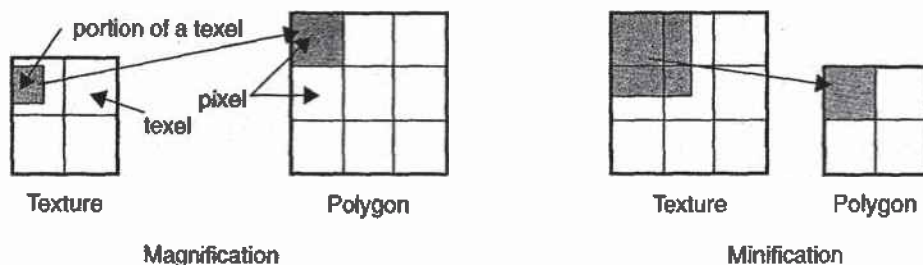


Figure 9-5 Texture Magnification and Minification

In some cases, it isn't obvious whether magnification or minification is called for. If the mipmap needs to be stretched (or shrunk) in both the *x* and *y* directions, then magnification (or minification) is needed. If the mipmap needs to be stretched in one direction and shrunk in the other, OpenGL

makes a choice between magnification and minification that in most cases gives the best result possible. It's best to try to avoid these situations by using texture coordinates that map without such distortion. (See "Computing Appropriate Texture Coordinates" on page 358.)

The following lines are examples of how to use `glTexParameter*()` to specify the magnification and minification filtering methods:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,  
                GL_NEAREST);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,  
                GL_NEAREST);
```

The first argument to `glTexParameter*()` is either `GL_TEXTURE_2D` or `GL_TEXTURE_1D`, depending on whether you're working with two- or one-dimensional textures. For the purposes of this discussion, the second argument is either `GL_TEXTURE_MAG_FILTER` or `GL_TEXTURE_MIN_FILTER` to indicate whether you're specifying the filtering method for magnification or minification. The third argument specifies the filtering method; Table 9-1 lists the possible values.

Parameter	Values
<code>GL_TEXTURE_MAG_FILTER</code>	<code>GL_NEAREST</code> or <code>GL_LINEAR</code>
<code>GL_TEXTURE_MIN_FILTER</code>	<code>GL_NEAREST</code> , <code>GL_LINEAR</code> , <code>GL_NEAREST_MIPMAP_NEAREST</code> , <code>GL_NEAREST_MIPMAP_LINEAR</code> , <code>GL_LINEAR_MIPMAP_NEAREST</code> , or <code>GL_LINEAR_MIPMAP_LINEAR</code>

Table 9-1 Filtering Methods for Magnification and Minification

If you choose `GL_NEAREST`, the texel with coordinates nearest the center of the pixel is used for both magnification and minification. This can result in aliasing artifacts (sometimes severe). If you choose `GL_LINEAR`, a weighted linear average of the 2×2 array of texels that lie nearest to the center of the pixel is used, again for both magnification and minification. When the texture coordinates are near the edge of the texture map, the nearest 2×2 array of texels might include some that are outside the texture map. In these cases, the texel values used depend on whether `GL_REPEAT` or `GL_CLAMP` is in effect and whether you've assigned a border for the texture. (See "Using a Texture's Borders" on page 337.) `GL_NEAREST` requires less computation than `GL_LINEAR` and therefore might execute more quickly, but `GL_LINEAR` provides smoother results.

With magnification, even if you've supplied mipmaps, the largest texture map (*level* = 0) is always used. With minification, you can choose a filtering method that uses the most appropriate one or two mipmaps, as described in the next paragraph. (If `GL_NEAREST` or `GL_LINEAR` is specified with minification, the largest texture map is used.)

As shown in Table 9-1, four additional filtering choices are available when minifying with mipmaps. Within an individual mipmap, you can choose the nearest texel value with `GL_NEAREST_MIPMAP_NEAREST`, or you can interpolate linearly by specifying `GL_LINEAR_MIPMAP_NEAREST`. Using the nearest texels is faster but yields less desirable results. The particular mipmap chosen is a function of the amount of minification required, and there's a cutoff point from the use of one particular mipmap to the next. To avoid a sudden transition, use `GL_NEAREST_MIPMAP_LINEAR` or `GL_LINEAR_MIPMAP_LINEAR` to linearly interpolate texel values from the two nearest best choices of mipmaps. `GL_NEAREST_MIPMAP_LINEAR` selects the nearest texel in each of the two maps and then interpolates linearly between these two values. `GL_LINEAR_MIPMAP_LINEAR` uses linear interpolation to compute the value in each of two maps and then interpolates linearly between these two values. As you might expect, `GL_LINEAR_MIPMAP_LINEAR` generally produces the smoothest results, but it requires the most computation and therefore might be the slowest.

Texture Objects

Texture objects are an important new feature in release 1.1 of OpenGL. A texture object stores texture data and makes it readily available. You can now control many textures and go back to textures that have been previously loaded into your texture resources. Using texture objects is usually the fastest way to apply textures, resulting in big performance gains, because it is almost always much faster to bind (reuse) an existing texture object than it is to reload a texture image using `glTexImage2D()`.

Also, some implementations support a limited *working set* of high-performance textures. You can use texture objects to load your most often used textures into this limited area.

To use texture objects for your texture data, take these steps.

1. Generate texture names.
2. Initially bind (create) texture objects to texture data, including the image arrays and texture properties.

-
3. If your implementation supports a working set of high-performance textures, see if you have enough space for all your texture objects. If there isn't enough space, you may wish to establish priorities for each texture object so that more often used textures stay in the working set.
 4. Bind and rebind texture objects, making their data currently available for rendering textured models.

Naming A Texture Object

Any nonzero unsigned integer may be used as a texture name. To avoid accidentally reusing names, consistently use `glGenTextures()` to provide unused texture names.

```
void glGenTextures(GLsizei n, GLuint *textureNames);
```

Returns *n* currently unused names for texture objects in the array *textureNames*. The names returned in *textureNames* do not have to be a contiguous set of integers.

The names in *textureNames* are marked as used, but they acquire texture state and dimensionality (1D or 2D) only when they are first bound.

Zero is a reserved texture name and is never returned as a texture name by `glGenTextures()`.

`glIsTexture()` determines if a texture name is actually in use. If a texture name was returned by `glGenTextures()` but has not yet been bound (calling `glBindTexture()` with the name at least once), then `glIsTexture()` returns `GL_FALSE`.

```
GLboolean glIsTexture(GLuint textureName);
```

Returns `GL_TRUE` if *textureName* is the name of a texture that has been bound and has not been subsequently deleted. Returns `GL_FALSE` if *textureName* is zero or *textureName* is a nonzero value that is not the name of an existing texture.

Creating and Using Texture Objects

The same routine, `glBindTexture()`, both creates and uses texture objects. When a texture name is initially bound (used with `glBindTexture()`), a new texture object is created with default values for the texture image and texture properties. Subsequent calls to `glTexImage*()`, `glTexSubImage*()`, `glCopyTexImage*()`, `glCopyTexSubImage*()`, `glTexParameter*()`, and `glPrioritizeTextures()` store data in the texture object. The texture object may contain a texture image and associated mipmap images (if any), including associated data such as width, height, border width, internal format, resolution of components, and texture properties. Saved texture properties include minification and magnification filters, wrapping modes, border color, and texture priority.

When a texture object is subsequently bound once again, its data becomes the current texture state. (The state of the previously bound texture is replaced.)

```
void glBindTexture(GLenum target, GLuint textureName);
```

`glBindTexture()` does three things. When using *textureName* of an unsigned integer other than zero for the first time, a new texture object is created and assigned that name. When binding to a previously created texture object, that texture object becomes active. When binding to a *textureName* value of zero, OpenGL stops using texture objects and returns to the unnamed default texture.

When a texture object is initially bound (that is, created), it assumes the dimensionality of *target*, which is either `GL_TEXTURE_1D` or `GL_TEXTURE_2D`. Immediately upon its initial binding, the state of texture object is equivalent to the state of the default `GL_TEXTURE_1D` or `GL_TEXTURE_2D` (depending upon its dimensionality) at the initialization of OpenGL. In this initial state, texture properties such as minification and magnification filters, wrapping modes, border color, and texture priority are set to their default values.

In Example 9-5, two texture objects are created in `init()`. In `display()`, each texture object is used to render a different four-sided polygon.

Example 9-5 Binding Texture Objects: `texbind.c`

```
#define checkImageWidth 64
#define checkImageHeight 64
static GLubyte checkImage[checkImageHeight][checkImageWidth][4];
```

```

static GLubyte otherImage[checkImageHeight][checkImageWidth][4];

static GLuint texName[2];

void makeCheckImages(void)
{
    int i, j, c;

    for (i = 0; i < checkImageHeight; i++) {
        for (j = 0; j < checkImageWidth; j++) {
            c = (((i&0x8)==0)^((j&0x8)==0))*255;
            checkImage[i][j][0] = (GLubyte) c;
            checkImage[i][j][1] = (GLubyte) c;
            checkImage[i][j][2] = (GLubyte) c;
            checkImage[i][j][3] = (GLubyte) 255;
            c = (((i&0x10)==0)^((j&0x10)==0))*255;
            otherImage[i][j][0] = (GLubyte) c;
            otherImage[i][j][1] = (GLubyte) 0;
            otherImage[i][j][2] = (GLubyte) 0;
            otherImage[i][j][3] = (GLubyte) 255;
        }
    }
}

void init(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_FLAT);
    glEnable(GL_DEPTH_TEST);

    makeCheckImages();
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    glGenTextures(2, texName);
    glBindTexture(GL_TEXTURE_2D, texName[0]);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,
                checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                checkImage);

    glBindTexture(GL_TEXTURE_2D, texName[1]);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);

```



```

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,
                .checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                otherImage);
    glEnable(GL_TEXTURE_2D);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindTexture(GL_TEXTURE_2D, texName[0]);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
    glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 1.0, 0.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
    glTexCoord2f(1.0, 0.0); glVertex3f(0.0, -1.0, 0.0);
    glEnd();
    glBindTexture(GL_TEXTURE_2D, texName[1]);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
    glTexCoord2f(0.0, 1.0); glVertex3f(1.0, 1.0, 0.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(2.41421, 1.0, -1.41421);
    glTexCoord2f(1.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
    glEnd();
    glFlush();
}

```

Whenever a texture object is bound once again, you may edit the contents of the bound texture object. Any commands you call that change the texture image or other properties change the contents of the currently bound texture object as well as the current texture state.

In Example 9-5, after completion of `display()`, you are still bound to the texture named by the contents of `texName[1]`. Be careful that you don't call a spurious texture routine that changes the data in that texture object.

When using mipmaps, all related mipmaps of a single texture image must be put into a single texture object. In Example 9-4, levels 0-5 of a mipmapped texture image are put into a single texture object named *texName*.

Cleaning Up Texture Objects

As you bind and unbind texture objects, their data still sits around somewhere among your texture resources. If texture resources are limited, deleting textures may be one way to free up resources.

```
void glDeleteTextures(GLsizei n, const GLuint *textureNames);
```

Deletes *n* texture objects, named by elements in the array *textureNames*. The freed texture names may now be reused (for example, by `glGenTextures()`).

If a texture that is currently bound is deleted, the binding reverts to the default texture, as if `glBindTexture()` were called with zero for the value of *textureName*. Attempts to delete nonexistent texture names or the texture name of zero are ignored without generating an error.

A Working Set of Resident Textures

Some OpenGL implementations support a working set of high-performance textures, which are said to be resident. Typically, these implementations have specialized hardware to perform texture operations and a limited hardware cache to store texture images. In this case, using texture objects is recommended, because you are able to load many textures into the working set and then control them.

If all the textures required by the application exceed the size of the cache, some textures cannot be resident. If you want to find out if a single texture is currently resident, bind its object, and then use `glGetTexParameter*v()` to find out the value associated with the `GL_TEXTURE_RESIDENT` state. If you want to know about the texture residence status of many textures, use `glAreTexturesResident()`.

```
GLboolean glAreTexturesResident(GLsizei n, const
                                GLuint* textureNames, GLboolean *residences);
```

Queries the texture residence status of the n texture objects, named in the array *textureNames*; *residences* is an array in which texture residence status is returned for the corresponding texture objects in the array *textureNames*. If all the named textures in *textureNames* are resident, the `glAreTexturesResident()` function returns `GL_TRUE`, and the contents of the array *residences* are undisturbed. If any texture in *textureNames* is not resident, then `glAreTexturesResident()` returns `GL_FALSE` and the elements in *residences*, which correspond to nonresident texture objects in *textureNames*, are also set to `GL_FALSE`.

Note that `glAreTexturesResident()` returns the current residence status. Texture resources are very dynamic, and texture residence status may change at any time. Some implementations cache textures when they are first used. It may be necessary to draw with the texture before checking residency.

If your OpenGL implementation does not establish a working set of high-performance textures, then the texture objects are always considered resident. In that case, `glAreTexturesResident()` always returns `GL_TRUE` and basically provides no information.

Texture Residence Strategies

If you can create a working set of textures and want to get the best texture performance possible, you really have to know the specifics of your implementation and application. For example, with a visual simulation or video game, you have to maintain performance in all situations. In that case, you should never access a nonresident texture. For these applications, you want to load up all your textures upon initialization and make them all resident. If you don't have enough texture memory available, you may need to reduce the size, resolution, and levels of mipmaps for your texture images, or you may use `glTexSubImage*()` to repeatedly reuse the same texture memory.

For applications that create textures "on the fly," nonresident textures may be unavoidable. If some textures are used more frequently than others, you may assign a higher priority to those texture objects to increase their likelihood of being resident. Deleting texture objects also frees up space. Short of that, assigning a lower priority to a texture object may make it first

in line for being moved out of the working set, as resources dwindle. `glPrioritizeTextures()` is used to assign priorities to texture objects.

```
void glPrioritizeTextures(GLsizei n, const GLuint *textureNames,  
                          const GLclampf *priorities);
```

Assigns the n texture objects, named in the array *textureNames*, the texture residence priorities in the corresponding elements of the array *priorities*. The priority values in the array *priorities* are clamped to the range [0.0, 1.0] before being assigned. Zero indicates the lowest priority; these textures are least likely to be resident. One indicates the highest priority.

`glPrioritizeTextures()` does not require that any of the textures in *textureNames* be bound. However, the priority might not have any effect on a texture object until it is initially bound.

`glTexParameter*()` also may be used to set a single texture's priority, but only if the texture is currently bound. In fact, use of `glTexParameter*()` is the only way to set the priority of a default texture.

If texture objects have equal priority, typical implementations of OpenGL apply a least recently used (LRU) strategy to decide which texture objects to move out of the working set. If you know that your OpenGL implementation has this behavior, then having equal priorities for all texture objects creates a reasonable LRU system for reallocating texture resources.

If your implementation of OpenGL doesn't use an LRU strategy for texture objects of equal priority (or if you don't know how it decides), you can implement your own LRU strategy by carefully maintaining the texture object priorities. When a texture is used (bound), you can maximize its priority, which reflects its recent use. Then, at regular (time) intervals, you can degrade the priorities of all texture objects.

Note: Fragmentation of texture memory can be a problem, especially if you're deleting and creating lots of new textures. Although it is even possible that you can load all the texture objects into a working set by binding them in one sequence, binding them in a different sequence may leave some textures nonresident.

Texture Functions

In all the examples so far in this chapter, the values in the texture map have been used directly as colors to be painted on the surface being rendered. You can also use the values in the texture map to modulate the color that the surface would be rendered without texturing, or to blend the color in the texture map with the original color of the surface. You choose one of four texturing functions by supplying the appropriate arguments to `glTexEnv*()`.

```
void glTexEnvf(GLenum target, GLenum pname, TYPE param);  
void glTexEnvfv(GLenum target, GLenum pname, TYPE *param);
```

Sets the current texturing function. *target* must be `GL_TEXTURE_ENV`. If *pname* is `GL_TEXTURE_ENV_MODE`, *param* can be `GL_DECAL`, `GL_REPLACE`, `GL_MODULATE`, or `GL_BLEND`, to specify how texture values are to be combined with the color values of the fragment being processed. If *pname* is `GL_TEXTURE_ENV_COLOR`, *param* is an array of four floating-point values representing R, G, B, and A components. These values are used only if the `GL_BLEND` texture function has been specified as well.

The combination of the texturing function and the base internal format determine how the textures are applied for each component of the texture. The texturing function operates on selected components of the texture and the color values that would be used with no texturing. (Note that the selection is performed after the pixel-transfer function has been applied.) Recall that when you specify your texture map with `glTexImage*D()`, the third argument is the internal format to be selected for each texel.

Table 9-2 and Table 9-3 show how the texturing function and base internal format determine the texturing application formula used for each component of the texture. There are six base internal formats (the letters in parentheses represent their values in the tables): `GL_ALPHA` (A), `GL_LUMINANCE` (L), `GL_LUMINANCE_ALPHA` (L and A), `GL_INTENSITY` (I), `GL_RGB` (C), and `GL_RGBA` (C and A). Other internal formats specify

desired resolutions of the texture components and can be matched to one of these six base internal formats.

Base Internal Format	Replace Texture Function	Modulate Texture Function
GL_ALPHA	$C = C_f$ $A = A_t$	$C = C_f$ $A = A_f A_t$
GL_LUMINANCE	$C = L_b$ $A = A_f$	$C = C_f L_b$ $A = A_f$
GL_LUMINANCE_ALPHA	$C = L_b$ $A = A_t$	$C = C_f L_b$ $A = A_f A_t$
GL_INTENSITY	$C = I_b$ $A = I_t$	$C = C_f I_b$ $A = A_f I_t$
GL_RGB	$C = C_b$ $A = A_f$	$C = C_f C_b$ $A = A_f$
GL_RGBA	$C = C_b$ $A = A_t$	$C = C_f C_b$ $A = A_f A_t$

Table 9-2 Replace and Modulate Texture Functions

Base Internal Format	Decal Texture Function	Blend Texture Function
GL_ALPHA	undefined	$C = C_f$ $A = A_f A_t$
GL_LUMINANCE	undefined	$C = C_f(1 - I_b) + C_c I_b$ $A = A_f$
GL_LUMINANCE_ALPHA	undefined	$C = C_f(1 - I_b) + C_c I_b$ $A = A_f A_t$
GL_INTENSITY	undefined	$C = C_f(1 - I_b) + C_c I_b$ $A = A_f(1 - I_b) + A_c I_b$
GL_RGB	$C = C_b$ $A = A_f$	$C = C_f(1 - C_b) + C_c C_b$ $A = A_f$
GL_RGBA	$C = C_f(1 - A_b) + C_t A_b$ $A = A_f$	$C = C_f(1 - C_b) + C_c C_b$ $A = A_f A_t$

Table 9-3 Decal and Blend Texture Functions

Note: In Table 9-2 and Table 9-3, a subscript of *t* indicates a texture value, *f* indicates the incoming fragment value, *c* indicates the values assigned with `GL_TEXTURE_ENV_COLOR`, and no subscript indicates the final, computed value. Also in the tables, multiplication of a color triple by a scalar means multiplying each of the R, G, and B components by the scalar; multiplying (or adding) two color triples means multiplying (or adding) each component of the second by the corresponding component of the first.

The decal texture function makes sense only for the RGB and RGBA internal formats (remember that texture mapping doesn't work in color-index mode). With the RGB internal format, the color that would have been painted in the absence of any texture mapping (the fragment's color) is replaced by the texture color, and its alpha is unchanged. With the RGBA internal format, the fragment's color is blended with the texture color in a ratio determined by the texture alpha, and the fragment's alpha is unchanged. You use the decal texture function in situations where you want to apply an opaque texture to an object—if you were drawing a soup can with an opaque label, for example. The decal texture function also can be used to apply an alpha blended texture, such as an insignia onto an airplane wing.

The replacement texture function is similar to decal; in fact, for the RGB internal format, they are exactly the same. With all the internal formats, the component values are either replaced or left alone.

For modulation, the fragment's color is modulated by the contents of the texture map. If the base internal format is `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA`, or `GL_INTENSITY`, the color values are multiplied by the same value, so the texture map modulates between the fragment's color (if the luminance or intensity is 1) to black (if it's 0). For the `GL_RGB` and `GL_RGBA` internal formats, each of the incoming color components is multiplied by a corresponding (possibly different) value in the texture. If there's an alpha value, it's multiplied by the fragment's alpha. Modulation is a good texture function for use with lighting, since the lit polygon color can be used to attenuate the texture color. Most of the texture-mapping examples in the color plates use modulation for this reason. White, specular polygons are often used to render lit, textured objects, and the texture image provides the diffuse color.

The blending texture function is the only function that uses the color specified by `GL_TEXTURE_ENV_COLOR`. The luminance, intensity, or color value is used somewhat like an alpha value to blend the fragment's

color with the `GL_TEXTURE_ENV_COLOR`. (See “Sample Uses of Blending” on page 217 for the billboard example, which uses a blended texture.)

Assigning Texture Coordinates

As you draw your texture-mapped scene, you must provide both object coordinates and texture coordinates for each vertex. After transformation, the object coordinates determine where on the screen that particular vertex is rendered. The texture coordinates determine which texel in the texture map is assigned to that vertex. In exactly the same way that colors are interpolated between two vertices of shaded polygons and lines, texture coordinates are also interpolated between vertices. (Remember that textures are rectangular arrays of data.)

Texture coordinates can comprise one, two, three, or four coordinates. They're usually referred to as the *s*, *t*, *r*, and *q* coordinates to distinguish them from object coordinates (*x*, *y*, *z*, and *w*) and from evaluator coordinates (*u* and *v*; see Chapter 12). For one-dimensional textures, you use the *s* coordinate; for two-dimensional textures, you use *s* and *t*. In Release 1.1, the *r* coordinate is ignored. (Some implementations have 3D texture mapping as an extension, and that extension uses the *r* coordinate.) The *q* coordinate, like *w*, is typically given the value 1 and can be used to create homogeneous coordinates; it's described as an advanced feature in “The *q* Coordinate” on page 372. The command to specify texture coordinates, `glTexCoord*()`, is similar to `glVertex*()`, `glColor*()`, and `glNormal*()`—it comes in similar variations and is used the same way between `glBegin()` and `glEnd()` pairs. Usually, texture-coordinate values range from 0 to 1; values can be assigned outside this range, however, with the results described in “Repeating and Clamping Textures” on page 360.

```
void glTexCoord{1234}{sifd}(TYPE coords);  
void glTexCoord{1234}{sifd}v(TYPE *coords);
```

Sets the current texture coordinates (s, t, r, q) . Subsequent calls to `glVertex*()` result in those vertices being assigned the current texture coordinates. With `glTexCoord1*()`, the s coordinate is set to the specified value, t and r are set to 0, and q is set to 1. Using `glTexCoord2*()` allows you to specify s and t ; r and q are set to 0 and 1, respectively. With `glTexCoord3*()`, q is set to 1 and the other coordinates are set as specified. You can specify all coordinates with `glTexCoord4*()`. Use the appropriate suffix (s, i, f, or d) and the corresponding value for *TYPE* (GLshort, GLint, GLfloat, or GLdouble) to specify the coordinates' data type. You can supply the coordinates individually, or you can use the vector version of the command to supply them in a single array. Texture coordinates are multiplied by the 4x4 texture matrix before any texture mapping occurs. (See "The Texture Matrix Stack" on page 371.) Note that integer texture coordinates are interpreted directly rather than being mapped to the range $[-1, 1]$ as normal coordinates are.

The next section discusses how to calculate appropriate texture coordinates. Instead of explicitly assigning them yourself, you can choose to have texture coordinates calculated automatically by OpenGL as a function of the vertex coordinates. (See "Automatic Texture-Coordinate Generation" on page 364.)

Computing Appropriate Texture Coordinates

Two-dimensional textures are square or rectangular images that are typically mapped to the polygons that make up a polygonal model. In the simplest case, you're mapping a rectangular texture onto a model that's also rectangular—for example, your texture is a scanned image of a brick wall, and your rectangle is to represent a brick wall of a building. Suppose the brick wall is square and the texture is square, and you want to map the whole texture to the whole wall. The texture coordinates of the texture square are $(0, 0)$, $(1, 0)$, $(1, 1)$, and $(0, 1)$ in counterclockwise order. When you're drawing the wall, just give those four coordinate sets as the texture coordinates as you specify the wall's vertices in counterclockwise order.

Now suppose that the wall is two-thirds as high as it is wide, and that the texture is again square. To avoid distorting the texture, you need to map the wall to a portion of the texture map so that the aspect ratio of the texture is preserved. Suppose that you decide to use the lower two-thirds of the

texture map to texture the wall. In this case, use texture coordinates of (0,0), (1,0), (1,2/3), and (0,2/3) for the texture coordinates as the wall vertices are traversed in a counterclockwise order.

As a slightly more complicated example, suppose you'd like to display a tin can with a label wrapped around it on the screen. To obtain the texture, you purchase a can, remove the label, and scan it in. Suppose the label is 4 units tall and 12 units around, which yields an aspect ratio of 3 to 1. Since textures must have aspect ratios of 2^n to 1, you can either simply not use the top third of the texture, or you can cut and paste the texture until it has the necessary aspect ratio. Suppose you decide not to use the top third. Now suppose the tin can is a cylinder approximated by thirty polygons of length 4 units (the height of the can) and width 12/30 (1/30 of the circumference of the can). You can use the following texture coordinates for each of the thirty approximating rectangles:

1: (0, 0), (1/30, 0), (1/30, 2/3), (0, 2/3)

2: (1/30, 0), (2/30, 0), (2/30, 2/3), (1/30, 2/3)

3: (2/30, 0), (3/30, 0), (3/30, 2/3), (2/30, 2/3)

...

30: (29/30, 0), (1, 0), (1, 2/3), (29/30, 2/3)

Only a few curved surfaces such as cones and cylinders can be mapped to a flat surface without geodesic distortion. Any other shape requires some distortion. In general, the higher the curvature of the surface, the more distortion of the texture is required.

If you don't care about texture distortion, it's often quite easy to find a reasonable mapping. For example, consider a sphere whose surface coordinates are given by $(\cos \theta \cos \phi, \cos \theta \sin \phi, \sin \theta)$, where $0 \leq \theta \leq 2\pi$, and $0 \leq \phi \leq \pi$. The θ - ϕ rectangle can be mapped directly to a rectangular texture map, but the closer you get to the poles, the more distorted the texture is. The entire top edge of the texture map is mapped to the north pole, and the entire bottom edge to the south pole. For other surfaces, such as that of a torus (doughnut) with a large hole, the natural surface coordinates map to the texture coordinates in a way that produces only a little distortion, so it might be suitable for many applications. Figure 9-6 shows two tori, one with a small hole (and therefore a lot of distortion near the center) and one with a large hole (and only a little distortion).

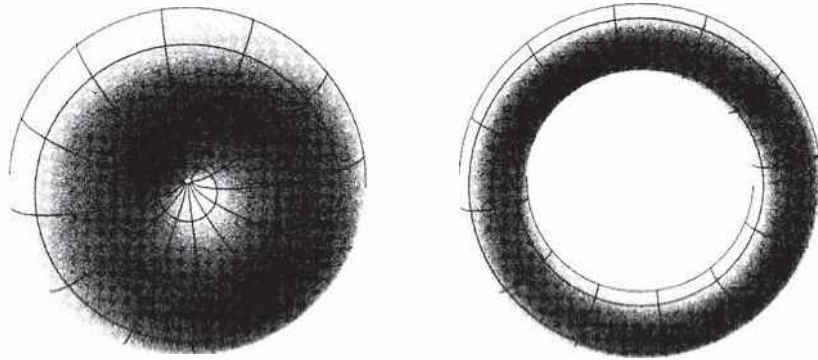


Figure 9-6 Texture-Map Distortion

If you're texturing spline surfaces generated with evaluators (see Chapter 12), the u and v parameters for the surface can sometimes be used as texture coordinates. In general, however, there's a large artistic component to successfully mapping textures to polygonal approximations of curved surfaces.

Repeating and Clamping Textures

You can assign texture coordinates outside the range $[0, 1]$ and have them either clamp or repeat in the texture map. With repeating textures, if you have a large plane with texture coordinates running from 0.0 to 10.0 in both directions, for example, you'll get 100 copies of the texture tiled together on the screen. During repeating, the integer part of texture coordinates is ignored, and copies of the texture map tile the surface. For most applications where the texture is to be repeated, the texels at the top of the texture should match those at the bottom, and similarly for the left and right edges.

The other possibility is to clamp the texture coordinates: Any values greater than 1.0 are set to 1.0, and any values less than 0.0 are set to 0.0. Clamping is useful for applications where you want a single copy of the texture to appear on a large surface. If the surface-texture coordinates range from 0.0 to 10.0 in both directions, one copy of the texture appears in the lower corner of the surface. If you've chosen `GL_LINEAR` as the filtering method

(see “Filtering” on page 344), an equally weighted combination of the border color and the texture color is used, as follows.

- When repeating, the 2×2 array wraps to the opposite edge of the texture. Thus, texels on the right edge are averaged with those on the left, and top and bottom texels are also averaged.
- If there is a border, then the texel from the border is used in the weighting. Otherwise, `GL_TEXTURE_BORDER_COLOR` is used. (If you’ve chosen `GL_NEAREST` as the filtering method, the border color is completely ignored.)

Note that if you are using clamping, you can avoid having the rest of the surface affected by the texture. To do this, use alpha values of 0 for the edges (or borders, if they are specified) of the texture. The decal texture function directly uses the texture’s alpha value in its calculations. If you are using one of the other texture functions, you may also need to enable blending with good source and destination factors. (See “Blending” on page 214.)

To see the effects of wrapping, you must have texture coordinates that venture beyond [0.0, 1.0]. Start with Example 9-1, and modify the texture coordinates for the squares by mapping the texture coordinates from 0.0 to 3.0 as follows:

```
glBegin(GL_QUADS);
  glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
  glTexCoord2f(0.0, 3.0); glVertex3f(-2.0, 1.0, 0.0);
  glTexCoord2f(3.0, 3.0); glVertex3f(0.0, 1.0, 0.0);
  glTexCoord2f(3.0, 0.0); glVertex3f(0.0, -1.0, 0.0);

  glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
  glTexCoord2f(0.0, 3.0); glVertex3f(1.0, 1.0, 0.0);
  glTexCoord2f(3.0, 3.0); glVertex3f(2.41421, 1.0, -1.41421);
  glTexCoord2f(3.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
glEnd();
```

With `GL_REPEAT` wrapping, the result is as shown in Figure 9-7.



Figure 9-7 Repeating a Texture

In this case, the texture is repeated in both the *s* and *t* directions, since the following calls are made to `glTexParameter*()`:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

If `GL_CLAMP` is used instead of `GL_REPEAT` for each direction, you see something similar to Figure 9-8.



Figure 9-8 Clamping a Texture

You can also clamp in one direction and repeat in the other, as shown in Figure 9-9.



Figure 9-9 Repeating and Clamping a Texture

You've now seen all the possible arguments for `glTexParameter*()`, which is summarized here.

```
void glTexParameter{if}(GLenum target, GLenum pname, TYPE param);
void glTexParameter{if}v(GLenum target, GLenum pname,
    TYPE *param);
```

Sets various parameters that control how a texture is treated as it's applied to a fragment or stored in a texture object. The *target* parameter is either `GL_TEXTURE_2D` or `GL_TEXTURE_1D` to indicate a two- or one-dimensional texture. The possible values for *pname* and *param* are shown in Table 9-4. You can use the vector version of the command to supply an array of values for `GL_TEXTURE_BORDER_COLOR`, or you can supply individual values for other parameters using the nonvector version. If these values are supplied as integers, they're converted to floating-point according to Table 4-1 on page 164; they're also clamped to the range [0,1].

Parameter	Values
<code>GL_TEXTURE_WRAP_S</code>	<code>GL_CLAMP</code> , <code>GL_REPEAT</code>
<code>GL_TEXTURE_WRAP_T</code>	<code>GL_CLAMP</code> , <code>GL_REPEAT</code>
<code>GL_TEXTURE_MAG_FILTER</code>	<code>GL_NEAREST</code> , <code>GL_LINEAR</code>
<code>GL_TEXTURE_MIN_FILTER</code>	<code>GL_NEAREST</code> , <code>GL_LINEAR</code> , <code>GL_NEAREST_MIPMAP_NEAREST</code> , <code>GL_NEAREST_MIPMAP_LINEAR</code> , <code>GL_LINEAR_MIPMAP_NEAREST</code> , <code>GL_LINEAR_MIPMAP_LINEAR</code>
<code>GL_TEXTURE_BORDER_COLOR</code>	any four values in [0.0, 1.0]
<code>GL_TEXTURE_PRIORITY</code>	[0.0, 1.0] for the current texture object

Table 9-4 `glTexParameter*()` Parameters

Try This

Figure 9-8 and Figure 9-9 are drawn using `GL_NEAREST` for the minification and magnification filter. What happens if you change the filter values to `GL_LINEAR`? Why?



Automatic Texture-Coordinate Generation

You can use texture mapping to make contours on your models or to simulate the reflections from an arbitrary environment on a shiny model. To achieve these effects, let OpenGL automatically generate the texture coordinates for you, rather than explicitly assigning them with `glTexCoord*()`. To generate texture coordinates automatically, use the command `glTexGen()`.

```
void glTexGen(efd)(GLenum coord, GLenum pname, TYPE param);  
void glTexGen(efd)v(GLenum coord, GLenum pname, TYPE *param);
```

Specifies the functions for automatically generating texture coordinates. The first parameter, *coord*, must be `GL_S`, `GL_T`, `GL_R`, or `GL_Q` to indicate whether texture coordinate *s*, *t*, *r*, or *q* is to be generated. The *pname* parameter is `GL_TEXTURE_GEN_MODE`, `GL_OBJECT_PLANE`, or `GL_EYE_PLANE`. If it's `GL_TEXTURE_GEN_MODE`, *param* is an integer (or, in the vector version of the command, points to an integer) that's either `GL_OBJECT_LINEAR`, `GL_EYE_LINEAR`, or `GL_SPHERE_MAP`. These symbolic constants determine which function is used to generate the texture coordinate. With either of the other possible values for *pname*, *param* is a pointer to an array of values (for the vector version) specifying parameters for the texture-generation function.

The different methods of texture-coordinate generation have different uses. Specifying the reference plane in object coordinates is best for when a texture image remains fixed to a moving object. Thus, `GL_OBJECT_LINEAR` would be used for putting a wood grain on a table top. Specifying the reference plane in eye coordinates (`GL_EYE_LINEAR`) is best for producing dynamic contour lines on moving objects. `GL_EYE_LINEAR` may be used by specialists in geosciences, who are drilling for oil or gas. As the drill goes deeper into the ground, the drill may be rendered with different colors to represent the layers of rock at increasing depths. `GL_SPHERE_MAP` is predominantly used for environment mapping. (See "Environment Mapping" on page 369.)

Creating Contours

When `GL_TEXTURE_GEN_MODE` and `GL_OBJECT_LINEAR` are specified, the generation function is a linear combination of the object coordinates of the vertex (x_0, y_0, z_0, w_0) :

generated coordinate = $p_1x_0 + p_2y_0 + p_3z_0 + p_4w_0$

The p_1, \dots, p_4 values are supplied as the *param* argument to `glTexGen*v()`, with *pname* set to `GL_OBJECT_PLANE`. With p_1, \dots, p_4 correctly normalized, this function gives the distance from the vertex to a plane. For example, if $p_2 = p_3 = p_4 = 0$ and $p_1 = 1$, the function gives the distance between the vertex and the plane $x = 0$. The distance is positive on one side of the plane, negative on the other, and zero if the vertex lies on the plane.

Initially in Example 9-6, equally spaced contour lines are drawn on a teapot; the lines indicate the distance from the plane $x = 0$. The coefficients for the plane $x = 0$ are in this array:

```
static GLfloat xequalzero[] = {1.0, 0.0, 0.0, 0.0};
```

Since only one property is being shown (the distance from the plane), a one-dimensional texture map suffices. The texture map is a constant green color, except that at equally spaced intervals it includes a red mark. Since the teapot is sitting on the x - y plane, the contours are all perpendicular to its base. Plate 18a shows the picture drawn by the program.

In the same example, pressing the 's' key changes the parameters of the reference plane to

```
static GLfloat slanted[] = {1.0, 1.0, 1.0, 0.0};
```

the contour stripes are parallel to the plane $x + y + z = 0$, slicing across the teapot at an angle, as shown in Plate 18b. To restore the reference plane to its initial value, $x = 0$, press the 'x' key.

Example 9-6 Automatic Texture-Coordinate Generation: `texgen.c`

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>

#define stripeImageWidth 32
GLubyte stripeImage[4*stripeImageWidth];
```



```

static GLuint texName;

void makeStripeImage(void)
{
    int j;

    for (j = 0; j < stripeImageWidth; j++) {
        stripeImage[4*j] = (GLubyte) ((j<=4) ? 255 : 0);
        stripeImage[4*j+1] = (GLubyte) ((j>4) ? 255 : 0);
        stripeImage[4*j+2] = (GLubyte) 0;
        stripeImage[4*j+3] = (GLubyte) 255;
    }
}

/* planes for texture coordinate generation */
static GLfloat xequalzero[] = {1.0, 0.0, 0.0, 0.0};
static GLfloat slanted[] = {1.0, 1.0, 1.0, 0.0};
static GLfloat *currentCoeff;
static GLenum currentPlane;
static GLint currentGenMode;

void init(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_SMOOTH);

    makeStripeImage();
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    glGenTextures(1, &texName);
    glBindTexture(GL_TEXTURE_1D, texName);
    glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR);
    glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR);
    glTexImage1D(GL_TEXTURE_1D, 0, GL_RGBA, stripeImageWidth, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, stripeImage);

    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    currentCoeff = xequalzero;
    currentGenMode = GL_OBJECT_LINEAR;
    currentPlane = GL_OBJECT_PLANE;
    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
    glTexGenfv(GL_S, currentPlane, currentCoeff);
}

```

```

glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_1D);
glEnable(GL_CULL_FACE);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_AUTO_NORMAL);
glEnable(GL_NORMALIZE);
glFrontFace(GL_CW);
glCullFace(GL_BACK);
glMaterialf (GL_FRONT, GL_SHININESS, 64.0);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix ();
    glRotatef(45.0, 0.0, 0.0, 1.0);
    glBindTexture(GL_TEXTURE_1D, texName);
    glutSolidTeapot(2.0);
    glPopMatrix ();
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho (-3.5, 3.5, -3.5*(GLfloat)h/(GLfloat)w,
                3.5*(GLfloat)h/(GLfloat)w, -3.5, 3.5);
    else
        glOrtho (-3.5*(GLfloat)w/(GLfloat)h,
                3.5*(GLfloat)w/(GLfloat)h, -3.5, 3.5, -3.5, 3.5);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void keyboard (unsigned char key, int x, int y)
{
    switch (key) {
        case 'e':
        case 'E':
            currentGenMode = GL_EYE_LINEAR;
            currentPlane = GL_EYE_PLANE;
            glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
    }
}

```

```

        glTexGenfv(GL_S, currentPlane, currentCoeff);
        glutPostRedisplay();
        break;
    case 'o':
    case 'O':
        currentGenMode = GL_OBJECT_LINEAR;
        currentPlane = GL_OBJECT_PLANE;
        glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
        glTexGenfv(GL_S, currentPlane, currentCoeff);
        glutPostRedisplay();
        break;
    case 's':
    case 'S':
        currentCoeff = slanted;
        glTexGenfv(GL_S, currentPlane, currentCoeff);
        glutPostRedisplay();
        break;
    case 'x':
    case 'X':
        currentCoeff = xequalzero;
        glTexGenfv(GL_S, currentPlane, currentCoeff);
        glutPostRedisplay();
        break;
    case 27:
        exit(0);
        break;
    default:
        break;
}
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(256, 256);
    glutInitWindowPosition(100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}

```

You enable texture-coordinate generation for the *s* coordinate by passing `GL_TEXTURE_GEN_S` to `glEnable()`. To generate other coordinates, enable them with `GL_TEXTURE_GEN_T`, `GL_TEXTURE_GEN_R`, or `GL_TEXTURE_GEN_Q`. Use `glDisable()` with the appropriate constant to disable coordinate generation. Also note the use of `GL_REPEAT` to cause the contour lines to be repeated across the teapot.

The `GL_OBJECT_LINEAR` function calculates the texture coordinates in the model's coordinate system. Initially in Example 9-6, the `GL_OBJECT_LINEAR` function is used, so the contour lines remain perpendicular to the base of the teapot, no matter how the teapot is rotated or viewed. However, if you press the 'e' key, the texture generation mode is changed from `GL_OBJECT_LINEAR` to `GL_EYE_LINEAR`, and the contour lines are calculated relative to the eye coordinate system. (Pressing the 'o' key restores `GL_OBJECT_LINEAR` as the texture generation mode.) If the reference plane is $x = 0$, the result is a teapot with red stripes parallel to the y - z plane from the eye's point of view, as shown in Plate 18c. Mathematically, you are multiplying the vector $(p_1 p_2 p_3 p_4)$ by the inverse of the modelview matrix to obtain the values used to calculate the distance to the plane. The texture coordinate is generated with the following function:

generated coordinate = $p_1' x_e + p_2' y_e + p_3' z_e + p_4' w_e$

where $(p_1' p_2' p_3' p_4') = (p_1 p_2 p_3 p_4)M^{-1}$

In this case, (x_e, y_e, z_e, w_e) are the eye coordinates of the vertex, and p_1, \dots, p_4 are supplied as the *param* argument to `glTexGen*()` with *pname* set to `GL_EYE_PLANE`. The primed values are calculated only at the time they're specified so this operation isn't as computationally expensive as it looks.

In all these examples, a single texture coordinate is used to generate contours. The *s* and *t* texture coordinates can be generated independently, however, to indicate the distances to two different planes. With a properly constructed two-dimensional texture map, the resulting two sets of contours can be viewed simultaneously. For an added level of complexity, you can calculate the *s* coordinate using `GL_OBJECT_LINEAR` and the *t* coordinate using `GL_EYE_LINEAR`.

Environment Mapping

The goal of environment mapping is to render an object as if it were perfectly reflective, so that the colors on its surface are those reflected to

the eye from its surroundings. In other words, if you look at a perfectly polished, perfectly reflective silver object in a room, you see the walls, floor, and other objects in the room reflected off the object. (A classic example of using environment mapping is the evil, morphing cyborg in the film *Terminator 2*.) The objects whose reflections you see depend on the position of your eye and on the position and surface angles of the silver object. To perform environment mapping, all you have to do is create an appropriate texture map and then have OpenGL generate the texture coordinates for you.

Environment mapping is an approximation based on the assumption that the items in the environment are far away compared to the surfaces of the shiny object—that is, it's a small object in a large room. With this assumption, to find the color of a point on the surface, take the ray from the eye to the surface, and reflect the ray off the surface. The direction of the reflected ray completely determines the color to be painted there. Encoding a color for each direction on a flat texture map is equivalent to putting a polished perfect sphere in the middle of the environment and taking a picture of it with a camera that has a lens with a very long focal length placed far away. Mathematically, the lens has an infinite focal length and the camera is infinitely far away. The encoding therefore covers a circular region of the texture map, tangent to the top, bottom, left, and right edges of the map. The texture values outside the circle make no difference, as they are never accessed in environment mapping.

To make a perfectly correct environment texture map, you need to obtain a large silvered sphere, take a photograph of it in some environment with a camera located an infinite distance away and with a lens that has an infinite focal length, and scan in the photograph. To approximate this result, you can use a scanned-in photograph of an environment taken with an extremely wide-angle (or fish-eye) lens. Plate 21 shows a photograph taken with such a lens and the results when that image is used as an environment map.

Once you've created a texture designed for environment mapping, you need to invoke OpenGL's environment-mapping algorithm. This algorithm finds the point on the surface of the sphere with the same tangent surface as the point on the object being rendered, and it paints the object's point with the color visible on the sphere at the corresponding point.

To automatically generate the texture coordinates to support environment mapping, use this code in your program:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);  
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
```

```
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

The `GL_SPHERE_MAP` constant creates the proper texture coordinates for the environment mapping. As shown, you need to specify it for both the *s* and *t* directions. However, you don't have to specify any parameters for the texture-coordinate generation function.

The `GL_SPHERE_MAP` texture function generates texture coordinates using the following mathematical steps.

1. *u* is the unit vector pointing from the origin to the vertex (in eye coordinates).
2. *n'* is the current normal vector, after transformation to eye coordinates.
3. *r* is the reflection vector, $(r_x \ r_y \ r_z)^T$, which is calculated by $\mathbf{u} - 2\mathbf{n}'\mathbf{n}'^T\mathbf{u}$.
4. Then an interim value, *m*, is calculated by $m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$.
5. Finally, the *s* and *t* texture coordinates are calculated by $s = r_x/m + \frac{1}{2}$ and $t = r_y/m + \frac{1}{2}$.

Advanced Features

Advanced

This section describes how to manipulate the texture matrix stack and how to use the *q* coordinate. Both techniques are considered advanced, since you don't need them for many applications of texture mapping.



The Texture Matrix Stack

Just as your model coordinates are transformed by a matrix before being rendered, texture coordinates are multiplied by a 4x4 matrix before any texture mapping occurs. By default, the texture matrix is the identity, so the texture coordinates you explicitly assign or those that are automatically generated remain unchanged. By modifying the texture matrix while redrawing an object, however, you can make the texture slide over the surface, rotate around it, stretch and shrink, or any combination of the three. In fact, since the texture matrix is a completely general 4x4 matrix, effects such as perspective can be achieved.

When the four texture coordinates (s, t, r, q) are multiplied by the texture matrix, the resulting vector ($s' t' r' q'$) is interpreted as homogeneous texture coordinates. In other words, the texture map is indexed by s'/q' and t'/q' . (Remember that r'/q' is ignored in standard OpenGL, but may be used by implementations that support a 3D texture extension.) The texture matrix is actually the top matrix on a stack, which must have a stack depth of at least two matrices. All the standard matrix-manipulation commands such as `glPushMatrix()`, `glPopMatrix()`, `glMultMatrix()`, and `glRotate*`() can be applied to the texture matrix. To modify the current texture matrix, you need to set the matrix mode to `GL_TEXTURE`, as follows:

```
glMatrixMode(GL_TEXTURE); /* enter texture matrix mode */
glRotated(...);
/* ... other matrix manipulations ... */
glMatrixMode(GL_MODELVIEW); /* back to modelview mode */
```

The q Coordinate

The mathematics of the q coordinate in a general four-dimensional texture coordinate is as described in the previous section. You can make use of q in cases where more than one projection or perspective transformation is needed. For example, suppose you want to model a spotlight that has some nonuniform pattern—brighter in the center, perhaps, or noncircular, because of flaps or lenses that modify the shape of the beam. You can emulate shining such a light on a flat surface by making a texture map that corresponds to the shape and intensity of a light, and then projecting it on the surface in question using projection transformations. Projecting the cone of light onto surfaces in the scene requires a perspective transformation ($q \neq 1$), since the lights might shine on surfaces that aren't perpendicular to them. A second perspective transformation occurs because the viewer sees the scene from a different (but perspective) point of view. (See Plate 27 for an example, and see “Fast Shadows and Lighting Effects Using Texture Mapping” by Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli, SIGGRAPH 1992 Proceedings, (*Computer Graphics*, 26:2, July 1992, p. 249–252) for more details.)

Another example might arise if the texture map to be applied comes from a photograph that itself was taken in perspective. As with spotlights, the final view depends on the combination of two perspective transformations.

The Framebuffer



Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Understand what buffers make up the framebuffer and how they're used
- Clear selected buffers and enable them for writing
- Control the parameters of the scissoring, alpha, stencil, and depth-buffer tests that are applied to pixels
- Perform dithering and logical operations
- Use the accumulation buffer for such purposes as scene antialiasing

An important goal of almost every graphics program is to draw pictures on the screen. The screen is composed of a rectangular array of pixels, each capable of displaying a tiny square of color at that point in the image. After the rasterization stage (including texturing and fog), the data are not yet pixels, but are fragments. Each fragment has coordinate data which corresponds to a pixel, as well as color and depth values. Then each fragment undergoes a series of tests and operations, some of which have been previously described (See "Blending" in Chapter 6) and others that are discussed in this chapter.

If the tests and operations are survived, the fragment values are ready to become pixels. To draw these pixels, you need to know what color they are, which is the information that's stored in the color buffer. Whenever data is stored uniformly for each pixel, such storage for all the pixels is called a *buffer*. Different buffers might contain different amounts of data per pixel, but within a given buffer, each pixel is assigned the same amount of data. A buffer that stores a single bit of information about pixels is called a *bitplane*.

As shown in Figure 10-1, the lower-left pixel in an OpenGL window is pixel (0, 0), corresponding to the window coordinates of the lower-left corner of the 1x1 region occupied by this pixel. In general, pixel (x, y) fills the region bounded by x on the left, x+1 on the right, y on the bottom, and y+1 on the top.

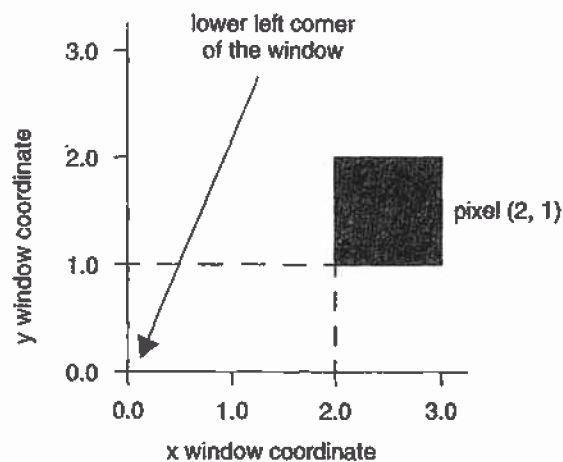


Figure 10-1 Region Occupied by a Pixel

As an example of a buffer, let's look more closely at the color buffer, which holds the color information that's to be displayed on the screen. Assume

that the screen is 1280 pixels wide and 1024 pixels high and that it's a full 24-bit color screen—in other words, there are 2^{24} (or 16,777,216) different colors that can be displayed. Since 24 bits translates to 3 bytes (8 bits/byte), the color buffer in this example has to store at least 3 bytes of data for each of the 1,310,720 (1280*1024) pixels on the screen. A particular hardware system might have more or fewer pixels on the physical screen as well as more or less color data per pixel. Any particular color buffer, however, has the same amount of data saved for each pixel on the screen.

The color buffer is only one of several buffers that hold information about a pixel. For example, in “A Hidden-Surface Removal Survival Kit” on page 171, you learned that the depth buffer holds depth information for each pixel. The color buffer itself can consist of several subbuffers. The *framebuffer* on a system comprises all of these buffers. With the exception of the color buffer(s), you don't view these other buffers directly; instead, you use them to perform such tasks as hidden-surface elimination, antialiasing of an entire scene, stenciling, drawing smooth motion, and other operations.

This chapter describes all the buffers that can exist in an OpenGL implementation and how they're used. It also discusses the series of tests and pixel operations that are performed before any data is written to the viewable color buffer. Finally, it explains how to use the accumulation buffer, which is used to accumulate images that are drawn into the color buffer. This chapter has the following major sections.

- “Buffers and Their Uses” on page 376 describes the possible buffers, what they're for, and how to clear them and enable them for writing.
- “Testing and Operating on Fragments” on page 382 explains the scissoring, alpha, stencil, and depth-buffer tests that occur after a pixel's position and color have been calculated but before this information is drawn on the screen. Several operations—blending, dithering, and logical operations—can also be performed before a fragment updates the screen.
- “The Accumulation Buffer” on page 394 describes how to perform several advanced techniques using the accumulation buffer. These techniques include antialiasing an entire scene, using motion blur, and simulating photographic depth of field.

Buffers and Their Uses

An OpenGL system can manipulate the following buffers:

- Color buffers: front-left, front-right, back-left, back-right, and any number of auxiliary color buffers
- Depth buffer
- Stencil buffer
- Accumulation buffer

Your particular OpenGL implementation determines which buffers are available and how many bits per pixel each holds. Additionally, you can have multiple visuals, or window types, that have different buffers available. Table 10-1 lists the parameters to use with `glGetIntegerv()` to query your OpenGL system about per-pixel buffer storage for a particular visual.

Note: If you're using the X Window System, you're guaranteed, at a minimum, to have a visual with one color buffer for use in RGBA mode with associated stencil, depth, and accumulation buffers that have color components of nonzero size. Also, if your X Window System implementation supports a Pseudo-Color visual, you are also guaranteed to have one OpenGL visual that has a color buffer for use in color-index mode with associated depth and stencil buffers. You'll probably want to use `glXGetConfig()` to query your visuals; see Appendix C and the *OpenGL Reference Manual* for more information about this routine.

Parameter	Meaning
GL_RED_BITS, GL_GREEN_BITS, GL_BLUE_BITS, GL_ALPHA_BITS	Number of bits per R, G, B, or A component in the color buffers
GL_INDEX_BITS	Number of bits per index in the color buffers
GL_DEPTH_BITS	Number of bits per pixel in the depth buffer
GL_STENCIL_BITS	Number of bits per pixel in the stencil buffer
GL_ACCUM_RED_BITS, GL_ACCUM_GREEN_BITS, GL_ACCUM_BLUE_BITS, GL_ACCUM_ALPHA_BITS	Number of bits per R, G, B, or A component in the accumulation buffer

Table 10-1 Query Parameters for Per-Pixel Buffer Storage

Color Buffers

The color buffers are the ones to which you usually draw. They contain either color-index or RGB color data and may also contain alpha values. An OpenGL implementation that supports stereoscopic viewing has left and right color buffers for the left and right stereo images. If stereo isn't supported, only the left buffers are used. Similarly, double-buffered systems have front and back buffers, and a single-buffered system has the front buffers only. Every OpenGL implementation must provide a front-left color buffer.

Optional, nondisplayable auxiliary color buffers may also be supported. OpenGL doesn't specify any particular uses for these buffers, so you can define and use them however you please. For example, you might use them for saving an image that you use repeatedly. Then rather than redrawing the image, you can just copy it from an auxiliary buffer into the usual color buffers. (See the description of `glCopyPixels()` in "Reading, Writing, and Copying Pixel Data" on page 290 for more information about how to do this.)

You can use `GL_STEREO` or `GL_DOUBLEBUFFER` with `glGetBooleanv()` to find out if your system supports stereo (that is, has left and right buffers) or double-buffering (has front and back buffers). To find out how many, if any, auxiliary buffers are present, use `glGetIntegerv()` with `GL_AUX_BUFFERS`.

Depth Buffer

The depth buffer stores a depth value for each pixel. As described in "A Hidden-Surface Removal Survival Kit" on page 171, depth is usually measured in terms of distance to the eye, so pixels with larger depth-buffer values are overwritten by pixels with smaller values. This is just a useful convention, however, and the depth buffer's behavior can be modified as described in "Depth Test" on page 391. The depth buffer is sometimes called the *z buffer* (the *z* comes from the fact that *x* and *y* values measure horizontal and vertical displacement on the screen, and the *z* value measures distance perpendicular to the screen).

Stencil Buffer

One use for the stencil buffer is to restrict drawing to certain portions of the screen, just as a cardboard stencil can be used with a can of spray paint to make fairly precise painted images. For example, if you want to draw an image as it would appear through an odd-shaped windshield, you can store an image of the windshield's shape in the stencil buffer, and then draw the

entire scene. The stencil buffer prevents anything that wouldn't be visible through the windshield from being drawn. Thus, if your application is a driving simulation, you can draw all the instruments and other items inside the automobile once, and as the car moves, only the outside scene need be updated.

Accumulation Buffer

The accumulation buffer holds RGBA color data just like the color buffers do in RGBA mode. (The results of using the accumulation buffer in color-index mode are undefined.) It's typically used for accumulating a series of images into a final, composite image. With this method, you can perform operations like scene antialiasing by supersampling an image and then averaging the samples to produce the values that are finally painted into the pixels of the color buffers. You don't draw directly into the accumulation buffer; accumulation operations are always performed in rectangular blocks, which are usually transfers of data to or from a color buffer.

Clearing Buffers

In graphics programs, clearing the screen (or any of the buffers) is typically one of the most expensive operations you can perform—on a 1280×1024 monitor, it requires touching well over a million pixels. For simple graphics applications, the clear operation can take more time than the rest of the drawing. If you need to clear not only the color buffer but also the depth and stencil buffers, the clear operation can be three times as expensive.

To address this problem, some machines have hardware that can clear more than one buffer at once. The OpenGL clearing commands are structured to take advantage of architectures like this. First, you specify the values to be written into each buffer to be cleared. Then you issue a single command to perform the clear operation, passing in a list of all the buffers to be cleared. If the hardware is capable of simultaneous clears, they all occur at once; otherwise, each buffer is cleared sequentially.

The following commands set the clearing values for each buffer.

```
void glClearColor(GLclampf red, GLclampf green, GLclampf blue,  
                 GLclampf alpha);  
void glClearIndex(GLfloat index);  
void glClearDepth(GLclampd depth);  
void glClearStencil(GLint s);  
void glClearAccum(GLfloat red, GLfloat green, GLfloat blue,  
                 GLfloat alpha);
```

Specifies the current clearing values for the color buffer (in RGBA mode), the color buffer (in color-index mode), the depth buffer, the stencil buffer, and the accumulation buffer. The `GLclampf` and `GLclampd` types (clamped `GLfloat` and clamped `GLdouble`) are clamped to be between 0.0 and 1.0. The default depth-clearing value is -1.0; all the other default clearing values are 0. The values set with the clear commands remain in effect until they're changed by another call to the same command.

After you've selected your clearing values and you're ready to clear the buffers, use `glClear()`.

```
void glClear(GLbitfield mask);
```

Clears the specified buffers. The value of *mask* is the bitwise logical OR of some combination of `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`, `GL_STENCIL_BUFFER_BIT`, and `GL_ACCUM_BUFFER_BIT` to identify which buffers are to be cleared. `GL_COLOR_BUFFER_BIT` clears either the RGBA color or the color-index buffer, depending on the mode of the system at the time. When you clear the color or color-index buffer, all the color buffers that are enabled for writing (see the next section) are cleared. The pixel ownership test, scissor test, and dithering, if enabled, are applied to the clearing operation. Masking operations, such as `glColorMask()` and `glIndexMask()`, are also effective. The alpha test, stencil test, and depth test do not affect the operation of `glClear()`.

Selecting Color Buffers for Writing and Reading

The results of a drawing or reading operation can go into or come from any of the color buffers: front, back, front-left, back-left, front-right, back-right, or any of the auxiliary buffers. You can choose an individual buffer to be the drawing or reading target. For drawing, you can also set the target to draw into more than one buffer at the same time. You use `glDrawBuffer()` to select the buffers to be written and `glReadBuffer()` to select the buffer as

the source for `glReadPixels()`, `glCopyPixels()`, `glCopyTexImage*()`, and `glCopyTexSubImage*()`.

If you are using double-buffering, you usually want to draw only in the back buffer (and swap the buffers when you're finished drawing). In some situations, you might want to treat a double-buffered window as though it were single-buffered by calling `glDrawBuffer()` to enable you to draw to both front and back buffers at the same time.

`glDrawBuffer()` is also used to select buffers to render stereo images (`GL*LEFT` and `GL*RIGHT`) and to render into auxiliary buffers (`GL_AUXi`).

void `glDrawBuffer(GLenum mode)`;

Selects the color buffers enabled for writing or clearing. Disables buffers enabled by previous calls to `glDrawBuffer()`. More than one buffer may be enabled at one time. The value of *mode* can be one of the following:

<code>GL_FRONT</code>	<code>GL_FRONT_LEFT</code>	<code>GL_AUX<i>i</i></code>
<code>GL_BACK</code>	<code>GL_FRONT_RIGHT</code>	<code>GL_FRONT_AND_BACK</code>
<code>GL_LEFT</code>	<code>GL_BACK_LEFT</code>	<code>GL_NONE</code>
<code>GL_RIGHT</code>	<code>GL_BACK_RIGHT</code>	

Arguments that omit `LEFT` or `RIGHT` refer to both the left and right buffers; similarly, arguments that omit `FRONT` or `BACK` refer to both. The *i* in `GL_AUXi` is a digit identifying a particular auxiliary buffer.

By default, *mode* is `GL_FRONT` for single-buffered contexts and `GL_BACK` for double-buffered contexts.

Note: You can enable drawing to nonexistent buffers as long as you enable drawing to at least one buffer that does exist. If none of the specified buffers exist, an error results.

void `glReadBuffer(GLenum mode)`;

Selects the color buffer enabled as the source for reading pixels for subsequent calls to `glReadPixels()`, `glCopyPixels()`, `glCopyTexImage*()`, and `glCopyTexSubImage*()`. Disables buffers enabled by previous calls to `glReadBuffer()`. The value of *mode* can be one of the following:

<code>GL_FRONT</code>	<code>GL_FRONT_LEFT</code>	<code>GL_AUX<i>i</i></code>
<code>GL_BACK</code>	<code>GL_FRONT_RIGHT</code>	

GL_LEFT GL_BACK_LEFT
GL_RIGHT GL_BACK_RIGHT

By default, *mode* is GL_FRONT for single-buffered contexts and GL_BACK for double-buffered contexts.

Note: You must enable reading from a buffer that does exist or an error results.

Masking Buffers

Before OpenGL writes data into the enabled color, depth, or stencil buffers, a masking operation is applied to the data, as specified with one of the following commands. A bitwise logical AND is performed with each mask and the corresponding data to be written.

```
void glIndexMask(GLuint mask);  
void glColorMask(GLboolean red, GLboolean green, GLboolean blue,  
                  GLboolean alpha);  
void glDepthMask(GLboolean flag);  
void glStencilMask(GLuint mask);
```

Sets the masks used to control writing into the indicated buffers. The mask set by `glIndexMask()` applies only in color-index mode. If a 1 appears in *mask*, the corresponding bit in the color-index buffer is written; where a 0 appears, the bit isn't written. Similarly, `glColorMask()` affects drawing in RGBA mode only. The *red*, *green*, *blue*, and *alpha* values control whether the corresponding component is written. (GL_TRUE means it is written.) If *flag* is GL_TRUE for `glDepthMask()`, the depth buffer is enabled for writing; otherwise, it's disabled. The mask for `glStencilMask()` is used for stencil data in the same way as the mask is used for color-index data in `glIndexMask()`. The default values of all the GLboolean masks are GL_TRUE, and the default values for the two GLuint masks are all 1's.

You can do plenty of tricks with color masking in color-index mode. For example, you can use each bit in the index as a different layer and set up interactions between arbitrary layers with appropriate settings of the color map. You can create overlays and underlays, and do so-called color-map animations. (See Chapter 14 for examples of using color masking.) Masking in RGBA mode is useful less often, but you can use it for loading separate image files into the red, green, and blue bitplanes, for example.

You've seen one use for disabling the depth buffer in "Three-Dimensional Blending with the Depth Buffer" on page 222. Disabling the depth buffer for writing can also be useful if a common background is desired for a series of frames, and you want to add some features that may be obscured by parts of the background. For example, suppose your background is a forest, and you would like to draw repeated frames with the same trees, but with objects moving among them. After the trees are drawn with their depths recorded in the depth buffer, then the image of the trees is saved, and the new items are drawn with the depth buffer disabled for writing. As long as the new items don't overlap each other, the picture is correct. To draw the next frame, restore the image of the trees and continue. You don't need to restore the values in the depth buffer. This trick is most useful if the background is extremely complex—so complex that it's much faster just to recopy the image into the color buffer than to recompute it from the geometry.

Masking the stencil buffer can allow you to use a multiple-bit stencil buffer to hold multiple stencils (one per bit). You might use this technique to perform capping as explained in "Stencil Test" on page 385 or to implement the Game of Life as described in "Life in the Stencil Buffer" on page 526.

Note: The mask specified by `glStencilMask()` controls which stencil bitplanes are written. This mask isn't related to the mask that's specified as the third parameter of `glStencilFunc()`, which specifies which bitplanes are considered by the stencil function.

Testing and Operating on Fragments

When you draw geometry, text, or images on the screen, OpenGL performs several calculations to rotate, translate, scale, determine the lighting, project the object(s) into perspective, figure out which pixels in the window are affected, and determine what colors those pixels should be drawn. Many of the earlier chapters in this book give some information about how to control these operations. After OpenGL determines that an individual fragment should be generated and what its color should be, several processing stages remain that control how and whether the fragment is drawn as a pixel into the framebuffer. For example, if it's outside a rectangular region or if it's farther from the viewpoint than the pixel that's already in the framebuffer, it isn't drawn. In another stage, the fragment's color is blended with the color of the pixel already in the framebuffer.

This section describes both the complete set of tests that a fragment must pass before it goes into the framebuffer and the possible final operations that can be performed on the fragment as it's written. The tests and operations occur in the following order; if a fragment is eliminated in an early test, none of the later tests or operations take place.

1. Scissor test
2. Alpha test
3. Stencil test
4. Depth test
5. Blending
6. Dithering
7. Logical operation

Each of these tests and operations is described in detail in the following sections.

Scissor Test

You can define a rectangular portion of your window and restrict drawing to take place within it by using the `glScissor()` command. If a fragment lies inside the rectangle, it passes the scissor test.

```
void glScissor(GLint x, GLint y, GLsizei width, GLsizei height);
```

Sets the location and size of the scissor rectangle (also known as the scissor box). The parameters define the lower-left corner (x, y), and the width and height of the rectangle. Pixels that lie inside the rectangle pass the scissor test. Scissoring is enabled and disabled by passing `GL_SCISSOR_TEST` to `glEnable()` and `glDisable()`. By default, the rectangle matches the size of the window and scissoring is disabled.

The scissor test is just a version of a stencil test using a rectangular region of the screen. It's fairly easy to create a blindingly fast hardware implementation of scissoring, while a given system might be much slower at stenciling—perhaps because the stenciling is performed in software.



Advanced

An advanced use of scissoring is performing nonlinear projection. First divide the window into a regular grid of subregions, specifying viewport and scissor parameters that limit rendering to one region at a time. Then project the entire scene to each region using a different projection matrix.

To determine whether scissoring is enabled and to obtain the values that define the scissor rectangle, you can use `GL_SCISSOR_TEST` with `glIsEnabled()` and `GL_SCISSOR_BOX` with `glGetIntegerv()`.

Alpha Test

In RGBA mode, the alpha test allows you to accept or reject a fragment based on its alpha value. The alpha test is enabled and disabled by passing `GL_ALPHA_TEST` to `glEnable()` and `glDisable()`. To determine whether the alpha test is enabled, use `GL_ALPHA_TEST` with `glIsEnabled()`.

If enabled, the test compares the incoming alpha value with a reference value. The fragment is accepted or rejected depending on the result of the comparison. Both the reference value and the comparison function are set with `glAlphaFunc()`. By default, the reference value is zero, the comparison function is `GL_ALWAYS`, and the alpha test is disabled. To obtain the alpha comparison function or reference value, use `GL_ALPHA_TEST_FUNC` or `GL_ALPHA_TEST_REF` with `glGetIntegerv()`.

```
void glAlphaFunc(GLenum func, GLclampf ref);
```

Sets the reference value and comparison function for the alpha test. The reference value *ref* is clamped to be between zero and one. The possible values for *func* and their meaning are listed in Table 10-2.

Parameter	Meaning
<code>GL_NEVER</code>	Never accept the fragment
<code>GL_ALWAYS</code>	Always accept the fragment
<code>GL_LESS</code>	Accept fragment if fragment alpha < reference alpha
<code>GL_LEQUAL</code>	Accept fragment if fragment alpha ≤ reference alpha
<code>GL_EQUAL</code>	Accept fragment if fragment alpha = reference alpha

Table 10-2 `glAlphaFunc()` Parameter Values

Parameter	Meaning
GL_EQUAL	Accept fragment if fragment alpha \geq reference alpha
GL_GREATER	Accept fragment if fragment alpha $>$ reference alpha
GL_NOTEQUAL	Accept fragment if fragment alpha \neq reference alpha

Table 10-2 glAlphaFunc() Parameter Values (continued)

One application for the alpha test is to implement a transparency algorithm. Render your entire scene twice, the first time accepting only fragments with alpha values of one, and the second time accepting fragments with alpha values that aren't equal to one. Turn the depth buffer on during both passes, but disable depth buffer writing during the second pass.

Another use might be to make decals with texture maps where you can see through certain parts of the decals. Set the alphas in the decals to 0.0 where you want to see through, set them to 1.0 otherwise, set the reference value to 0.5 (or anything between 0.0 and 1.0), and set the comparison function to GL_GREATER. The decal has see-through parts, and the values in the depth buffer aren't affected. This technique, called billboarding, is described in "Sample Uses of Blending" on page 217.

Stencil Test

The stencil test takes place only if there is a stencil buffer. (If there is no stencil buffer, the stencil test always passes.) Stenciling applies a test that compares a reference value with the value stored at a pixel in the stencil buffer. Depending on the result of the test, the value in the stencil buffer is modified. You can choose the particular comparison function used, the reference value, and the modification performed with the glStencilFunc() and glStencilOp() commands.

```
void glStencilFunc(GLenum func, GLint ref, GLuint mask);
```

Sets the comparison function (*func*), reference value (*ref*), and a mask (*mask*) for use with the stencil test. The reference value is compared to the value in the stencil buffer using the comparison function, but the comparison applies only to those bits where the corresponding bits of the mask are 1. The function can be GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GEQUAL, GL_GREATER, or

GL_NOTEQUAL. If it's GL_LESS, for example, then the fragment passes if *ref* is less than the value in the stencil buffer. If the stencil buffer contains *s* bitplanes, the low-order *s* bits of *mask* are bitwise ANDed with the value in the stencil buffer and with the reference value before the comparison is performed. The masked values are all interpreted as nonnegative values. The stencil test is enabled and disabled by passing GL_STENCIL_TEST to glEnable() and glDisable(). By default, *func* is GL_ALWAYS, *ref* is 0, *mask* is all 1's, and stenciling is disabled.

```
void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);
```

Specifies how the data in the stencil buffer is modified when a fragment passes or fails the stencil test. The three functions *fail*, *zfail*, and *zpass* can be GL_KEEP, GL_ZERO, GL_REPLACE, GL_INCR, GL_DECR, or GL_INVERT. They correspond to keeping the current value, replacing it with zero, replacing it with the reference value, incrementing it, decrementing it, and bitwise-inverting it. The result of the increment and decrement functions is clamped to lie between zero and the maximum unsigned integer value ($2^s - 1$ if the stencil buffer holds *s* bits). The *fail* function is applied if the fragment fails the stencil test; if it passes, then *zfail* is applied if the depth test fails and *zpass* if the depth test passes, or if no depth test is performed. (See "Depth Test" on page 391.) By default, all three stencil operations are GL_KEEP.

Stencil Queries

You can obtain the values for all six stencil-related parameters by using the query function glGetIntegerv() and one of the values shown in Table 10-3. You can also determine whether the stencil test is enabled by passing GL_STENCIL_TEST to glIsEnabled().

Query Value	Meaning
GL_STENCIL_FUNC	Stencil function
GL_STENCIL_REF	Stencil reference value
GL_STENCIL_VALUE_MASK	Stencil mask
GL_STENCIL_FAIL	Stencil fail action
GL_STENCIL_PASS_DEPTH_FAIL	Stencil pass and depth buffer fail action

Table 10-3 Query Values for the Stencil Test

Query Value	Meaning
GL_STENCIL_PASS_DEPTH_PASS	Stencil pass and depth buffer pass action

Table 10-3 Query Values for the Stencil Test (continued)

Stencil Examples

Probably the most typical use of the stencil test is to mask out an irregularly shaped region of the screen to prevent drawing from occurring within it (as in the windshield example in “Buffers and Their Uses” on page 376). To do this, fill the stencil mask with zeros, and then draw the desired shape in the stencil buffer with 1’s. You can’t draw geometry directly into the stencil buffer, but you can achieve the same result by drawing into the color buffer and choosing a suitable value for the *zpass* function (such as `GL_REPLACE`). (You can use `glDrawPixels()` to draw pixel data directly into the stencil buffer.) Whenever drawing occurs, a value is also written into the stencil buffer (in this case, the reference value). To prevent the stencil-buffer drawing from affecting the contents of the color buffer, set the color mask to zero (or `GL_FALSE`). You might also want to disable writing into the depth buffer.

After you’ve defined the stencil area, set the reference value to one, and the comparison function such that the fragment passes if the reference value is equal to the stencil-plane value. During drawing, don’t modify the contents of the stencil planes.

Example 10-1 demonstrates how to use the stencil test in this way. Two tori are drawn, with a diamond-shaped cutout in the center of the scene. Within the diamond-shaped stencil mask, a sphere is drawn. In this example, drawing into the stencil buffer takes place only when the window is redrawn, so the color buffer is cleared after the stencil mask has been created.

Example 10-1 Using the Stencil Test: `stencil.c`

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>

#define YELLOWMAT 1
#define BLUEMAT 2

void init (void)
```

```

{
    GLfloat yellow_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
    GLfloat yellow_specular[] = { 1.0, 1.0, 1.0, 1.0 };

    GLfloat blue_diffuse[] = { 0.1, 0.1, 0.7, 1.0 };
    GLfloat blue_specular[] = { 0.1, 1.0, 1.0, 1.0 };

    GLfloat position_one[] = { 1.0, 1.0, 1.0, 0.0 };

    glNewList(YELLOWMAT, GL_COMPILE);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, yellow_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, yellow_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 64.0);
    glEndList();

    glNewList(BLUEMAT, GL_COMPILE);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, blue_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, blue_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 45.0);
    glEndList();

    glLightfv(GL_LIGHT0, GL_POSITION, position_one);

    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHTING);
    glEnable(GL_DEPTH_TEST);

    glClearStencil(0x0);
    glEnable(GL_STENCIL_TEST);
}

/* Draw a sphere in a diamond-shaped section in the
 * middle of a window with 2 tori.
 */
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    /* draw blue sphere where the stencil is 1 */
    glStencilFunc (GL_EQUAL, 0x1, 0x1);
    glStencilOp (GL_KEEP, GL_KEEP, GL_KEEP);
    glCallList (BLUEMAT);
    glutSolidSphere (0.5, 15, 15);

    /* draw the tori where the stencil is not 1 */
    glStencilFunc (GL_NOTEQUAL, 0x1, 0x1);
    glPushMatrix();

```

```

    glRotatef (45.0, 0.0, 0.0, 1.0);
    glRotatef (45.0, 0.0, 1.0, 0.0);
    glCallList (YELLOWMAT);
    glutSolidTorus (0.275, 0.85, 15, 15);
    glPushMatrix();
        glRotatef (90.0, 1.0, 0.0, 0.0);
        glutSolidTorus (0.275, 0.85, 15, 15);
    glPopMatrix();
glPopMatrix();
}

/* Whenever the window is reshaped, redefine the
 * coordinate system and redraw the stencil area.
 */
void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);

    /* create a diamond shaped stencil area */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        gluOrtho2D(-3.0, 3.0, -3.0*(GLfloat)h/(GLfloat)w,
                  3.0*(GLfloat)h/(GLfloat)w);
    else
        gluOrtho2D(-3.0*(GLfloat)w/(GLfloat)h,
                  3.0*(GLfloat)w/(GLfloat)h, -3.0, 3.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    glClear(GL_STENCIL_BUFFER_BIT);
    glStencilFunc (GL_ALWAYS, 0x1, 0x1);
    glStencilOp (GL_REPLACE, GL_REPLACE, GL_REPLACE);
    glBegin(GL_QUADS);
        glVertex2f (-1.0, 0.0);
        glVertex2f (0.0, 1.0);
        glVertex2f (1.0, 0.0);
        glVertex2f (0.0, -1.0);
    glEnd();

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (GLfloat) w/(GLfloat) h, 3.0, 7.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -5.0);
}

```



```

/* Main Loop
 * Be certain to request stencil bits.
 */
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB
                        | GLUT_DEPTH | GLUT_STENCIL);
    glutInitWindowSize (400, 400);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}

```

The following examples illustrate other uses of the stencil test. (See Chapter 14 for additional ideas.)

- **Capping**—Suppose you're drawing a closed convex object (or several of them, as long as they don't intersect or enclose each other) made up of several polygons, and you have a clipping plane that may or may not slice off a piece of it. Suppose that if the plane does intersect the object, you want to cap the object with some constant-colored surface, rather than seeing the inside of it. To do this, clear the stencil buffer to zeros, and begin drawing with stenciling enabled and the stencil comparison function set to always accept fragments. Invert the value in the stencil planes each time a fragment is accepted. After all the objects are drawn, regions of the screen where no capping is required have zeros in the stencil planes, and regions requiring capping are nonzero. Reset the stencil function so that it draws only where the stencil value is nonzero, and draw a large polygon of the capping color across the entire screen.
- **Overlapping translucent polygons**—Suppose you have a translucent surface that's made up of polygons that overlap slightly. If you simply use alpha blending, portions of the underlying objects are covered by more than one transparent surface, which doesn't look right. Use the stencil planes to make sure that each fragment is covered by at most one portion of the transparent surface. Do this by clearing the stencil planes to zeros, drawing only when the stencil plane is zero, and incrementing the value in the stencil plane when you draw.

-
- Stippling—Suppose you want to draw an image with a stipple pattern. (See “Displaying Points, Lines, and Polygons” on page 49 for more information about stippling.) You can do this by writing the stipple pattern into the stencil buffer, and then drawing conditionally on the contents of the stencil buffer. After the original stipple pattern is drawn, the stencil buffer isn’t altered while drawing the image, so the object gets stippled by the pattern in the stencil planes.

Depth Test

For each pixel on the screen, the depth buffer keeps track of the distance between the viewpoint and the object occupying that pixel. Then if the specified depth test passes, the incoming depth value replaces the one already in the depth buffer.

The depth buffer is generally used for hidden-surface elimination. If a new candidate color for that pixel appears, it’s drawn only if the corresponding object is closer than the previous object. In this way, after the entire scene has been rendered, only objects that aren’t obscured by other items remain. Initially, the clearing value for the depth buffer is a value that’s as far from the viewpoint as possible, so the depth of any object is nearer than that value. If this is how you want to use the depth buffer, you simply have to enable it by passing `GL_DEPTH_TEST` to `glEnable()` and remember to clear the depth buffer before you redraw each frame. (See “Clearing Buffers” on page 378.) You can also choose a different comparison function for the depth test with `glDepthFunc()`.

```
void glDepthFunc(GLenum func);
```

Sets the comparison function for the depth test. The value for *func* must be `GL_NEVER`, `GL_ALWAYS`, `GL_LESS`, `GL_LEQUAL`, `GL_EQUAL`, `GL_GEQUAL`, `GL_GREATER`, or `GL_NOTEQUAL`. An incoming fragment passes the depth test if its *z* value has the specified relation to the value already stored in the depth buffer. The default is `GL_LESS`, which means that an incoming fragment passes the test if its *z* value is less than that already stored in the depth buffer. In this case, the *z* value represents the distance from the object to the viewpoint, and smaller values mean the corresponding objects are closer to the viewpoint.

Blending, Dithering, and Logical Operations

Once an incoming fragment has passed all the tests described in the previous section, it can be combined with the current contents of the color buffer in one of several ways. The simplest way, which is also the default, is to overwrite the existing values. Alternatively, if you're using RGBA mode and you want the fragment to be translucent or antialiased, you might average its value with the value already in the buffer (blending). On systems with a small number of available colors, you might want to dither color values to increase the number of colors available at the cost of a loss in resolution. In the final stage, you can use arbitrary bitwise logical operations to combine the incoming fragment and the pixel that's already written.

Blending

Blending combines the incoming fragment's R, G, B, and alpha values with those of the pixel already stored at the location. Different blending operations can be applied, and the blending that occurs depends on the values of the incoming alpha value and the alpha value (if any) stored at the pixel. (See "Blending" on page 214 for an extensive discussion of this topic.)

Dithering

On systems with a small number of color bitplanes, you can improve the color resolution at the expense of spatial resolution by dithering the color in the image. Dithering is like halftoning in newspapers. Although *The New York Times* has only two colors—black and white—it can show photographs by representing the shades of gray with combinations of black and white dots. Comparing a newspaper image of a photo (having no shades of gray) with the original photo (with grayscale) makes the loss of spatial resolution obvious. Similarly, systems with a small number of color bitplanes may dither values of red, green, and blue on neighboring pixels for the perception of a wider range of colors.

The dithering operation that takes place is hardware-dependent; all OpenGL allows you to do is to turn it on and off. In fact, on some machines, enabling dithering might do nothing at all, which makes sense if the machine already has high color resolution. To enable and disable dithering, pass `GL_DITHER` to `glEnable()` and `glDisable()`. Dithering is enabled by default.

Dithering applies in both RGBA and color-index mode. The colors or color indices alternate in some hardware-dependent way between the two nearest possibilities. For example, in color-index mode, if dithering is enabled and the color index to be painted is 4.4, then 60% of the pixels may be painted with index 4 and 40% of the pixels with index 5. (Many dithering algorithms are possible, but a dithered value produced by any algorithm must depend upon only the incoming value and the fragment's x and y coordinates.) In RGBA mode, dithering is performed separately for each component (including alpha). To use dithering in color-index mode, you generally need to arrange the colors in the color map appropriately in ramps, otherwise, bizarre images might result.

Logical Operations

The final operation on a fragment is the *logical operation*, such as an OR, XOR, or INVERT, which is applied to the incoming fragment values (source) and/or those currently in the color buffer (destination). Such fragment operations are especially useful on bit-bit-type machines, on which the primary graphics operation is copying a rectangle of data from one place in the window to another, from the window to processor memory, or from memory to the window. Typically, the copy doesn't write the data directly into memory but instead allows you to perform an arbitrary logical operation on the incoming data and the data already present; then it replaces the existing data with the results of the operation.

Since this process can be implemented fairly cheaply in hardware, many such machines are available. As an example of using a logical operation, XOR can be used to draw on an image in an undoable way; simply XOR the same drawing again, and the original image is restored. As another example, when using color-index mode, the color indices can be interpreted as bit patterns. Then you can compose an image as combinations of drawings on different layers, use writemasks to limit drawing to different sets of bitplanes, and perform logical operations to modify different layers.

You enable and disable logical operations by passing `GL_INDEX_LOGIC_OP` or `GL_COLOR_LOGIC_OP` to `glEnable()` and `glDisable()` for color-index mode or RGBA mode, respectively. You also must choose among the sixteen logical operations with `glLogicOp()`, or you'll just get the effect of the default value, `GL_COPY`. (For backward compatibility with OpenGL Version 1.0, `glEnable(GL_LOGIC_OP)` also enables logical operation in color-index mode.)

```
void glLogicOp(GLenum opcode);
```

Selects the logical operation to be performed, given an incoming (source) fragment and the pixel currently stored in the color buffer (destination). Table 10-4 shows the possible values for *opcode* and their meaning (*s* represents source and *d* destination). The default value is `GL_COPY`.

Parameter	Operation	Parameter	Operation
<code>GL_CLEAR</code>	0	<code>GL_AND</code>	$s \wedge d$
<code>GL_COPY</code>	s	<code>GL_OR</code>	$s \vee d$
<code>GL_NOOP</code>	d	<code>GL_NAND</code>	$\neg(s \wedge d)$
<code>GL_SET</code>	1	<code>GL_NOR</code>	$\neg(s \vee d)$
<code>GL_COPY_INVERTED</code>	$\neg s$	<code>GL_XOR</code>	$s \text{ XOR } d$
<code>GL_INVERT</code>	$\neg d$	<code>GL_EQUIV</code>	$\neg(s \text{ XOR } d)$
<code>GL_AND_REVERSE</code>	$s \wedge \neg d$	<code>GL_AND_INVERTED</code>	$\neg s \wedge d$
<code>GL_OR_REVERSE</code>	$s \vee \neg d$	<code>GL_OR_INVERTED</code>	$\neg s \vee d$

Table 10-4 Sixteen Logical Operations

The Accumulation Buffer

Advanced



The accumulation buffer can be used for such things as scene antialiasing, motion blur, simulating photographic depth of field, and calculating the soft shadows that result from multiple light sources. Other techniques are possible, especially in combination with some of the other buffers. (See *The Accumulation Buffer: Hardware Support for High-Quality Rendering* by Paul Haerberli and Kurt Akeley (SIGGRAPH 1990 Proceedings, p. 309–318) for more information on the uses for the accumulation buffer.)

OpenGL graphics operations don't write directly into the accumulation buffer. Typically, a series of images is generated in one of the standard color buffers, and these are accumulated, one at a time, into the accumulation buffer. When the accumulation is finished, the result is copied back into a color buffer for viewing. To reduce rounding errors, the accumulation buffer may have higher precision (more bits per color) than the standard

color buffers. Rendering a scene several times obviously takes longer than rendering it once, but the result is higher quality. You can decide what trade-off between quality and rendering time is appropriate for your application.

You can use the accumulation buffer the same way a photographer can use film for multiple exposures. A photographer typically creates a multiple exposure by taking several pictures of the same scene without advancing the film. If anything in the scene moves, that object appears blurred. Not surprisingly, a computer can do more with an image than a photographer can do with a camera. For example, a computer has exquisite control over the viewpoint, but a photographer can't shake a camera a predictable and controlled amount. (See "Clearing Buffers" on page 378 for information about how to clear the accumulation buffer; use `glAccum()` to control it.)

```
void glAccum(GLenum op, GLfloat value);
```

Controls the accumulation buffer. The *op* parameter selects the operation, and *value* is a number to be used in that operation. The possible operations are `GL_ACCUM`, `GL_LOAD`, `GL_RETURN`, `GL_ADD`, and `GL_MULT`.

- `GL_ACCUM` reads each pixel from the buffer currently selected for reading with `glReadBuffer()`, multiplies the R, G, B, and alpha values by *value*, and adds the result to the accumulation buffer.
- `GL_LOAD` does the same thing, except that the values replace those in the accumulation buffer rather than being added to them.
- `GL_RETURN` takes values from the accumulation buffer, multiplies them by *value*, and places the result in the color buffer(s) enabled for writing.
- `GL_ADD` and `GL_MULT` simply add or multiply the value of each pixel in the accumulation buffer by *value* and then return it to the accumulation buffer. For `GL_MULT`, *value* is clamped to be in the range `[-1.0,1.0]`. For `GL_ADD`, no clamping occurs.

Scene Antialiasing

To perform scene antialiasing, first clear the accumulation buffer and enable the front buffer for reading and writing. Then loop several times

(say, n) through code that jitters and draws the image (*jittering* is moving the image to a slightly different position), accumulating the data with

```
glAccum(GL_ACCUM, 1.0/n);
```

and finally calling

```
glAccum(GL_RETURN, 1.0);
```

Note that this method is a bit faster if, on the first pass through the loop, `GL_LOAD` is used and clearing the accumulation buffer is omitted. See Table 10-5 for possible jittering values. With this code, the image is drawn n times before the final image is drawn. If you want to avoid showing the user the intermediate images, draw into a color buffer that's not displayed, accumulate from that, and use the `GL_RETURN` call to draw into a displayed buffer (or into a back buffer that you subsequently swap to the front).

You could instead present a user interface that shows the viewed image improving as each additional piece is accumulated and that allows the user to halt the process when the image is good enough. To accomplish this, in the loop that draws successive images, call `glAccum()` with `GL_RETURN` after each accumulation, using `16.0/1.0`, `16.0/2.0`, `16.0/3.0`, ... as the second argument. With this technique, after one pass, $1/16$ of the final image is shown, after two passes, $2/16$ is shown, and so on. After the `GL_RETURN`, the code should check to see if the user wants to interrupt the process. This interface is slightly slower, since the resultant image must be copied in after each pass.

To decide what n should be, you need to trade off speed (the more times you draw the scene, the longer it takes to obtain the final image) and quality (the more times you draw the scene, the smoother it gets, until you make maximum use of the accumulation buffer's resolution). Plates 22 and 23 show improvements made using scene antialiasing.

Example 10-2 defines two routines for jittering that you might find useful: `accPerspective()` and `accFrustum()`. The routine `accPerspective()` is used in place of `gluPerspective()`, and the first four parameters of both routines are the same. To jitter the viewing frustum for scene antialiasing, pass the x and y jitter values (of less than one pixel) to the fifth and sixth parameters of `accPerspective()`. Also pass `0.0` for the seventh and eighth parameters to `accPerspective()` and a nonzero value for the ninth parameter (to prevent

division by zero inside `accPerspective()`). These last three parameters are used for depth-of-field effects, which are described later in this chapter.

Example 10-2 Routines for Jittering the Viewing Volume: `accpersp.c`

```
#define PI_ 3.14159265358979323846

void accFrustum(GLdouble left, GLdouble right, GLdouble bottom,
               GLdouble top, GLdouble near, GLdouble far, GLdouble pixdx,
               GLdouble pixdy, GLdouble eyedx, GLdouble eyedy,
               GLdouble focus)
{
    GLdouble xysize, yysize;
    GLdouble dx, dy;
    GLint viewport[4];

    glGetIntegerv (GL_VIEWPORT, viewport);

    xysize = right - left;
    yysize = top - bottom;
    dx = -(pixdx*xysize/(GLdouble) viewport[2] +
           eyedx*near/focus);
    dy = -(pixdy*yysize/(GLdouble) viewport[3] +
           eyedy*near/focus);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum (left + dx, right + dx, bottom + dy, top + dy,
              near, far);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef (-eyedx, -eyedy, 0.0);
}

void accPerspective(GLdouble fovy, GLdouble aspect,
                  GLdouble near, GLdouble far, GLdouble pixdx, GLdouble pixdy,
                  GLdouble eyedx, GLdouble eyedy, GLdouble focus)
{
    GLdouble fov2, left, right, bottom, top;
    fov2 = ((fovy*PI_) / 180.0) / 2.0;

    top = near / (fcos(fov2) / fsin(fov2));
    bottom = -top;
    right = top * aspect;
    left = -right;
}
```



```

        accFrustum (left, right, bottom, top, near, far,
                   pixdx, pixdy, eyedx, eyedy, focus);
    }

```

Example 10-3 uses these two routines to perform scene antialiasing.

Example 10-3 Scene Antialiasing: accpersp.c

```

#include <GL/gl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <math.h>
#include <GL/glut.h>
#include "jitter.h"

void init(void)
{
    GLfloat mat_ambient[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 0.0, 0.0, 10.0, 1.0 };
    GLfloat lm_ambient[] = { 0.2, 0.2, 0.2, 1.0 };

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 50.0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lm_ambient);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
    glShadeModel (GL_FLAT);

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClearAccum(0.0, 0.0, 0.0, 0.0);
}

void displayObjects(void)
{
    GLfloat torus_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
    GLfloat cube_diffuse[] = { 0.0, 0.7, 0.7, 1.0 };
    GLfloat sphere_diffuse[] = { 0.7, 0.0, 0.7, 1.0 };
    GLfloat octa_diffuse[] = { 0.7, 0.4, 0.4, 1.0 };

    glPushMatrix ();
    glTranslatef (0.0, 0.0, -5.0);
    glRotatef (30.0, 1.0, 0.0, 0.0);

```

```

glPushMatrix ();
glTranslatef (-0.80, 0.35, 0.0);
glRotatef (100.0, 1.0, 0.0, 0.0);
glMaterialfv(GL_FRONT, GL_DIFFUSE, torus_diffuse);
glutSolidTorus (0.275, 0.85, 16, 16);
glPopMatrix ();

glPushMatrix ();
glTranslatef (-0.75, -0.50, 0.0);
glRotatef (45.0, 0.0, 0.0, 1.0);
glRotatef (45.0, 1.0, 0.0, 0.0);
glMaterialfv(GL_FRONT, GL_DIFFUSE, cube_diffuse);
glutSolidCube (1.5);
glPopMatrix ();

glPushMatrix ();
glTranslatef (0.75, 0.60, 0.0);
glRotatef (30.0, 1.0, 0.0, 0.0);
glMaterialfv(GL_FRONT, GL_DIFFUSE, sphere_diffuse);
glutSolidSphere (1.0, 16, 16);
glPopMatrix ();

glPushMatrix ();
glTranslatef (0.70, -0.90, 0.25);
glMaterialfv(GL_FRONT, GL_DIFFUSE, octa_diffuse);
glutSolidOctahedron ();
glPopMatrix ();

glPopMatrix ();
}

#define ACSIZE 8

void display(void)
{
    GLint viewport[4];
    int jitter;

    glGetIntegerv (GL_VIEWPORT, viewport);

    glClear(GL_ACCUM_BUFFER_BIT);
    for (jitter = 0; jitter < ACSIZE; jitter++) {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        accPerspective (50.0,
            (GLdouble) viewport[2]/(GLdouble) viewport[3],
            1.0, 15.0, j8[jitter].x, j8[jitter].y, 0.0, 0.0, 1.0);
        displayObjects ();
    }
}

```

```

        glAccum(GL_ACCUM, 1.0/ACSIZE);
    }
    glAccum (GL_RETURN, 1.0);
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
}

/* Main Loop
 * Be certain you request an accumulation buffer.
 */
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB
                        | GLUT_ACCUM | GLUT_DEPTH);
    glutInitWindowSize (250, 250);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init();
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}

```

You don't have to use a perspective projection to perform scene antialiasing. You can antialias a scene with orthographic projection simply by using `glTranslate*()` to jitter the scene. Keep in mind that `glTranslate*()` operates in world coordinates, but you want the apparent motion of the scene to be less than one pixel, measured in screen coordinates. Thus, you must reverse the world-coordinate mapping by calculating the jittering translation values, using its width or height in world coordinates divided by its viewport size. Then multiply that world-coordinate value by the amount of jitter to determine how much the scene should be moved in world coordinates to get a predictable jitter of less than one pixel. Example 10-4

shows how the `display()` and `reshape()` routines might look with a world-coordinate width and height of 4.5.

Example 10-4 Jittering with an Orthographic Projection: `accanti.c`

```
#define ACSIZE 8

void display(void)
{
    GLint viewport[4];
    int jitter;

    glGetIntegerv (GL_VIEWPORT, viewport);

    glClear(GL_ACCUM_BUFFER_BIT);
    for (jitter = 0; jitter < ACSIZE; jitter++) {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glPushMatrix ();
/*      Note that 4.5 is the distance in world space between
 *      left and right and bottom and top.
 *      This formula converts fractional pixel movement to
 *      world coordinates.
 */
        glTranslatef (j8[jitter].x*4.5/viewport[2],
                     j8[jitter].y*4.5/viewport[3], 0.0);
        displayObjects ();
        glPopMatrix ();
        glAccum(GL_ACCUM, 1.0/ACSIZE);
    }
    glAccum (GL_RETURN, 1.0);
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho (-2.25, 2.25, -2.25*h/w, 2.25*h/w, -10.0, 10.0);
    else
        glOrtho (-2.25*w/h, 2.25*w/h, -2.25, 2.25, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

Motion Blur

Similar methods can be used to simulate motion blur, as shown in Plate 7 and Figure 10-2. Suppose your scene has some stationary and some moving objects in it, and you want to make a motion-blurred image extending over a small interval of time. Set up the accumulation buffer in the same way, but instead of spatially jittering the images, jitter them temporally. The entire scene can be made successively dimmer by calling

```
glAccum (GL_MULT, decayFactor);
```

as the scene is drawn into the accumulation buffer, where *decayFactor* is a number from 0.0 to 1.0. Smaller numbers for *decayFactor* cause the object to appear to be moving faster. You can transfer the completed scene with the object's current position and "vapor trail" of previous positions from the accumulation buffer to the standard color buffer with

```
glAccum (GL_RETURN, 1.0);
```

The image looks correct even if the items move at different speeds, or if some of them are accelerated. As before, the more jitter points (temporal, in this case) you use, the better the final image, at least up to the point where you begin to lose resolution due to finite precision in the accumulation buffer. You can combine motion blur with antialiasing by jittering in both the spatial and temporal domains, but you pay for higher quality with longer rendering times.

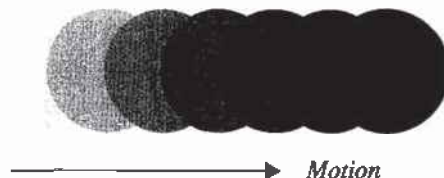


Figure 10-2 Motion-Blurred Object

Depth of Field

A photograph made with a camera is in perfect focus only for items lying on a single plane a certain distance from the film. The farther an item is from this plane, the more out of focus it is. The depth of field for a camera is a region about the plane of perfect focus where items are out of focus by a small enough amount.

Under normal conditions, everything you draw with OpenGL is in focus (unless your monitor's bad, in which case everything is out of focus). The accumulation buffer can be used to approximate what you would see in a photograph where items are more and more blurred as their distance from a plane of perfect focus increases. It isn't an exact simulation of the effects produced in a camera, but the result looks similar to what a camera would produce.

To achieve this result, draw the scene repeatedly using calls with different argument values to `glFrustum()`. Choose the arguments so that the position of the viewpoint varies slightly around its true position and so that each frustum shares a common rectangle that lies in the plane of perfect focus, as shown in Figure 10-3. The results of all the renderings should be averaged in the usual way using the accumulation buffer.

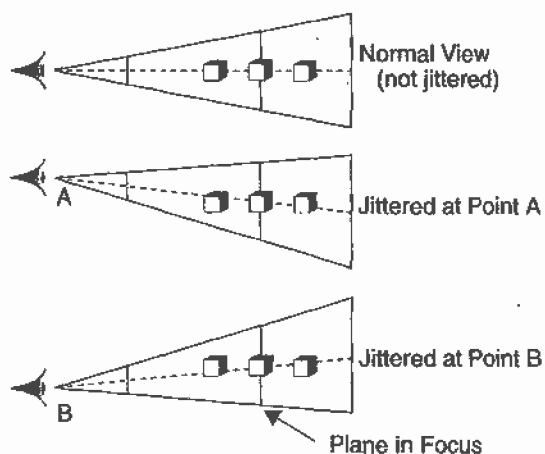


Figure 10-3 Jittered Viewing Volume for Depth-of-Field Effects

Plate 10 shows an image of five teapots drawn using the depth-of-field effect. The gold teapot (second from the left) is in focus, and the other teapots get progressively blurrier, depending upon their distance from the focal plane (gold teapot). The code to draw this image is shown in Example 10-5 (which assumes `accPerspective()` and `accFrustum()` are defined as described in Example 10-2). The scene is drawn eight times, each with a slightly jittered viewing volume, by calling `accPerspective()`. As you recall, with scene antialiasing, the fifth and sixth parameters jitter the viewing volumes in the x and y directions. For the depth-of-field effect, however, you want to jitter the volume while holding it stationary at the focal plane. The focal plane is the depth value defined by the ninth (last) parameter to `accPerspective()`, which is $z = 5.0$ in this example. The

amount of blur is determined by multiplying the x and y jitter values (seventh and eighth parameters of `accPerspective()`) by a constant. Determining the constant is not a science; experiment with values until the depth of field is as pronounced as you want. (Note that in Example 10-5, the fifth and sixth parameters to `accPerspective()` are set to 0.0, so scene antialiasing is turned off.)

Example 10-5 Depth-of-Field Effect: `dof.c`

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <math.h>
#include "jitter.h"

void init(void)
{
    GLfloat ambient[] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat position[] = { 0.0, 3.0, 3.0, 0.0 };

    GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
    GLfloat local_view[] = { 0.0 };

    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, position);

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
    glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, local_view);

    glFrontFace (GL_CW);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);
    glEnable(GL_DEPTH_TEST);

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClearAccum(0.0, 0.0, 0.0, 0.0);
    /* make teapot display list */
    teapotList = glGenLists(1);
    glNewList (teapotList, GL_COMPILE);
    glutSolidTeapot (0.5);
}
```

```

    glEndList ();
}

void renderTeapot (GLfloat x, GLfloat y, GLfloat z,
    GLfloat ambr, GLfloat ambg, GLfloat ambb,
    GLfloat difr, GLfloat difg, GLfloat difb,
    GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)
{
    GLfloat mat[4];

    glPushMatrix();
    glTranslatef (x, y, z);
    mat[0] = ambr; mat[1] = ambg; mat[2] = ambb; mat[3] = 1.0;
    glMaterialfv (GL_FRONT, GL_AMBIENT, mat);
    mat[0] = difr; mat[1] = difg; mat[2] = difb;
    glMaterialfv (GL_FRONT, GL_DIFFUSE, mat);
    mat[0] = specr; mat[1] = specg; mat[2] = specb;
    glMaterialfv (GL_FRONT, GL_SPECULAR, mat);
    glMaterialf (GL_FRONT, GL_SHININESS, shine*128.0);
    glCallList(teapotList);
    glPopMatrix();
}

void display(void)
{
    int jitter;
    GLint viewport[4];

    glGetIntegerv (GL_VIEWPORT, viewport);
    glClear(GL_ACCUM_BUFFER_BIT);

    for (jitter = 0; jitter < 8; jitter++) {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        accPerspective (45.0,
            (GLdouble) viewport[2]/(GLdouble) viewport[3],
            1.0, 15.0, 0.0, 0.0,
            0.33*j8[jitter].x, 0.33*j8[jitter].y, 5.0);

        /*    ruby, gold, silver, emerald, and cyan teapots    */
        renderTeapot (-1.1, -0.5, -4.5, 0.1745, 0.01175,
            0.01175, 0.61424, 0.04136, 0.04136,
            0.727811, 0.626959, 0.626959, 0.6);
        renderTeapot (-0.5, -0.5, -5.0, 0.24725, 0.1995,
            0.0745, 0.75164, 0.60648, 0.22648,
            0.628281, 0.555802, 0.366065, 0.4);
    }
}

```



```

        renderTeapot (0.2, -0.5, -5.5, 0.19225, 0.19225,
                     0.19225, 0.50754, 0.50754, 0.50754,
                     0.508273, 0.508273, 0.508273, 0.4);
        renderTeapot (1.0, -0.5, -6.0, 0.0215, 0.1745, 0.0215,
                     0.07568, 0.61424, 0.07568, 0.633,
                     0.727811, 0.633, 0.6);
        renderTeapot (1.8, -0.5, -6.5, 0.0, 0.1, 0.06, 0.0,
                     0.50980392, 0.50980392, 0.50196078,
                     0.50196078, 0.50196078, .25);
        glAccum (GL_ACCUM, 0.125);
    }
    glAccum (GL_RETURN, 1.0);
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
}

/* Main Loop
 * Be certain you request an accumulation buffer.
 */
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB
                        | GLUT_ACCUM | GLUT_DEPTH);
    glutInitWindowSize (400, 400);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init();
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}

```

Soft Shadows

To accumulate soft shadows due to multiple light sources, render the shadows with one light turned on at a time, and accumulate them together. This can be combined with spatial jittering to antialias the scene at the same time. (See “Shadows” on page 519 for more information about drawing shadows.)

Jittering

If you need to take nine or sixteen samples to antialias an image, you might think that the best choice of points is an equally spaced grid across the pixel. Surprisingly, this is not necessarily true. In fact, sometimes it's a good idea to take points that lie in adjacent pixels. You might want a uniform distribution or a normalized distribution, clustering toward the center of the pixel. (The aforementioned SIGGRAPH paper discusses these issues.) In addition, Table 10-5 shows a few sets of reasonable jittering values to be used for some selected sample counts. Most of the examples in the table are uniformly distributed in the pixel, and all lie within the pixel.

Count	Values
2	{0.25, 0.75}, {0.75, 0.25}
3	{0.5033922635, 0.8317967229}, {0.7806016275, 0.2504380877}, {0.2261828938, 0.4131553612}
4	{0.375, 0.25}, {0.125, 0.75}, {0.875, 0.25}, {0.625, 0.75}
5	{0.5, 0.5}, {0.3, 0.1}, {0.7, 0.9}, {0.9, 0.3}, {0.1, 0.7}
6	{0.4646464646, 0.4646464646}, {0.1313131313, 0.7979797979}, {0.5353535353, 0.8686868686}, {0.8686868686, 0.5353535353}, {0.7979797979, 0.1313131313}, {0.2020202020, 0.2020202020}
8	{0.5625, 0.4375}, {0.0625, 0.9375}, {0.3125, 0.6875}, {0.6875, 0.8125}, {0.8125, 0.1875}, {0.9375, 0.5625}, {0.4375, 0.0625}, {0.1875, 0.3125}
9	{0.5, 0.5}, {0.1666666666, 0.9444444444}, {0.5, 0.1666666666}, {0.5, 0.8333333333}, {0.1666666666, 0.2777777777}, {0.8333333333, 0.3888888888}, {0.1666666666, 0.6111111111}, {0.8333333333, 0.7222222222}, {0.8333333333, 0.0555555555}
12	{0.4166666666, 0.625}, {0.9166666666, 0.875}, {0.25, 0.375}, {0.4166666666, 0.125}, {0.75, 0.125}, {0.0833333333, 0.125}, {0.75, 0.625}, {0.25, 0.875}, {0.5833333333, 0.375}, {0.9166666666, 0.375}, {0.0833333333, 0.625}, {0.5833333333, 0.875}

Table 10-5 Sample Jittering Values

Count	Values
16	{0.375, 0.4375}, {0.625, 0.0625}, {0.875, 0.1875}, {0.125, 0.0625}, {0.375, 0.6875}, {0.875, 0.4375}, {0.625, 0.5625}, {0.375, 0.9375}, {0.625, 0.3125}, {0.125, 0.5625}, {0.125, 0.8125}, {0.375, 0.1875}, {0.875, 0.9375}, {0.875, 0.6875}, {0.125, 0.3125}, {0.625, 0.8125}

Table 10-5 Sample Jittering Values (continued)

Tessellators and Quadrics



Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Render concave filled polygons by first tessellating them into convex polygons, which can be rendered using standard OpenGL routines.
- Use the GLU library to create quadrics objects to render and model the surfaces of spheres and cylinders and to tessellate disks (circles) and partial disks (arcs).

The OpenGL library (GL) is designed for low-level operations, both streamlined and accessible to hardware acceleration. The OpenGL Utility Library (GLU) complements the OpenGL library, supporting higher-level operations. Some of the GLU operations are covered in other chapters. Mipmapping (`gluBuild*DMipmaps()`) and image scaling (`gluScaleImage()`) are discussed along with other facets of texture mapping in Chapter 9. Several matrix transformation GLU routines (`gluOrtho2D()`, `gluPerspective()`, `gluLookAt()`, `gluProject()`, and `gluUnProject()`) are described in Chapter 3. The use of `gluPickMatrix()` is explained in Chapter 13. The GLU NURBS facilities, which are built atop OpenGL evaluators, are covered in Chapter 12. Only two GLU topics remain: polygon tessellators and quadric surfaces, and those topics are discussed in this chapter.

To optimize performance, the basic OpenGL only renders convex polygons, but the GLU contains routines to tessellate concave polygons into convex ones, which the basic OpenGL can handle. Where the basic OpenGL operates upon simple primitives, such as points, lines, and filled polygons, the GLU can create higher-level objects, such as the surfaces of spheres, cylinders, and cones.

This chapter has the following major sections.

- “Polygon Tessellation” on page 410 explains how to tessellate convex polygons into easier-to-render convex polygons.
- “Quadrics: Rendering Spheres, Cylinders, and Disks” on page 428 describes how to generate spheres, cylinders, circles and arcs, including data such as surface normals and texture coordinates.

Polygon Tessellation

As discussed in “Describing Points, Lines, and Polygons” on page 37, OpenGL can directly display only simple convex polygons. A polygon is simple if the edges intersect only at vertices, there are no duplicate vertices, and exactly two edges meet at any vertex. If your application requires the display of concave polygons, polygons containing holes, or polygons with intersecting edges, those polygons must first be subdivided into simple convex polygons before they can be displayed. Such subdivision is called *tessellation*, and the GLU provides a collection of routines that perform tessellation. These routines take as input arbitrary contours, which describe hard-to-render polygons, and they return some combination of triangles, triangle meshes, triangle fans, or lines.

Figure 11-1 shows some contours of polygons that require tessellation: from left to right, a concave polygon, a polygon with a hole, and a self-intersecting polygon.

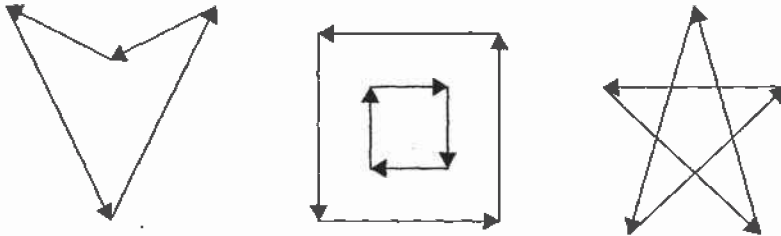


Figure 11-1 Contours That Require Tessellation

If you think a polygon may need tessellation, follow these typical steps.

1. Create a new tessellation object with `gluNewTess()`.
2. Use `gluTessCallback()` several times to register callback functions to perform operations during the tessellation. The trickiest case for a callback function is when the tessellation algorithm detects an intersection and must call the function registered for the `GLU_TESS_COMBINE` callback.
3. Specify tessellation properties by calling `gluTessProperty()`. The most important property is the winding rule, which determines the regions that should be filled and those that should remain unshaded.
4. Create and render tessellated polygons by specifying the contours of one or more closed polygons. If the data for the object is static, encapsulate the tessellated polygons in a display list. (If you don't have to recalculate the tessellation over and over again, using display lists is more efficient.)
5. If you need to tessellate something else, you may reuse your tessellation object. If you are forever finished with your tessellation object, you may delete it with `gluDeleteTess()`.

Note: The tessellator described here was introduced in version 1.2 of the GLU. If you are using an older version of the GLU, you must use routines described in "Describing GLU Errors" on page 426. To query which version of GLU you have, use `gluGetString(GLU_VERSION)`, which returns a string with your GLU version number. If you don't seem to have `gluGetString()` in your GLU, then you have GLU 1.0, which did not yet have the `gluGetString()` routine.

Create a Tessellation Object

As a complex polygon is being described and tessellated, it has associated data, such as the vertices, edges, and callback functions. All this data is tied to a single tessellation object. To perform tessellation, your program first has to create a tessellation object using the routine `gluNewTess()`.

```
GLUtesselator* gluNewTess(void);
```

Creates a new tessellation object and returns a pointer to it. A null pointer is returned if the creation fails.

A single tessellation object can be reused for all your tessellations. This object is required only because library routines might need to do their own tessellations, and they should be able to do so without interfering with any tessellation that your program is doing. It might also be useful to have multiple tessellation objects if you want to use different sets of callbacks for different tessellations. A typical program, however, allocates a single tessellation object and uses it for all its tessellations. There's no real need to free it because it uses a small amount of memory. On the other hand, it never hurts to be tidy.

Tessellation Callback Routines

After you create a tessellation object, you must provide a series of callback routines to be called at appropriate times during the tessellation. After specifying the callbacks, you describe the contours of one or more polygons using GLU routines. When the description of the contours is complete, the tessellation facility invokes your callback routines as necessary.

Any functions that are omitted are simply not called during the tessellation, and any information they might have returned to your program is lost. All are specified by the single routine `gluTessCallback()`.

```
void gluTessCallback(GLUtesselator *tessobj, GLenum type, void (*fn)());
```

Associates the callback function *fn* with the tessellation object *tessobj*. The type of the callback is determined by the parameter *type*, which can be `GLU_TESS_BEGIN`, `GLU_TESS_BEGIN_DATA`, `GLU_TESS_EDGE_FLAG`, `GLU_TESS_EDGE_FLAG_DATA`, `GLU_TESS_VERTEX`, `GLU_TESS_VERTEX_DATA`, `GLU_TESS_END`, `GLU_TESS_END_DATA`, `GLU_TESS_COMBINE`, `GLU_TESS_COMBINE_DATA`, `GLU_TESS_ERROR`,

and `GLU_TESS_ERROR_DATA`. The twelve possible callback functions have the following prototypes:

```
GLU_TESS_BEGIN          void begin(GLenum type);
GLU_TESS_BEGIN_DATA     void begin(GLenum type,
                                   void *user_data);
GLU_TESS_EDGE_FLAG      void edgeFlag(GLboolean flag);
GLU_TESS_EDGE_FLAG_DATA void edgeFlag(GLboolean flag,
                                   void *user_data);
GLU_TESS_VERTEX         void vertex(void *vertex_data);
GLU_TESS_VERTEX_DATA    void vertex(void *vertex_data,
                                   void *user_data);
GLU_TESS_END            void end(void);
GLU_TESS_END_DATA       void end(void *user_data);
GLU_TESS_ERROR          void error(GLenum errno);
GLU_TESS_ERROR_DATA     void error(GLenum errno, void
*user_data);
GLU_TESS_COMBINE        void combine(GLdouble coords[3],
                                   void *vertex_data[4],
                                   GLfloat weight[4],
                                   void **outData);
GLU_TESS_COMBINE_DATA   void combine(GLdouble coords[3],
                                   void *vertex_data[4],
                                   GLfloat weight[4],
                                   void **outData,
                                   void *user_data);
```

To change a callback routine, simply call `gluTessCallback()` with the new routine. To eliminate a callback routine without replacing it with a new one, pass `gluTessCallback()` a null pointer for the appropriate function.

As tessellation proceeds, the callback routines are called in a manner similar to how you use the OpenGL commands `glBegin()`, `glEdgeFlag*()`, `glVertex*()`, and `glEnd()`. (See “Marking Polygon Boundary Edges” on page 62 for more information about `glEdgeFlag*()`.) The combine callback is used to create new vertices where edges intersect. The error callback is invoked during the tessellation only if something goes wrong.

For every tessellator object created, a `GLU_TESS_BEGIN` callback is invoked with one of four possible parameters: `GL_TRIANGLE_FAN`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLES`, and `GL_LINE_LOOP`. When the tessellator decomposes the polygons, the tessellation algorithm will decide which type of triangle primitive is most efficient to use. (If the `GLU_TESS_BOUNDARY_ONLY` property is enabled, then `GL_LINE_LOOP` is used for rendering.)

Since edge flags make no sense in a triangle fan or triangle strip, if there is a callback associated with `GLU_TESS_EDGE_FLAG` that enables edge flags, the `GLU_TESS_BEGIN` callback is called only with `GL_TRIANGLES`. The `GLU_TESS_EDGE_FLAG` callback works exactly analogously to the OpenGL `glEdgeFlag*()` call.

After the `GLU_TESS_BEGIN` callback routine is called and before the callback associated with `GLU_TESS_END` is called, some combination of the `GLU_TESS_EDGE_FLAG` and `GLU_TESS_VERTEX` callbacks is invoked (usually by calls to `gluTessVertex()`, which is described on page 423). The associated edge flags and vertices are interpreted exactly as they are in OpenGL between `glBegin()` and the matching `glEnd()`.

If something goes wrong, the error callback is passed a GLU error number. A character string describing the error is obtained using the routine `gluErrorString()`. (See “Describing GLU Errors” on page 426 for more information about this routine.)

Example 11-1 shows a portion of `tess.c`, where a tessellation object is created and several callbacks are registered.

Example 11-1 Registering Tessellation Callbacks: `tess.c`

```
/* a portion of init() */
tobj = gluNewTess();
gluTessCallback(tobj, GLU_TESS_VERTEX,
                (GLvoid (*) ()) &glVertex3dv);
gluTessCallback(tobj, GLU_TESS_BEGIN,
                (GLvoid (*) ()) &beginCallback);
gluTessCallback(tobj, GLU_TESS_END,
                (GLvoid (*) ()) &endCallback);
```

```

gluTessCallback(tobj, GLU_TESS_ERROR,
                (GLvoid (*) ()) &errorCallback);

/* the callback routines registered by gluTessCallback() */

void beginCallback(GLenum which)
{
    glBegin(which);
}

void endCallback(void)
{
    glEnd();
}

void errorCallback(GLenum errorCode)
{
    const GLubyte *estring;

    estring = gluErrorString(errorCode);
    fprintf (stderr, "Tessellation Error: %s\n", estring);
    exit (0);
}

```

In Example 11-1, the registered `GLU_TESS_VERTEX` callback is simply `glVertex3dv()`, and only the coordinates at each vertex are passed along. However, if you want to specify more information at every vertex, such as a color value, a surface normal vector, or texture coordinate, you'll have to make a more complex callback routine. Example 11-2 shows the start of another tessellated object, further along in program `tess.c`. The registered function `vertexCallback()` expects to receive a parameter that is a pointer to six double-length floating point values: the *x*, *y*, and *z* coordinates and the red, green, and blue color values, respectively, for that vertex.

Example 11-2 Vertex and Combine Callbacks: `tess.c`

```

/* a different portion of init() */
gluTessCallback(tobj, GLU_TESS_VERTEX,
                (GLvoid (*) ()) &vertexCallback);
gluTessCallback(tobj, GLU_TESS_BEGIN,
                (GLvoid (*) ()) &beginCallback);
gluTessCallback(tobj, GLU_TESS_END,
                (GLvoid (*) ()) &endCallback);
gluTessCallback(tobj, GLU_TESS_ERROR,
                (GLvoid (*) ()) &errorCallback);
gluTessCallback(tobj, GLU_TESS_COMBINE,
                (GLvoid (*) ()) &combineCallback);

```

```

/* new callback routines registered by these calls */
void vertexCallback(GLvoid *vertex)
{
    const GLdouble *pointer;

    pointer = (GLdouble *) vertex;
    glColor3dv(pointer+3);
    glVertex3dv(vertex);
}

void combineCallback(GLdouble coords[3],
                    GLdouble *vertex_data[4],
                    GLfloat weight[4], GLdouble **dataOut )
{
    GLdouble *vertex;
    int i;

    vertex = (GLdouble *) malloc(6 * sizeof(GLdouble));
    vertex[0] = coords[0];
    vertex[1] = coords[1];
    vertex[2] = coords[2];
    for (i = 3; i < 7; i++)
        vertex[i] = weight[0] * vertex_data[0][i]
                    + weight[1] * vertex_data[1][i]
                    + weight[2] * vertex_data[2][i]
                    + weight[3] * vertex_data[3][i];
    *dataOut = vertex;
}

```

Example 11-2 also shows the use of the `GLU_TESS_COMBINE` callback. Whenever the tessellation algorithm examines the input contours, detects an intersection, and decides it must create a new vertex, the `GLU_TESS_COMBINE` callback is invoked. The callback is also called when the tessellator decides to merge features of two vertices that are very close to one another. The newly created vertex is a linear combination of up to four existing vertices, referenced by `vertex_data[0..3]` in Example 11-2. The coefficients of the linear combination are given by `weight[0..3]`; these weights sum to 1.0. `coords` gives the location of the new vertex.

The registered callback routine must allocate memory for another vertex, perform a weighted interpolation of data using `vertex_data` and `weight`, and return the new vertex pointer as `dataOut`. `combineCallback()` in Example 11-2 interpolates the RGB color value. The function allocates a six-element array, puts the *x*, *y*, and *z* coordinates in the first three elements, and then puts the weighted average of the RGB color values in the last three elements.

User-Specified Data

Six kinds of callbacks can be registered. Since there are two versions of each kind of callback, there are twelve callbacks in all. For each kind of callback, there is one with user-specified data and one without. The user-specified data is given by the application to `gluTessBeginPolygon()` and is then passed, unaltered, to each `*DATA` callback routine. With `GLU_TESS_BEGIN_DATA`, the user-specified data may be used for “per-polygon” data. If you specify both versions of a particular callback, the callback with *user_data* is used, and the other is ignored. So, although there are twelve callbacks, you can have a maximum of six callback functions active at any time.

For instance, Example 11-2 uses smooth shading, so `vertexCallback()` specifies an RGB color for every vertex. If you want to do lighting and smooth shading, the callback would specify a surface normal for every vertex. However, if you want lighting and flat shading, you might specify only one surface normal for every polygon, not for every vertex. In that case, you might choose to use the `GLU_TESS_BEGIN_DATA` callback and pass the vertex coordinates and surface normal in the *user_data* pointer.

Tessellation Properties

Prior to tessellation and rendering, you may use `gluTessProperty()` to set several properties to affect the tessellation algorithm. The most important and complicated of these properties is the winding rule, which determines what is considered “interior” and “exterior.”

```
void gluTessProperty(GLUtessellator *tessobj, GLenum property,  
                    GLdouble value);
```

For the tessellation object *tessobj*, the current value of *property* is set to *value*. *property* is one of `GLU_TESS_BOUNDARY_ONLY`, `GLU_TESS_TOLERANCE`, or `GLU_TESS_WINDING_RULE`.

If *property* is `GLU_TESS_BOUNDARY_ONLY`, *value* is either `GL_TRUE` or `GL_FALSE`. When set to `GL_TRUE`, polygons are no longer tessellated into filled polygons; line loops are drawn to outline the contours that separate the polygon interior and exterior. The default value is `GL_FALSE`. (See `gluTessNormal()` to see how to control the winding direction of the contours.)

If *property* is `GLU_TESS_TOLERANCE`, *value* is a distance used to calculate whether two vertices are close together enough to be merged by the `GLU_TESS_COMBINE` callback. The tolerance value is multiplied by the largest coordinate magnitude of an input vertex to determine the maximum distance any feature can move as a result of a single merge operation. Feature merging may not be supported by your implementation, and the tolerance value is only a hint. The default tolerance value is zero.

The `GLU_TESS_WINDING_RULE` *property* determines which parts of the polygon are on the interior and which are the exterior and should not be filled. *value* can be one of `GLU_TESS_WINDING_ODD` (the default), `GLU_TESS_WINDING_NONZERO`, `GLU_TESS_WINDING_POSITIVE`, `GLU_TESS_WINDING_NEGATIVE`, or `GLU_TESS_WINDING_ABS_GEQ_TWO`.

Winding Numbers and Winding Rules

For a single contour, the winding number of a point is the signed number of revolutions we make around that point while traveling once around the contour (where a counterclockwise revolution is positive and a clockwise revolution is negative). When there are several contours, the individual winding numbers are summed. This procedure associates a signed integer value with each point in the plane. Note that the winding number is the same for all points in a single region.

Figure 11-2 shows three sets of contours and winding numbers for points inside those contours. In the left set, all three contours are counterclockwise, so each nested interior region adds one to the winding number. For the middle set, the two interior contours are drawn clockwise, so the winding number decreases and actually becomes negative.

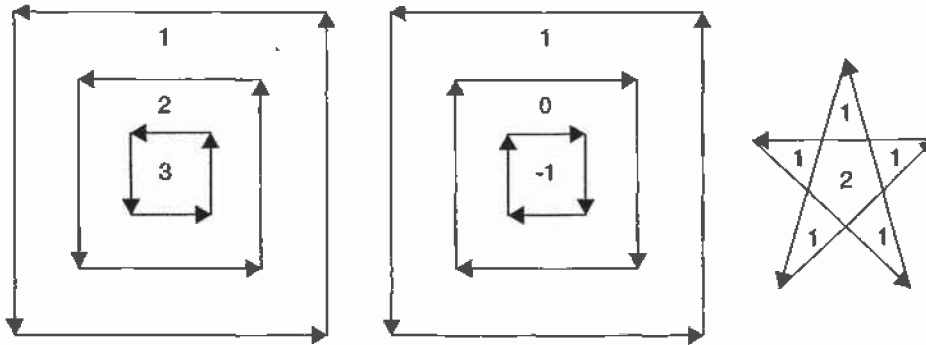


Figure 11-2 Winding Numbers for Sample Contours

The winding rule classifies a region as *inside* if its winding number belongs to the chosen category (odd, nonzero, positive, negative, or “absolute value of greater than or equal to two”). The odd and nonzero rules are common ways to define the interior. The positive, negative, and “absolute value ≥ 2 ” winding rules have some limited use for polygon CSG (computational solid geometry) operations.

The program `tesswind.c` demonstrates the effects of winding rules. The four sets of contours shown in Figure 11-3 are rendered. The user can then cycle through the different winding rule properties to see their effects. For each winding rule, the dark areas represent interiors. Note the effect of clockwise and counterclockwise winding.

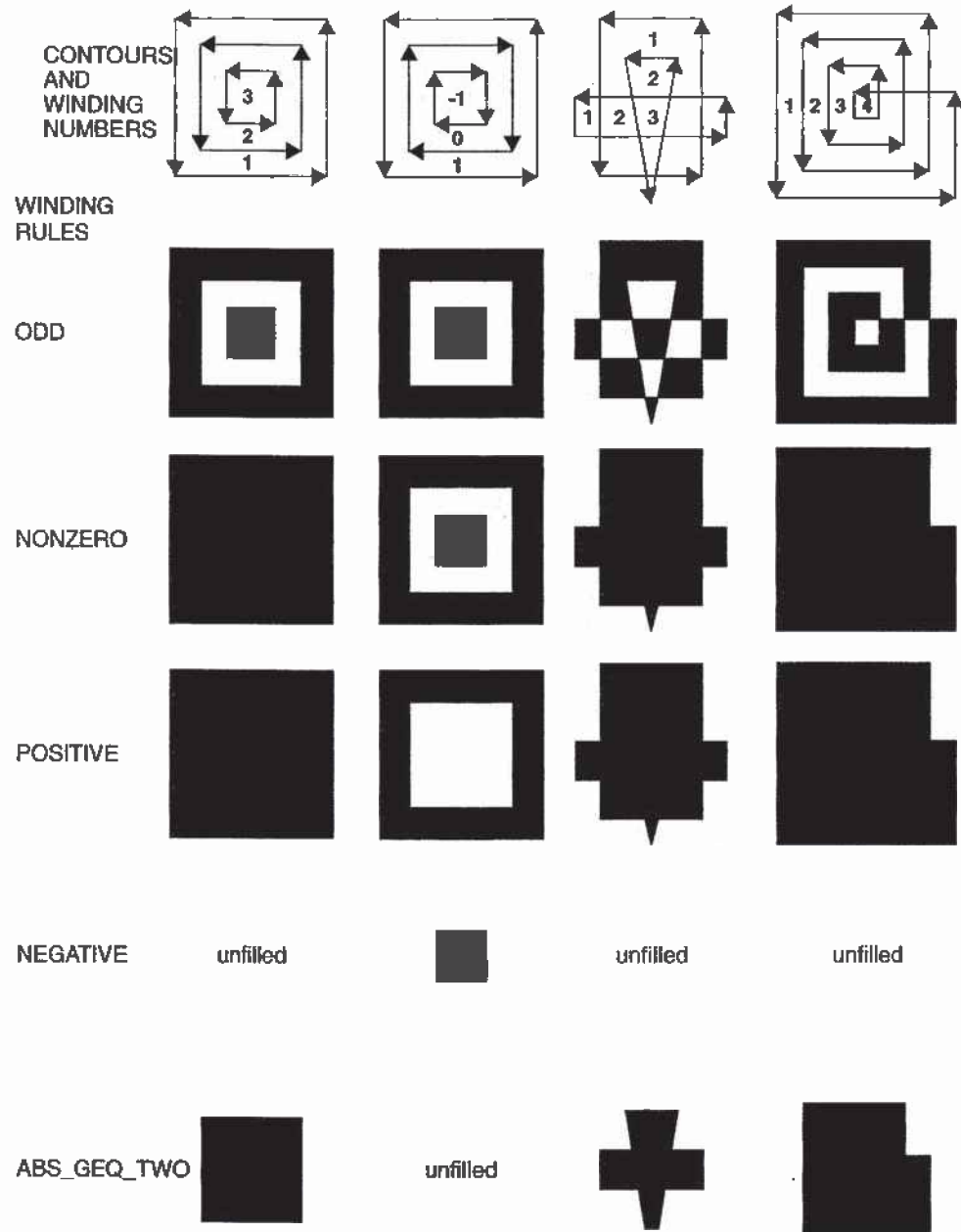


Figure 11-3 How Winding Rules Define Interiors

CSG Uses for Winding Rules

`GLU_TESS_WINDING_ODD` and `GLU_TESS_WINDING_NONZERO` are the most commonly used winding rules. They work for the most typical cases of shading.

The winding rules are also designed for computational solid geometry (CSG) operations. They make it easy to find the union, difference, or intersection (Boolean operations) of several contours.

First, assume that each contour is defined so that the winding number is zero for each exterior region and one for each interior region. (Each contour must not intersect itself.) Under this model, counterclockwise contours define the outer boundary of the polygon, and clockwise contours define holes. Contours may be nested, but a nested contour must be oriented oppositely from the contour that contains it.

If the original polygons do not satisfy this description, they can be converted to this form by first running the tessellator with the `GLU_TESS_BOUNDARY_ONLY` property turned on. This returns a list of contours satisfying the restriction just described. By creating two tessellator objects, the callbacks from one tessellator can be fed directly as input to the other.

Given two or more polygons of the preceding form, CSG operations can be implemented as follows.

- **UNION**—To calculate the union of several contours, draw all input contours as a single polygon. The winding number of each resulting region is the number of original polygons that cover it. The union can be extracted by using the `GLU_TESS_WINDING_NONZERO` or `GLU_TESS_WINDING_POSITIVE` winding rules. Note that with the nonzero winding rule, we would get the same result if all contour orientations were reversed.
- **INTERSECTION**—This only works for two contours at a time. Draw a single polygon using two contours. Extract the result using `GLU_TESS_WINDING_ABS_GEQ_TWO`.
- **DIFFERENCE**—Suppose you want to compute $A \text{ diff } (B \text{ union } C \text{ union } D)$. Draw a single polygon consisting of the unmodified contours from A , followed by the contours of B , C , and D , with their vertex order reversed. To extract the result, use the `GLU_TESS_WINDING_POSITIVE` winding rule. (If B , C , and D are the result of a `GLU_TESS_BOUNDARY_ONLY` operation, an alternative to reversing

the vertex order is to use `gluTessNormal()` to reverse the sign of the supplied normal.

Other Tessellation Property Routines

There are complementary routines, which work alongside `gluTessProperty()`. `gluGetTessProperty()` retrieves the current values of tessellator properties. If the tessellator is being used to generate wire frame outlines instead of filled polygons, `gluTessNormal()` can be used to determine the winding direction of the tessellated polygons.

```
void gluGetTessProperty(GLUtessellator *tessobj, GLenum property,  
                       GLdouble *value);
```

For the tessellation object *tessobj*, the current value of *property* is returned to *value*. Values for *property* and *value* are the same as for `gluTessProperty()`.

```
void gluTessNormal(GLUtessellator *tessobj, GLdouble x, GLdouble y,  
                  GLdouble z);
```

For the tessellation object *tessobj*, `gluTessNormal()` defines a normal vector, which controls the winding direction of generated polygons. Before tessellation, all input data is projected into a plane perpendicular to the normal. Then, all output triangles are oriented counterclockwise, with respect to the normal. (Clockwise orientation can be obtained by reversing the sign of the supplied normal.) The default normal is (0, 0, 0).

If you have some knowledge about the location and orientation of the input data, then using `gluTessNormal()` can increase the speed of the tessellation. For example, if you know that all polygons lie on the x-y plane, call `gluTessNormal(tessobj, 0, 0, 1)`.

The default normal is (0, 0, 0), and its effect is not immediately obvious. In this case, it is expected that the input data lies approximately in a plane, and a plane is fitted to the vertices, no matter how they are truly connected. The sign of the normal is chosen so that the sum of the signed areas of all input contours is nonnegative (where a counterclockwise contour has a positive area). Note that if the input data does not lie approximately in a plane, then projection perpendicular to the computed normal may substantially change the geometry.

Polygon Definition

After all the tessellation properties have been set and the callback actions have been registered, it is finally time to describe the vertices that compromise input contours and tessellate the polygons.

```
void gluTessBeginPolygon (GLUtesselator *tessobj, void *user_data);  
void gluTessEndPolygon (GLUtesselator *tessobj);
```

Begins and ends the specification of a polygon to be tessellated and associates a tessellation object, *tessobj*, with it. *user_data* points to a user-defined data structure, which is passed along all the `GLU_TESS_*_DATA` callback functions that have been bound.

Calls to `gluTessBeginPolygon()` and `gluTessEndPolygon()` surround the definition of one or more contours. When `gluTessEndPolygon()` is called, the tessellation algorithm is implemented, and the tessellated polygons are generated and rendered. The callback functions and tessellation properties that were bound and set to the tessellation object using `gluTessCallback()` and `gluTessProperty()` are used.

```
void gluTessBeginContour (GLUtesselator *tessobj);  
void gluTessEndContour (GLUtesselator *tessobj);
```

Begins and ends the specification of a closed contour, which is a portion of a polygon. A closed contour consists of zero or more calls to `gluTessVertex()`, which defines the vertices. The last vertex of each contour is automatically linked to the first.

In practice, a minimum of three vertices is needed for a meaningful contour.

```
void gluTessVertex (GLUtesselator *tessobj, GLdouble coords[3],  
                  void *vertex_data);
```

Specifies a vertex in the current contour for the tessellation object. *coords* contains the three-dimensional vertex coordinates, and *vertex_data* is a pointer that's sent to the callback associated with `GLU_TESS_VERTEX` or `GLU_TESS_VERTEX_DATA`. Typically, *vertex_data* contains vertex coordinates, surface normals, texture coordinates, color information, or whatever else the application may find useful.

In the program `tess.c`, a portion of which is shown in Example 11-3, two polygons are defined. One polygon is a rectangular contour with a triangular hole inside, and the other is a smooth-shaded, self-intersecting, five-pointed star. For efficiency, both polygons are stored in display lists. The first polygon consists of two contours; the outer one is wound counterclockwise, and the “hole” is wound clockwise. For the second polygon, the *star* array contains both the coordinate and color data, and its tessellation callback, `vertexCallback()`, uses both.

It is important that each vertex is in a different memory location because the vertex data is not copied by `gluTessVertex()`; only the pointer (*vertex_data*) is saved. A program that reuses the same memory for several vertices may not get the desired result.

Note: In `gluTessVertex()`, it may seem redundant to specify the vertex coordinate data twice, for both the *coords* and *vertex_data* parameters; however, both are necessary. *coords* refers only to the vertex coordinates. *vertex_data* uses the coordinate data, but may also use other information for each vertex.

Example 11-3 Polygon Definition: `tess.c`

```
GLdouble rect[4][3] = {50.0, 50.0, 0.0,
                       200.0, 50.0, 0.0,
                       200.0, 200.0, 0.0,
                       50.0, 200.0, 0.0};

GLdouble tri[3][3] = {75.0, 75.0, 0.0,
                     125.0, 175.0, 0.0,
                     175.0, 75.0, 0.0};

GLdouble star[5][6] = {250.0, 50.0, 0.0, 1.0, 0.0, 1.0,
                       325.0, 200.0, 0.0, 1.0, 1.0, 0.0,
                       400.0, 50.0, 0.0, 0.0, 1.0, 1.0,
                       250.0, 150.0, 0.0, 1.0, 0.0, 0.0,
                       400.0, 150.0, 0.0, 0.0, 1.0, 0.0};

startList = glGenLists(2);
tobj = gluNewTess();
gluTessCallback(tobj, GLU_TESS_VERTEX,
               (GLvoid (*) ()) &glVertex3dv);
gluTessCallback(tobj, GLU_TESS_BEGIN,
               (GLvoid (*) ()) &beginCallback);
gluTessCallback(tobj, GLU_TESS_END,
               (GLvoid (*) ()) &endCallback);
gluTessCallback(tobj, GLU_TESS_ERROR,
               (GLvoid (*) ()) &errorCallback);
```

```

glNewList(startList, GL_COMPILE);
glShadeModel(GL_FLAT);
gluTessBeginPolygon(tobj, NULL);
    gluTessBeginContour(tobj);
        gluTessVertex(tobj, rect[0], rect[0]);
        gluTessVertex(tobj, rect[1], rect[1]);
        gluTessVertex(tobj, rect[2], rect[2]);
        gluTessVertex(tobj, rect[3], rect[3]);
    gluTessEndContour(tobj);
    gluTessBeginContour(tobj);
        gluTessVertex(tobj, tri[0], tri[0]);
        gluTessVertex(tobj, tri[1], tri[1]);
        gluTessVertex(tobj, tri[2], tri[2]);
    gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
glEndList();

gluTessCallback(tobj, GLU_TESS_VERTEX,
                (GLvoid (*) ()) &vertexCallback);
gluTessCallback(tobj, GLU_TESS_BEGIN,
                (GLvoid (*) ()) &beginCallback);
gluTessCallback(tobj, GLU_TESS_END,
                (GLvoid (*) ()) &endCallback);
gluTessCallback(tobj, GLU_TESS_ERROR,
                (GLvoid (*) ()) &errorCallback);
gluTessCallback(tobj, GLU_TESS_COMBINE,
                (GLvoid (*) ()) &combineCallback);

glNewList(startList + 1, GL_COMPILE);
glShadeModel(GL_SMOOTH);
gluTessProperty(tobj, GLU_TESS_WINDING_RULE,
                GLU_TESS_WINDING_POSITIVE);
gluTessBeginPolygon(tobj, NULL);
    gluTessBeginContour(tobj);
        gluTessVertex(tobj, star[0], star[0]);
        gluTessVertex(tobj, star[1], star[1]);
        gluTessVertex(tobj, star[2], star[2]);
        gluTessVertex(tobj, star[3], star[3]);
        gluTessVertex(tobj, star[4], star[4]);
    gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
glEndList();

```

Deleting a Tessellator Object

If you no longer need a tessellation object, you can delete it and free all associated memory with `gluDeleteTess()`.

```
void gluDeleteTess(GLUtessellator *tessobj);
```

Deletes the specified tessellation object, *tessobj*, and frees all associated memory.

Tessellator Performance Tips

For best performance, remember these rules.

1. Cache the output of the tessellator in a display list or other user structure. To obtain the post-tessellation vertex coordinates, tessellate the polygons while in feedback mode. (See “Feedback” on page 491.)
2. Use `gluTessNormal()` to supply the polygon normal.
3. Use the same tessellator object to render many polygons rather than allocate a new tessellator for each one. (In a multithreaded, multiprocessor environment, you may get better performance using several tessellators.)

Describing GLU Errors

The GLU provides a routine for obtaining a descriptive string for an error code. This routine is not limited to tessellation but is also used for NURBS and quadrics errors, as well as errors in the base GL. (See “Error Handling” on page 501 for information about OpenGL’s error handling facility.)

Backward Compatibility

If you are using the 1.0 or 1.1 version of GLU, you have a much less powerful tessellator available. The 1.0/1.1 tessellator handles only simple nonconvex polygons or simple polygons containing holes. It does not properly tessellate intersecting contours (no COMBINE callback), nor process per-polygon data.

The 1.0/1.1 tessellator has some similarities to the current tessellator. `gluNewTess()` and `gluDeleteTess()` are used for both tessellators. The main vertex specification routine remains `gluTessVertex()`. The callback mechanism is controlled by `gluTessCallback()`, although there are only five callback functions that can be registered, a subset of the current twelve.

Here are the prototypes for the 1.0/1.1 tessellator. The 1.0/1.1 tessellator still works in GLU 1.2, but its use is no longer recommended.

```
void gluBeginPolygon(GLUtriangulatorObj *tessobj);
void gluNextContour(GLUtriangulatorObj *tessobj, GLenum type);
void gluEndPolygon(GLUtriangulatorObj *tessobj);
```

The outermost contour must be specified first, and it does not require an initial call to `gluNextContour()`. For polygons without holes, only one contour is defined, and `gluNextContour()` is not used. If a polygon has multiple contours (that is, holes or holes within holes), the contours are specified one after the other, each preceded by `gluNextContour()`. `gluTessVertex()` is called for each vertex of a contour.

For `gluNextContour()`, `type` can be `GLU_EXTERIOR`, `GLU_INTERIOR`, `GLU_CCW`, `GLU_CW`, or `GLU_UNKNOWN`. These serve only as hints to the tessellation. If you get them right, the tessellation might go faster. If you get them wrong, they're ignored, and the tessellation still works. For polygons with holes, one contour is the exterior contour and the other's interior. The first contour is assumed to be of type `GLU_EXTERIOR`. Choosing clockwise and counterclockwise orientation is arbitrary in three dimensions; however, there are two different orientations in any plane, and the `GLU_CCW` and `GLU_CW` types should be used consistently. Use `GLU_UNKNOWN` if you don't have a clue.

It is highly recommended that you convert GLU 1.0/1.1 code to the new tessellation interface for GLU 1.2 by following these steps.

1. Change references to the major data structure type from `GLUtriangulatorObj` to `GLUtesselator`. In GLU 1.2, `GLUtriangulatorObj` and `GLUtesselator` are defined to be the same type.
2. Convert `gluBeginPolygon()` to two commands: `gluTessBeginPolygon()` and `gluTessBeginContour()`. All contours must be explicitly started, including the first one.
3. Convert `gluNextContour()` to both `gluTessEndContour()` and `gluTessBeginContour()`. You have to end the previous contour before starting the next one.

-
4. Convert `gluEndPolygon()` to both `gluTessEndContour()` and `gluTessEndPolygon()`. The final contour must be closed.
 5. Change references to constants to `gluTessCallback()`. In GLU 1.2, `GLU_BEGIN`, `GLU_VERTEX`, `GLU_END`, `GLU_ERROR`, and `GLU_EDGE_FLAG` are defined as synonyms for `GLU_TESS_BEGIN`, `GLU_TESS_VERTEX`, `GLU_TESS_END`, `GLU_TESS_ERROR`, and `GLU_TESS_EDGE_FLAG`.

Quadrics: Rendering Spheres, Cylinders, and Disks

The base OpenGL library only provides support for modeling and rendering simple points, lines, and convex filled polygons. Neither 3D objects, nor commonly used 2D objects such as circles, are directly available.

Throughout this book, you've been using GLUT to create some 3D objects. The GLU also provides routines to model and render tessellated, polygonal approximations for a variety of 2D and 3D shapes (spheres, cylinders, disks, and parts of disks), which can be calculated with quadric equations. This includes routines to draw the quadric surfaces in a variety of styles and orientations. Quadric surfaces are defined by the following general quadratic equation:

$$a_1x^2 + a_2y^2 + a_3z^2 + a_4xy + a_5yz + a_6xz + a_7x + a_8y + a_9z + a_{10} = 0$$

(See David Rogers' *Procedural Elements for Computer Graphics*. New York, NY: McGraw-Hill Book Company, 1985.) Creating and rendering a quadric surface is similar to using the tessellator. To use a quadrics object, follow these steps.

1. To create a quadrics object, use `gluNewQuadric()`.
2. Specify the rendering attributes for the quadrics object (unless you're satisfied with the default values).
 - a. Use `gluQuadricOrientation()` to control the winding direction and differentiate the interior from the exterior.
 - b. Use `gluQuadricDrawStyle()` to choose between rendering the object as points, lines, or filled polygons.
 - c. For lit quadrics objects, use `gluQuadricNormals()` to specify one normal per vertex or one normal per face. The default is that no normals are generated at all.

- d. For textured quadrics objects, use `gluQuadricTexture()` if you want to generate texture coordinates.
3. Prepare for problems by registering an error-handling routine with `gluQuadricCallback()`. Then, if an error occurs during rendering, the routine you've specified is invoked.
4. Now invoke the rendering routine for the desired type of quadrics object: `gluSphere()`, `gluCylinder()`, `gluDisk()`, or `gluPartialDisk()`. For best performance for static data, encapsulate the quadrics object in a display list.
5. When you're completely finished with it, destroy this object with `gluDeleteQuadric()`. If you need to create another quadric, it's best to reuse your quadrics object.

Manage Quadrics Objects

A quadrics object consists of parameters, attributes, and callbacks that are stored in a data structure of type `GLUquadricObj`. A quadrics object may generate vertices, normals, texture coordinates, and other data, all of which may be used immediately or stored in a display list for later use. The following routines create, destroy, and report upon errors of a quadrics object.

```
GLUquadricObj* gluNewQuadric (void);
```

Creates a new quadrics object and returns a pointer to it. A null pointer is returned if the routine fails.

```
void gluDeleteQuadric (GLUquadricObj *qobj);
```

Destroys the quadrics object *qobj* and frees up any memory used by it.

```
void gluQuadricCallback (GLUquadricObj *qobj, GLenum which, void (*fn)());
```

Defines a function *fn* to be called in special circumstances. `GLU_ERROR` is the only legal value for *which*, so *fn* is called when an error occurs. If *fn* is `NULL`, any existing callback is erased.

For `GLU_ERROR`, *fn* is called with one parameter, which is the error code. `gluErrorString()` can be used to convert the error code into an ASCII string.

Control Quadrics Attributes

The following routines affect the kinds of data generated by the quadrics routines. Use these routines before you actually specify the primitives.

Example 11-4, `quadric.c`, on page 433, demonstrates changing the drawing style and the kind of normals generated as well as creating quadrics objects, error handling, and drawing the primitives.

```
void gluQuadricDrawStyle (GLUquadricObj *qobj, GLenum drawStyle);
```

For the quadrics object *qobj*, *drawStyle* controls the rendering style. Legal values for *drawStyle* are `GLU_POINT`, `GLU_LINE`, `GLU_SILHOUETTE`, and `GLU_FILL`.

`GLU_POINT` and `GLU_LINE` specify that primitives should be rendered as a point at every vertex or a line between each pair of connected vertices.

`GLU_SILHOUETTE` specifies that primitives are rendered as lines, except that edges separating coplanar faces are not drawn. This is most often used for `gluDisk()` and `gluPartialDisk()`.

`GLU_FILL` specifies rendering by filled polygons, where the polygons are drawn in a counterclockwise fashion with respect to their normals. This may be affected by `gluQuadricOrientation()`.

```
void gluQuadricOrientation (GLUquadricObj *qobj, GLenum orientation);
```

For the quadrics object *qobj*, *orientation* is either `GLU_OUTSIDE` (the default) or `GLU_INSIDE`, which controls the direction in which normals are pointing.

For `gluSphere()` and `gluCylinder()`, the definitions of outside and inside are obvious. For `gluDisk()` and `gluPartialDisk()`, the positive *z* side of the disk is considered to be outside.

```
void gluQuadricNormals (GLUquadricObj *qobj, GLenum normals);
```

For the quadrics object *qobj*, *normals* is one of `GLU_NONE` (the default), `GLU_FLAT`, or `GLU_SMOOTH`.

`gluQuadricNormals()` is used to specify when to generate normal vectors. `GLU_NONE` means that no normals are generated and is intended for use

without lighting. `GLU_FLAT` generates one normal for each facet, which is often best for lighting with flat shading. `GLU_SMOOTH` generates one normal for every vertex of the quadric, which is usually best for lighting with smooth shading.

```
void gluQuadricTexture (GLUquadricObj *qobj,  
                       GLboolean textureCoords);
```

For the quadrics object *qobj*, *textureCoords* is either `GL_FALSE` (the default) or `GL_TRUE`. If the value of *textureCoords* is `GL_TRUE`, then texture coordinates are generated for the quadrics object. The manner in which the texture coordinates are generated varies, depending upon the type of quadrics object rendered.

Quadrics Primitives

The following routines actually generate the vertices and other data that constitute a quadrics object. In each case, *qobj* refers to a quadrics object created by `gluNewQuadric()`.

```
void gluSphere (GLUquadricObj *qobj, GLdouble radius,  
               GLint slices, GLint stacks);
```

Draws a sphere of the given *radius*, centered around the origin, (0, 0, 0). The sphere is subdivided around the z axis into a number of *slices* (similar to longitude) and along the z axis into a number of *stacks* (latitude).

If texture coordinates are also generated by the quadrics facility, the *t* coordinate ranges from 0.0 at $z = -\text{radius}$ to 1.0 at $z = \text{radius}$, with *t* increasing linearly along longitudinal lines. Meanwhile, *s* ranges from 0.0 at the +y axis, to 0.25 at the +x axis, to 0.5 at the -y axis, to 0.75 at the -x axis, and back to 1.0 at the +y axis.

```
void gluCylinder (GLUquadricObj *qobj, GLdouble baseRadius,  
                 GLdouble topRadius, GLdouble height,  
                 GLint slices, GLint stacks);
```

Draws a cylinder oriented along the z axis, with the base of the cylinder at $z = 0$ and the top at $z = \text{height}$. Like a sphere, the cylinder is subdivided around the z axis into a number of *slices* and along the z axis into a number of *stacks*. *baseRadius* is the radius of the cylinder at $z = 0$.

topRadius is the radius of the cylinder at $z = \text{height}$. If *topRadius* is set to zero, then a cone is generated.

If texture coordinates are generated by the quadrics facility, then the t coordinate ranges linearly from 0.0 at $z = 0$ to 1.0 at $z = \text{height}$. The s texture coordinates are generated the same way as they are for a sphere.

Note: The cylinder is not closed at the top or bottom. The disks at the base and at the top are not drawn.

```
void gluDisk (GLUquadricObj *qobj, GLdouble innerRadius,
             GLdouble outerRadius, GLint slices, GLint rings);
```

Draws a disk on the $z = 0$ plane, with a radius of *outerRadius* and a concentric circular hole with a radius of *innerRadius*. If *innerRadius* is 0, then no hole is created. The disk is subdivided around the z axis into a number of *slices* (like slices of pizza) and also about the z axis into a number of concentric *rings*.

With respect to orientation, the $+z$ side of the disk is considered to be "outside"; that is, any normals generated point along the $+z$ axis. Otherwise, the normals point along the $-z$ axis.

If texture coordinates are generated by the quadrics facility, then the texture coordinates are generated linearly such that where $R = \text{outerRadius}$, the values for s and t at $(R, 0, 0)$ is $(1, 0.5)$, at $(0, R, 0)$ they are $(0.5, 1)$, at $(-R, 0, 0)$ they are $(0, 0.5)$, and at $(0, -R, 0)$ they are $(0.5, 0)$.

```
void gluPartialDisk (GLUquadricObj *qobj, GLdouble innerRadius,
                   GLdouble outerRadius, GLint slices, GLint rings,
                   GLdouble startAngle, GLdouble sweepAngle);
```

Draws a partial disk on the $z = 0$ plane. A partial disk is similar to a complete disk, in terms of *outerRadius*, *innerRadius*, *slices*, and *rings*. The difference is that only a portion of a partial disk is drawn, starting from *startAngle* through *startAngle + sweepAngle* (where *startAngle* and *sweepAngle* are measured in degrees, where 0 degrees is along the $+y$ axis, 90 degrees along the $+x$ axis, 180 along the $-y$ axis, and 270 along the $-x$ axis).

A partial disk handles orientation and texture coordinates in the same way as a complete disk.

Note: For all quadrics objects, it's better to use the **Radius, height*, and similar arguments to scale them rather than the `glScale*()` command so that the unit-length normals that are generated don't have to be renormalized. Set the *rings* and *stacks* arguments to values other than one to force lighting calculations at a finer granularity, especially if the material specularity is high.

Example 11-4 shows each of the quadrics primitives being drawn, as well as the effects of different drawing styles.

Example 11-4 Quadrics Objects: `quadric.c`

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdio.h>
#include <stdlib.h>

GLuint startList;

void errorCallback(GLenum errorCode)
{
    const GLubyte *estring;

    estrings = gluErrorString(errorCode);
    fprintf(stderr, "Quadric Error: %s\n", estrings);
    exit(0);
}

void init(void)
{
    GLUquadricObj *qobj;
    GLfloat mat_ambient[] = { 0.5, 0.5, 0.5, 1.0 };
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_shininess[] = { 50.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    GLfloat model_ambient[] = { 0.5, 0.5, 0.5, 1.0 };

    glClearColor(0.0, 0.0, 0.0, 0.0);

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, model_ambient);

    glEnable(GL_LIGHTING);
}
```

```

glEnable(GL_LIGHT0);
glEnable(GL_DEPTH_TEST);

/* Create 4 display lists, each with a different quadric object.
 * Different drawing styles and surface normal specifications
 * are demonstrated.
 */
startList = glGenLists(4);
qobj = gluNewQuadric();
gluQuadricCallback(qobj, GLU_ERROR, errorCallback);

gluQuadricDrawStyle(qobj, GLU_FILL); /* smooth shaded */
gluQuadricNormals(qobj, GLU_SMOOTH);
glNewList(startList, GL_COMPILE);
    gluSphere(qobj, 0.75, 15, 10);
glEndList();

gluQuadricDrawStyle(qobj, GLU_FILL); /* flat shaded */
gluQuadricNormals(qobj, GLU_FLAT);
glNewList(startList+1, GL_COMPILE);
    gluCylinder(qobj, 0.5, 0.3, 1.0, 15, 5);
glEndList();

gluQuadricDrawStyle(qobj, GLU_LINE); /* wireframe */
gluQuadricNormals(qobj, GLU_NONE);
glNewList(startList+2, GL_COMPILE);
    gluDisk(qobj, 0.25, 1.0, 20, 4);
glEndList();

gluQuadricDrawStyle(qobj, GLU_SILHOUETTE);
gluQuadricNormals(qobj, GLU_NONE);
glNewList(startList+3, GL_COMPILE);
    gluPartialDisk(qobj, 0.0, 1.0, 20, 4, 0.0, 225.0);
glEndList();
}

void display(void)
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();

    glEnable(GL_LIGHTING);
    glShadeModel (GL_SMOOTH);
    glTranslatef(-1.0, -1.0, 0.0);
    glCallList(startList);

    glShadeModel (GL_FLAT);

```

```

    glTranslatef(0.0, 2.0, 0.0);
    glPushMatrix();
    glRotatef(300.0, 1.0, 0.0, 0.0);
    glCallList(startList+1);
    glPopMatrix();

    glDisable(GL_LIGHTING);
    glColor3f(0.0, 1.0, 1.0);
    glTranslatef(2.0, -2.0, 0.0);
    glCallList(startList+2);

    glColor3f(1.0, 1.0, 0.0);
    glTranslatef(0.0, 2.0, 0.0);
    glCallList(startList+3);

    glPopMatrix();
    glFlush();
}

void reshape (int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-2.5, 2.5, -2.5*(GLfloat)h/(GLfloat)w,
                2.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
        glOrtho(-2.5*(GLfloat)w/(GLfloat)h,
                2.5*(GLfloat)w/(GLfloat)h, -2.5, 2.5, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 27:
            exit(0);
            break;
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);

```

```
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

Evaluators and NURBS



Chapter Objectives

Advanced



After reading this chapter, you'll be able to do the following:

- Use OpenGL evaluator commands to draw basic curves and surfaces
- Use the GLU's higher-level NURBS facility to draw more complex curves and surfaces

Note that this chapter presumes a number of prerequisites; they're listed in "Prerequisites" on page 439.

At the lowest level, graphics hardware draws points, line segments, and polygons, which are usually triangles and quadrilaterals. Smooth curves and surfaces are drawn by approximating them with large numbers of small line segments or polygons. However, many useful curves and surfaces can be described mathematically by a small number of parameters such as a few *control points*. Saving the 16 control points for a surface requires much less storage than saving 1000 triangles together with the normal vector information at each vertex. In addition, the 1000 triangles only approximate the true surface, but the control points accurately describe the real surface.

Evaluators provide a way to specify points on a curve or surface (or part of one) using only the control points. The curve or surface can then be rendered at any precision. In addition, normal vectors can be calculated for surfaces automatically. You can use the points generated by an evaluator in many ways—to draw dots where the surface would be, to draw a wireframe version of the surface, or to draw a fully lighted, shaded, and even textured version.

You can use evaluators to describe any polynomial or rational polynomial splines or surfaces of any degree. These include almost all splines and spline surfaces in use today, including B-splines, NURBS (Non-Uniform Rational B-Spline) surfaces, Bézier curves and surfaces, and Hermite splines. Since evaluators provide only a low-level description of the points on a curve or surface, they're typically used underneath utility libraries that provide a higher-level interface to the programmer. The GLU's NURBS facility is such a higher-level interface—the NURBS routines encapsulate lots of complicated code. Much of the final rendering is done with evaluators, but for some conditions (trimming curves, for example) the NURBS routines use planar polygons for rendering.

This chapter contains the following major sections.

- “Prerequisites” on page 439 discusses what knowledge is assumed for this chapter. It also gives several references where you can obtain this information.
- “Evaluators” on page 440 explains how evaluators work and how to control them using the appropriate OpenGL commands.
- “The GLU NURBS Interface” on page 455 describes the GLU routines for creating NURBS surfaces.

Prerequisites

Evaluators make splines and surfaces that are based on a Bézier (or Bernstein) basis. The defining formulas for the functions in this basis are given in this chapter, but the discussion doesn't include derivations or even lists of all their interesting mathematical properties. If you want to use evaluators to draw curves and surfaces using other bases, you must know how to convert your basis to a Bézier basis. In addition, when you render a Bézier surface or part of it using evaluators, you need to determine the granularity of your subdivision. Your decision needs to take into account the trade-off between high-quality (highly subdivided) images and high speed. Determining an appropriate subdivision strategy can be quite complicated—too complicated to be discussed here.

Similarly, a complete discussion of NURBS is beyond the scope of this book. The GLU NURBS interface is documented here, and programming examples are provided for readers who already understand the subject. In what follows, you already should know about NURBS control points, knot sequences, and trimming curves.

If you lack some of these prerequisites, the following references will help.

- Farin, Gerald E., *Curves and Surfaces for Computer-Aided Geometric Design, Fourth Edition*. San Diego, CA: Academic Press, 1996.
- Farin, Gerald E., *NURB Curves and Surfaces: from Projective Geometry to Practical Use*. Wellesley, MA: A. K. Peters Ltd., 1995.
- Farin, Gerald E., editor, *NURBS for Curve and Surface Design*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1991.
- Hoschek, Josef and Dieter Lasser, *Fundamentals of Computer Aided Geometric Design*. Wellesley, MA: A. K. Peters Ltd., 1993.
- Piegl, Les and Wayne Tiller, *The NURBS Book*. New York, NY: Springer-Verlag, 1995.

Note: Some terms used in this chapter might have slightly different meanings in other books on spline curves and surfaces, since there isn't total agreement among the practitioners of this art. Generally, the OpenGL meanings are a bit more restrictive. For example, OpenGL evaluators always use Bézier bases; in other contexts, evaluators might refer to the same concept, but with an arbitrary basis.

Evaluators

A Bézier curve is a vector-valued function of one variable

$$C(u) = [X(u) \ Y(u) \ Z(u)]$$

where u varies in some domain (say $[0,1]$). A Bézier surface patch is a vector-valued function of two variables

$$S(u,v) = [X(u,v) \ Y(u,v) \ Z(u,v)]$$

where u and v can both vary in some domain. The range isn't necessarily three-dimensional as shown here. You might want two-dimensional output for curves on a plane or texture coordinates, or you might want four-dimensional output to specify RGBA information. Even one-dimensional output may make sense for gray levels.

For each u (or u and v , in the case of a surface), the formula for $C()$ (or $S()$) calculates a point on the curve (or surface). To use an evaluator, first define the function $C()$ or $S()$, enable it, and then use the `glEvalCoord1()` or `glEvalCoord2()` command instead of `glVertex*()`. This way, the curve or surface vertices can be used like any other vertices—to form points or lines, for example. In addition, other commands automatically generate series of vertices that produce a regular mesh uniformly spaced in u (or in u and v). One- and two-dimensional evaluators are similar, but the description is somewhat simpler in one dimension, so that case is discussed first.

One-Dimensional Evaluators

This section presents an example of using one-dimensional evaluators to draw a curve. It then describes the commands and equations that control evaluators.

One-Dimensional Example: A Simple Bézier Curve

The program shown in Example 12-1 draws a cubic Bézier curve using four control points, as shown in Figure 12-1.

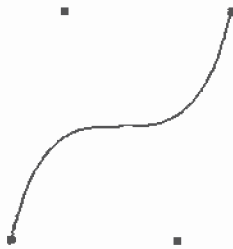


Figure 12-1 Bézier Curve

Example 12-1 Bézier Curve with Four Control Points: bezcurve.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <GL/glut.h>

GLfloat ctrlpnts[4][3] = {
    { -4.0, -4.0, 0.0}, { -2.0, 4.0, 0.0},
    { 2.0, -4.0, 0.0}, { 4.0, 4.0, 0.0}};

void init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_FLAT);
    glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpnts[0][0]);
    glEnable(GL_MAP1_VERTEX_3);
}

void display(void)
{
    int i;

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glBegin(GL_LINE_STRIP);
        for (i = 0; i <= 30; i++)
            glEvalCoord1f((GLfloat) i/30.0);
    glEnd();
    /* The following code displays the control points as dots. */
    glPointSize(5.0);
    glColor3f(1.0, 1.0, 0.0);
    glBegin(GL_POINTS);
        for (i = 0; i < 4; i++)
            glVertex3fv(&ctrlpnts[i][0]);
}
```

```

        glEnd();
        glFlush();
    }

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
                5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
    else
        glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
                5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}

```

A cubic Bézier curve is described by four control points, which appear in this example in the *ctrlpoints* array. This array is one of the arguments to *glMap1f*. All the arguments for this command are as follows:

GL_MAP1_VERTEX_3

Three-dimensional control points are provided and three-dimensional vertices are produced

0.0 Low value of parameter *u*

1.0 High value of parameter *u*

3 The number of floating-point values to advance in the data between one control point and the next

4 The order of the spline, which is the degree+1;
 in this case, the degree is 3 (since this is a cubic curve)

`&ctrlpoints[0][0]` Pointer to the first control point's data

Note that the second and third arguments control the parameterization of the curve—as the variable u ranges from 0.0 to 1.0, the curve goes from one end to the other. The call to `glEnable()` enables the one-dimensional evaluator for three-dimensional vertices.

The curve is drawn in the routine `display()` between the `glBegin()` and `glEnd()` calls. Since the evaluator is enabled, the command `glEvalCoord1f()` is just like issuing a `glVertex()` command with the coordinates of a vertex on the curve corresponding to the input parameter u .

Defining and Evaluating a One-Dimensional Evaluator

The Bernstein polynomial of degree n (or order $n+1$) is given by

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

If P_i represents a set of control points (one-, two-, three-, or even four-dimensional), then the equation

$$C(u) = \sum_{i=0}^n B_i^n(u) P_i$$

represents a Bézier curve as u varies from 0.0 to 1.0. To represent the same curve but allowing u to vary between u_1 and u_2 instead of 0.0 and 1.0, evaluate

$$C\left(\frac{u-u_1}{u_2-u_1}\right)$$

The command `glMap1()` defines a one-dimensional evaluator that uses these equations.

```
void glMap1(fd)(GLenum target, TYPE u1, TYPE u2, GLint stride,  
                  GLint order, const TYPE *points);
```

Defines a one-dimensional evaluator. The *target* parameter specifies what the control points represent, as shown in Table 12-1, and therefore how many values need to be supplied in *points*. The points can represent vertices, RGBA color data, normal vectors, or texture coordinates. For

example, with `GL_MAP1_COLOR_4`, the evaluator generates color data along a curve in four-dimensional (RGBA) color space. You also use the parameter values listed in Table 12-1 to enable each defined evaluator before you invoke it. Pass the appropriate value to `glEnable()` or `glDisable()` to enable or disable the evaluator.

The second two parameters for `glMap1*`, `u1` and `u2`, indicate the range for the variable `u`. The variable `stride` is the number of single- or double-precision values (as appropriate) in each block of storage. Thus, it's an offset value between the beginning of one control point and the beginning of the next.

The `order` is the degree plus one, and it should agree with the number of control points. The `points` parameter points to the first coordinate of the first control point. Using the example data structure for `glMap1*`, use the following for `points`:

```
(GLfloat *)(&ctlpoints[0].x)
```

Parameter	Meaning
<code>GL_MAP1_VERTEX_3</code>	<code>x, y, z</code> vertex coordinates
<code>GL_MAP1_VERTEX_4</code>	<code>x, y, z, w</code> vertex coordinates
<code>GL_MAP1_INDEX</code>	color index
<code>GL_MAP1_COLOR_4</code>	R, G, B, A
<code>GL_MAP1_NORMAL</code>	normal coordinates
<code>GL_MAP1_TEXTURE_COORD_1</code>	<code>s</code> texture coordinates
<code>GL_MAP1_TEXTURE_COORD_2</code>	<code>s, t</code> texture coordinates
<code>GL_MAP1_TEXTURE_COORD_3</code>	<code>s, t, r</code> texture coordinates
<code>GL_MAP1_TEXTURE_COORD_4</code>	<code>s, t, r, q</code> texture coordinates

Table 12-1 Types of Control Points for `glMap1*`

More than one evaluator can be evaluated at a time. If you have both a `GL_MAP1_VERTEX_3` and a `GL_MAP1_COLOR_4` evaluator defined and enabled, for example, then calls to `glEvalCoord1()` generate both a position and a color. Only one of the vertex evaluators can be enabled at a time, although you might have defined both of them. Similarly, only one of the

texture evaluators can be active. Other than that, however, evaluators can be used to generate any combination of vertex, normal, color, and texture-coordinate data. If more than one evaluator of the same type is defined and enabled, the one of highest dimension is used.

Use `glEvalCoord1()` to evaluate a defined and enabled one-dimensional map.

```
void glEvalCoord1{fd}(TYPE u);  
void glEvalCoord1{fd}v(TYPE *u);
```

Causes evaluation of the enabled one-dimensional maps. The argument *u* is the value (or a pointer to the value, in the vector version of the command) of the domain coordinate.

For evaluated vertices, values for color, color index, normal vectors, and texture coordinates are generated by evaluation. Calls to `glEvalCoord*`() do not use the current values for color, color index, normal vectors, and texture coordinates. `glEvalCoord*`() also leaves those values unchanged.

Defining Evenly Spaced Coordinate Values in One Dimension

You can use `glEvalCoord1()` with any values for *u*, but by far the most common use is with evenly spaced values, as shown previously in Example 12-1. To obtain evenly spaced values, define a one-dimensional grid using `glMapGrid1*`() and then apply it using `glEvalMesh1()`.

```
void glMapGrid1{fd}(GLint n, TYPE u1, TYPE u2);
```

Defines a grid that goes from *u1* to *u2* in *n* steps, which are evenly spaced:

```
void glEvalMesh1(GGLenum mode, GLint p1, GLint p2);
```

Applies the currently defined map grid to all enabled evaluators. The *mode* can be either `GL_POINT` or `GL_LINE`, depending on whether you want to draw points or a connected line along the curve. The call has exactly the same effect as issuing a `glEvalCoord1()` for each of the steps between and including *p1* and *p2*, where $0 \leq p1, p2 \leq n$. Programmatically, it's equivalent to the following:

```
glBegin(GL_POINTS); /* OR glBegin(GL_LINE_STRIP); */  
for (i = p1; i <= p2; i++)  
    glEvalCoord1(u1 + i*(u2-u1)/n);  
glEnd();
```


except that if $i = 0$ or $i = n$, then `glEvalCoord1()` is called with exactly $u1$ or $u2$ as its parameter.

Two-Dimensional Evaluators

In two dimensions, everything is similar to the one-dimensional case, except that all the commands must take two parameters, u and v , into account. Points, colors, normals, or texture coordinates must be supplied over a surface instead of a curve. Mathematically, the definition of a Bézier surface patch is given by

$$S(u, v) = \sum_{i=0}^n \sum_{j=0}^m B_i^n(u) B_j^m(v) P_{ij}$$

where P_{ij} are a set of $m \times n$ control points, and the B_i are the same Bernstein polynomials for one dimension. As before, the P_{ij} can represent vertices, normals, colors, or texture coordinates.

The procedure to use two-dimensional evaluators is similar to the procedure for one dimension.

1. Define the evaluator(s) with `glMap2*()`.
2. Enable them by passing the appropriate value to `glEnable()`.
3. Invoke them either by calling `glEvalCoord2()` between a `glBegin()` and `glEnd()` pair or by specifying and then applying a mesh with `glMapGrid2()` and `glEvalMesh2()`.

Defining and Evaluating a Two-Dimensional Evaluator

Use `glMap2*()` and `glEvalCoord2*()` to define and then invoke a two-dimensional evaluator.

```
void glMap2{fd}(GLenum target, TYPE u1, TYPE u2, GLint ustride,
                GLint uorder, TYPE v1, TYPE v2, GLint vstride,
                GLint vorder, TYPE points);
```

The *target* parameter can have any of the values in Table 12-1, except that the string MAP1 is replaced with MAP2. As before, these values are also used with `glEnable()` to enable the corresponding evaluator. Minimum and maximum values for both u and v are provided as $u1$, $u2$, $v1$, and $v2$. The parameters *ustride* and *vstride* indicate the number of single- or double-precision values (as appropriate) between independent settings

for these values, allowing users to select a subrectangle of control points out of a much larger array. For example, if the data appears in the form

```
GLfloat ctlpoints[100][100][3];
```

and you want to use the 4x4 subset beginning at `ctlpoints[20][30]`, choose *ustride* to be 100*3 and *vstride* to be 3. The starting point, *points*, should be set to `&ctlpoints[20][30][0]`. Finally, the order parameters, *uorder* and *vorder*, can be different, allowing patches that are cubic in one direction and quadratic in the other, for example.

```
void glEvalCoord2{fd}(TYPE u, TYPE v);  
void glEvalCoord2{fd}v(TYPE *values);
```

Causes evaluation of the enabled two-dimensional maps. The arguments *u* and *v* are the values (or a pointer to the *values* *u* and *v*, in the vector version of the command) for the domain coordinates. If either of the vertex evaluators is enabled (`GL_MAP2_VERTEX_3` or `GL_MAP2_VERTEX_4`), then the normal to the surface is computed analytically. This normal is associated with the generated vertex if automatic normal generation has been enabled by passing `GL_AUTO_NORMAL` to `glEnable()`. If it's disabled, the corresponding enabled normal map is used to produce a normal. If no such map exists, the current normal is used.

Two-Dimensional Example: A Bézier Surface

Example 12-2 draws a wireframe Bézier surface using evaluators, as shown in Figure 12-2. In this example, the surface is drawn with nine curved lines in each direction. Each curve is drawn as 30 segments. To get the whole program, add the `reshape()` and `main()` routines from Example 12-1.

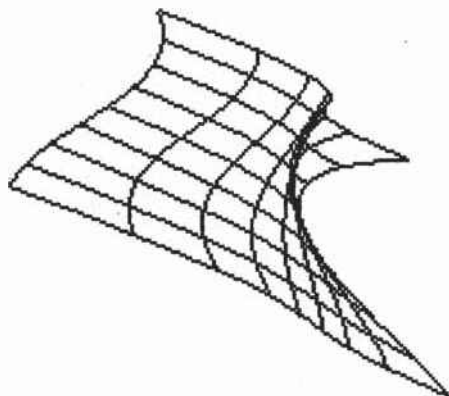


Figure 12-2 Bézier Surface

Example 12-2 Bézier Surface: bezsurf.c

```

#include <GL/gl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <GL/glut.h>

GLfloat ctrlpoints[4][4][3] = {
    {{-1.5, -1.5, 4.0}, {-0.5, -1.5, 2.0},
     {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
    {{-1.5, -0.5, 1.0}, {-0.5, -0.5, 3.0},
     {0.5, -0.5, 0.0}, {1.5, -0.5, -1.0}},
    {{-1.5, 0.5, 4.0}, {-0.5, 0.5, 0.0},
     {0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
    {{-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
     {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
};

void display(void)
{
    int i, j;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glPushMatrix ();
    glRotatef(85.0, 1.0, 1.0, 1.0);
    for (j = 0; j <= 8; j++) {
        glBegin(GL_LINE_STRIP);
        for (i = 0; i <= 30; i++)
            glEvalCoord2f((GLfloat)i/30.0, (GLfloat)j/8.0);
        glEnd();
    }
}

```

```

    glBegin(GL_LINE_STRIP);
    for (i = 0; i <= 30; i++)
        glEvalCoord2f((GLfloat)j/8.0, (GLfloat)i/30.0);
    glEnd();
}
glPopMatrix ();
glFlush();
}

void init(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
            0, 1, 12, 4, &ctrlpoints[0][0][0]);
    glEnable(GL_MAP2_VERTEX_3);
    glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
    glEnable(GL_DEPTH_TEST);
    glShadeModel (GL_FLAT);
}

```

Defining Evenly Spaced Coordinate Values in Two Dimensions

In two dimensions, the `glMapGrid2*` and `glEvalMesh2` commands are similar to the one-dimensional versions, except that both u and v information must be included.

```

void glMapGrid2(fd)(GLint nu, TYPE u1, TYPE u2,
                  GLint nv, TYPE v1, TYPE v2);
void glEvalMesh2(GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2);

```

Defines a two-dimensional map grid that goes from $u1$ to $u2$ in nu evenly spaced steps, from $v1$ to $v2$ in nv steps (`glMapGrid2*`), and then applies this grid to all enabled evaluators (`glEvalMesh2`). The only significant difference from the one-dimensional versions of these two commands is that in `glEvalMesh2` the *mode* parameter can be `GL_FILL` as well as `GL_POINT` or `GL_LINE`. `GL_FILL` generates filled polygons using the quad-mesh primitive. Stated precisely, `glEvalMesh2` is nearly equivalent to one of the following three code fragments. (It's nearly equivalent because when i is equal to nu or j to nv , the parameter is exactly equal to $u2$ or $v2$, not to $u1+nu*(u2-u1)/nu$, which might be slightly different due to round-off error.)

```

glBegin(GL_POINTS);           /* mode == GL_POINT */
for (i = nu1; i <= nu2; i++)
    for (j = nv1; j <= nv2; j++)
        glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
glEnd();

OR

for (i = nu1; i <= nu2; i++) { /* mode == GL_LINE */
    glBegin(GL_LINES);
        for (j = nv1; j <= nv2; j++)
            glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
        glEnd();
    }
for (j = nv1; j <= nv2; j++) {
    glBegin(GL_LINES);
        for (i = nu1; i <= nu2; i++)
            glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
        glEnd();
    }

OR

for (i = nu1; i < nu2; i++) { /* mode == GL_FILL */
    glBegin(GL_QUAD_STRIP);
        for (j = nv1; j <= nv2; j++) {
            glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
            glEvalCoord2(u1 + (i+1)*(u2-u1)/nu, v1+j*(v2-v1)/nv);
        }
    glEnd();
}

```

Example 12-3 shows the differences necessary to draw the same Bézier surface as Example 12-2, but using `glMapGrid2()` and `glEvalMesh2()` to subdivide the square domain into a uniform 8x8 grid. This program also adds lighting and shading, as shown in Figure 12-3.

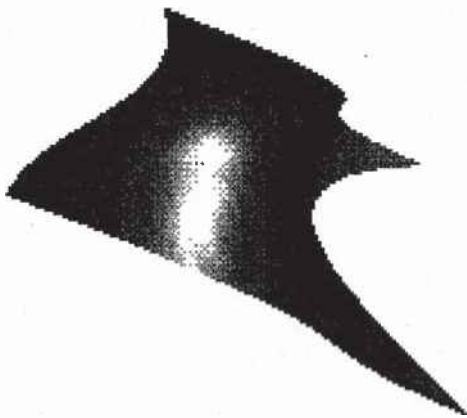


Figure 12-3 Lit, Shaded Bézier Surface Drawn with a Mesh

Example 12-3 Lit, Shaded Bézier Surface Using a Mesh: bezmesh.c

```
void initlights(void)
{
    GLfloat ambient[] = {0.2, 0.2, 0.2, 1.0};
    GLfloat position[] = {0.0, 0.0, 2.0, 1.0};
    GLfloat mat_diffuse[] = {0.6, 0.6, 0.6, 1.0};
    GLfloat mat_specular[] = {1.0, 1.0, 1.0, 1.0};
    GLfloat mat_shininess[] = {50.0};

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_POSITION, position);

    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(85.0, 1.0, 1.0, 1.0);
    glEvalMesh2(GL_FILL, 0, 20, 0, 20);
    glPopMatrix();
    glFlush();
}
```

```

void init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
           0, 1, 12, 4, &ctrlpoints[0][0][0]);
    glEnable(GL_MAP2_VERTEX_3);
    glEnable(GL_AUTO_NORMAL);
    glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
    initlights();
}

```

Using Evaluators for Textures

Example 12-4 enables two evaluators at the same time: The first generates three-dimensional points on the same Bézier surface as Example 12-3, and the second generates texture coordinates. In this case, the texture coordinates are the same as the u and v coordinates of the surface, but a special flat Bézier patch must be created to do this.

The flat patch is defined over a square with corners at (0, 0), (0, 1), (1, 0), and (1, 1); it generates (0, 0) at corner (0, 0), (0, 1) at corner (0, 1), and so on. Since it's of order two (linear degree plus one), evaluating this texture at the point (u , v) generates texture coordinates (s , t). It's enabled at the same time as the vertex evaluator, so both take effect when the surface is drawn. (See Plate 19.) If you want the texture to repeat three times in each direction, change every 1.0 in the array `texpts[][][]` to 3.0. Since the texture wraps in this example, the surface is rendered with nine copies of the texture map.

Example 12-4 Using Evaluators for Textures: texturesurf.c

```

#include <GL/gl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <GL/glut.h>
#include <math.h>

GLfloat ctrlpoints[4][4][3] = {
    {{ -1.5, -1.5, 4.0}, { -0.5, -1.5, 2.0},
      { 0.5, -1.5, -1.0}, { 1.5, -1.5, 2.0}},
    {{ -1.5, -0.5, 1.0}, { -0.5, -0.5, 3.0},
      { 0.5, -0.5, 0.0}, { 1.5, -0.5, -1.0}},
    {{ -1.5, 0.5, 4.0}, { -0.5, 0.5, 0.0},

```

```

        {0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
        {{ -1.5, 1.5, -2.0}, { -0.5, 1.5, -2.0},
        {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
    };
    GLfloat texpts[2][2][2] = {{{0.0, 0.0}, {0.0, 1.0}},
                                {{1.0, 0.0}, {1.0, 1.0}}};

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glEvalMesh2(GL_FILL, 0, 20, 0, 20);
    glFlush();
}
#define imageWidth 64
#define imageHeight 64
GLubyte image[3*imageWidth*imageHeight];

void makeImage(void)
{
    int i, j;
    float ti, tj;

    for (i = 0; i < imageWidth; i++) {
        ti = 2.0*3.14159265*i/imageWidth;
        for (j = 0; j < imageHeight; j++) {
            tj = 2.0*3.14159265*j/imageHeight;
            image[3*(imageHeight*i+j)] =
                (GLubyte) 127*(1.0+sin(ti));
            image[3*(imageHeight*i+j)+1] =
                (GLubyte) 127*(1.0+cos(2*tj));
            image[3*(imageHeight*i+j)+2] =
                (GLubyte) 127*(1.0+cos(ti+tj));
        }
    }
}

void init(void)
{
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
            0, 1, 12, 4, &ctrlpoints[0][0][0]);
    glMap2f(GL_MAP2_TEXTURE_COORD_2, 0, 1, 2, 2,
            0, 1, 4, 2, &texpts[0][0][0]);
    glEnable(GL_MAP2_TEXTURE_COORD_2);
    glEnable(GL_MAP2_VERTEX_3);
    glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
    makeImage();
}

```



```

glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);
glTexImage2D(GL_TEXTURE_2D, 0, 3, imageWidth, imageHeight, 0,
            GL_RGB, GL_UNSIGNED_BYTE, image);
glEnable(GL_TEXTURE_2D);
glEnable(GL_DEPTH_TEST);
glShadeModel (GL_FLAT);
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-4.0, 4.0, -4.0*(GLfloat)h/(GLfloat)w,
                4.0*(GLfloat)h/(GLfloat)w, -4.0, 4.0);
    else
        glOrtho(-4.0*(GLfloat)w/(GLfloat)h,
                4.0*(GLfloat)w/(GLfloat)h, -4.0, 4.0, -4.0, 4.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glRotatef(85.0, 1.0, 1.0, 1.0);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}

```

The GLU NURBS Interface

Although evaluators are the only OpenGL primitive available to draw curves and surfaces directly, and even though they can be implemented very efficiently in hardware, they're often accessed by applications through higher-level libraries. The GLU provides a NURBS (Non-Uniform Rational B-Spline) interface built on top of the OpenGL evaluator commands.

A Simple NURBS Example

If you understand NURBS, writing OpenGL code to manipulate NURBS curves and surfaces is relatively easy, even with lighting and texture mapping. Follow these steps to draw NURBS curves or untrimmed NURBS surfaces. (See "Trim a NURBS Surface" on page 464 for information about trimmed surfaces.)

1. If you intend to use lighting with a NURBS surface, call `glEnable()` with `GL_AUTO_NORMAL` to automatically generate surface normals. (Or you can calculate your own.)
2. Use `gluNewNurbsRenderer()` to create a pointer to a NURBS object, which is referred to when creating your NURBS curve or surface.
3. If desired, call `gluNurbsProperty()` to choose rendering values, such as the maximum size of lines or polygons that are used to render your NURBS object.
4. Call `gluNurbsCallback()` if you want to be notified when an error is encountered. (Error checking may slightly degrade performance but is still highly recommended.)
5. Start your curve or surface by calling `gluBeginCurve()` or `gluBeginSurface()`.
6. Generate and render your curve or surface. Call `gluNurbsCurve()` or `gluNurbsSurface()` at least once with the control points (rational or nonrational), knot sequence, and order of the polynomial basis function for your NURBS object. You might call these functions additional times to specify surface normals and/or texture coordinates.
7. Call `gluEndCurve()` or `gluEndSurface()` to complete the curve or surface.

Example 12-5 renders a NURBS surface in the shape of a symmetrical hill with control points ranging from -3.0 to 3.0 . The basis function is a cubic

B-spline, but the knot sequence is nonuniform, with a multiplicity of 4 at each endpoint, causing the basis function to behave like a Bézier curve in each direction. The surface is lighted, with a dark gray diffuse reflection and white specular highlights. Figure 12-4 shows the surface as a lit wireframe.

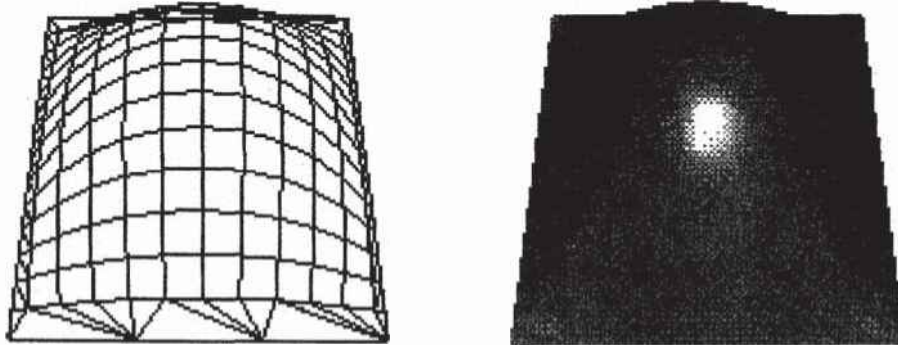


Figure 12-4 NURBS Surface

Example 12-5 NURBS Surface: surface.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>

GLfloat ctlpoints[4][4][3];
int showPoints = 0;

GLUnurbsObj *theNurb;

void init_surface(void)
{
    int u, v;
    for (u = 0; u < 4; u++) {
        for (v = 0; v < 4; v++) {
            ctlpoints[u][v][0] = 2.0*((GLfloat)u - 1.5);
            ctlpoints[u][v][1] = 2.0*((GLfloat)v - 1.5);

            if ( (u == 1 || u == 2) && (v == 1 || v == 2) )
                ctlpoints[u][v][2] = 3.0;
            else
                ctlpoints[u][v][2] = -3.0;
        }
    }
}
```

```

void nurbsError(GLenum errorCode)
{
    const GLubyte *estring;

    estring = gluErrorString(errorCode);
    fprintf (stderr, "Nurbs Error: %s\n", estring);
    exit (0);
}

void init(void)
{
    GLfloat mat_diffuse[] = { 0.7, 0.7, 0.7, 1.0 };
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_shininess[] = { 100.0 };

    glClearColor (0.0, 0.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);

    init_surface();

    theNurb = gluNewNurbsRenderer();
    gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 25.0);
    gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
    gluNurbsCallback(theNurb, GLU_ERROR,
                     (GLvoid (*)()) nurbsError);
}

void display(void)
{
    GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
    int i, j;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
    glRotatef(330.0, 1.,0.,0.);
    glScalef (0.5, 0.5, 0.5);

    gluBeginSurface(theNurb);

```

```

gluNurbsSurface(theNurb,
                8, knots, 8, knots,
                4 * 3, 3, &ctlpoints[0][0][0],
                4, 4, GL_MAP2_VERTEX_3);
gluEndSurface(theNurb);

if (showPoints) {
    glPointSize(5.0);
    glDisable(GL_LIGHTING);
    glColor3f(1.0, 1.0, 0.0);
    glBegin(GL_POINTS);
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            glVertex3f(ctlpoints[i][j][0],
                    ctlpoints[i][j][1], ctlpoints[i][j][2]);
        }
    }
    glEnd();
    glEnable(GL_LIGHTING);
}
glPopMatrix();
glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (45.0, (GLdouble)w/(GLdouble)h, 3.0, 8.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef (0.0, 0.0, -5.0);
}

```

```

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 'c':
        case 'C':
            showPoints = !showPoints;
            glutPostRedisplay();
            break;
        case 27:
            exit(0);
            break;
        default:
            break;
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutKeyboardFunc (keyboard);
    glutMainLoop();
    return 0;
}

```

Manage a NURBS Object

As shown in Example 12-5, `gluNewNurbsRenderer()` returns a new NURBS object, whose type is a pointer to a `GLUnurbsObj` structure. You must make this object before using any other NURBS routine. When you're done with a NURBS object, you may use `gluDeleteNurbsRenderer()` to free up the memory that was used.

GLUnurbsObj* gluNewNurbsRenderer (void);

Creates a new NURBS object, *obj*. Returns a pointer to the new object, or zero, if OpenGL cannot allocate memory for a new NURBS object.

```
void gluDeleteNurbsRenderer (GLUnurbsObj *nobj);
```

Destroys the NURBS object *nobj*.

Control NURBS Rendering Properties

A set of properties associated with a NURBS object affects the way the object is rendered. These properties include how the surface is rasterized (for example, filled or wireframe) and the precision of tessellation.

```
void gluNurbsProperty(GLUnurbsObj *nobj, GLenum property,  
                     GLfloat value);
```

Controls attributes of a NURBS object, *nobj*: The *property* argument specifies the property and can be `GLU_DISPLAY_MODE`, `GLU_CULLING`, `GLU_SAMPLING_METHOD`, `GLU_SAMPLING_TOLERANCE`, `GLU_PARAMETRIC_TOLERANCE`, `GLU_U_STEP`, `GLU_V_STEP`, or `GLU_AUTO_LOAD_MATRIX`. The *value* argument indicates what the property should be.

The default value for `GLU_DISPLAY_MODE` is `GLU_FILL`, which causes the surface to be rendered as polygons. If `GLU_OUTLINE_POLYGON` is used for the display-mode property, only the outlines of polygons created by tessellation are rendered. `GLU_OUTLINE_PATCH` renders the outlines of patches and trimming curves. (See "Create a NURBS Curve or Surface" on page 462.)

`GLU_CULLING` can speed up performance by not performing tessellation if the NURBS object falls completely outside the viewing volume; set this property to `GL_TRUE` to enable culling (the default is `GL_FALSE`).

Since a NURBS object is rendered as primitives, it's sampled at different values of its parameter(s) (*u* and *v*) and broken down into small line segments or polygons for rendering. If *property* is `GLU_SAMPLING_METHOD`, then *value* is set to one of `GLU_PATH_LENGTH` (which is the default), `GLU_PARAMETRIC_ERROR`, or `GLU_DOMAIN_DISTANCE`, which specifies how a NURBS curve or surface should be tessellated. When *value* is set to `GLU_PATH_LENGTH`, the surface is rendered so that the maximum length, in pixels, of the edges of tessellated polygons is no greater than what is specified by `GLU_SAMPLING_TOLERANCE`. When set to `GLU_PARAMETRIC_ERROR`, then the value specified by `GLU_PARAMETRIC_TOLERANCE` is the maximum distance, in pixels, between tessellated polygons and the

surfaces they approximate. When set to `GLU_DOMAIN_DISTANCE`, the application specifies, in parametric coordinates, how many sample points per unit length are taken in the *u* and *v* dimensions, using the values for `GLU_U_STEP` and `GLU_V_STEP`.

If *property* is `GLU_SAMPLING_TOLERANCE` and the sampling method is `GLU_PATH_LENGTH`, *value* controls the maximum length, in pixels, to use for tessellated polygons. The default value of 50.0 makes the largest sampled line segment or polygon edge 50.0 pixels long. If *property* is `GLU_PARAMETRIC_TOLERANCE` and the sampling method is `GLU_PARAMETRIC_ERROR`, *value* controls the maximum distance, in pixels, between the tessellated polygons and the surfaces they approximate. The default value for `GLU_PARAMETRIC_TOLERANCE` is 0.5, which makes the tessellated polygons within one-half pixel of the approximated surface. If the sampling method is `GLU_DOMAIN_DISTANCE` and *property* is either `GLU_U_STEP` or `GLU_V_STEP`, then *value* is the number of sample points per unit length taken along the *u* or *v* dimension, respectively, in parametric coordinates. The default for both `GLU_U_STEP` and `GLU_V_STEP` is 100.

The `GLU_AUTO_LOAD_MATRIX` property determines whether the projection matrix, modelview matrix, and viewport are downloaded from the OpenGL server (`GL_TRUE`, the default), or whether the application must supply these matrices with `gluLoadSamplingMatrices()` (`GL_FALSE`).

```
void gluLoadSamplingMatrices (GLUnurbsObj *nobj, const GLfloat  
modelMatrix[16], const GLfloat projMatrix[16], const GLint viewport[4]);
```

If the `GLU_AUTO_LOAD_MATRIX` is turned off, the modelview and projection matrices and the viewport specified in `gluLoadSamplingMatrices()` are used to compute sampling and culling matrices for each NURBS curve or surface.

If you need to query the current value for a NURBS property, you may use `gluGetNurbsProperty()`.

```
void gluGetNurbsProperty (GLUnurbsObj *nobj, GLenum property,  
                          GLfloat *value);
```

Given the *property* to be queried for the NURBS object *nobj*, return its current *value*.

Handle NURBS Errors

Since there are 37 different errors specific to NURBS functions, it's a good idea to register an error callback to let you know if you've stumbled into one of them. In Example 12-5, the callback function was registered with

```
gluNurbsCallback(theNurb, GLU_ERROR, (GLvoid (*)()) nurbsError);
```

```
void gluNurbsCallback (GLUnurbsObj *nobj, GLenum which,  
                      void (*fn)(GLenum errorCode));
```

which is the type of callback; it must be `GLU_ERROR`. When a NURBS function detects an error condition, *fn* is invoked with the error code as its only argument. *errorCode* is one of 37 error conditions, named `GLU_NURBS_ERROR1` through `GLU_NURBS_ERROR37`. Use `gluErrorString()` to describe the meaning of those error codes.

In Example 12-5, the `nurbsError()` routine was registered as the error callback function:

```
void nurbsError(GLenum errorCode)  
{  
    const GLubyte *estring;  
  
    estring = gluErrorString(errorCode);  
    fprintf (stderr, "Nurbs Error: %s\n", estring);  
    exit (0);  
}
```

Create a NURBS Curve or Surface

To render a NURBS surface, `gluNurbsSurface()` is bracketed by `gluBeginSurface()` and `gluEndSurface()`. The bracketing routines save and restore the evaluator state.

```
void gluBeginSurface (GLUnurbsObj *nobj);
void gluEndSurface (GLUnurbsObj *nobj);
```

After `gluBeginSurface()`, one or more calls to `gluNurbsSurface()` defines the attributes of the surface. Exactly one of these calls must have a surface type of `GL_MAP2_VERTEX_3` or `GL_MAP2_VERTEX_4` to generate vertices. Use `gluEndSurface()` to end the definition of a surface. Trimming of NURBS surfaces is also supported between `gluBeginSurface()` and `gluEndSurface()`. (See "Trim a NURBS Surface" on page 464.)

```
void gluNurbsSurface (GLUnurbsObj *nobj, GLint uknot_count,
                    GLfloat *uknot, GLint vknot_count, GLfloat *vknot,
                    GLint u_stride, GLint v_stride, GLfloat *ctlarray,
                    GLint uorder, GLint vorder, GLenum type);
```

Describes the vertices (or surface normals or texture coordinates) of a NURBS surface; *nobj*. Several of the values must be specified for both *u* and *v* parametric directions, such as the knot sequences (*uknot* and *vknot*), knot counts (*uknot_count* and *vknot_count*), and order of the polynomial (*uorder* and *vorder*) for the NURBS surface. Note that the number of control points isn't specified. Instead, it's derived by determining the number of control points along each parameter as the number of knots minus the order. Then, the number of control points for the surface is equal to the number of control points in each parametric direction, multiplied by one another. The *ctlarray* argument points to an array of control points.

The last parameter, *type*, is one of the two-dimensional evaluator types. Commonly, you might use `GL_MAP2_VERTEX_3` for nonrational or `GL_MAP2_VERTEX_4` for rational control points, respectively. You might also use other types, such as `GL_MAP2_TEXTURE_COORD_*` or `GL_MAP2_NORMAL` to calculate and assign texture coordinates or surface normals. For example, to create a lighted (with surface normals) and textured NURBS surface, you may need to call this sequence:

```
gluBeginSurface (nobj);
gluNurbsSurface (nobj, ..., GL_MAP2_TEXTURE_COORD_2);
gluNurbsSurface (nobj, ..., GL_MAP2_NORMAL);
gluNurbsSurface (nobj, ..., GL_MAP2_VERTEX_3);
gluEndSurface (nobj);
```

The *u_stride* and *v_stride* arguments represent the number of floating-point values between control points in each parametric direction. The evaluator type, as well as its order, affects the *u_stride* and *v_stride* values. In Example 12-5, *u_stride* is 12 ($4 * 3$) because there are three coordinates for each vertex (set by `GL_MAP2_VERTEX_3`) and four control points in the parametric *v* direction; *v_stride* is 3 because each vertex had three coordinates, and *v* control points are adjacent to one another.

Drawing a NURBS curve is similar to drawing a surface, except that all calculations are done with one parameter, *u*, rather than two. Also, for curves, `gluBeginCurve()` and `gluEndCurve()` are the bracketing routines.

```
void gluBeginCurve (GLUnurbsObj *nobj);  
void gluEndCurve (GLUnurbsObj *nobj);
```

After `gluBeginCurve()`, one or more calls to `gluNurbsCurve()` define the attributes of the surface. Exactly one of these calls must have a surface type of `GL_MAP1_VERTEX_3` or `GL_MAP1_VERTEX_4` to generate vertices. Use `gluEndCurve()` to end the definition of a surface.

```
void gluNurbsCurve (GLUnurbsObj *nobj, GLint uknot_count,  
                  GLfloat *uknot, GLint u_stride, GLfloat *ctlarray,  
                  GLint uorder, GLenum type);
```

Defines a NURBS curve for the object *nobj*. The arguments have the same meaning as those for `gluNurbsSurface()`. Note that this routine requires only one knot sequence and one declaration of the order of the NURBS object. If this curve is defined within a `gluBeginCurve()/gluEndCurve()` pair, then the type can be any of the valid one-dimensional evaluator types (such as `GL_MAP1_VERTEX_3` or `GL_MAP1_VERTEX_4`).

Trim a NURBS Surface

To create a trimmed NURBS surface with OpenGL, start as if you were creating an untrimmed surface. After calling `gluBeginSurface()` and `gluNurbsSurface()` but before calling `gluEndSurface()`, start a trim by calling `gluBeginTrim()`.

```
void gluBeginTrim (GLUnurbsObj *nobj);  
void gluEndTrim (GLUnurbsObj *nobj);
```

Marks the beginning and end of the definition of a trimming loop. A trimming loop is a set of oriented, trimming curve segments (forming a closed curve) that defines the boundaries of a NURBS surface.

You can create two kinds of trimming curves, a piecewise linear curve with `gluPwlCurve()` or a NURBS curve with `gluNurbsCurve()`. A piecewise linear curve doesn't look like what's conventionally called a curve, because it's a series of straight lines. A NURBS curve for trimming must lie within the unit square of parametric (u, v) space. The type for a NURBS trimming curve is usually `GLU_MAP1_TRIM2`. Less often, the type is `GLU_MAP1_TRIM3`, where the curve is described in a two-dimensional homogeneous space (u', v', w') by $(u, v) = (u'/w', v'/w')$.

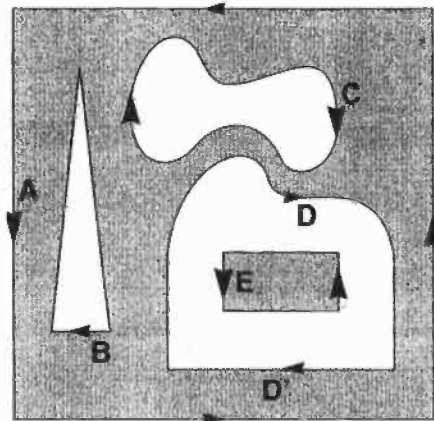
```
void gluPwlCurve (GLUnurbsObj *nobj, GLint count, GLfloat *array,  
                GLint stride, GLenum type);
```

Describes a piecewise linear trimming curve for the NURBS object *nobj*. There are *count* points on the curve, and they're given by *array*. The *type* can be either `GLU_MAP1_TRIM_2` (the most common) or `GLU_MAP1_TRIM_3` ((u, v, w) homogeneous parameter space). The *type* affects whether *stride*, the number of floating-point values to the next vertex, is 2 or 3.

You need to consider the orientation of trimming curves—that is, whether they're counterclockwise or clockwise—to make sure you include the desired part of the surface. If you imagine walking along a curve, everything to the left is included and everything to the right is trimmed away. For example, if your trim consists of a single counterclockwise loop, everything inside the loop is included. If the trim consists of two nonintersecting counterclockwise loops with nonintersecting interiors, everything inside either of them is included. If it consists of a counterclockwise loop with two clockwise loops inside it, the trimming region has two holes in it. The outermost trimming curve must be counterclockwise. Often, you run a trimming curve around the entire unit square to include everything within it, which is what you get by default by not specifying any trimming curves.

Trimming curves must be closed and nonintersecting. You can combine trimming curves, so long as the endpoints of the trimming curves meet to form a closed curve. You can nest curves, creating islands that float in space. Be sure to get the curve orientations right. For example, an error results if

you specify a trimming region with two counterclockwise curves, one enclosed within another: The region between the curves is to the left of one and to the right of the other, so it must be both included and excluded, which is impossible. Figure 12-5 illustrates a few valid possibilities.



```
gluBeginSurface();
gluNurbsSurface(...);
gluBeginTrim();
    gluPwlCurve(...); /* A */
gluEndTrim();
gluBeginTrim();
    gluPwlCurve(...); /* B */
gluEndTrim();
gluBeginTrim();
    gluNurbsCurve(...); /* C */
gluEndTrim();
gluBeginTrim();
    gluNurbsCurve(...); /* D */
    gluPwlCurve(...); /* D' */
gluEndTrim();
gluBeginTrim();
    gluPwlCurve(...); /* E */
gluEndTrim();
gluEndSurface();
```

Figure 12-5 Parametric Trimming Curves

Figure 12-6 shows the same small hill as in Figure 12-4, this time with a trimming curve that's a combination of a piecewise linear curve and a NURBS curve. The program that creates this figure is similar to that shown in Example 12-5; the differences are in the routines shown in Example 12-6.

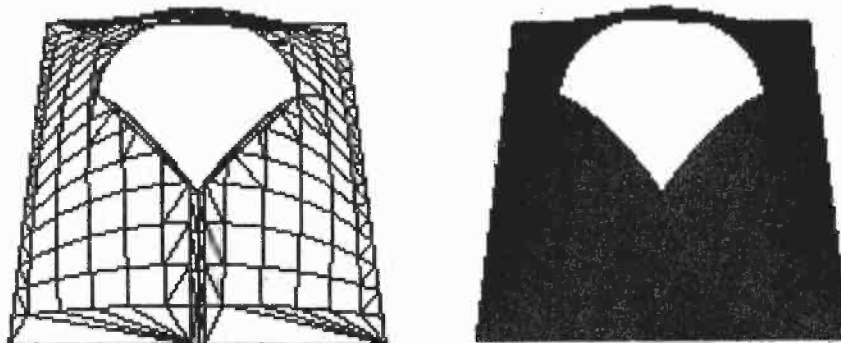


Figure 12-6 Trimmed NURBS Surface

Example 12-6 Trimming a NURBS Surface: trim.c

```
void display(void)
{
    GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
    GLfloat edgePt[5][2] = /* counter clockwise */
        {{0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0},
         {0.0, 0.0}};
    GLfloat curvePt[4][2] = /* clockwise */
        {{0.25, 0.5}, {0.25, 0.75}, {0.75, 0.75}, {0.75, 0.5}};
    GLfloat curveKnots[8] =
        {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
    GLfloat pwlPt[4][2] = /* clockwise */
        {{0.75, 0.5}, {0.5, 0.25}, {0.25, 0.5}};

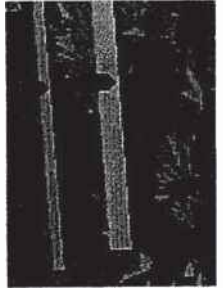
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(330.0, 1., 0., 0.);
    glScalef (0.5, 0.5, 0.5);

    gluBeginSurface(theNurb);
    gluNurbsSurface(theNurb, 8, knots, 8, knots,
        4 * 3, 3, &ctlpoints[0][0][0],
        4, 4, GL_MAP2_VERTEX_3);
    gluBeginTrim (theNurb);
        gluPwlCurve (theNurb, 5, &edgePt[0][0], 2,
            GLU_MAP1_TRIM_2);
    gluEndTrim (theNurb);
    gluBeginTrim (theNurb);
        gluNurbsCurve (theNurb, 8, curveKnots, 2,
            &curvePt[0][0], 4, GLU_MAP1_TRIM_2);
        gluPwlCurve (theNurb, 3, &pwlPt[0][0], 2,
            GLU_MAP1_TRIM_2);
    gluEndTrim (theNurb);
    gluEndSurface(theNurb);

    glPopMatrix();
    glFlush();
}
```

In Example 12-6, `gluBeginTrim()` and `gluEndTrim()` bracket each trimming curve. The first trim, with vertices defined by the array `edgePt[][]`, goes counterclockwise around the entire unit square of parametric space. This ensures that everything is drawn, provided it isn't removed by a clockwise trimming curve inside of it. The second trim is a combination of a NURBS trimming curve and a piecewise linear trimming curve. The NURBS curve ends at the points (0.9, 0.5) and (0.1, 0.5), where it is met by the piecewise linear curve, forming a closed clockwise curve.

Selection and Feedback



Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Create applications that allow the user to select a region of the screen or pick an object drawn on the screen
- Use the OpenGL feedback mode to obtain the results of rendering calculations

Some graphics applications simply draw static images of two- and three-dimensional objects. Other applications allow the user to identify objects on the screen and then to move, modify, delete, or otherwise manipulate those objects. OpenGL is designed to support exactly such interactive applications. Since objects drawn on the screen typically undergo multiple rotations, translations, and perspective transformations, it can be difficult for you to determine which object a user is selecting in a three-dimensional scene. To help you, OpenGL provides a selection mechanism that automatically tells you which objects are drawn inside a specified region of the window. You can use this mechanism together with a special utility routine to determine which object within the region the user is specifying, or *picking*, with the cursor.

Selection is actually a mode of operation for OpenGL; feedback is another such mode. In feedback mode, you use your graphics hardware and OpenGL to perform the usual rendering calculations. Instead of using the calculated results to draw an image on the screen, however, OpenGL returns (or feeds back) the drawing information to you. For example, if you want to draw three-dimensional objects on a plotter rather than the screen, you would draw the items in feedback mode, collect the drawing instructions, and then convert them to commands the plotter can understand.

In both selection and feedback modes, drawing information is returned to the application rather than being sent to the framebuffer, as it is in rendering mode. Thus, the screen remains frozen—no drawing occurs—while OpenGL is in selection or feedback mode. In these modes, the contents of the color, depth, stencil, and accumulation buffers are not affected. This chapter explains each of these modes in its own section:

- “Selection” on page 470 discusses how to use selection mode and related routines to allow a user of your application to pick an object drawn on the screen.
- “Feedback” on page 491 describes how to obtain information about what would be drawn on the screen and how that information is formatted.

Selection

Typically, when you’re planning to use OpenGL’s selection mechanism, you first draw your scene into the framebuffer, and then you enter selection mode and redraw the scene. However, once you’re in selection mode, the contents of the framebuffer don’t change until you exit selection mode.

When you exit selection mode, OpenGL returns a list of the primitives that intersect the viewing volume (remember that the viewing volume is defined by the current modelview and projection matrices and any additional clipping planes, as explained in Chapter 3.) Each primitive that intersects the viewing volume causes a selection *hit*. The list of primitives is actually returned as an array of integer-valued *names* and related data—the *hit records*—that correspond to the current contents of the *name stack*. You construct the name stack by loading names onto it as you issue primitive drawing commands while in selection mode. Thus, when the list of names is returned, you can use it to determine which primitives might have been selected on the screen by the user.

In addition to this selection mechanism, OpenGL provides a utility routine designed to simplify selection in some cases by restricting drawing to a small region of the viewport. Typically, you use this routine to determine which objects are drawn near the cursor, so that you can identify which object the user is picking. (You can also delimit a selection region by specifying additional clipping planes. Remember that these planes act in world space, not in screen space.) Since picking is a special case of selection, selection is described first in this chapter, and then picking.

The Basic Steps

To use the selection mechanism, you need to perform the following steps.

1. Specify the array to be used for the returned hit records with `glSelectBuffer()`.
2. Enter selection mode by specifying `GL_SELECT` with `glRenderMode()`.
3. Initialize the name stack using `glInitNames()` and `glPushName()`.
4. Define the viewing volume you want to use for selection. Usually this is different from the viewing volume you originally used to draw the scene, so you probably want to save and then restore the current transformation state with `glPushMatrix()` and `glPopMatrix()`.
5. Alternately issue primitive drawing commands and commands to manipulate the name stack so that each primitive of interest has an appropriate name assigned.
6. Exit selection mode and process the returned selection data (the hit records).

The following paragraphs describe `glSelectBuffer()` and `glRenderMode()`. In the next section, the commands to manipulate the name stack are described.

```
void glSelectBuffer(GLsizei size, GLuint *buffer);
```

Specifies the array to be used for the returned selection data. The *buffer* argument is a pointer to an array of unsigned integers into which the data is put, and *size* indicates the maximum number of values that can be stored in the array. You need to call `glSelectBuffer()` before entering selection mode.

```
GLint glRenderMode(GLenum mode);
```

Controls whether the application is in rendering, selection, or feedback mode. The *mode* argument can be one of `GL_RENDER` (the default), `GL_SELECT`, or `GL_FEEDBACK`. The application remains in a given mode until `glRenderMode()` is called again with a different argument. Before entering selection mode, `glSelectBuffer()` must be called to specify the selection array. Similarly, before entering feedback mode, `glFeedbackBuffer()` must be called to specify the feedback array. The return value for `glRenderMode()` has meaning if the current render mode (that is, not the *mode* parameter) is either `GL_SELECT` or `GL_FEEDBACK`. The return value is the number of selection hits or the number of values placed in the feedback array when either mode is exited; a negative value means that the selection or feedback array has overflowed. You can use `GL_RENDER_MODE` with `glGetIntegerv()` to obtain the current mode.

Creating the Name Stack

As mentioned in the previous section, the name stack forms the basis for the selection information that's returned to you. To create the name stack, first initialize it with `glInitNames()`, which simply clears the stack, and then add integer names to it while issuing corresponding drawing commands. As you might expect, the commands to manipulate the stack allow you to push a name onto it (`glPushName()`), pop a name off of it (`glPopName()`), and replace the name on the top of the stack with a different one (`glLoadName()`). Example 13-1 shows what your name-stack manipulation code might look like with these commands.

Example 13-1 Creating a Name Stack

```
glInitNames();
glPushName(0);

glPushMatrix(); /* save the current transformation state */

    /* create your desired viewing volume here */

    glLoadName(1);
    drawSomeObject();
    glLoadName(2);
    drawAnotherObject();
    glLoadName(3);
    drawYetAnotherObject();
    drawJustOneMoreObject();

glPopMatrix(); /* restore the previous transformation state*/
```

In this example, the first two objects to be drawn have their own names, and the third and fourth objects share a single name. With this setup, if either or both of the third and fourth objects causes a selection hit, only one hit record is returned to you. You can have multiple objects share the same name if you don't need to differentiate between them when processing the hit records.

void glInitNames(void);

Clears the name stack so that it's empty.

void glPushName(GLuint name);

Pushes *name* onto the name stack. Pushing a name beyond the capacity of the stack generates the error `GL_STACK_OVERFLOW`. The name stack's depth can vary among different OpenGL implementations, but it must be able to contain at least sixty-four names. You can use the parameter `GL_NAME_STACK_DEPTH` with `glGetIntegerv()` to obtain the depth of the name stack.

void glPopName(void);

Pops one name off the top of the name stack. Popping an empty stack generates the error `GL_STACK_UNDERFLOW`.

```
void glLoadName(GLuint name);
```

Replaces the value on the top of the name stack with *name*. If the stack is empty, which it is right after `glInitNames()` is called, `glLoadName()` generates the error `GL_INVALID_OPERATION`. To avoid this, if the stack is initially empty, call `glPushName()` at least once to put something on the name stack before calling `glLoadName()`.

Calls to `glPushName()`, `glPopName()`, and `glLoadName()` are ignored if you're not in selection mode. You might find that it simplifies your code to use these calls throughout your drawing code, and then use the same drawing code for both selection and normal rendering modes.

The Hit Record

In selection mode, a primitive that intersects the viewing volume causes a selection hit. Whenever a name-stack manipulation command is executed or `glRenderMode()` is called, OpenGL writes a hit record into the selection array if there's been a hit since the last time the stack was manipulated or `glRenderMode()` was called. With this process, objects that share the same name—for example, an object that's composed of more than one primitive—don't generate multiple hit records. Also, hit records aren't guaranteed to be written into the array until `glRenderMode()` is called.

Note: In addition to primitives, valid coordinates produced by `glRasterPos()` can cause a selection hit. Also, in the case of polygons, no hit occurs if the polygon would have been culled.

Each hit record consists of four items, in order.

- The number of names on the name stack when the hit occurred.
- Both the minimum and maximum window-coordinate *z* values of all vertices of the primitives that intersected the viewing volume since the last recorded hit. These two values, which lie in the range $[0,1]$, are each multiplied by $2^{32}-1$ and rounded to the nearest unsigned integer.
- The contents of the name stack at the time of the hit, with the bottommost element first.

When you enter selection mode, OpenGL initializes a pointer to the beginning of the selection array. Each time a hit record is written into the array, the pointer is updated accordingly. If writing a hit record would cause the number of values in the array to exceed the *size* argument specified with

`glSelectBuffer()`, OpenGL writes as much of the record as fits in the array and sets an overflow flag. When you exit selection mode with `glRenderMode()`, this command returns the number of hit records that were written (including a partial record if there was one), clears the name stack, resets the overflow flag, and resets the stack pointer. If the overflow flag had been set, the return value is `-1`.

A Selection Example

In Example 13-2, four triangles (green, red, and two yellow triangles, created by calling `drawTriangle()`) and a wireframe box representing the viewing volume (`drawViewVolume()`) are drawn to the screen. Then the triangles are rendered again (`selectObjects()`), but this time in selection mode. The corresponding hit records are processed in `processHits()`, and the selection array is printed out. The first triangle generates a hit, the second one doesn't, and the third and fourth ones together generate a single hit.

Example 13-2 Selection Example: `select.c`

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>

void drawTriangle (GLfloat x1, GLfloat y1, GLfloat x2,
                  GLfloat y2, GLfloat x3, GLfloat y3, GLfloat z)
{
    glBegin (GL_TRIANGLES);
    glVertex3f (x1, y1, z);
    glVertex3f (x2, y2, z);
    glVertex3f (x3, y3, z);
    glEnd ();
}

void drawViewVolume (GLfloat x1, GLfloat x2, GLfloat y1,
                    GLfloat y2, GLfloat z1, GLfloat z2)
{
    glColor3f (1.0, 1.0, 1.0);
    glBegin (GL_LINE_LOOP);
    glVertex3f (x1, y1, -z1);
    glVertex3f (x2, y1, -z1);
    glVertex3f (x2, y2, -z1);
```

```

    glVertex3f (x1, y2, -z1);
    glEnd ();

    glBegin (GL_LINE_LOOP);
    glVertex3f (x1, y1, -z2);
    glVertex3f (x2, y1, -z2);
    glVertex3f (x2, y2, -z2);
    glVertex3f (x1, y2, -z2);
    glEnd ();

    glBegin (GL_LINES); /* 4 lines */
    glVertex3f (x1, y1, -z1);
    glVertex3f (x1, y1, -z2);
    glVertex3f (x1, y2, -z1);
    glVertex3f (x1, y2, -z2);
    glVertex3f (x2, y1, -z1);
    glVertex3f (x2, y1, -z2);
    glVertex3f (x2, y2, -z1);
    glVertex3f (x2, y2, -z2);
    glEnd ();
}

void drawScene (void)
{
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective (40.0, 4.0/3.0, 1.0, 100.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
    gluLookAt (7.5, 7.5, 12.5, 2.5, 2.5, -5.0, 0.0, 1.0, 0.0);
    glColor3f (0.0, 1.0, 0.0); /* green triangle */
    drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -5.0);
    glColor3f (1.0, 0.0, 0.0); /* red triangle */
    drawTriangle (2.0, 7.0, 3.0, 7.0, 2.5, 8.0, -5.0);
    glColor3f (1.0, 1.0, 0.0); /* yellow triangles */
    drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, 0.0);
    drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -10.0);
    drawViewVolume (0.0, 5.0, 0.0, 5.0, 0.0, 10.0);
}

void processHits (GLint hits, GLuint buffer[])
{
    unsigned int i, j;
    GLuint names, *ptr;

    printf ("hits = %d\n", hits);
}

```

```

ptr = (GLuint *) buffer;
for (i = 0; i < hits; i++) { /* for each hit */
    names = *ptr;
    printf (" number of names for hit = %d\n", names); ptr++;
    printf("  z1 is %g;", (float) *ptr/0x7fffffff); ptr++;
    printf("  z2 is %g\n", (float) *ptr/0x7fffffff); ptr++;
    printf ("    the name is ");
    for (j = 0; j < names; j++) { /* for each name */
        printf ("%d ", *ptr); ptr++;
    }
    printf ("\n");
}
}

#define BUFSIZE 512

void selectObjects(void)
{
    GLuint selectBuf[BUFSIZE];
    GLint hits;

    glSelectBuffer (BUFSIZE, selectBuf);
    (void) glRenderMode (GL_SELECT);

    glInitNames ();
    glPushName (0);

    glPushMatrix ();
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glOrtho (0.0, 5.0, 0.0, 5.0, 0.0, 10.0);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
    glLoadName(1);
    drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -5.0);
    glLoadName(2);
    drawTriangle (2.0, 7.0, 3.0, 7.0, 2.5, 8.0, -5.0);
    glLoadName(3);
    drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, 0.0);
    drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -10.0);
    glPopMatrix ();
    glFlush ();

    hits = glRenderMode (GL_RENDER);
    processHits (hits, selectBuf);
}

```



```

void init (void)
{
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
}

void display(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    drawScene ();
    selectObjects ();
    glFlush();
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize (200, 200);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}

```

Picking

As an extension of the process described in the previous section, you can use selection mode to determine if objects are picked. To do this, you use a special picking matrix in conjunction with the projection matrix to restrict drawing to a small region of the viewport, typically near the cursor. Then you allow some form of input, such as clicking a mouse button, to initiate selection mode. With selection mode established and with the special picking matrix used, objects that are drawn near the cursor cause selection hits. Thus, during picking you're typically determining which objects are drawn near the cursor.

Picking is set up almost exactly like regular selection mode is, with the following major differences.

- Picking is usually triggered by an input device. In the following code examples, pressing the left mouse button invokes a function that performs picking.
- You use the utility routine `gluPickMatrix()` to multiply a special picking matrix onto the current projection matrix. This routine should be called prior to multiplying a standard projection matrix (such as `gluPerspective()` or `glOrtho()`). You'll probably want to save the contents of the projection matrix first, so the sequence of operations may look like this:

```
glMatrixMode (GL_PROJECTION);
glPushMatrix ();
glLoadIdentity ();
gluPickMatrix (...);
gluPerspective, glOrtho, gluOrtho2D, or glFrustum
/* ... draw scene for picking ; perform picking ... */
glPopMatrix();
```

Another completely different way to perform picking is described in “Object Selection Using the Back Buffer” on page 508. This technique uses color values to identify different components of an object.

```
void gluPickMatrix(GLdouble x, GLdouble y, GLdouble width,
                  GLdouble height, GLint viewport[4]);
```

Creates a projection matrix that restricts drawing to a small region of the viewport and multiplies that matrix onto the current matrix stack. The center of the picking region is (x, y) in window coordinates, typically the cursor location. *width* and *height* define the size of the picking region in screen coordinates. (You can think of the width and height as the sensitivity of the picking device.) *viewport[]* indicates the current viewport boundaries, which can be obtained by calling

```
glGetIntegerv(GL_VIEWPORT, GLint *viewport);
```

Advanced

The net result of the matrix created by `gluPickMatrix()` is to transform the clipping region into the unit cube $-1 \leq (x, y, z) \leq 1$ (or $-w \leq (wx, wy, wz) \leq w$). The picking matrix effectively performs an orthogonal transformation that maps a subregion of this unit cube to the unit cube. Since the transformation is arbitrary, you can make picking work for different sorts of regions—for example, for rotated rectangular portions of the window. In



certain situations, you might find it easier to specify additional clipping planes to define the picking region.

Example 13-3 illustrates simple picking. It also demonstrates how to use multiple names to identify different components of a primitive, in this case the row and column of a selected object. A 3×3 grid of squares is drawn, with each square a different color. The `board[3][3]` array maintains the current amount of blue for each square. When the left mouse button is pressed, the `pickSquares()` routine is called to identify which squares were picked by the mouse. Two names identify each square in the grid—one identifies the row, and the other the column. Also, when the left mouse button is pressed, the color of all squares under the cursor position changes.

Example 13-3 Picking Example: `picksquare.c`

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <stdio.h>
#include <GL/glut.h>

int board[3][3]; /* amount of color for each square */

/* Clear color value for every square on the board */
void init(void)
{
    int i, j;
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            board[i][j] = 0;
    glClearColor (0.0, 0.0, 0.0, 0.0);
}
```

```

void drawSquares(GLenum mode)
{
    GLuint i, j;
    for (i = 0; i < 3; i++) {
        if (mode == GL_SELECT)
            glLoadName (i);
        for (j = 0; j < 3; j++) {
            if (mode == GL_SELECT)
                glPushName (j);
            glColor3f ((GLfloat) i/3.0, (GLfloat) j/3.0,
                      (GLfloat) board[i][j]/3.0);
            glRecti (i, j, i+1, j+1);
            if (mode == GL_SELECT)
                glPopName ();
        }
    }
}

/* processHits prints out the contents of the
 * selection array.
 */
void processHits (GLint hits, GLuint buffer[])
{
    unsigned int i, j;
    GLuint ii, jj, names, *ptr;

    printf ("hits = %d\n", hits);
    ptr = (GLuint *) buffer;
    for (i = 0; i < hits; i++) { /* for each hit */
        names = *ptr;
        printf (" number of names for this hit = %d\n", names);
        ptr++;
        printf (" z1 is %g;", (float) *ptr/0x7fffffff); ptr++;
        printf (" z2 is %g\n", (float) *ptr/0x7fffffff); ptr++;
        printf (" names are ");
        for (j = 0; j < names; j++) { /* for each name */
            printf ("%d ", *ptr);
            if (j == 0) /* set row and column */
                ii = *ptr;
            else if (j == 1)
                jj = *ptr;
            ptr++;
        }
        printf ("\n");
        board[ii][jj] = (board[ii][jj] + 1) % 3;
    }
}

```

```

#define BUFSIZE 512

void pickSquares(int button, int state, int x, int y)
{
    GLuint selectBuf[BUFSIZE];
    GLint hits;
    GLint viewport[4];

    if (button != GLUT_LEFT_BUTTON || state != GLUT_DOWN)
        return;

    glGetIntegerv (GL_VIEWPORT, viewport);

    glSelectBuffer (BUFSIZE, selectBuf);
    (void) glRenderMode (GL_SELECT);

    glInitNames();
    glPushName(0);

    glMatrixMode (GL_PROJECTION);
    glPushMatrix ();
    glLoadIdentity ();
    /* create 5x5 pixel picking region near cursor location */
    gluPickMatrix ((GLdouble) x, (GLdouble) (viewport[3] - y),
                   5.0, 5.0, viewport);
    gluOrtho2D (0.0, 3.0, 0.0, 3.0);
    drawSquares (GL_SELECT);

    glMatrixMode (GL_PROJECTION);
    glPopMatrix ();
    glFlush ();

    hits = glRenderMode (GL_RENDER);
    processHits (hits, selectBuf);
    glutPostRedisplay();
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    drawSquares (GL_RENDER);
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, w, h);
}

```

```

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D (0.0, 3.0, 0.0, 3.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (100, 100);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutMouseFunc (pickSquares);
    glutReshapeFunc (reshape);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}

```

Picking with Multiple Names and a Hierarchical Model

Multiple names can also be used to choose parts of a hierarchical object in a scene. For example, if you were rendering an assembly line of automobiles, you might want the user to move the mouse to pick the third bolt on the left front tire of the third car in line. A different name can be used to identify each level of hierarchy: which car, which tire, and finally which bolt. As another example, one name can be used to describe a single molecule among other molecules, and additional names can differentiate individual atoms within that molecule.

Example 13-4 is a modification of Example 3-4 which draws an automobile with four identical wheels, each of which has five identical bolts. Code has been added to manipulate the name stack with the object hierarchy.

Example 13-4 Creating Multiple Names

```

draw_wheel_and_bolts()
{
    long i;

    draw_wheel_body();
    for (i = 0; i < 5; i++) {
        glPushMatrix();

```

```

        glRotate(72.0*i, 0.0, 0.0, 1.0);
        glTranslatef(3.0, 0.0, 0.0);
        glPushName(i);
            draw_bolt_body();
        glPopName();
        glPopMatrix();
    }
}

draw_body_and_wheel_andBolts()
(
    draw_car_body();
    glPushMatrix();
        glTranslate(40, 0, 20); /* first wheel position*/
        glPushName(1); /* name of wheel number 1 */
            draw_wheel_andBolts();
        glPopName();
    glPopMatrix();
    glPushMatrix();
        glTranslate(40, 0, -20); /* second wheel position */
        glPushName(2); /* name of wheel number 2 */
            draw_wheel_andBolts();
        glPopName();
    glPopMatrix();

    /* draw last two wheels similarly */
)

```

Example 13-5 uses the routines in Example 13-4 to draw three different cars, numbered 1, 2, and 3.

Example 13-5 Using Multiple Names

```

draw_three_cars()
(
    glInitNames();
    glPushMatrix();
        translate_to_first_car_position();
        glPushName(1);
            draw_body_and_wheel_andBolts();
        glPopName();
    glPopMatrix();

    glPushMatrix();
        translate_to_second_car_position();
        glPushName(2);
            draw_body_and_wheel_andBolts();
        glPopName();
)

```

```

glPopMatrix();

glPushMatrix();
    translate_to_third_car_position();
    glPushName(3);
        draw_body_and_wheel_and_bolts();
    glPopName();
glPopMatrix();
}

```

Assuming that picking is performed, the following are some possible name-stack return values and their interpretations. In these examples, at most one hit record is returned; also, *d1* and *d2* are depth values.

2 <i>d1 d2</i> 2 1	Car 2, wheel 1
1 <i>d1 d2</i> 3	Car 3 body
3 <i>d1 d2</i> 1 1 0	Bolt 0 on wheel 1 on car 1
empty	The pick was outside all cars

The last interpretation assumes that the bolt and wheel don't occupy the same picking region. A user might well pick both the wheel and the bolt, yielding two hits. If you receive multiple hits, you have to decide which hit to process, perhaps by using the depth values to determine which picked object is closest to the viewpoint. The use of depth values is explored further in the next section.

Picking and Depth Values

Example 13-6 demonstrates how to use depth values when picking to determine which object is picked. This program draws three overlapping rectangles in normal rendering mode. When the left mouse button is pressed, the `pickRects()` routine is called. This routine returns the cursor position, enters selection mode, initializes the name stack, and multiplies the picking matrix onto the stack before the orthographic projection matrix. A selection hit occurs for each rectangle the cursor is over when the left mouse button is clicked. Finally, the contents of the selection buffer are examined to identify which named objects were within the picking region near the cursor.

The rectangles in this program are drawn at different depth, or *z*, values. Since only one name is used to identify all three rectangles, only one hit can be recorded. However, if more than one rectangle is picked, that single hit has different minimum and maximum *z* values.

Example 13-6 Picking with Depth Values: pickdepth.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>

void init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
    glDepthRange(0.0, 1.0); /* The default z mapping */
}

void drawRects(GLenum mode)
{
    if (mode == GL_SELECT)
        glLoadName(1);
    glBegin(GL_QUADS);
    glColor3f(1.0, 1.0, 0.0);
    glVertex3i(2, 0, 0);
    glVertex3i(2, 6, 0);
    glVertex3i(6, 6, 0);
    glVertex3i(6, 0, 0);
    glEnd();
    if (mode == GL_SELECT)
        glLoadName(2);
    glBegin(GL_QUADS);
    glColor3f(0.0, 1.0, 1.0);
    glVertex3i(3, 2, -1);
    glVertex3i(3, 8, -1);
    glVertex3i(8, 8, -1);
    glVertex3i(8, 2, -1);
    glEnd();
    if (mode == GL_SELECT)
        glLoadName(3);
    glBegin(GL_QUADS);
    glColor3f(1.0, 0.0, 1.0);
    glVertex3i(0, 2, -2);
    glVertex3i(0, 7, -2);
    glVertex3i(5, 7, -2);
    glVertex3i(5, 2, -2);
    glEnd();
}
```

```

void processHits(GLint hits, GLuint buffer[])
{
    unsigned int i, j;
    GLuint names, *ptr;

    printf("hits = %d\n", hits);
    ptr = (GLuint *) buffer;
    for (i = 0; i < hits; i++) ( /* for each hit */
        names = *ptr;
        printf(" number of names for hit = %d\n", names); ptr++;
        printf(" z1 is %g;", (float) *ptr/0x7fffffff); ptr++;
        printf(" z2 is %g\n", (float) *ptr/0x7fffffff); ptr++;
        printf(" the name is ");
        for (j = 0; j < names; j++) ( /* for each name */
            printf("%d ", *ptr); ptr++;
        )
        printf("\n");
    )
}

#define BUFSIZE 512

void pickRects(int button, int state, int x, int y)
{
    GLuint selectBuf[BUFSIZE];
    GLint hits;
    GLint viewport[4];

    if (button != GLUT_LEFT_BUTTON || state != GLUT_DOWN)
        return;
    glGetIntegerv(GL_VIEWPORT, viewport);

    glSelectBuffer(BUFSIZE, selectBuf);
    (void) glRenderMode(GL_SELECT);

    glInitNames();
    glPushName(0);

    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    /* create 5x5 pixel picking region near cursor location */
    gluPickMatrix((GLdouble) x, (GLdouble) (viewport[3] - y),
        5.0, 5.0, viewport);
    glOrtho(0.0, 8.0, 0.0, 8.0, -0.5, 2.5);
    drawRects(GL_SELECT);
    glPopMatrix();
}

```

```

    glFlush();

    hits = glRenderMode(GL_RENDER);
    processHits(hits, selectBuf);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    drawRects(GL_RENDER);
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 8.0, 0.0, 8.0, -0.5, 2.5);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize (200, 200);
    glutInitWindowPosition (100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutMouseFunc(pickRects);
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}

```



Try This

- Modify Example 13-6 to add additional calls to `glPushName()` so that multiple names are on the stack when the selection hit occurs. What will the contents of the selection buffer be?
- By default, `glDepthRange()` sets the mapping of the z values to `[0.0,1.0]`. Try modifying the `glDepthRange()` values and see how it affects the z values that are returned in the selection array.

Hints for Writing a Program That Uses Selection

Most programs that allow a user to interactively edit some geometry provide a mechanism for the user to pick items or groups of items for editing. For two-dimensional drawing programs (for example, text editors, page-layout programs, and circuit-design programs), it might be easier to do your own picking calculations instead of using the OpenGL picking mechanism. Often, it's easy to find bounding boxes for two-dimensional objects and to organize them in some hierarchical data structure to speed up searches. For example, picking that uses the OpenGL style in a VLSI layout program containing millions of rectangles can be relatively slow. However, using simple bounding-box information when rectangles are typically aligned with the screen could make picking in such a program extremely fast. The code is probably simpler to write, too.

As another example, since only geometric objects cause hits, you might want to create your own method for picking text. Setting the current raster position is a geometric operation, but it effectively creates only a single pickable point at the current raster position, which is typically at the lower-left corner of the text. If your editor needs to manipulate individual characters within a text string, some other picking mechanism must be used. You could draw little rectangles around each character during picking mode, but it's almost certainly easier to handle text as a special case.

If you decide to use OpenGL picking, organize your program and its data structures so that it's easy to draw appropriate lists of objects in either selection or normal drawing mode. This way, when the user picks something, you can use the same data structures for the pick operation that you use to display the items on the screen. Also, consider whether you want to allow the user to select multiple objects. One way to do this is to store a bit for each item indicating whether it's selected (however, this method requires traversing your entire list of items to find the selected items). You might find it useful to maintain a list of pointers to selected items to speed up this search. It's probably a good idea to keep the selection bit for each item as well, since when you're drawing the entire picture, you might want to draw selected items differently (for example, in a different color or with a selection box around them). Finally, consider the selection user interface. You might want to allow the user to do the following:

- Select an item
- Sweep-select a group of items (see the next paragraphs for a description of this behavior)
- Add an item to the selection

-
- Add a sweep selection to the current selections
 - Delete an item from a selection
 - Choose a single item from a group of overlapping items

A typical solution for a two-dimensional drawing program might work as follows.

1. All selection is done by pointing with the mouse cursor and using the left mouse button. In what follows, *cursor* means the cursor tied to the mouse, and *button* means the left mouse button.
2. Clicking on an item selects it and deselects all other currently selected items. If the cursor is on top of multiple items, the smallest is selected. (In three dimensions, many other strategies work to disambiguate a selection.)
3. Clicking down where there is no item, holding the button down while dragging the cursor, and then releasing the button selects all the items in a screen-aligned rectangle whose corners are determined by the cursor positions when the button went down and where it came up. This is called a *sweep selection*. All items not in the swept-out region are deselected. (You must decide whether an item is selected only if it's completely within the sweep region, or if any part of it falls within the region. The completely within strategy usually works best.)
4. If the Shift key is held down and the user clicks on an item that isn't currently selected, that item is added to the selected list. If the clicked-upon item is selected, it's deleted from the selection list.
5. If a sweep selection is performed with the Shift key pressed, the items swept out are added to the current selection.
6. In an extremely cluttered region, it's often hard to do a sweep selection. When the button goes down, the cursor might lie on top of some item, and normally that item would be selected. You can make any operation a sweep selection, but a typical user interface interprets a button-down on an item plus a mouse motion as a select-plus-drag operation. To solve this problem, you can have an enforced sweep selection by holding down, say, the Alt key. With this, the following set of operations constitutes a sweep selection: Alt-button down, sweep, button up. Items under the cursor when the button goes down are ignored.
7. If the Shift key is held during this sweep selection, the items enclosed in the sweep region are added to the current selection.

-
8. Finally, if the user clicks on multiple items, select just one of them. If the cursor isn't moved (or maybe not moved more than a pixel), and the user clicks again in the same place, deselect the item originally selected, and select a different item under the cursor. Use repeated clicks at the same point to cycle through all the possibilities.

Different rules can apply in particular situations. In a text editor, you probably don't have to worry about characters on top of each other, and selections of multiple characters are always contiguous characters in the document. Thus, you need to mark only the first and last selected characters to identify the complete selection. With text, often the best way to handle selection is to identify the positions between characters rather than the characters themselves. This allows you to have an empty selection when the beginning and end of the selection are between the same pair of characters; it also allows you to put the cursor before the first character in the document or after the final one with no special-case code.

In three-dimensional editors, you might provide ways to rotate and zoom between selections, so sophisticated schemes for cycling through the possible selections might be unnecessary. On the other hand, selection in three dimensions is difficult because the cursor's position on the screen usually gives no indication of its depth.

Feedback

Feedback is similar to selection in that once you're in either mode, no pixels are produced and the screen is frozen. Drawing does not occur; instead, information about primitives that would have been rendered is sent back to the application. The key difference between selection and feedback modes is what information is sent back. In selection mode, assigned names are returned to an array of integer values. In feedback mode, information about transformed primitives is sent back to an array of floating-point values. The values sent back to the feedback array consist of tokens that specify what type of primitive (point, line, polygon, image, or bitmap) has been processed and transformed, followed by vertex, color, or other data for that primitive. The values returned are fully transformed by lighting and viewing operations. Feedback mode is initiated by calling `glRenderMode()` with `GL_FEEDBACK` as the argument.

Here's how you enter and exit feedback mode.

1. Call `glFeedbackBuffer()` to specify the array to hold the feedback information. The arguments to this command describe what type of data and how much of it gets written into the array.
2. Call `glRenderMode()` with `GL_FEEDBACK` as the argument to enter feedback mode. (For this step, you can ignore the value returned by `glRenderMode()`.) After this point, primitives aren't rasterized to produce pixels until you exit feedback mode, and the contents of the framebuffer don't change.
3. Draw your primitives. While issuing drawing commands, you can make several calls to `glPassThrough()` to insert markers into the returned feedback data and thus facilitate parsing.
4. Exit feedback mode by calling `glRenderMode()` with `GL_RENDER` as the argument if you want to return to normal drawing mode. The integer value returned by `glRenderMode()` is the number of values stored in the feedback array.
5. Parse the data in the feedback array.

```
void glFeedbackBuffer(GLsizei size, GLenum type, GLfloat *buffer);
```

Establishes a buffer for the feedback data: *buffer* is a pointer to an array where the data is stored. The *size* argument indicates the maximum number of values that can be stored in the array. The *type* argument describes the information fed back for each vertex in the feedback array; its possible values and their meaning are shown in Table 13-1. `glFeedbackBuffer()` must be called before feedback mode is entered. In the table, *k* is 1 in color-index mode and 4 in RGBA mode.

<i>type</i> Argument	Coordinates	Color	Texture	Total Values
<code>GL_2D</code>	<i>x, y</i>	-	-	2
<code>GL_3D</code>	<i>x, y, z</i>	-	-	3
<code>GL_3D_COLOR</code>	<i>x, y, z</i>	<i>k</i>	-	3 + <i>k</i>
<code>GL_3D_COLOR_TEXTURE</code>	<i>x, y, z</i>	<i>k</i>	4	7 + <i>k</i>
<code>GL_4D_COLOR_TEXTURE</code>	<i>x, y, z, w</i>	<i>k</i>	4	8 + <i>k</i>

Table 13-1 `glFeedbackBuffer()` *type* Values

The Feedback Array

In feedback mode, each primitive that would be rasterized (or each call to `glBitmap()`, `glDrawPixels()`, or `glCopyPixels()`, if the raster position is valid) generates a block of values that's copied into the feedback array. The number of values is determined by the *type* argument to `glFeedbackBuffer()`, as listed in Table 13-1. Use the appropriate value for the type of primitives you're drawing: `GL_2D` or `GL_3D` for unlit two- or three-dimensional primitives, `GL_3D_COLOR` for lit, three-dimensional primitives, and `GL_3D_COLOR_TEXTURE` or `GL_4D_COLOR_TEXTURE` for lit, textured, three- or four-dimensional primitives.

Each block of feedback values 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 pixel rectangles. In addition, pass-through markers that you've explicitly created can be returned in the array; the next section explains these markers in more detail. Table 13-2 shows the syntax for the feedback array; remember that the data associated with each returned vertex is as described in Table 13-1. Note that a polygon can have *n* vertices returned. Also, the *x*, *y*, *z* coordinates returned by feedback are window coordinates; if *w* is returned, it's in clip coordinates. For bitmaps and pixel rectangles, the coordinates returned are those of the current raster position. In the table, note that `GL_LINE_RESET_TOKEN` is returned only when the line stipple is reset for that line segment.

Primitive Type	Code	Associated Data
Point	<code>GL_POINT_TOKEN</code>	vertex
Line	<code>GL_LINE_TOKEN</code> or <code>GL_LINE_RESET_TOKEN</code>	vertex vertex
Polygon	<code>GL_POLYGON_TOKEN</code>	<i>n</i> vertex vertex ... vertex
Bitmap	<code>GL_BITMAP_TOKEN</code>	vertex
Pixel Rectangle	<code>GL_DRAW_PIXEL_TOKEN</code> or <code>GL_COPY_PIXEL_TOKEN</code>	vertex
Pass-through	<code>GL_PASS_THROUGH_TOKEN</code>	a floating-point number

Table 13-2 Feedback Array Syntax

Using Markers in Feedback Mode

Feedback occurs after transformations, lighting, polygon culling, and interpretation of polygons by `glPolygonMode()`. It might also occur after polygons with more than three edges are broken up into triangles (if your particular OpenGL implementation renders polygons by performing this decomposition). Thus, it might be hard for you to recognize the primitives you drew in the feedback data you receive. To help parse the feedback data, call `glPassThrough()` as needed in your sequence of drawing commands to insert a marker. You might use the markers to separate the feedback values returned from different primitives, for example. This command causes `GL_PASS_THROUGH_TOKEN` to be written into the feedback array, followed by the floating-point value you pass in as an argument.

```
void glPassThrough(GLfloat token);
```

Inserts a marker into the stream of values written into the feedback array, if called in feedback mode. The marker consists of the code `GL_PASS_THROUGH_TOKEN` followed by a single floating-point value, *token*. This command has no effect when called outside of feedback mode. Calling `glPassThrough()` between `glBegin()` and `glEnd()` generates a `GL_INVALID_OPERATION` error.

A Feedback Example

Example 13-7 demonstrates the use of feedback mode. This program draws a lit, three-dimensional scene in normal rendering mode. Then, feedback mode is entered, and the scene is redrawn. Since the program draws lit, untextured, three-dimensional objects, the type of feedback data is `GL_3D_COLOR`. Since RGBA mode is used, each unclipped vertex generates seven values for the feedback buffer: *x*, *y*, *z*, *r*, *g*, *b*, and *a*.

In feedback mode, the program draws two lines as part of a line strip and then inserts a pass-through marker. Next, a point is drawn at `(-100.0, -100.0, -100.0)`, which falls outside the orthographic viewing volume and thus doesn't put any values into the feedback array. Finally, another pass-through marker is inserted, and another point is drawn.

Example 13-7 Feedback Mode: feedback.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>

void init(void)
{
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
}

void drawGeometry (GLenum mode)
{
    glBegin (GL_LINE_STRIP);
    glNormal3f (0.0, 0.0, 1.0);
    glVertex3f (30.0, 30.0, 0.0);
    glVertex3f (50.0, 60.0, 0.0);
    glVertex3f (70.0, 40.0, 0.0);
    glEnd ();
    if (mode == GL_FEEDBACK)
        glPassThrough (1.0);
    glBegin (GL_POINTS);
    glVertex3f (-100.0, -100.0, -100.0); /* will be clipped */
    glEnd ();
    if (mode == GL_FEEDBACK)
        glPassThrough (2.0);
    glBegin (GL_POINTS);
    glNormal3f (0.0, 0.0, 1.0);
    glVertex3f (50.0, 50.0, 0.0);
    glEnd ();
}

void print3DcolorVertex (GLint size, GLint *count,
                        GLfloat *buffer)
{
    int i;

    printf (" ");
    for (i = 0; i < 7; i++) {
        printf ("%4.2f ", buffer[size-(*count)]);
        *count = *count - 1;
    }
    printf ("\n");
}
```

```

void printBuffer(GLint size, GLfloat *buffer)
{
    GLint count;
    GLfloat token;

    count = size;
    while (count) {
        token = buffer[size-count]; count--;
        if (token == GL_PASS_THROUGH_TOKEN) {
            printf ("GL_PASS_THROUGH_TOKEN\n");
            printf (" %4.2f\n", buffer[size-count]);
            count--;
        }
        else if (token == GL_POINT_TOKEN) {
            printf ("GL_POINT_TOKEN\n");
            print3DcolorVertex (size, &count, buffer);
        }
        else if (token == GL_LINE_TOKEN) {
            printf ("GL_LINE_TOKEN\n");
            print3DcolorVertex (size, &count, buffer);
            print3DcolorVertex (size, &count, buffer);
        }
        else if (token == GL_LINE_RESET_TOKEN) {
            printf ("GL_LINE_RESET_TOKEN\n");
            print3DcolorVertex (size, &count, buffer);
            print3DcolorVertex (size, &count, buffer);
        }
    }
}

void display(void)
{
    GLfloat feedBuffer[1024];
    GLint size;

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glOrtho (0.0, 100.0, 0.0, 100.0, 0.0, 1.0);

    glClearColor (0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    drawGeometry (GL_RENDER);

    glFeedbackBuffer (1024, GL_3D_COLOR, feedBuffer);
    (void) glRenderMode (GL_FEEDBACK);
    drawGeometry (GL_FEEDBACK);
}

```

```

    size = glRenderMode (GL_RENDER);
    printBuffer (size, feedBuffer);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (100, 100);
    glutInitWindowPosition (100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}

```

Running this program generates the following output:

```

GL_LINE_RESET_TOKEN
 30.00 30.00 0.00 0.84 0.84 0.84 1.00
 50.00 60.00 0.00 0.84 0.84 0.84 1.00
GL_LINE_TOKEN
 50.00 60.00 0.00 0.84 0.84 0.84 1.00
 70.00 40.00 0.00 0.84 0.84 0.84 1.00
GL_PASS_THROUGH_TOKEN
 1.00
GL_PASS_THROUGH_TOKEN
 2.00
GL_POINT_TOKEN
 50.00 50.00 0.00 0.84 0.84 0.84 1.00

```

Thus, the line strip drawn with these commands results in two primitives:

```

glBegin(GL_LINE_STRIP);
    glNormal3f (0.0, 0.0, 1.0);
    glVertex3f (30.0, 30.0, 0.0);
    glVertex3f (50.0, 60.0, 0.0);
    glVertex3f (70.0, 40.0, 0.0);
glEnd();

```

The first primitive begins with `GL_LINE_RESET_TOKEN`, which indicates that the primitive is a line segment and that the line stipple is reset. The second primitive begins with `GL_LINE_TOKEN`, so it's also a line segment, but the line stipple isn't reset and hence continues from where the previous line segment left off. Each of the two vertices for these lines generates seven values for the feedback array. Note that the RGBA values for all four vertices in these two lines are (0.84, 0.84, 0.84, 1.0), which is a very light gray color

with the maximum alpha value. These color values are a result of the interaction of the surface normal and lighting parameters.

Since no feedback data is generated between the first and second pass-through markers, you can deduce that any primitives drawn between the first two calls to `glPassThrough()` were clipped out of the viewing volume. Finally, the point at (50.0, 50.0, 0.0) is drawn, and its associated data is copied into the feedback array.

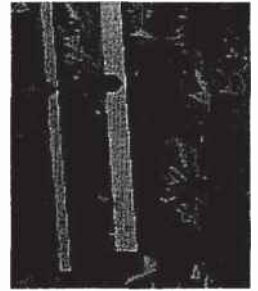
Note: In both feedback and selection modes, information on objects is returned prior to any fragment tests. Thus, objects that would not be drawn due to failure of the scissor, alpha, depth, or stencil tests may still have their data processed and returned in both feedback and selection modes.



Try This

Make changes to Example 13-7 and see how they affect the feedback values that are returned. For example, change the coordinate values of `glOrtho()`. Change the lighting variables, or eliminate lighting altogether and change the feedback type to `GL_3D`. Or add more primitives to see what other geometry (such as filled polygons) contributes to the feedback array.

Now That You Know



Chapter Objectives

This chapter doesn't have objectives in the same way that previous chapters do. It's simply a collection of topics that describe ideas you might find useful for your application. Some topics, such as error handling, don't fit into other categories, but are too short for an entire chapter.

OpenGL is kind of a bag of low-level tools; now that you know about those tools, you can use them to implement higher-level functions. This chapter presents several examples of such higher-level capabilities.

This chapter discusses a variety of techniques based on OpenGL commands that illustrate some of the not-so-obvious uses to which you can put these commands. The examples are in no particular order and aren't related to each other. The idea is to read the section headings and skip to the examples that you find interesting. For your convenience, the headings are listed and explained briefly here.

Note: Most of the examples in the rest of this guide are complete and can be compiled and run as is. In this chapter, however, there are no complete programs, and you have to do a bit of work on your own to make them run.

- “Error Handling” on page 501 tells you how to check for OpenGL error conditions.
- “Which Version Am I Using?” on page 503 describes how to find out details about the implementation, including the version number. This can be useful for writing applications that are backward compatible with earlier versions of OpenGL.
- “Extensions to the Standard” on page 505 presents techniques to identify and use vendor-specific extensions to the OpenGL standard.
- “Cheesy Translucency” on page 506 explains how to use polygon stippling to achieve translucency; this is particularly useful when you don't have blending hardware available.
- “An Easy Fade Effect” on page 506 shows how to use polygon stippling to create the effect of a fade into the background.
- “Object Selection Using the Back Buffer” on page 508 describes how to use the back buffer in a double-buffered system to handle simple object picking.
- “Cheap Image Transformation” on page 509 discusses how to draw a distorted version of a bitmapped image by drawing each pixel as a quadrilateral.
- “Displaying Layers” on page 511 explains how to display multiple different layers of materials and indicate where the materials overlap.
- “Antialiased Characters” on page 512 describes how to draw smoother fonts.
- “Drawing Round Points” on page 514 describes how to draw near-round points.
- “Interpolating Images” on page 514 shows how to smoothly blend from one image to the another.

-
- “Making Decals” on page 515 explains how to draw two images, where one is a sort of decal that should always appear on top of the other.
 - “Drawing Filled, Concave Polygons Using the Stencil Buffer” on page 516 tells you how to draw concave polygons, nonsimple polygons, and polygons with holes by using the stencil buffer.
 - “Finding Interference Regions” on page 518 describes how to determine where three-dimensional pieces overlap.
 - “Shadows” on page 519 describes how to draw shadows of lit objects.
 - “Hidden-Line Removal” on page 521 discusses how to draw a wireframe object with hidden lines removed by using the stencil buffer.
 - “Texture-Mapping Applications” on page 523 describes several clever uses for texture mapping, such as rotating and warping images.
 - “Drawing Depth-Buffered Images” on page 523 tells you how to combine images in a depth-buffered environment.
 - “Dirichlet Domains” on page 524 explains how to find the Dirichlet domain of a set of points using the depth buffer.
 - “Life in the Stencil Buffer” on page 526 explains how to implement the Game of Life using the stencil buffer.
 - “Alternative Uses for `glDrawPixels()` and `glCopyPixels()`” on page 527 describes how to use these two commands for such effects as fake video, airbrushing, and transposed images.

Error Handling

The truth is, your program will make mistakes. Use of error-handling routines are essential during development and are highly recommended for commercially released applications. (Unless you can give a 100% guarantee your program will never generate an OpenGL error condition. Get real!) OpenGL has simple error-handling routines for the base GL and GLU libraries.

When OpenGL detects an error (in either the base GL or GLU), it records a current error code. The command that caused the error is ignored, so it has no effect on OpenGL state or on the framebuffer contents. (If the error recorded was `GL_OUT_OF_MEMORY`, however, the results of the command are undefined.) Once recorded, the current error code isn't cleared—that is,

additional errors aren't recorded—until you call the query command `glGetError()`, which returns the current error code. After you've queried and cleared the current error code, or if there's no error to begin with, `glGetError()` returns `GL_NO_ERROR`.

GLenum glGetError(void);

Returns the value of the error flag. When an error occurs in either the GL or GLU, the error flag is set to the appropriate error code value. If `GL_NO_ERROR` is returned, there has been no detectable error since the last call to `glGetError()`, or since the GL was initialized. No other errors are recorded until `glGetError()` is called, the error code is returned, and the flag is reset to `GL_NO_ERROR`.

It is strongly recommended that you call `glGetError()` at least once in each `display()` routine. Table 14-1 lists the basic defined OpenGL error codes.

Error Code	Description
<code>GL_INVALID_ENUM</code>	GLenum argument out of range
<code>GL_INVALID_VALUE</code>	Numeric argument out of range
<code>GL_INVALID_OPERATION</code>	Operation illegal in current state
<code>GL_STACK_OVERFLOW</code>	Command would cause a stack overflow
<code>GL_STACK_UNDERFLOW</code>	Command would cause a stack underflow
<code>GL_OUT_OF_MEMORY</code>	Not enough memory left to execute command

Table 14-1 OpenGL Error Codes

There are also thirty-seven GLU NURBS errors (with non-descriptive constant names, `GLU_NURBS_ERROR1`, `GLU_NURBS_ERROR2`, and so on), fourteen tessellator errors (`GLU_TESS_MISSING_BEGIN_POLYGON`, `GLU_TESS_MISSING_END_POLYGON`, `GLU_TESS_MISSING_BEGIN_CONTOUR`, `GLU_TESS_MISSING_END_CONTOUR`, `GLU_TESS_COORD_TOO_LARGE`, `GLU_TESS_NEED_COMBINE_CALLBACK`, and eight generically named `GLU_TESS_ERROR*`), and `GLU_INCOMPATIBLE_GL_VERSION`. Also, the GLU defines the error codes `GLU_INVALID_ENUM`, `GLU_INVALID_VALUE`, and `GLU_OUT_OF_MEMORY`, which have the same meaning as the related OpenGL codes.

To obtain a printable, descriptive string corresponding to either a GL or GLU error code, use the GLU routine `gluErrorString()`.

```
const GLubyte* gluErrorString(GLenum errorCode);
```

Returns a pointer to a descriptive string that corresponds to the OpenGL or GLU error number passed in *errorCode*.

In Example 14-1, a simple error handling routine is shown.

Example 14-1 Querying and Printing an Error

```
GLenum errCode;
const GLubyte *errString;

if ((errCode = glGetError()) != GL_NO_ERROR) {
    errString = gluErrorString(errCode);
    fprintf (stderr, "OpenGL Error: %s\n", errString);
}
```

Note: The string returned by `gluErrorString()` must not be altered or freed by the application.

Which Version Am I Using?

The portability of OpenGL applications is one of OpenGL's attractive features. However, new versions of OpenGL introduce new features, which may introduce backward compatibility problems. In addition, you may want your application to perform equally well on a variety of implementations. For example, you might make texture mapping the default rendering mode on one machine, but only have flat shading on another. You can use `glGetString()` to obtain release information about your OpenGL implementation.

```
const GLubyte* glGetString(GLenum name);
```

Returns a pointer to a string that describes an aspect of the OpenGL implementation. *name* can be one of the following: `GL_VENDOR`, `GL_RENDERER`, `GL_VERSION`, or `GL_EXTENSIONS`.

`GL_VENDOR` returns the name of the company responsible for the OpenGL implementation. `GL_RENDERER` returns an identifier of the renderer,

which is usually the hardware platform. For more about `GL_EXTENSIONS`, see the next section, “Extensions to the Standard” on page 505.

`GL_VERSION` returns a string that identifies the version number of this implementation of OpenGL. 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. For example, if this OpenGL implementation is from the fictitious XYZ Corporation, the string returned might be

`1.1.4 XYZ-OS 3.2`

which means that this implementation is XYZ’s fourth release of an OpenGL library that conforms to the specification for OpenGL Version 1.1. It probably also means this is release 3.2 of XYZ’s proprietary operating system.

Another way to query the version number for OpenGL is to look for the symbolic constant (use the preprocessor statement `#ifdef`) named `GL_VERSION_1_1`. The absence of the constant `GL_VERSION_1_1` means that you have OpenGL Version 1.0.

Note: If running from client to server, such as when performing indirect rendering with the OpenGL extension to the X Window System, the client and server may be different versions. If your client version is ahead of your server, your client might request an operation that is not supported on your server.

Utility Library Version

`gluGetString()` is a query function for the Utility Library (GLU) and is similar to `glGetString()`.

```
const GLubyte* gluGetString(GLenum name);
```

Returns a pointer to a string that describes an aspect of the OpenGL implementation. *name* can be one of the following: `GLU_VERSION`, or `GLU_EXTENSIONS`.

Note that `gluGetString()` was not available in GLU 1.0. Another way to query the version number for GLU is to look for the symbolic constant `GLU_VERSION_1_1`. The absence of the constant `GLU_VERSION_1_1` means that you have GLU 1.0.

Extensions to the Standard

OpenGL has a formal written specification that describes what operations comprise the library. An individual vendor or a group of vendors may decide to include additional functionality to their released implementation.

New routine and symbolic constant names clearly indicate whether a feature is part of the OpenGL standard or a vendor-specific extension. To make a vendor-specific name, the vendor appends a company identifier (in uppercase) and, if needed, additional information, such as a machine name. For example, if XYZ Corporation wants to add a new routine and symbolic constant, they might be of the form `glCommandXYZ()` and `GL_DEFINITION_XYZ`. If XYZ Corporation wants to have an extension that is available only on its FooBar graphics board, then the names might be `glCommandXYZfb()` and `GL_DEFINITION_XYZ_FB`.

If two or more vendors agree to implement the same extension, then the procedures and constants are suffixed with the more generic `EXT` (`glCommandEXT()` and `GL_DEFINITION_EXT`).

If you want to know if a particular extension is supported on your implementation, use `glGetString(GL_EXTENSIONS)`. This returns a list of all the extensions in the implementation, separated by spaces. If you want to find out if a specific extension is supported, use the code in Example 14-2 to search through the list and match the extension name. Return `GL_TRUE`, if it is; `GL_FALSE`, if it isn't.

Example 14-2 Find Out If An Extension Is Supported

```
static GLboolean QueryExtension(char *extName)
{
    char *p = (char *) glGetString(GL_EXTENSIONS);
```

```

char *end = p + strlen(p);
while (p < end) {
    int n = strcspn(p, " ");
    if ((strlen(extName)==n) && (strncmp(extName,p,n)==0)) {
        return GL_TRUE;
    }
    p += (n + 1);
}
return GL_FALSE;
}

```

Cheesy Translucency

You can use polygon stippling to simulate a translucent material. This is an especially good solution for systems that don't have blending hardware. Since polygon stipple patterns are 32x32 bits, or 1024 bits, you can go from opaque to transparent in 1023 steps. (In practice, that's many more steps than you need!) For example, if you want a surface that lets through 29 percent of the light, simply make up a stipple pattern where 29 percent (roughly 297) of the pixels in the mask are zero and the rest are one. Even if your surfaces have the same translucency, don't use the same stipple pattern for each one, as they cover exactly the same bits on the screen. Make up a different pattern for each by randomly selecting the appropriate number of pixels to be zero. (See "Displaying Points, Lines, and Polygons" on page 49 for more information about polygon stippling.)

If you don't like the effect with random pixels turned on, you can use regular patterns, but they don't work as well when transparent surfaces are stacked. This is often not a problem because most scenes have relatively few translucent regions that overlap. In a picture of an automobile with translucent windows, your line of sight can go through at most two windows, and usually it's only one.

An Easy Fade Effect

Suppose you have an image that you want to fade gradually to some background color. Define a series of polygon stipple patterns, each of which has more bits turned on so that they represent denser and denser patterns. Then use these patterns repeatedly with a polygon large enough to cover

the region over which you want to fade. For example, suppose you want to fade to black in 16 steps. First define 16 different pattern arrays:

```
GLubyte stips[16][4*32];
```

Then load them in such a way that each has one-sixteenth of the pixels in a 32x32 stipple pattern turned on and that the bitwise OR of all the stipple patterns is all ones. After that, the following code does the trick:

```
draw_the_picture();
glColor3f(0.0, 0.0, 0.0); /* set color to black */
for (i = 0; i < 16; i++) {
    glPolygonStipple(&stips[i][0]);
    draw_a_polygon_large_enough_to_cover_the_whole_region();
}
```

In some OpenGL implementations, you might get better performance by first compiling the stipple patterns into display lists. During your initialization, do something like this:

```
#define STIP_OFFSET 100
for (i = 0; i < 16; i++) {
    glNewList(i+STIP_OFFSET, GL_COMPILE);
    glPolygonStipple(&stips[i][0]);
    glEndList();
}
```

Then, replace this line in the first code fragment

```
glPolygonStipple(&stips[i][0]);
```

with

```
glCallList(i);
```

By compiling the command to set the stipple into a display list, OpenGL might be able to rearrange the data in the *stips[][]* array into the hardware-specific form required for maximum stipple-setting speed.

Another application for this technique is if you're drawing a changing picture and want to leave some blur behind that gradually fades out to give some indication of past motion. For example, suppose you're simulating a planetary system and you want to leave trails on the planets to show a recent portion of their path. Again, assuming you want to fade in sixteen

steps, set up the stipple patterns as before (using the display-list version, say), and have the main simulation loop look something like this:

```
current_stipple = 0;
while (1) {
    draw_the_next_frame();
    glCallList(current_stipple++);
    if (current_stipple == 16) current_stipple = 0;
    glColor3f(0.0, 0.0, 0.0); /* set color to black */
    draw_a_polygon_large_enough_to_cover_the_whole_region();
}
```

Each time through the loop, you clear one-sixteenth of the pixels. Any pixel that hasn't had a planet on it for sixteen frames is certain to be cleared to black. Of course, if your system supports blending in hardware, it's easier to blend in a certain amount of background color with each frame. (See "Displaying Points, Lines, and Polygons" on page 49 for polygon stippling details, Chapter 7 for more information about display lists, and "Blending" on page 214 for information about blending.)

Object Selection Using the Back Buffer

Although the OpenGL selection mechanism (see "Selection" on page 470) is powerful and flexible, it can be cumbersome to use. Often, the situation is simple: Your application draws a scene composed of a substantial number of objects; the user points to an object with the mouse, and the application needs to find the item under the tip of the cursor.

One way to do this requires your application to be running in double-buffer mode. When the user picks an object, the application redraws the entire scene in the back buffer, but instead of using the normal colors for objects, it encodes some kind of object identifier for each object's color. The application then simply reads back the pixel under the cursor, and the value of that pixel encodes the number of the picked object. If many picks are expected for a single, static picture, you can read the entire color buffer once and look in your copy for each attempted pick, rather than read back each pixel individually.

Note that this scheme has an advantage over standard selection in that it picks the object that's in front if multiple objects appear at the same pixel, one behind the other. Since the image with false colors is drawn in the back buffer, the user never sees it; you can redraw the back buffer (or copy it from the front buffer) before swapping the buffers. In color-index mode, the

encoding is simple—send the object identifier as the index. In RGBA mode, encode the bits of the identifier into the R, G, and B components.

Be aware that you can run out of identifiers if there are too many objects in the scene. For example, suppose you're running in color-index mode on a system that has 4-bit buffers for color-index information (16 possible different indices) in each of the color buffers, but the scene has thousands of pickable items. To address this issue, the picking can be done in a few passes. To think about this in concrete terms, assume there are fewer than 4096 items, so all the object identifiers can be encoded in 12 bits. In the first pass, draw the scene using indices composed of the 4 high-order bits, then use the second and third passes to draw the middle 4 bits and the 4 low-order bits. After each pass, read the pixel under the cursor, extract the bits, and pack them together at the end to get the object identifier.

With this method, the picking takes three times as long, but that's often acceptable. Note that after you have the high-order 4 bits, you eliminate 15/16 of all objects, so you really need to draw only 1/16 of them for the second pass. Similarly, after the second pass, 255 of the 256 possible items have been eliminated. The first pass thus takes about as long as drawing a single frame does, but the second and third passes can be up to 16 and 256 times as fast.

If you're trying to write portable code that works on different systems, break up your object identifiers into chunks that fit on the lowest common denominator of those systems. Also, keep in mind that your system might perform automatic dithering in RGB mode. If this is the case, turn off dithering.

Cheap Image Transformation

If you want to draw a distorted version of a bitmapped image (perhaps simply stretched or rotated, or perhaps drastically modified by some mathematical function), there are many possibilities. You can use the image as a texture map, which allows you to scale, rotate, or otherwise distort the image. If you just want to scale the image, you can use `glPixelZoom()`.

In many cases, you can achieve good results by drawing the image of each pixel as a quadrilateral. Although this scheme doesn't produce images that are as nice as those you would get by applying a sophisticated filtering algorithm (and it might not be sufficient for sophisticated users), it's a lot quicker.

To make the problem more concrete, assume that the original image is m pixels by n pixels, with coordinates chosen from $[0, m-1] \times [0, n-1]$. Let the distortion functions be $x(m,n)$ and $y(m,n)$. For example, if the distortion is simply a zooming by a factor of 3.2, then $x(m,n) = 3.2*m$ and $y(m,n) = 3.2*n$. The following code draws the distorted image:

```
glShadeModel(GL_FLAT);
glScale(3.2, 3.2, 1.0);
for (j=0; j < n; j++) {
    glBegin(GL_QUAD_STRIP);
    for (i=0; i <= m; i++) {
        glVertex2i(i,j);
        glVertex2i(i, j+1);
        set_color(i,j);
    }
    glEnd();
}
```

This code draws each transformed pixel in a solid color equal to that pixel's color and scales the image size by 3.2. The routine `set_color()` stands for whatever the appropriate OpenGL command is to set the color of the image pixel.

The following is a slightly more complex version that distorts the image using the functions $x(i,j)$ and $y(i,j)$:

```
glShadeModel(GL_FLAT);
for (j=0; j < n; j++) {
    glBegin(GL_QUAD_STRIP);
    for (i=0; i <= m; i++) {
        glVertex2i(x(i,j), y(i,j));
        glVertex2i(x(i,j+1), y(i,j+1));
        set_color(i,j);
    }
    glEnd();
}
```

An even better distorted image can be drawn with the following code:

```
glShadeModel(GL_SMOOTH);
for (j=0; j < (n-1); j++) {
    glBegin(GL_QUAD_STRIP);
    for (i=0; i < m; i++) {
        set_color(i,j);
        glVertex2i(x(i,j), y(i,j));
        set_color(i,j+1);
        glVertex2i(x(i,j+1), y(i,j+1));
    }
}
```

```
    glEnd();  
}
```

This code smoothly interpolates color across each quadrilateral. Note that this version produces one fewer quadrilateral in each dimension than do the flat-shaded versions, because the color image is being used to specify colors at the quadrilateral vertices. In addition, you can antialias the polygons with the appropriate blending function (`GL_SRC_ALPHA`, `GL_ONE`) to get an even nicer image.

Displaying Layers

In some applications such as semiconductor layout programs, you want to display multiple different layers of materials and indicate where the materials overlap each other.

As a simple example, suppose you have three different substances that can be layered. At any point, eight possible combinations of layers can occur, as shown in Table 14-2.

	Layer 1	Layer 2	Layer 3	Color
0	absent	absent	absent	black
1	present	absent	absent	red
2	absent	present	absent	green
3	present	present	absent	blue
4	absent	absent	present	pink
5	present	absent	present	yellow
6	absent	present	present	white
7	present	present	present	gray

Table 14-2 Eight Combinations of Layers

You want your program to display eight different colors, depending on the layers present. One arbitrary possibility is shown in the last column of the table. To use this method, use color-index mode and load your color map so that entry 0 is black, entry 1 is red, entry 2 is green, and so on. Note that if the numbers from 0 through 7 are written in binary, the 4 bit is turned

on whenever layer 3 appears, the 2 bit whenever layer 2 appears, and the 1 bit whenever layer 1 appears.

To clear the window, set the writemask to 7 (all three layers) and set the clearing color to 0. To draw your image, set the color to 7, and then when you want to draw something in layer n , set the writemask to n . In other types of applications, it might be necessary to selectively erase in a layer, in which case you would use the writemasks just discussed, but set the color to 0 instead of 7. (See “Masking Buffers” on page 381 for more information about writemasks.)

Antialiased Characters

Using the standard technique for drawing characters with `glBitmap()`, drawing each pixel of a character is an all-or-nothing affair—the pixel is either turned on or not. If you’re drawing black characters on a white background, for example, the resulting pixels are either black or white, never a shade of gray. Much smoother, higher-quality images can be achieved if intermediate colors are used when rendering characters (grays, in this example).

Assuming that you’re drawing black characters on a white background, imagine a highly magnified picture of the pixels on the screen, with a high-resolution character outline superimposed on it, as shown in the left side of Figure 14-1.

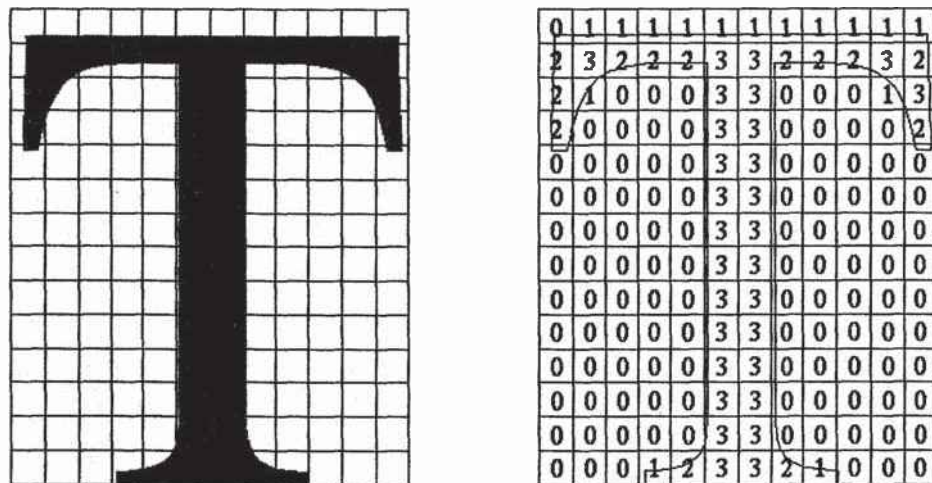


Figure 14-1 Antialiased Characters

Notice that some of the pixels are completely enclosed by the character's outline and should be painted black; some pixels are completely outside the outline and should be painted white; but many pixels should ideally be painted some shade of gray, where the darkness of the gray corresponds to the amount of black in the pixel. If this technique is used, the resulting image on the screen looks better.

If speed and memory usage are of no concern, each character can be drawn as a small image instead of as a bitmap. If you're using RGBA mode, however, this method might require up to 32 bits per pixel of the character to be stored and drawn, instead of the 1 bit per pixel in a standard character. Alternatively, you could use one 8-bit index per pixel and convert these indices to RGBA by table lookup during transfer. In many cases, a compromise is possible that allows you to draw the character with a few gray levels between black and white (say, two or three), and the resulting font description requires only 2 or 3 bits per pixel of storage.

The numbers in the right side of Figure 14-1 indicate the approximate percentage coverage of each pixel: 0 means approximately empty, 1 means approximately one-third coverage, 2 means two-thirds, and 3 means completely covered. If pixels labeled 0 are painted white, pixels labeled 3 are painted black, and pixels labeled 1 and 2 are painted one-third and two-thirds black, respectively, the resulting character looks quite good. Only 2 bits are required to store the numbers 0, 1, 2, and 3, so for 2 bits per pixel, four levels of gray can be saved.

There are basically two methods to implement antialiased characters, depending on whether you're in RGBA or color-index mode.

In RGBA mode, define three different character bitmaps, corresponding to where 1, 2, and 3 appear in Figure 14-1. Set the color to white, and clear for the background. Set the color to one-third gray (RGB = (0.666, 0.666, 0.666)), and draw all the pixels with a 1 in them. Then set RGB = (0.333, 0.333, 0.333), draw with the 2 bitmap, and use RGB = (0.0, 0.0, 0.0) for the 3 bitmap. What you're doing is defining three different fonts and redrawing the string three times, where each pass fills in the bits of the appropriate color densities.

In color-index mode, you can do exactly the same thing, but if you're willing to set up the color map correctly and use writemasks, you can get away with only two bitmaps per character and two passes per string. In the preceding example, set up one bitmap that has a 1 wherever 1 or 3 appears in the character. Set up a second bitmap that has a 1 wherever a 2 or a 3 appears. Load the color map so that 0 gives white, 1 gives light gray, 2 gives

dark gray, and 3 gives black. Set the color to 3 (11 in binary) and the writemask to 1, and draw the first bitmap. Then change the writemask to 2, and draw the second. Where 0 appears in Figure 14-1, nothing is drawn in the framebuffer. Where 1, 2, and 3 appear, 1, 2, and 3 appear in the framebuffer.

For this example with only four gray levels, the savings is small—two passes instead of three. If eight gray levels were used instead, the RGBA method would require seven passes, and the color-map masking technique would require only three. With sixteen gray levels, the comparison is fifteen passes to four passes. (See “Masking Buffers” on page 381 for more information about writemasks and “Bitmaps and Fonts” on page 279 for more information about drawing bitmaps.)



Try This

- Can you see how to do RGBA rendering using no more images than the optimized color-index case? Hint: How are RGB fragments normally merged into the color buffer when antialiasing is desired?

Drawing Round Points

Draw near-round, aliased points by enabling point antialiasing, turning blending off, and using an alpha function that passes only fragments with alpha greater than 0.5. (See “Antialiasing” on page 226 and “Blending” on page 214 for more information about these topics.)

Interpolating Images

Suppose you have a pair of images (where *image* can mean a bitmap image, or a picture generated using geometry in the usual way), and you want to smoothly blend from one to the other. This can be done easily using the alpha component and appropriate blending operations. Let’s say you want to accomplish the blending in ten steps, where image A is shown in frame 0 and image B is shown in frame 9. The obvious approach is to draw image A with alpha equal to $(9-i)/9$ and image B with an alpha of $i/9$ in frame i .

The problem with this method is that both images must be drawn in each frame. A faster approach is to draw image A in frame 0. To get frame 1, blend in $1/9$ of image B and $8/9$ of what’s there. For frame 2, blend in $1/8$ of image

B with 7/8 of what's there. For frame 3, blend in 1/7 of image B with 6/7 of what's there, and so on. For the last step, you're just drawing 1/1 of image B blended with 0/1 of what's left, yielding image B exactly.

To see that this works, if for frame i you have

$$\frac{(9-i)A}{9} + \frac{iB}{9}$$

and you blend in $B/(9-i)$ with $(8-i)/(9-i)$ of what's there, you get

$$\frac{B}{9-i} + \frac{8-i}{9-i} \left[\frac{(9-i)A}{9} + \frac{iB}{9} \right] = \frac{9-(i+1)A}{9} + \frac{(i+1)B}{9}$$

(See "Blending" on page 214.)

Making Decals

Suppose you're drawing a complex three-dimensional picture using depth-buffering to eliminate the hidden surfaces. Suppose further that one part of your picture is composed of coplanar figures A and B, where B is a sort of decal that should always appear on top of figure A.

Your first approach might be to draw B after you've drawn A, setting the depth-buffering function to replace on greater or equal. Due to the finite precision of the floating-point representations of the vertices, however, round-off error can cause polygon B to be sometimes a bit in front and sometimes a bit behind figure A. Here's one solution to this problem.

1. Disable the depth buffer for writing, and render A.
2. Enable the depth buffer for writing, and render B.
3. Disable the color buffer for writing, and render A again.
4. Enable the color buffer for writing.

Note that during the entire process, the depth-buffer test is enabled. In step 1, A is rendered wherever it should be, but none of the depth-buffer values are changed; thus, in step 2, wherever B appears over A, B is guaranteed to be drawn. Step 3 simply makes sure that all of the depth values under A are updated correctly, but since RGBA writes are disabled, the color pixels are unaffected. Finally, step 4 returns the system to the default state (writing is enabled both in the depth buffer and in the color buffer).

If a stencil buffer is available, the following simpler technique works.

1. Configure the stencil buffer to write one if the depth test passes, and zero otherwise. Render A.
2. Configure the stencil buffer to make no stencil value change, but to render only where stencil values are one. Disable the depth-buffer test and its update. Render B.

With this method, it's not necessary to initialize the contents of the stencil buffer at any time, because the stencil value of all pixels of interest (that is, those rendered by A) are set when A is rendered. Be sure to reenable the depth test and disable the stencil test before additional polygons are drawn. (See "Selecting Color Buffers for Writing and Reading" on page 379, "Depth Test" on page 391, and "Stencil Test" on page 385.)

Drawing Filled, Concave Polygons Using the Stencil Buffer

Consider the concave polygon 1234567 shown in Figure 14-2. Imagine that it's drawn as a series of triangles: 123, 134, 145, 156, 167, all of which are shown in the figure. The heavier line represents the original polygon boundary. Drawing all these triangles divides the buffer into nine regions A, B, C, ..., I, where region I is outside all the triangles.

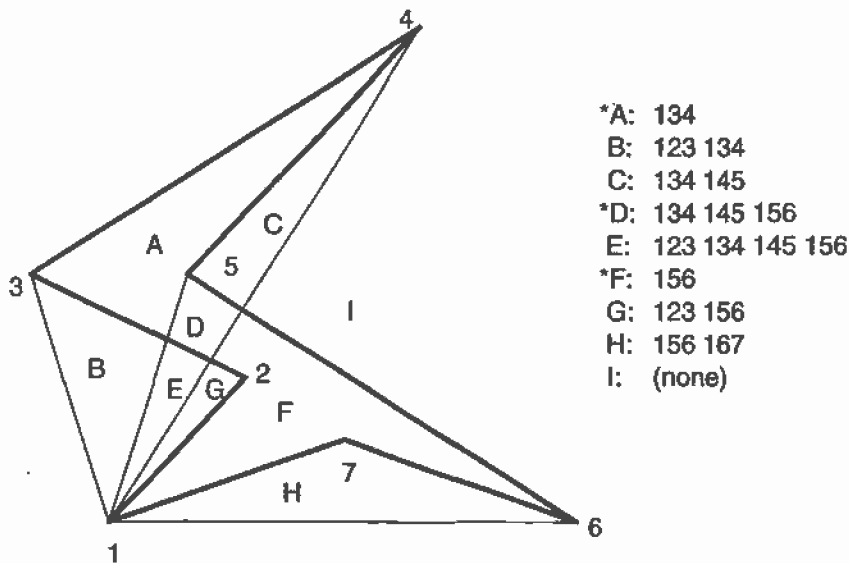


Figure 14-2 Concave Polygon

In the text of the figure, each of the region names is followed by a list of the triangles that cover it. Regions A, D, and F make up the original polygon; note that these three regions are covered by an odd number of triangles. Every other region is covered by an even number of triangles (possibly zero). Thus, to render the inside of the concave polygon, you just need to render regions that are enclosed by an odd number of triangles. This can be done using the stencil buffer, with a two-pass algorithm.

First, clear the stencil buffer and disable writing into the color buffer. Next, draw each of the triangles in turn, using the `GL_INVERT` function in the stencil buffer. (For best performance, use triangle fans.) This flips the value between zero and a nonzero value every time a triangle is drawn that covers a pixel. After all the triangles are drawn, if a pixel is covered an even number of times, the value in the stencil buffers is zero; otherwise, it's nonzero. Finally, draw a large polygon over the whole region (or redraw the triangles), but allow drawing only where the stencil buffer is nonzero.

Note: There's a slight generalization of the preceding technique, where you don't need to start with a polygon vertex. In the 1234567 example, let P be any point on or off the polygon. Draw the triangles: P12, P23, P34, P45, P56, P67, and P71. Regions covered by an odd number of triangles are inside; other regions are outside. This is a generalization in that if P happens to be one of the polygon's edges, one of the triangles is empty.

This technique can be used to fill both nonsimple polygons (polygons whose edges cross each other) and polygons with holes. The following example illustrates how to handle a complicated polygon with two regions, one four-sided and one five-sided. Assume further that there's a triangular and a four-sided hole (it doesn't matter in which regions the holes lie). Let the two regions be *abcd* and *efghi*, and the holes *jkl* and *mnop*. Let *z* be any point on the plane. Draw the following triangles:

zab zbc zcd zda zef zfg zgh zhi zie zjk zkl zlj zmn zno zop zpm

Mark regions covered by an odd number of triangles as *in*, and those covered by an even number as *out*. (See "Stencil Test" on page 385 for more information about the stencil buffer.)

Finding Interference Regions

If you're designing a mechanical part made from smaller three-dimensional pieces, you often want to display regions where the pieces overlap. In many cases, such regions indicate design errors where parts of a machine interfere with each other. In the case of moving parts, it can be even more valuable, since a search for interfering regions can be done through a complete mechanical cycle of the design. The method for doing this is complicated, and the description here might be too brief. Complete details can be found in the paper *Interactive Inspection of Solids: Cross-sections and Interferences*, by Jarek Rossignac, Abe Megahed, and Bengt-Olaf Schneider (SIGGRAPH 1992 Proceedings).

The method is related to the capping algorithm described in "Stencil Test" on page 385. The idea is to pass an arbitrary clipping plane through the objects that you want to test for interference, and then determine when a portion of the clipping plane is inside more than one object at a time. For a static image, the clipping plane can be moved manually to highlight interfering regions; for a dynamic image, it might be easier to use a grid of clipping planes to search for all possible interferences.

Draw each of the objects you want to check and clip them against the clipping plane. Note which pixels are inside the object at that clipping plane using an odd-even count in the stencil buffer, as explained in the preceding section. (For properly formed objects, a point is inside the object if a ray drawn from that point to the eye intersects an odd number of surfaces of the object.) To find interferences, you need to find pixels in the framebuffer where the clipping plane is in the interior of two or more

regions at once; in other words, in the intersection of the interiors of any pair of objects.

If multiple objects need to be tested for mutual intersection, store 1 bit every time some intersection appears, and another bit wherever the clipping buffer is inside any of the objects (the union of the objects' interiors). For each new object, determine its interior, find the intersection of that interior with the union of the interiors of the objects so far tested, and keep track of the intersection points. Then add the interior points of the new object to the union of the other objects' interiors.

You can perform the operations described in the preceding paragraph by using different bits in the stencil buffer together with various masking operations. Three bits of stencil buffer are required per pixel—one for the toggling to determine the interior of each object, one for the union of all interiors discovered so far, and one for the regions where interference has occurred so far. To make this discussion more concrete, assume the 1 bit of the stencil buffer is for toggling interior/exterior, the 2 bit is the running union, and the 4 bit is for interferences so far. For each object that you're going to render, clear the 1 bit (using a stencil mask of one and clearing to zero), then toggle the 1 bit by keeping the stencil mask as one and using the `GL_INVERT` stencil operation.

You can find intersections and unions of the bits in the stencil buffers using the stenciling operations. For example, to make bits in buffer 2 be the union of the bits in buffers 1 and 2, mask the stencil to those 2 bits, and draw something over the entire object with the stencil function set to pass if anything nonzero occurs. This happens if the bits in buffer 1, buffer 2, or both are turned on. If the comparison succeeds, write a 1 in buffer 2. Also, make sure that drawing in the color buffer is disabled. An intersection calculation is similar—set the function to pass only if the value in the two buffers is equal to 3 (bits turned on in both buffers 1 and 2). Write the result into the correct buffer. (See “Stencil Test” on page 385.)

Shadows

Every possible projection of three-dimensional space to three-dimensional space can be achieved with a suitable 4×4 invertible matrix and homogeneous coordinates. If the matrix isn't invertible but has rank 3, it projects three-dimensional space onto a two-dimensional plane. Every such possible projection can be achieved with a suitable rank-3 4×4 matrix. To find the shadow of an arbitrary object on an arbitrary plane from an

arbitrary light source (possibly at infinity), you need to find a matrix representing that projection, multiply it on the matrix stack, and draw the object in the shadow color. Keep in mind that you need to project onto each plane that you're calling the "ground."

As a simple illustration, assume the light is at the origin, and the equation of the ground plane is $ax+by+c+d=0$. Given a vertex $S=(sx, sy, sz, 1)$, the line from the light through S includes all points αS , where α is an arbitrary real number. The point where this line intersects the plane occurs when

$$\alpha(a*sz+b*sy+c*sz) + d = 0,$$

so

$$\alpha = -d/(a*sx+b*sy+c*sz).$$

Plugging this back into the line, we get

$$-d(sx, sy, sz)/(a*sx+b*sy+c*sz)$$

for the point of intersection.

The matrix that maps S to this point for every S is

$$\begin{bmatrix} -d & 0 & 0 & a \\ 0 & -d & 0 & b \\ 0 & 0 & -d & c \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

This matrix can be used if you first translate the world so that the light is at the origin.

If the light is from an infinite source, all you have is a point S and a direction $D = (dx, dy, dz)$. Points along the line are given by

$$S + \alpha D$$

Proceeding as before, the intersection of this line with the plane is given by

$$a(sx+\alpha dx)+b(sy+\alpha dy)+c(sz+\alpha dz)+d = 0$$

Solving for α , plugging that back into the equation for a line, and then determining a projection matrix gives

$$\begin{bmatrix} b*dy+c*dz & -a*dy & -a*dz & 0 \\ -b*dx & a*dx+c*dz & -b*dz & 0 \\ -c*dx & -c*dy & a*dx+b*dy & 0 \\ -d*dx & -d*dy & -d*dz & a*dx+b*dy*c*dz \end{bmatrix}$$

This matrix works given the plane and an arbitrary direction vector. There's no need to translate anything first. (See Chapter 3 and Appendix F.)

Hidden-Line Removal

If you want to draw a wireframe object with hidden lines removed, one approach is to draw the outlines using lines and then fill the interiors of the polygons making up the surface with polygons having the background color. With depth-buffering enabled, this interior fill covers any outlines that would be obscured by faces closer to the eye. This method would work, except that there's no guarantee that the interior of the object falls entirely inside the polygon's outline; in fact, it might overlap it in various places.

There's an easy, two-pass solution using either polygon offset or the stencil buffer. Polygon offset is usually the preferred technique, since polygon offset is almost always faster than stencil buffer. Both methods are described here, so you can see how both approaches to the problem work.

Hidden-Line Removal with Polygon Offset

To use polygon offset to accomplish hidden-line removal, the object is drawn twice. The highlighted edges are drawn in the foreground color, using filled polygons but with the polygon mode `GL_LINE` to rasterize it as a wireframe. Then the filled polygons are drawn with the default polygon mode, which fills the interior of the wireframe, and with enough polygon offset to nudge the filled polygons a little farther from the eye. With the polygon offset, the interior recedes just enough that the highlighted edges are drawn without unpleasant visual artifacts.

```
glEnable(GL_DEPTH_TEST);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
set_color(foreground);
draw_object_with_filled_polygons();

glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(1.0, 1.0);
set_color(background);
draw_object_with_filled_polygons();
glDisable(GL_POLYGON_OFFSET_FILL);
```

You may need to adjust the amount of offset needed (for wider lines, for example). (See “Polygon Offset” on page 247 for more information.)

Hidden-Line Removal with the Stencil Buffer

Using the stencil buffer for hidden-line removal is a more complicated procedure. For each polygon, you’ll need to clear the stencil buffer, and then draw the outline both in the framebuffer and in the stencil buffer. Then when you fill the interior, enable drawing only where the stencil buffer is still clear. To avoid doing an entire stencil-buffer clear for each polygon, an easy way to clear it is simply to draw 0’s into the buffer using the same polygon outline. In this way, you need to clear the entire stencil buffer only once.

For example, the following code represents the inner loop you might use to perform such hidden-line removal. Each polygon is outlined in the foreground color, filled with the background color, and then outlined again in the foreground color. The stencil buffer is used to keep the fill color of each polygon from overwriting its outline. To optimize performance, the stencil and color parameters are changed only twice per loop by using the same values both times the polygon outline is drawn.

```
glEnable(GL_STENCIL_TEST);
glEnable(GL_DEPTH_TEST);
glClear(GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_ALWAYS, 0, 1);
glStencilOp(GL_INVERT, GL_INVERT, GL_INVERT);
set_color(foreground);
for (i=0; i < max; i++) {
    outline_polygon(i);
    set_color(background);
    glStencilFunc(GL_EQUAL, 0, 1);
    glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
    fill_polygon(i);
    set_color(foreground);
    glStencilFunc(GL_ALWAYS, 0, 1);
    glStencilOp(GL_INVERT, GL_INVERT, GL_INVERT);
    outline_polygon(i);
}
```

(See “Stencil Test” on page 385.)

Texture-Mapping Applications

Texture mapping is quite powerful, and it can be used in some interesting ways. Here are a few advanced applications of texture mapping.

- **Antialiased text**—Define a texture map for each character at a relatively high resolution, and then map them onto smaller areas using the filtering provided by texturing. This also makes text appear correctly on surfaces that aren't aligned with the screen, but are tilted and have some perspective distortion.
- **Antialiased lines**—These can be done like antialiased text: Make the line in the texture several pixels wide, and use the texture filtering to antialias the lines.
- **Image scaling and rotation**—If you put an image into a texture map and use that texture to map onto a polygon, rotating and scaling the polygon effectively rotates and scales the image.
- **Image warping**—As in the preceding example, store the image as a texture map, but map it to some spline-defined surface (use evaluators). As you warp the surface, the image follows the warping.
- **Projecting images**—Put the image in a texture map, and project it as a spotlight, creating a slide projector effect. (See "The q Coordinate" on page 372 for more information about how to model a spotlight using textures.)

(See Chapter 3 for information about rotating and scaling, Chapter 9 for more information about creating textures, and Chapter 12 for details on evaluators.)

Drawing Depth-Buffered Images

For complex static backgrounds, the rendering time for the geometric description of the background can be greater than the time it takes to draw a pixel image of the rendered background. If there's a fixed background and a relatively simple changing foreground, you may want to draw the background and its associated depth-buffered version as an image rather than render it geometrically. The foreground might also consist of items that are time-consuming to render, but whose framebuffer images and depth buffers are available. You can render these items into a depth-buffered environment using a two-pass algorithm.

For example, if you're drawing a model of a molecule made of spheres, you might have an image of a beautifully rendered sphere and its associated depth-buffer values that were calculated using Phong shading or ray-tracing or by using some other scheme that isn't directly available through OpenGL. To draw a complex model, you might be required to draw hundreds of such spheres, which should be depth-buffered together.

To add a depth-buffered image to the scene, first draw the image's depth-buffer values into the depth buffer using `glDrawPixels()`. Then enable depth-buffering, set the writemask to zero so that no drawing occurs, and enable stenciling such that the stencil buffers get drawn whenever a write to the depth buffer occurs.

Then draw the image into the color buffer, masked by the stencil buffer you've just written so that writing occurs only when there's a 1 in the stencil buffer. During this write, set the stenciling function to zero out the stencil buffer so that it's automatically cleared when it's time to add the next image to the scene. If the objects are to be moved nearer to or farther from the viewer, you need to use an orthographic projection; in these cases, you use `GL_DEPTH_BIAS` with `glPixelTransfer*()` to move the depth image. (See "Coordinate System Survival Kit" on page 35, "Depth Test" on page 391, "Stencil Test" on page 385, and Chapter 8 for details on `glDrawPixels()` and `glPixelTransfer*()`.)

Dirichlet Domains

Given a set S of points on a plane, the Dirichlet domain or Voronoi polygon of one of the points is the set of all points in the plane closer to that point than to any other point in the set S . These points provide the solution to many problems in computational geometry. Figure 14-3 shows outlines of the Dirichlet domains for a set of points.

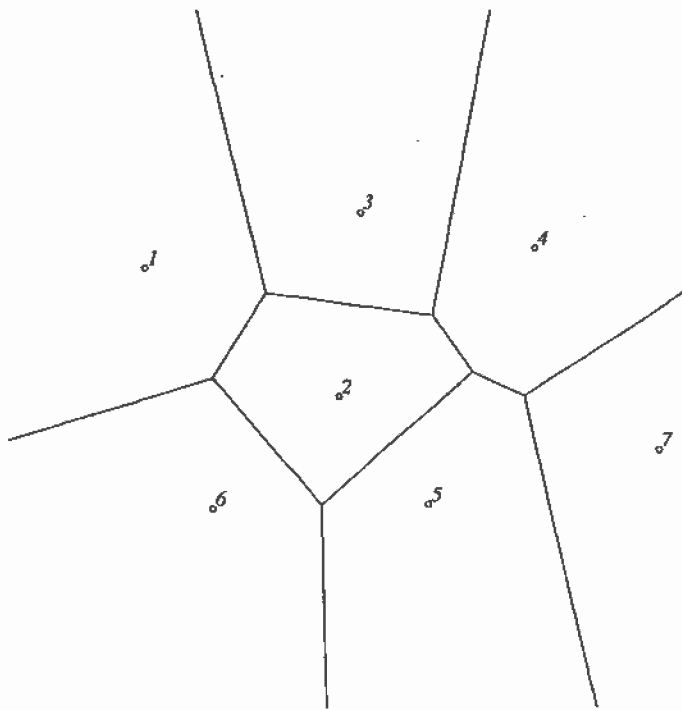


Figure 14-3 Dirichlet Domains

If you draw a depth-buffered cone with its apex at the point in a different color than each of the points in S , the Dirichlet domain for each point is drawn in that color. The easiest way to do this is to precompute a cone's depth in an image and use the image as the depth-buffer values as described in the preceding section. You don't need an image to draw in the framebuffer as in the case of shaded spheres, however. While you're drawing into the depth buffer, use the stencil buffer to record the pixels where drawing should occur by first clearing it and then writing nonzero values wherever the depth test succeeds. To draw the Dirichlet region, draw a polygon over the entire window, but enable drawing only where the stencil buffers are nonzero.

You can do this perhaps more easily by rendering cones of uniform color with a simple depth buffer, but a good cone might require thousands of polygons. The technique described in this section can render much higher-quality cones much more quickly. (See "A Hidden-Surface Removal Survival Kit" on page 171 and "Depth Test" on page 391.)

Life in the Stencil Buffer

The Game of Life, invented by John Conway, is played on a rectangular grid where each grid location is "alive" or "dead." To calculate the next generation from the current one, count the number of live neighbors for each grid location (the eight adjacent grid locations are neighbors). A grid location is alive in generation $n+1$ if it was alive in generation n and has exactly two or three live neighbors, or if it was dead in generation n and has exactly three live neighbors. In all other cases, it is dead in generation $n+1$. This game generates some incredibly interesting patterns given different initial configurations. (See Martin Gardner, "Mathematical Games," *Scientific American*, vol. 223, no. 4, October 1970, p. 120–123.) Figure 14-4 shows six generations from a game.

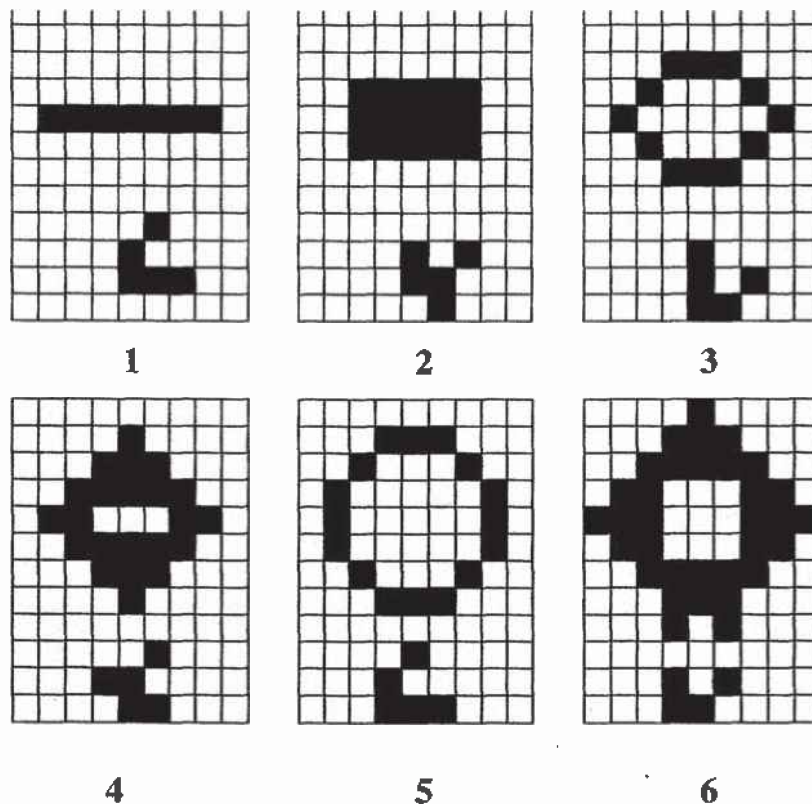


Figure 14-4 Six Generations from the Game of Life

One way to create this game using OpenGL is to use a multipass algorithm. Keep the data in the color buffer, one pixel for each grid point. Assume that black (all zeros) is the background color, and the color of a live pixel is

nonzero. Initialize by clearing the depth and stencil buffers to zero, set the depth-buffer writemask to zero, and set the depth comparison function so that it passes on not-equal. To iterate, read the image off the screen, enable drawing into the depth buffer, and set the stencil function so that it increments whenever a depth comparison succeeds but leaves the stencil buffer unchanged otherwise. Disable drawing into the color buffer.

Next, draw the image eight times, offset one pixel in each vertical, horizontal, and diagonal direction. When you're done, the stencil buffer contains a count of the number of live neighbors for each pixel. Enable drawing to the color buffer, set the color to the color for live cells, and set the stencil function to draw only if the value in the stencil buffer is 3 (three live neighbors). In addition, if this drawing occurs, decrement the value in the stencil buffer. Then draw a rectangle covering the image; this paints each cell that has exactly three live neighbors with the "alive" color.

At this point, the stencil buffers contain 0, 1, 2, 4, 5, 6, 7, 8, and the values under the 2's are correct. The values under 0, 1, 4, 5, 6, 7, and 8 must be cleared to the "dead" color. Set the stencil function to draw whenever the value is not 2, and to zero the stencil values in all cases. Then draw a large polygon of the "dead" color across the entire image. You're done.

For a usable demonstration program, you might want to zoom the grid up to a size larger than a single pixel; it's hard to see detailed patterns with a single pixel per grid point. (See "Coordinate System Survival Kit" on page 35, "Depth Test" on page 391, and "Stencil Test" on page 385.)

Alternative Uses for `glDrawPixels()` and `glCopyPixels()`

You might think of `glDrawPixels()` as a way to draw a rectangular region of pixels to the screen. Although this is often what it's used for, some other interesting uses are outlined here.

- **Video**—Even if your machine doesn't have special video hardware, you can display short movie clips by repeatedly drawing frames with `glDrawPixels()` in the same region of the back buffer and then swapping the buffers. The size of the frames you can display with reasonable performance using this method depends on your hardware's drawing speed, so you might be limited to 100×100 pixel movies (or smaller) if you want smooth fake video.

-
- **Airbrush**—In a paint program, your airbrush (or paintbrush) shape can be simulated using alpha values. The color of the paint is represented as the color values. To paint with a circular brush in blue, repeatedly draw a blue square with `glDrawPixels()` where the alpha values are largest in the center and taper to zero at the edges of a circle centered in the square. Draw using a blending function that uses alpha of the incoming color and $(1-\text{alpha})$ of the color already at the pixel. If the alpha values in the brush are all much less than one, you have to paint over an area repeatedly to get a solid color. If the alpha values are near one, each brush stroke pretty much obliterates the colors underneath.
 - **Filtered Zooms**—If you zoom a pixel image by a nonintegral amount, OpenGL effectively uses a box filter, which can lead to rather severe aliasing effects. To improve the filtering, jitter the resulting image by amounts less than a pixel and redraw it multiple times, using alpha blending to average the resulting pixels. The result is a filtered zoom.
 - **Transposing Images**—You can swap same-size images in place with `glCopyPixels()` using the XOR operation. With this method, you can avoid having to read the images back into processor memory. If A and B represent the two images, the operation looks like this:
 - a. $A = A \text{ XOR } B$
 - b. $B = A \text{ XOR } B$
 - c. $A = A \text{ XOR } B$

Order of Operations

This book describes all the operations performed between when vertices are initially specified and fragments are finally written into the framebuffer. The chapters of this book are arranged in an order that facilitates learning rather than in the exact order in which these operations are actually performed. Sometimes the exact order of operations doesn't matter—for example, surfaces can be converted to polygons and then transformed, or transformed first and then converted to polygons, with identical results—and different implementations of OpenGL might do things differently.

This appendix describes a possible order; any implementation is required to give equivalent results. If you want more details than are presented here, see the *OpenGL Reference Manual*.

This appendix has the following major sections:

- “Overview” on page 530
- “Geometric Operations” on page 530
- “Pixel Operations” on page 532
- “Fragment Operations” on page 533
- “Odds and Ends” on page 533

Overview

This section gives an overview of the order of operations, as shown in Figure A-1. Geometric data (vertices, lines, and polygons) follows the path through the row of boxes that include evaluators and per-vertex operations, while pixel data (pixels, images, and bitmaps) is treated differently for part of the process. Both types of data undergo the rasterization and per-fragment operations before the final pixel data is written into the framebuffer.

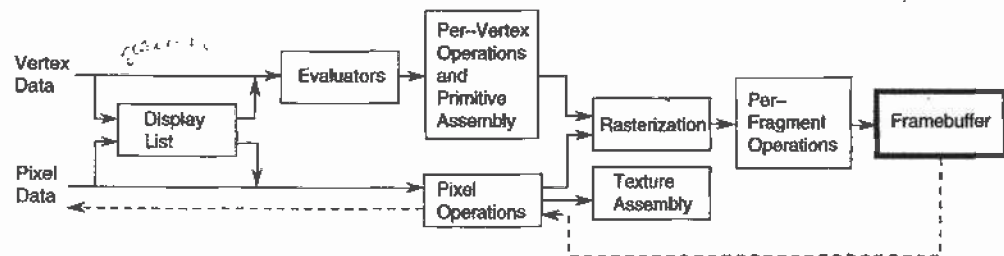


Figure A-1 Order of Operations

All data, whether it describes geometry or pixels, can be saved in a display list or processed immediately. When a display list is executed, the data is sent from the display list just as if it were sent by the application.

All geometric primitives are eventually described by vertices. If evaluators are used, that data is converted to vertices and treated as vertices from then on. Vertex data may also be stored in and used from specialized vertex arrays. Per-vertex calculations are performed on each vertex, followed by rasterization to fragments. For pixel data, pixel operations are performed, and the results are either stored in the texture memory, used for polygon stippling, or rasterized to fragments.

Finally, the fragments are subjected to a series of per-fragment operations, after which the final pixel values are drawn into the framebuffer.

Geometric Operations

Geometric data, whether it comes from a display list, an evaluator, the vertices of a rectangle, or as raw data, consists of a set of vertices and the type of primitive it describes (a vertex, line, or polygon). Vertex data includes not only the (x, y, z, w) coordinates, but also a normal vector,

texture coordinates, a RGBA color, a color index, material properties, and edge-flag data. All these elements except the vertex's coordinates can be specified in any order, and default values exist as well. As soon as the vertex command `glVertex*()` is issued, the components are padded, if necessary, to four dimensions (using $z = 0$ and $w = 1$), and the current values of all the elements are associated with the vertex. The complete set of vertex data is then processed. (If vertex arrays are used, vertex data may be batch processed and processed vertices may be reused.)

Per-Vertex Operations

In the per-vertex operations stage of processing, each vertex's spatial coordinates are transformed by the modelview matrix, while the normal vector is transformed by that matrix's inverse transpose and renormalized if specified. If automatic texture generation is enabled, new texture coordinates are generated from the transformed vertex coordinates, and they replace the vertex's old texture coordinates. The texture coordinates are then transformed by the current texture matrix and passed on to the primitive assembly step.

Meanwhile, the lighting calculations, if enabled, are performed using the transformed vertex and normal vector coordinates, and the current material, lights, and lighting model. These calculations generate new colors or indices that are clamped or masked to the appropriate range and passed on to the primitive assembly step.

Primitive Assembly

Primitive assembly differs, depending on whether the primitive is a point, a line, or a polygon. If flat shading is enabled, the colors or indices of all the vertices in a line or polygon are set to the same value. If special clipping planes are defined and enabled, they're used to clip primitives of all three types. (The clipping-plane equations are transformed by the inverse transpose of the modelview matrix when they're specified.) Point clipping simply passes or rejects vertices; line or polygon clipping can add additional vertices depending on how the line or polygon is clipped. After this clipping, the spatial coordinates of each vertex are transformed by the projection matrix, and the results are clipped against the standard viewing planes $x = \pm w$, $y = \pm w$, and $z = \pm w$.

If selection is enabled, any primitive not eliminated by clipping generates a selection-hit report, and no further processing is performed. Without selection, perspective division by w occurs and the viewport and depth-range operations are applied. Also, if the primitive is a polygon, it's then subjected to a culling test (if culling is enabled). A polygon might convert to vertices or lines, depending on the polygon mode.

Finally, points, lines, and polygons are rasterized to fragments, taking into account polygon or line stipples, line width, and point size. Rasterization involves determining which squares of an integer grid in window coordinates are occupied by the primitive. If antialiasing is enabled, coverage (the portion of the square that is occupied by the primitive) is also computed. Color and depth values are also assigned to each such square. If polygon offset is enabled, depth values are slightly modified by a calculated offset value.

Pixel Operations

Pixels from host memory are first unpacked into the proper number of components. The OpenGL unpacking facility handles a number of different formats. Next, the data is scaled, biased, and processed using a pixel map. The results are clamped to an appropriate range depending on the data type and then either written in the texture memory for use in texture mapping or rasterized to fragments.

If pixel data is read from the framebuffer, pixel-transfer operations (scale, bias, mapping, and clamping) are performed. The results are packed into an appropriate format and then returned to processor memory.

The pixel copy operation is similar to a combination of the unpacking and transfer operations, except that packing and unpacking is unnecessary, and only a single pass is made through the transfer operations before the data is written back into the framebuffer.

Texture Memory

OpenGL Version 1.1 provides additional control over texture memory. Texture image data can be specified from framebuffer memory, as well as processor memory. All or a portion of a texture image may be replaced. Texture data may be stored in texture objects, which can be loaded into texture memory. If there are too many texture objects to fit into texture

memory at the same time, the textures that have the highest priorities remain in the texture memory.

Fragment Operations

If texturing is enabled, a texel is generated from texture memory for each fragment and applied to the fragment. Then fog calculations are performed, if they're enabled, followed by the application of coverage (antialiasing) values, if antialiasing is enabled.

Next comes scissoring, followed by the alpha test (in RGBA mode only), the stencil test, and the depth-buffer test. If in RGBA mode, blending is performed. Blending is followed by dithering and logical operation. All these operations may be disabled.

The fragment is then masked by a color mask or an index mask, depending on the mode, and drawn into the appropriate buffer. If fragments are being written into the stencil or depth buffer, masking occurs after the stencil and depth tests, and the results are drawn into the framebuffer without performing the blending, dithering, or logical operation.

Odds and Ends

Matrix operations deal with the current matrix stack, which can be the modelview, the projection, or the texture matrix stack. The commands `glMultMatrix*()`, `glLoadMatrix*()`, and `glLoadIdentity()` are applied to the top matrix on the stack, while `glTranslate*()`, `glRotate*()`, `glScale*()`, `glOrtho()`, and `glFrustum()` are used to create a matrix that's multiplied by the top matrix. When the modelview matrix is modified, its inverse transpose is also generated for normal vector transformation.

The commands that set the current raster position are treated exactly like a vertex command up until when rasterization would occur. At this point, the value is saved and is used in the rasterization of pixel data.

The various `glClear()` commands bypass all operations except scissoring, dithering, and writemasking.

State Variables

This appendix lists the queryable OpenGL state variables, their default values, and the commands for obtaining the values of these variables. The *OpenGL Reference Manual* contains detailed information on all the commands and constants discussed in this appendix. This appendix has these major sections:

- “The Query Commands” on page 536
- “OpenGL State Variables” on page 537

The Query Commands

In addition to the basic commands to obtain the values of simple state variables (commands such as `glGetIntegerv()` and `glIsEnabled()`, which are described in “Basic State Management” on page 48), there are other specialized commands to return more complex state variables. The prototypes for these specialized commands are listed here. Some of these routines, such as `glGetError()` and `glGetString()`, have been discussed in more detail elsewhere in the book.

To find out when you need to use these commands and their corresponding symbolic constants, use the tables in the next section, “OpenGL State Variables” on page 537. Also see the *OpenGL Reference Manual*.

```
void glGetClipPlane(GLenum plane, GLdouble *equation);
GLenum glGetError(void);
void glGetLight{if}v(GLenum light, GLenum pname, TYPE *params);
void glGetMap{ifd}v(GLenum target, GLenum query, TYPE *v);
void glGetMaterial{if}v(GLenum face, GLenum pname, TYPE *params);
void glGetPixelMap{f ui us}v(GLenum map, TYPE *values);
void glGetPolygonStipple(GLubyte *mask);
const GLubyte * glGetString(GLenum name);
void glGetTexEnv{if}v(GLenum target, GLenum pname, TYPE *params);
void glGetTexGen{ifd}v(GLenum coord, GLenum pname, TYPE *params);
void glGetTexImage(GLenum target, GLint level, GLenum format, GLenum
    type, GLvoid *pixels);
void glGetTexLevelParameter{if}v(GLenum target, GLint level, GLenum
    pname, TYPE *params);
void glGetTexParameter{if}v(GLenum target, GLenum pname, TYPE
    *params);
void gluGetNurbsProperty(GLUnurbsObj *nobj, GLenum property, GLfloat
    *value);
const GLubyte * gluGetString(GLenum name);
void gluGetTessProperty(GLUtesselator *tess, GLenum which, GLdouble
    *data);
```

OpenGL State Variables

The following pages contain tables that list the names of queryable state variables. For each variable, the tables list a description of it, its attribute group, its initial or minimum value, and the suggested `glGet*()` command to use for obtaining it. State variables that can be obtained using `glGetBooleanv()`, `glGetIntegerv()`, `glGetFloatv()`, or `glGetDoublev()` are listed with just one of these commands—the one that’s most appropriate given the type of data to be returned. (Some vertex array variables can be queried only with `glGetPointerv()`.) These state variables can’t be obtained using `glIsEnabled()`. However, state variables for which `glIsEnabled()` is listed as the query command can also be obtained using `glGetBooleanv()`, `glGetIntegerv()`, `glGetFloatv()`, and `glGetDoublev()`. State variables for which any other command is listed as the query command can be obtained only by using that command.

If one or more attribute groups are listed, the state variable belongs to the listed group or groups. If no attribute group is listed, the variable doesn’t belong to any group. `glPushAttrib()`, `glPushClientAttrib()`, `glPopAttrib()`, and `glPopClientAttrib()` may be used to save and restore all state values that belong to an attribute group. (See “Attribute Groups” on page 78 for more information.)

All queryable state variables, except the implementation-dependent ones, have initial values. If no initial value is listed, you need to consult either the section where that variable is discussed or the *OpenGL Reference Manual* to determine its initial value.

Current Values and Associated Data

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_CURRENT_COLOR	Current color	current	1, 1, 1, 1	glGetIntegerv(), glGetFloatv()
GL_CURRENT_INDEX	Current color index	current	1	glGetIntegerv(), glGetFloatv()
GL_CURRENT_TEXTURE_COORDS	Current texture coordinates	current	0, 0, 0, 1	glGetFloatv()
GL_CURRENT_NORMAL	Current normal	current	0, 0, 1	glGetFloatv()
GL_CURRENT_RASTER_POSITION	Current raster position	current	0, 0, 0, 1	glGetFloatv()
GL_CURRENT_RASTER_DISTANCE	Current raster distance	current	0	glGetFloatv()
GL_CURRENT_RASTER_COLOR	Color associated with raster position	current	1, 1, 1, 1	glGetIntegerv(), glGetFloatv()
GL_CURRENT_RASTER_INDEX	Color index associated with raster position	current	1	glGetIntegerv(), glGetFloatv()
GL_CURRENT_RASTER_TEXTURE_COORDS	Texture coordinates associated with raster position	current	0, 0, 0, 1	glGetFloatv()
GL_CURRENT_RASTER_POSITION_VALID	Raster position valid bit	current	GL_TRUE	glGetBooleanv()
GL_EDGE_FLAG	Edge flag	current	GL_TRUE	glGetBooleanv()

Table B-1 State Variables for Current Values and Associated Data

Vertex Array

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_VERTEX_ARRAY	Vertex array enable	vertex-array	GL_FALSE	glIsEnabled()
GL_VERTEX_ARRAY_SIZE	Coordinates per vertex	vertex-array	4	glGetIntegerv()
GL_VERTEX_ARRAY_TYPE	Type of vertex coordinates	vertex-array	GL_FLOAT	glGetIntegerv()
GL_VERTEX_ARRAY_STRIDE	Stride between vertices	vertex-array	0	glGetIntegerv()
GL_VERTEX_ARRAY_POINTER	Pointer to the vertex array	vertex-array	NULL	glGetPointerv()
GL_NORMAL_ARRAY	Normal array enable	vertex-array	GL_FALSE	glIsEnabled()
GL_NORMAL_ARRAY_TYPE	Type of normal coordinates	vertex-array	GL_FLOAT	glGetIntegerv()
GL_NORMAL_ARRAY_STRIDE	Stride between normals	vertex-array	0	glGetIntegerv()
GL_NORMAL_ARRAY_POINTER	Pointer to the normal array	vertex-array	NULL	glGetPointerv()
GL_COLOR_ARRAY	RGBA color array enable	vertex-array	GL_FALSE	glIsEnabled()
GL_COLOR_ARRAY_SIZE	Colors per vertex	vertex-array	4	glGetIntegerv()
GL_COLOR_ARRAY_TYPE	Type of color components	vertex-array	GL_FLOAT	glGetIntegerv()
GL_COLOR_ARRAY_STRIDE	Stride between colors	vertex-array	0	glGetIntegerv()
GL_COLOR_ARRAY_POINTER	Pointer to the color array	vertex-array	NULL	glGetPointerv()
GL_INDEX_ARRAY	Color-index array enable	vertex-array	GL_FALSE	glIsEnabled()

Table B-2 Vertex Array State Variables

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_INDEX_ARRAY_TYPE	Type of color indices	vertex-array	GL_FLOAT	glGetIntegerv()
GL_INDEX_ARRAY_STRIDE	Stride between color indices	vertex-array	0	glGetIntegerv()
GL_INDEX_ARRAY_POINTER	Pointer to the index array	vertex-array	NULL	glGetPointerv()
GL_TEXTURE_COORD_ARRAY	Texture coordinate array enable	vertex-array	GL_FALSE	glIsEnabled()
GL_TEXTURE_COORD_ARRAY_SIZE	Texture coordinates per element	vertex-array	4	glGetIntegerv()
GL_TEXTURE_COORD_ARRAY_TYPE	Type of texture coordinates	vertex-array	GL_FLOAT	glGetIntegerv()
GL_TEXTURE_COORD_ARRAY_STRIDE	Stride between texture coordinates	vertex-array	0	glGetIntegerv()
GL_TEXTURE_COORD_ARRAY_POINTER	Pointer to the texture coordinate array	vertex-array	NULL	glGetPointerv()
GL_EDGE_FLAG_ARRAY	Edge flag array enable	vertex-array	GL_FALSE	glIsEnabled()
GL_EDGE_FLAG_ARRAY_STRIDE	Stride between edge flags	vertex-array	0	glGetIntegerv()
GL_EDGE_FLAG_ARRAY_POINTER	Pointer to the edge flag array	vertex-array	NULL	glGetPointerv()

Table B-2 Vertex Array State Variables (continued)

Transformation

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_MODELVIEW_MATRIX	Modelview matrix stack	—	Identity	glGetFloatv()
GL_PROJECTION_MATRIX	Projection matrix stack	—	Identity	glGetFloatv()
GL_TEXTURE_MATRIX	Texture matrix stack	—	Identity	glGetFloatv()
GL_VIEWPORT	Viewport origin and extent	viewport	—	glGetIntegerv()
GL_DEPTH_RANGE	Depth range near and far	viewport	0, 1	glGetFloatv()
GL_MODELVIEW_STACK_DEPTH	Modelview matrix stack pointer	—	1	glGetIntegerv()
GL_PROJECTION_STACK_DEPTH	Projection matrix stack pointer	—	1	glGetIntegerv()
GL_TEXTURE_STACK_DEPTH	Texture matrix stack pointer	—	1	glGetIntegerv()
GL_MATRIX_MODE	Current matrix mode	transform	GL_MODELVIEW	glGetIntegerv()
GL_NORMALIZE	Current normal normalization on/off	transform/ enable	GL_FALSE	glIsEnabled()
GL_CLIP_PLANE _{<i>i</i>}	User clipping plane coefficients	transform	0, 0, 0, 0	glGetClipPlane()
GL_CLIP_PLANE _{<i>i</i>}	<i>i</i> th user clipping plane enabled	transform/ enable	GL_FALSE	glIsEnabled()

Table B-3 Transformation State Variables

Coloring

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_FOG_COLOR	Fog color	fog	0, 0, 0, 0	glGetFloatv()
GL_FOG_INDEX	Fog index	fog	0	glGetFloatv()
GL_FOG_DENSITY	Exponential fog density	fog	1.0	glGetFloatv()
GL_FOG_START	Linear fog start	fog	0.0	glGetFloatv()
GL_FOG_END	Linear fog end	fog	1.0	glGetFloatv()
GL_FOG_MODE	Fog mode	fog	GL_EXP	glGetIntegerv()
GL_FOG	True if fog enabled	fog/enable	GL_FALSE	glIsEnabled()
GL_SHADE_MODEL	glShadeModel() setting	lighting	GL_SMOOTH	glGetIntegerv()

Table B-4 Coloring State Variables

Lighting

See also Table 5-1 on page 180 and Table 5-3 on page 196 for initial values.

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_LIGHTING	True if lighting is enabled	lighting /enable	GL_FALSE	glIsEnabled()
GL_COLOR_MATERIAL	True if color tracking is enabled	lighting	GL_FALSE	glIsEnabled()
GL_COLOR_MATERIAL_PARAMETER	Material properties tracking current color	lighting	GL_AMBIENT_AND_DIFFUSE	glGetIntegerv()
GL_COLOR_MATERIAL_FACE	Face(s) affected by color tracking	lighting	GL_FRONT_AND_BACK	glGetIntegerv()
GL_AMBIENT	Ambient material color	lighting	(0.2, 0.2, 0.2, 1.0)	glGetMaterialfv()
GL_DIFFUSE	Diffuse material color	lighting	(0.8, 0.8, 0.8, 1.0)	glGetMaterialfv()
GL_SPECULAR	Specular material color	lighting	(0.0, 0.0, 0.0, 1.0)	glGetMaterialfv()
GL_EMISSION	Emissive material color	lighting	(0.0, 0.0, 0.0, 1.0)	glGetMaterialfv()
GL_SHININESS	Specular exponent of material	lighting	0.0	glGetMaterialfv()
GL_LIGHT_MODEL_AMBIENT	Ambient scene color	lighting	(0.2, 0.2, 0.2, 1.0)	glGetFloatv()
GL_LIGHT_MODEL_LOCAL_VIEWER	Viewer is local	lighting	GL_FALSE	glGetBooleanv()
GL_LIGHT_MODEL_TWO_SIDE	Use two-sided lighting	lighting	GL_FALSE	glGetBooleanv()

Table B-5 Lighting State Variables

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_AMBIENT	Ambient intensity of light i	lighting	(0.0,0.0,0.0,1.0)	glGetLightfv()
GL_DIFFUSE	Diffuse intensity of light i	lighting	—	glGetLightfv()
GL_SPECULAR	Specular intensity of light i	lighting	—	glGetLightfv()
GL_POSITION	Position of light i	lighting	(0.0, 0.0, 1.0, 0.0)	glGetLightfv()
GL_CONSTANT_ATTENUATION	Constant attenuation factor	lighting	1.0	glGetLightfv()
GL_LINEAR_ATTENUATION	Linear attenuation factor	lighting	0.0	glGetLightfv()
GL_QUADRATIC_ATTENUATION	Quadratic attenuation factor	lighting	0.0	glGetLightfv()
GL_SPOT_DIRECTION	Spotlight direction of light i	lighting	(0.0, 0.0, -1.0)	glGetLightfv()
GL_SPOT_EXPONENT	Spotlight exponent of light i	lighting	0.0	glGetLightfv()
GL_SPOT_CUTOFF	Spotlight angle of light i	lighting	180.0	glGetLightfv()
GL_LIGHT <i>i</i>	True if light i enabled	lighting /enable	GL_FALSE	glIsEnabled()
GL_COLOR_INDEXES	c_a , c_d , and c_s for color-index lighting	lighting /enable	0, 1, 1	glGetMaterialfv()

Table B-5 Lighting State Variables (continued)

Rasterization

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_POINT_SIZE	Point size	point	1.0	glGetFloatv()
GL_POINT_SMOOTH	Point antialiasing on	point/enable	GL_FALSE	glIsEnabled()
GL_LINE_WIDTH	Line width	line	1.0	glGetFloatv()
GL_LINE_SMOOTH	Line antialiasing on	line/enable	GL_FALSE	glIsEnabled()
GL_LINE_STIPPLE_PATTERN	Line stipple	line	1's	glGetIntegerv()
GL_LINE_STIPPLE_REPEAT	Line stipple repeat	line	1	glGetIntegerv()
GL_LINE_STIPPLE	Line stipple enable	line/enable	GL_FALSE	glIsEnabled()
GL_CULL_FACE	Polygon culling enabled	polygon/enable	GL_FALSE	glIsEnabled()
GL_CULL_FACE_MODE	Cull front-/back-facing polygons	polygon	GL_BACK	glGetIntegerv()
GL_FRONT_FACE	Polygon front-face CW/CCW indicator	polygon	GL_CCW	glGetIntegerv()
GL_POLYGON_SMOOTH	Polygon antialiasing on	polygon/enable	GL_FALSE	glIsEnabled()
GL_POLYGON_MODE	Polygon rasterization mode (front and back)	polygon	GL_FILL	glGetIntegerv()
GL_POLYGON_OFFSET_FACTOR	Polygon offset factor	polygon	0	glGetFloatv()
GL_POLYGON_OFFSET_BIAS	Polygon offset bias	polygon	0	glGetFloatv()

Table B-6 Rasterization State Variables

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_POLYGON_OFFSET_POINT	Polygon offset enable for GL_POINT mode rasterization	polygon/enable	GL_FALSE	glIsEnabled()
GL_POLYGON_OFFSET_LINE	Polygon offset enable for GL_LINE mode rasterization	polygon/enable	GL_FALSE	glIsEnabled()
GL_POLYGON_OFFSET_FILL	Polygon offset enable for GL_FILL mode rasterization	polygon/enable	GL_FALSE	glIsEnabled()
GL_POLYGON_STIPPLE	Polygon stipple enable	polygon/enable	GL_FALSE	glIsEnabled()
—	Polygon stipple pattern	polygon-stipple	1's	glGetPolygon-Stipple()

Table B-6 Rasterization State Variables (continued)

Texturing

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_TEXTURE_1D	True if 1-D texturing enabled	texture/enable	GL_FALSE	glIsEnabled()
GL_TEXTURE_2D	True if 2-D texturing enabled	texture/enable	GL_FALSE	glIsEnabled()
GL_TEXTURE_BINDING_1D	Texture object bound to GL_TEXTURE_1D	texture	GL_FALSE	glGetIntegerv()
GL_TEXTURE_BINDING_2D	Texture object bound to GL_TEXTURE_2D	texture	GL_FALSE	glGetIntegerv()
GL_TEXTURE	x -D texture image at level of detail i	—	—	glGetTexImage()
GL_TEXTURE_WIDTH	x -D texture image i 's width	—	0	glGetTexLevelParameter*()
GL_TEXTURE_HEIGHT	x -D texture image i 's height	—	0	glGetTexLevelParameter*()
GL_TEXTURE_BORDER	x -D texture image i 's border width	—	0	glGetTexLevelParameter*()
GL_TEXTURE_INTERNAL_FORMAT	x -D texture image i 's internal image format	—	1	glGetTexLevelParameter*()
GL_TEXTURE_RED_SIZE	x -D texture image i 's red resolution	—	0	glGetTexLevelParameter*()
GL_TEXTURE_GREEN_SIZE	x -D texture image i 's green resolution	—	0	glGetTexLevelParameter*()
GL_TEXTURE_BLUE_SIZE	x -D texture image i 's blue resolution	—	0	glGetTexLevelParameter*()
GL_TEXTURE_ALPHA_SIZE	x -D texture image i 's alpha resolution	—	0	glGetTexLevelParameter*()

Table B-7 Texturing State Variables

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_TEXTURE_LUMINANCE_SIZE	x -D texture image i 's luminance resolution	—	0	glGetTexLevelParameter*()
GL_TEXTURE_INTENSITY_SIZE	x -D texture image i 's intensity resolution	—	0	glGetTexLevelParameter*()
GL_TEXTURE_BORDER_COLOR	Texture border color	texture	0, 0, 0, 0	glGetTexParameter*()
GL_TEXTURE_MIN_FILTER	Texture minification function	texture	GL_NEAREST_MIPMAP_LINEAR	glGetTexParameter*()
GL_TEXTURE_MAG_FILTER	Texture magnification function	texture	GL_LINEAR	glGetTexParameter*()
GL_TEXTURE_WRAP_x	Texture wrap mode (x is S or T)	texture	GL_REPEAT	glGetTexParameter*()
GL_TEXTURE_PRIORITY	Texture object priority	texture	1	glGetTexParameter*()
GL_TEXTURE_RESIDENCY	Texture residency	texture	GL_FALSE	glGetTexParameteriv()
GL_TEXTURE_ENV_MODE	Texture application function	texture	GL_MODULATE	glGetTexEnviv()
GL_TEXTURE_ENV_COLOR	Texture environment color	texture	0, 0, 0, 0	glGetTexEnvfv()
GL_TEXTURE_GEN_x	Texgen enabled (x is S, T, R, or Q)	texture/ enable	GL_FALSE	glIsEnabled()
GL_EYE_PLANE	Texgen plane equation coefficients	texture	—	glGetTexGenfv()
GL_OBJECT_PLANE	Texgen object linear coefficients	texture	—	glGetTexGenfv()

Table B-7 Texturing State Variables (continued)

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_TEXTURE_GEN_MODE	Function used for texgen	texture	GL_EYE_LINEAR	glGetTexGeniv()

Table B-7 Texturing State Variables (continued)

Pixel Operations

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_SCISSOR_TEST	Scissoring enabled	scissor/enable	GL_FALSE	glIsEnabled()
GL_SCISSOR_BOX	Scissor box	scissor	—	glGetIntegerv()
GL_ALPHA_TEST	Alpha test enabled	color-buffer/enable	GL_FALSE	glIsEnabled()
GL_ALPHA_TEST_FUNC	Alpha test function	color-buffer	GL_ALWAYS	glGetIntegerv()
GL_ALPHA_TEST_REF	Alpha test reference value	color-buffer	0	glGetIntegerv()
GL_STENCIL_TEST	Stenciling enabled	stencil-buffer/enable	GL_FALSE	glIsEnabled()
GL_STENCIL_FUNC	Stencil function	stencil-buffer	GL_ALWAYS	glGetIntegerv()
GL_STENCIL_VALUE_MASK	Stencil mask	stencil-buffer	1's	glGetIntegerv()
GL_STENCIL_REF	Stencil reference value	stencil-buffer	0	glGetIntegerv()

Table B-8 Pixel Operations

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_STENCIL_FAIL	Stencil fail action	stencil-buffer	GL_KEEP	glGetIntegerv()
GL_STENCIL_PASS_DEPTH_FAIL	Stencil depth buffer fail action	stencil-buffer	GL_KEEP	glGetIntegerv()
GL_STENCIL_PASS_DEPTH_PASS	Stencil depth buffer pass action	stencil-buffer	GL_KEEP	glGetIntegerv()
GL_DEPTH_TEST	Depth buffer enabled	depth-buffer/enable	GL_FALSE	glIsEnabled()
GL_DEPTH_FUNC	Depth buffer test function	depth-buffer	GL_LESS	glGetIntegerv()
GL_BLEND	Blending enabled	color-buffer/enable	GL_FALSE	glIsEnabled()
GL_BLEND_SRC	Blending source function	color-buffer	GL_ONE	glGetIntegerv()
GL_BLEND_DST	Blending destination function	color-buffer	GL_ZERO	glGetIntegerv()
GL_DITHER	Dithering enabled	color-buffer/enable	GL_TRUE	glIsEnabled()
GL_INDEX_LOGIC_OP	Color index logical operation enabled	color-buffer/enable	GL_FALSE	glIsEnabled()
GL_COLOR_LOGIC_OP	RGBA color logical operation enabled	color-buffer/enable	GL_FALSE	glIsEnabled()
GL_LOGIC_OP_MODE	Logical operation function	color-buffer	GL_COPY	glGetIntegerv()

Table B-8 Pixel Operations (continued)

Framebuffer Control

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_DRAW_BUFFER	Buffers selected for drawing	color-buffer	—	glGetIntegerv()
GL_INDEX_WRITEMASK	Color-index writemask	color-buffer	1's	glGetIntegerv()
GL_COLOR_WRITEMASK	Color write enables; R, G, B, or A	color-buffer	GL_TRUE	glGetBooleanv()
GL_DEPTH_WRITEMASK	Depth buffer enabled for writing	depth-buffer	GL_TRUE	glGetBooleanv()
GL_STENCIL_WRITEMASK	Stencil-buffer writemask	stencil-buffer	1's	glGetIntegerv()
GL_COLOR_CLEAR_VALUE	Color-buffer clear value (RGBA mode)	color-buffer	0, 0, 0, 0	glGetFloatv()
GL_INDEX_CLEAR_VALUE	Color-buffer clear value (color-index mode)	color-buffer	0	glGetFloatv()
GL_DEPTH_CLEAR_VALUE	Depth-buffer clear value	depth-buffer	1	glGetIntegerv()
GL_STENCIL_CLEAR_VALUE	Stencil-buffer clear value	stencil-buffer	0	glGetIntegerv()
GL_ACCUM_CLEAR_VALUE	Accumulation-buffer clear value	accum-buffer	0	glGetFloatv()

Table B-9 Framebuffer Control State Variables

Pixels

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_UNPACK_SWAP_BYTES	Value of GL_UNPACK_SWAP_BYTES	pixel-store	GL_FALSE	glGetBooleanv()
GL_UNPACK_LSB_FIRST	Value of GL_UNPACK_LSB_FIRST	pixel-store	GL_FALSE	glGetBooleanv()
GL_UNPACK_ROW_LENGTH	Value of GL_UNPACK_ROW_LENGTH	pixel-store	0	glGetIntegerv()
GL_UNPACK_SKIP_ROWS	Value of GL_UNPACK_SKIP_ROWS	pixel-store	0	glGetIntegerv()
GL_UNPACK_SKIP_PIXELS	Value of GL_UNPACK_SKIP_PIXELS	pixel-store	0	glGetIntegerv()
GL_UNPACK_ALIGNMENT	Value of GL_UNPACK_ALIGNMENT	pixel-store	4	glGetIntegerv()
GL_PACK_SWAP_BYTES	Value of GL_PACK_SWAP_BYTES	pixel-store	GL_FALSE	glGetBooleanv()
GL_PACK_LSB_FIRST	Value of GL_PACK_LSB_FIRST	pixel-store	GL_FALSE	glGetBooleanv()
GL_PACK_ROW_LENGTH	Value of GL_PACK_ROW_LENGTH	pixel-store	0	glGetIntegerv()
GL_PACK_SKIP_ROWS	Value of GL_PACK_SKIP_ROWS	pixel-store	0	glGetIntegerv()
GL_PACK_SKIP_PIXELS	Value of GL_PACK_SKIP_PIXELS	pixel-store	0	glGetIntegerv()
GL_PACK_ALIGNMENT	Value of GL_PACK_ALIGNMENT	pixel-store	4	glGetIntegerv()
GL_MAP_COLOR	True if colors are mapped	pixel	GL_FALSE	glGetBooleanv()
GL_MAP_STENCIL	True if stencil values are mapped	pixel	GL_FALSE	glGetBooleanv()
GL_INDEX_SHIFT	Value of GL_INDEX_SHIFT	pixel	0	glGetIntegerv()

Table B-10 Pixel State Variables

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_INDEX_OFFSET	Value of GL_INDEX_OFFSET	pixel	0	glGetIntegerv()
GL_x_SCALE	Value of GL_x_SCALE; <i>x</i> is GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, or GL_DEPTH	pixel	1	glGetFloatv()
GL_x_BIAS	Value of GL_x_BIAS; <i>x</i> is one of GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, or GL_DEPTH	pixel	0	glGetFloatv()
GL_ZOOM_X	<i>x</i> zoom factor	pixel	1.0	glGetFloatv()
GL_ZOOM_Y	<i>y</i> zoom factor	pixel	1.0	glGetFloatv()
GL_x	glPixelMap() translation tables; <i>x</i> is a map name from Table 8-1	—	0's	glGetPixelMap*()
GL_x_SIZE	Size of table <i>x</i>	—	1	glGetIntegerv()
GL_READ_BUFFER	Read source buffer	pixel	—	glGetIntegerv()

Table B-10 Pixel State Variables (continued)

Evaluators

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_ORDER	1D map order	—	1	glGetMapiv()
GL_ORDER	2D map orders	—	1, 1	glGetMapiv()
GL_COEFF	1D control points	—	—	glGetMapfv()
GL_COEFF	2D control points	—	—	glGetMapfv()
GL_DOMAIN	1D domain endpoints	—	—	glGetMapfv()
GL_DOMAIN	2D domain endpoints	—	—	glGetMapfv()
GL_MAP1_x	1D map enables: <i>x</i> is map type	eval/enable	GL_FALSE	glIsEnabled()
GL_MAP2_x	2D map enables: <i>x</i> is map type	eval/enable	GL_FALSE	glIsEnabled()
GL_MAP1_GRID_DOMAIN	1D grid endpoints	eval	0, 1	glGetFloatv()
GL_MAP2_GRID_DOMAIN	2D grid endpoints	eval	0, 1; 0, 1	glGetFloatv()
GL_MAP1_GRID_SEGMENTS	1D grid divisions	eval	1	glGetFloatv()
GL_MAP2_GRID_SEGMENTS	2D grid divisions	eval	1,1	glGetFloatv()
GL_AUTO_NORMAL	True if automatic normal generation enabled	eval	GL_FALSE	glIsEnabled()

Table B-11 Evaluator State Variables

Hints

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_PERSPECTIVE_CORRECTION_HINT	Perspective correction hint	hint	GL_DONT_CARE	glGetIntegerv()
GL_POINT_SMOOTH_HINT	Point smooth hint	hint	GL_DONT_CARE	glGetIntegerv()
GL_LINE_SMOOTH_HINT	Line smooth hint	hint	GL_DONT_CARE	glGetIntegerv()
GL_POLYGON_SMOOTH_HINT	Polygon smooth hint	hint	GL_DONT_CARE	glGetIntegerv()
GL_FOG_HINT	Fog hint	hint	GL_DONT_CARE	glGetIntegerv()

Table B-12 Hint State Variables

Implementation-Dependent Values

State Variable	Description	Attribute Group	Minimum Value	Get Command
GL_MAX_LIGHTS	Maximum number of lights	—	8	glGetIntegerv()
GL_MAX_CLIP_PLANES	Maximum number of user clipping planes	—	6	glGetIntegerv()
GL_MAX_MODELVIEW_STACK_DEPTH	Maximum modelview-matrix stack depth	—	32	glGetIntegerv()
GL_MAX_PROJECTION_STACK_DEPTH	Maximum projection-matrix stack depth	—	2	glGetIntegerv()
GL_MAX_TEXTURE_STACK_DEPTH	Maximum depth of texture matrix stack	—	2	glGetIntegerv()
GL_SUBPIXEL_BITS	Number of bits of subpixel precision in <i>x</i> and <i>y</i>	—	4	glGetIntegerv()
GL_MAX_TEXTURE_SIZE	See discussion in “Texture Proxy” on page 330	—	64	glGetIntegerv()
GL_MAX_PIXEL_MAP_TABLE	Maximum size of a glPixelMap() translation table	—	32	glGetIntegerv()
GL_MAX_NAME_STACK_DEPTH	Maximum selection-name stack depth	—	64	glGetIntegerv()
GL_MAX_LIST_NESTING	Maximum display-list call nesting	—	64	glGetIntegerv()
GL_MAX_EVAL_ORDER	Maximum evaluator polynomial order	—	8	glGetIntegerv()

Table B-13 Implementation-Dependent State Variables

State Variable	Description	Attribute Group	Minimum Value	Get Command
GL_MAX_VIEWPORT_DIMS	Maximum viewport dimensions	—	—	glGetIntegerv()
GL_MAX_ATTRIB_STACK_DEPTH	Maximum depth of the attribute stack	—	16	glGetIntegerv()
GL_MAX_CLIENT_ATTRIB_STACK_DEPTH	Maximum depth of the client attribute stack	—	16	glGetIntegerv()
GL_AUX_BUFFERS	Number of auxiliary buffers	—	0	glGetBooleanv()
GL_RGBA_MODE	True if color buffers store RGBA	—	—	glGetBooleanv()
GL_INDEX_MODE	True if color buffers store indices	—	—	glGetBooleanv()
GL_DOUBLEBUFFER	True if front and back buffers exist	—	—	glGetBooleanv()
GL_STEREO	True if left and right buffers exist	—	—	glGetBooleanv()
GL_POINT_SIZE_RANGE	Range (low to high) of antialiased point sizes	—	1, 1	glGetFloatv()
GL_POINT_SIZE_GRANULARITY	Antialiased point-size granularity	—	—	glGetFloatv()
GL_LINE_WIDTH_RANGE	Range (low to high) of antialiased line widths	—	1, 1	glGetFloatv()
GL_LINE_WIDTH_GRANULARITY	Antialiased line-width granularity	—	—	glGetFloatv()

Table B-13 Implementation-Dependent State Variables (continued)

Implementation-Dependent Pixel Depths

State Variable	Description	Attribute Group	Minimum Value	Get Command
GL_RED_BITS	Number of bits per red component in color buffers	—	—	glGetIntegerv()
GL_GREEN_BITS	Number of bits per green component in color buffers	—	—	glGetIntegerv()
GL_BLUE_BITS	Number of bits per blue component in color buffers	—	—	glGetIntegerv()
GL_ALPHA_BITS	Number of bits per alpha component in color buffers	—	—	glGetIntegerv()
GL_INDEX_BITS	Number of bits per index in color buffers	—	—	glGetIntegerv()
GL_DEPTH_BITS	Number of depth-buffer bitplanes	—	—	glGetIntegerv()
GL_STENCIL_BITS	Number of stencil bitplanes	—	—	glGetIntegerv()
GL_ACCUM_RED_BITS	Number of bits per red component in the accumulation buffer	—	—	glGetIntegerv()
GL_ACCUM_GREEN_BITS	Number of bits per green component in the accumulation buffer	—	—	glGetIntegerv()
GL_ACCUM_BLUE_BITS	Number of bits per blue component in the accumulation buffer	—	—	glGetIntegerv()
GL_ACCUM_ALPHA_BITS	Number of bits per alpha component in the accumulation buffer	—	—	glGetIntegerv()

Table B-14 Implementation-Dependent Pixel-Depth State Variables

Miscellaneous

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_LIST_BASE	Setting of glListBase()	list	0	glGetIntegerv()
GL_LIST_INDEX	Number of display list under construction; 0 if none	—	0	glGetIntegerv()
GL_LIST_MODE	Mode of display list under construction; undefined if none	—	0	glGetIntegerv()
GL_ATTRIB_STACK_DEPTH	Attribute stack pointer	—	0	glGetIntegerv()
GL_CLIENT_ATTRIB_STACK_DEPTH	Client attribute stack pointer	—	0	glGetIntegerv()
GL_NAME_STACK_DEPTH	Name stack depth	—	0	glGetIntegerv()
GL_RENDER_MODE	glRenderMode() setting	—	GL_RENDER	glGetIntegerv()
GL_SELECTION_BUFFER_POINTER	Pointer to selection buffer	select	0	glGetPointerv()
GL_SELECTION_BUFFER_SIZE	Size of selection buffer	select	0	glGetIntegerv()
GL_FEEDBACK_BUFFER_POINTER	Pointer to feedback buffer	feedback	0	glGetPointerv()
GL_FEEDBACK_BUFFER_SIZE	Size of feedback buffer	feedback	0	glGetIntegerv()
GL_FEEDBACK_BUFFER_TYPE	Type of feedback buffer	feedback	GL_2D	glGetIntegerv()
—	Current error code(s)	—	0	glGetError()

Table B-15 Miscellaneous State Variables

OpenGL and Window Systems

OpenGL is available on many different platforms and works with many different window systems. OpenGL is designed to complement window systems, not duplicate their functionality. Therefore, OpenGL performs geometric and image rendering in two and three dimensions, but it does not manage windows or handle input events.

However, the basic definitions of most window systems don't support a library as sophisticated as OpenGL, with its complex and diverse pixel formats, including depth, stencil, and accumulation buffers, as well as double-buffering. For most window systems, some routines are added to extend the window system to support OpenGL.

This appendix introduces the extensions defined for several window and operating systems: the X Window System, the Apple Mac OS, OS/2 Warp from IBM, and Microsoft Windows NT and Windows 95. You need to have some knowledge of the window systems to fully understand this appendix.

This appendix has the following major sections:

- "GLX: OpenGL Extension for the X Window System" on page 562
- "AGL: OpenGL Extension to the Apple Macintosh" on page 566
- "PGL: OpenGL Extension for IBM OS/2 Warp" on page 570
- "WGL: OpenGL Extension for Microsoft Windows NT and Windows 95" on page 574

GLX: OpenGL Extension for the X Window System

In the X Window System, OpenGL rendering is made available as an extension to X in the formal X sense. GLX is an extension to the X protocol (and its associated API) for communicating OpenGL commands to an extended X server. Connection and authentication are accomplished with the normal X mechanisms.

As with other X extensions, there is a defined network protocol for OpenGL's rendering commands encapsulated within the X byte stream, so client-server OpenGL rendering is supported. Since performance is critical in three-dimensional rendering, the OpenGL extension to X allows OpenGL to bypass the X server's involvement in data encoding, copying, and interpretation and instead render directly to the graphics pipeline.

The X Visual is the key data structure to maintain pixel format information about the OpenGL window. A variable of data type `XVisualInfo` keeps track of pixel information, including pixel type (RGBA or color index), single or double-buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers. The standard X Visuals (for example, `PseudoColor`, `TrueColor`) do not describe the pixel format details, so each implementation must extend the number of X Visuals supported.

The GLX routines are discussed in more detail in the *OpenGL Reference Manual*. Integrating OpenGL applications with the X Window System and the Motif widget set is discussed in great detail in *OpenGL Programming for the X Window System* by Mark Kilgard (Reading, MA: Addison-Wesley Developers Press, 1996), which includes full source code examples. If you absolutely want to learn about the internals of GLX, you may want to read the GLX specification, which can be found at

<ftp://sgigate.sgi.com/pub/opengl/doc/>

Initialization

Use `glXQueryExtension()` and `glXQueryVersion()` to determine whether the GLX extension is defined for an X server and, if so, which version is present. `glXQueryExtensionsString()` returns extension information about the client-server connection. `glXGetClientString()` returns information about the client library, including extensions and version number. `glXQueryServerString()` returns similar information about the server.

`glXChooseVisual()` returns a pointer to an `XVisualInfo` structure describing the visual that meets the client's specified attributes. You can query a visual about its support of a particular OpenGL attribute with `glXGetConfig()`.

Controlling Rendering

Several GLX routines are provided for creating and managing an OpenGL rendering context. You can use such a context to render off-screen if you want. Routines are also provided for such tasks as synchronizing execution between the X and OpenGL streams, swapping front and back buffers, and using an X font.

Managing an OpenGL Rendering Context

An OpenGL rendering context is created with `glXCreateContext()`. One of the arguments to this routine allows you to request a direct rendering context that bypasses the X server as described previously. (Note that to do direct rendering, the X server connection must be local, and the OpenGL implementation needs to support direct rendering.) `glXCreateContext()` also allows display-list and texture-object indices and definitions to be shared by multiple rendering contexts. You can determine whether a GLX context is direct with `glXIsDirect()`.

To make a rendering context current, use `glXMakeCurrent()`; `glXGetCurrentContext()` returns the current context. You can also obtain the current drawable with `glXGetCurrentDrawable()` and the current X Display with `glXGetCurrentDisplay()`. Remember that only one context can be current for any thread at any one time. If you have multiple contexts, you can copy selected groups of OpenGL state variables from one context to another with `glXCopyContext()`. When you're finished with a particular context, destroy it with `glXDestroyContext()`.

Off-Screen Rendering

To render off-screen, first create an X Pixmap and then pass this as an argument to `glXCreateGLXPixmap()`. Once rendering is completed, you can destroy the association between the X and GLX Pixmap with `glXDestroyGLXPixmap()`. (Off-screen rendering isn't guaranteed to be supported for direct renderers.)

Synchronizing Execution

To prevent X requests from executing until any outstanding OpenGL rendering is completed, call `glXWaitGL()`. Then, any previously issued OpenGL commands are guaranteed to be executed before any X rendering calls made after `glXWaitGL()`. Although the same result can be achieved with `glFinish()`, `glXWaitGL()` doesn't require a round trip to the server and thus is more efficient in cases where the client and server are on separate machines.

To prevent an OpenGL command sequence from executing until any outstanding X requests are completed, use `glXWaitX()`. This routine guarantees that previously issued X rendering calls are executed before any OpenGL calls made after `glXWaitX()`.

Swapping Buffers

For drawables that are double-buffered, the front and back buffers can be exchanged by calling `glXSwapBuffers()`. An implicit `glFlush()` is done as part of this routine.

Using an X Font

A shortcut for using X fonts in OpenGL is provided with the command `glXUseXFont()`. This routine builds display lists, each of which calls `glBitmap()`, for each requested character from the specified font and font size.

GLX Prototypes

Initialization

Determine whether the GLX extension is defined on the X server:

```
Bool glXQueryExtension ( Display *dpy, int *errorBase, int *eventBase );
```

Query version and extension information for client and server:

```
Bool glXQueryVersion ( Display *dpy, int *major, int *minor );
```

```
const char* glXGetClientString ( Display *dpy, int name );
```

```
const char* glXQueryServerString ( Display *dpy, int screen, int name );
```

```
const char* glXQueryExtensionsString ( Display *dpy, int screen );
```

Obtain the desired visual:

```
XVisualInfo* glXChooseVisual ( Display *dpy, int screen,  
int *attribList );  
  
int glXGetConfig ( Display *dpy, XVisualInfo *visual, int attrib,  
int *value );
```

Controlling Rendering

Manage or query an OpenGL rendering context:

```
GLXContext glXCreateContext ( Display *dpy, XVisualInfo *visual,  
GLXContext shareList, Bool direct );  
  
void glXDestroyContext ( Display *dpy, GLXContext context );  
  
void glXCopyContext ( Display *dpy, GLXContext source,  
GLXContext dest, unsigned long mask );  
  
Bool glXIsDirect ( Display *dpy, GLXContext context );  
  
Bool glXMakeCurrent ( Display *dpy, GLXDrawable draw,  
GLXContext context );  
  
GLXContext glXGetCurrentContext (void);  
  
Display* glXGetCurrentDisplay (void);  
  
GLXDrawable glXGetCurrentDrawable (void);
```

Perform off-screen rendering:

```
GLXPixmap glXCreateGLXPixmap ( Display *dpy, XVisualInfo *visual,  
Pixmap pixmap );  
  
void glXDestroyGLXPixmap ( Display *dpy, GLXPixmap pix );
```

Synchronize execution:

```
void glXWaitGL (void);  
void glXWaitX (void);
```

Exchange front and back buffers:

```
void glXSwapBuffers ( Display *dpy, GLXDrawable drawable );
```

Use an X font:

```
void glXUseXFont ( Font font, int first, int count, int listBase );
```

AGL: OpenGL Extension to the Apple Macintosh

This section covers the routines defined as the OpenGL extension to the Apple Macintosh (AGL), as defined by Template Graphics Software. An understanding of the way the Macintosh handles graphics rendering (QuickDraw) is required. The *Macintosh Toolbox Essentials* and *Imaging With QuickDraw* manuals from the *Inside Macintosh* series are also useful to have at hand.

For more information (including how to obtain the OpenGL software library for the Power Macintosh), you may want to check out the web site for OpenGL information at Template Graphics Software:

<http://www.sd.tgs.com/Products/opengl.htm>

For the Macintosh, OpenGL rendering is made available as a library that is either compiled in or resident as an extension for an application that wishes to make use of it. OpenGL is implemented in software for systems that do not possess hardware acceleration. Where acceleration is available (through the QuickDraw 3D Accelerator), those capabilities that match the OpenGL pipeline are used with the remaining functionality being provided through software rendering.

The data type `AGLPixelFmtID` (the AGL equivalent to `XVisualInfo`) maintains pixel information, including pixel type (RGBA or color index), single- or double-buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers.

In contrast to other OpenGL implementations on other systems (such as the X Window System), the client/server model is not used. However, you may still need to call `glFlush()` since some hardware accelerators buffer the OpenGL pipeline and require a flush to empty it.

Initialization

Use `aglQueryVersion()` to determine what version of OpenGL for the Macintosh is available.

The capabilities of underlying graphics devices and your requirements for rendering buffers are resolved using `aglChoosePixelFormat()`. Use `aglListPixelFmts()` to find the particular formats supported by a graphics device. Given a pixel format, you can determine which attributes are available by using `aglGetConfig()`.

Rendering and Contexts

Several AGL routines are provided for creating and managing an OpenGL rendering context. You can use such a context to render into either a window or an off-screen graphics world. Routines are also provided that allow you to swap front and back rendering buffers, adjust buffers in response to a move, resize or graphics device change event, and use Macintosh fonts. For software rendering (and in some cases, hardware-accelerated rendering) the rendering buffers are created in your application memory space. For the application to work properly you must provide sufficient memory for these buffers in your application's SIZE resource.

Managing an OpenGL Rendering Context

An OpenGL rendering context is created (at least one context per window being rendered into) with `aglCreateContext()`. This takes the pixel format you selected as a parameter and uses it to initialize the context.

Use `aglMakeCurrent()` to make a rendering context current. Only one context can be current for a thread of control at any time. This indicates which drawable is to be rendered into and which context to use with it. It's possible for more than one context to be used (not simultaneously) with a particular drawable. Two routines allow you to determine which is the current rendering context and drawable being rendered into: `aglGetCurrentContext()` and `aglGetCurrentDrawable()`.

If you have multiple contexts, you can copy selected groups of OpenGL state variables from one context to another with `aglCopyContext()`. When a particular context is finished with, it should be destroyed by calling `aglDestroyContext()`.

On-screen Rendering

With the OpenGL extensions for the Apple Macintosh you can choose whether window clipping is performed when writing to the screen and whether the cursor is hidden during screen writing operations. This is important since these two items may affect how fast rendering can be performed. Call `aglSetOptions()` to select these options.

Off-screen Rendering

To render off-screen, first create an off-screen graphics world in the usual way, and pass the handle into `aglCreateAGLPixmap()`. This routine returns

a drawable that can be used with `aglMakeCurrent()`. Once rendering is completed, you can destroy the association with `aglDestroyAGLPixmap()`.

Swapping Buffers

For drawables that are double-buffered (as per the pixel format of the current rendering context), call `aglSwapBuffers()` to exchange the front and back buffers. An implicit `glFlush()` is performed as part of this routine.

Updating the Rendering Buffers

The Apple Macintosh toolbox requires you to perform your own event handling and does not provide a way for libraries to automatically hook in to the event stream. So that the drawables maintained by OpenGL can adjust to changes in drawable size, position and pixel depth, `aglUpdateCurrent()` is provided.

This routine must be called by your event processing code whenever one of these events occurs in the current drawable. Ideally the scene should be rerendered after a update call to take into account the changes made to the rendering buffers.

Using an Apple Macintosh Font

A shortcut for using Macintosh fonts is provided with `aglUseFont()`. This routine builds display lists, each of which calls `glBitmap()`, for each requested character from the specified font and font size.

Error Handling

An error-handling mechanism is provided for the Apple Macintosh OpenGL extension. When an error occurs you can call `aglGetError()` to get a more precise description of what caused the error.

AGL Prototypes

Initialization

Determine AGL version:

```
GLboolean aglQueryVersion ( int *major, int *minor );
```

Pixel format selection, availability, and capability:

```
AGLPixelFmtID aglChoosePixelFormat ( GDHandle *dev, int ndev,  
int *attrs );  
int aglListPixelFormats ( GDHandle dev, AGLPixelFormatID **fmts );  
GLboolean aglGetConfig ( AGLPixelFormatID pix, int attrib, int *value );
```

Controlling Rendering

Manage an OpenGL rendering context:

```
AGLContext aglCreateContext ( AGLPixelFormatID pix,  
AGLContext shareList );  
GLboolean aglDestroyContext ( AGLContext context );  
GLboolean aglCopyContext ( AGLContext source, AGLContext dest,  
GLuint mask );  
GLboolean aglMakeCurrent ( AGLDrawable drawable,  
AGLContext context );  
GLboolean aglSetOptions ( int opts );  
AGLContext aglGetCurrentContext (void);  
AGLDrawable aglGetCurrentDrawable (void);
```

Perform off-screen rendering:

```
AGLPixmap aglCreateAGLPixmap ( AGLPixelFormatID pix,  
GWorldPtr pixmap );  
GLboolean aglDestroyAGLPixmap ( AGLPixmap pix );
```

Exchange front and back buffers:

```
GLboolean aglSwapBuffers ( AGLDrawable drawable );
```

Update the current rendering buffers:

```
GLboolean aglUpdateCurrent (void);
```

Use a Macintosh font:

```
GLboolean aglUseFont ( int familyID, int size, int first, int count,  
int listBase );
```

Find the cause of an error:

```
GLenum aglGetError (void);
```

PGL: OpenGL Extension for IBM OS/2 Warp

OpenGL rendering for IBM OS/2 Warp is accomplished by using PGL routines added to integrate OpenGL into the standard IBM Presentation Manager. OpenGL with PGL supports both a direct OpenGL context (which is often faster) and an indirect context (which allows some integration of Gpi and OpenGL rendering).

The data type VISUALCONFIG (the PGL equivalent to XVisualInfo) maintains the visual configuration, including pixel type (RGBA or color index), single- or double-buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers.

To get more information (including how to obtain the OpenGL software library for IBM OS/2 Warp, Version 3.0), you may want to start at

<http://www.austin.ibm.com/software/OpenGL/>

Packaged along with the software is the document, *OpenGL On OS/2 Warp*, which provides more detailed information. OpenGL support is included with the base operating system with OS/2 Warp Version 4.

Initialization

Use `pglQueryCapability()` and `pglQueryVersion()` to determine whether the OpenGL is supported on this machine and, if so, how it is supported and which version is present. `pglChooseConfig()` returns a pointer to an VISUALCONFIG structure describing the visual configuration that best meets the client's specified attributes. A list of the particular visual configurations supported by a graphics device can be found using `pglQueryConfigs()`.

Controlling Rendering

Several PGL routines are provided for creating and managing an OpenGL rendering context, capturing the contents of a bitmap, synchronizing execution between the Presentation Manager and OpenGL streams, swapping front and back buffers, using a color palette, and using an OS/2 logical font.

Managing an OpenGL Rendering Context

An OpenGL rendering context is created with `pglCreateContext()`. One of the arguments to this routine allows you to request a direct rendering context that bypasses the Gpi and render to a PM window, which is generally faster. You can determine whether a OpenGL context is direct with `pglIsIndirect()`.

To make a rendering context current, use `pglMakeCurrent()`; `pglGetCurrentContext()` returns the current context. You can also obtain the current window with `pglGetCurrentWindow()`. You can copy some OpenGL state variables from one context to another with `pglCopyContext()`. When you're finished with a particular context, destroy it with `pglDestroyContext()`.

Access the Bitmap of the Front Buffer

To lock access to the bitmap representation of the contents of the front buffer, use `pglGrabFrontBitmap()`. An implicit `glFlush()` is performed, and you can read the bitmap, but its contents are effectively read-only. Immediately after access is completed, you should call `pglReleaseFrontBitmap()` to restore write access to the front buffer.

Synchronizing Execution

To prevent Gpi rendering requests from executing until any outstanding OpenGL rendering is completed, call `pglWaitGL()`. Then, any previously issued OpenGL commands are guaranteed to be executed before any Gpi rendering calls made after `pglWaitGL()`.

To prevent an OpenGL command sequence from executing until any outstanding Gpi requests are completed, use `pglWaitPM()`. This routine guarantees that previously issued Gpi rendering calls are executed before any OpenGL calls made after `pglWaitPM()`.

Note: OpenGL and Gpi rendering can be integrated in the same window only if the OpenGL context is an indirect context.

Swapping Buffers

For windows that are double-buffered, the front and back buffers can be exchanged by calling `pglSwapBuffers()`. An implicit `glFlush()` is done as part of this routine.

Using a Color Index Palette

When you are running in 8-bit (256 color) mode, you have to worry about color palette management. For windows with a color index Visual Configuration, call `pglSelectColorIndexPalette()` to tell OpenGL what color-index palette you want to use with your context. A color palette must be selected before the context is initially bound to a window. In RGBA mode, OpenGL sets up a palette automatically.

Using an OS/2 Logical Font

A shortcut for using OS/2 logical fonts in OpenGL is provided with the command `pglUseFont()`. This routine builds display lists, each of which calls `glBitmap()`, for each requested character from the specified font and font size.

PGL Prototypes

Initialization

Determine whether OpenGL is supported and, if so, its version number:

```
long pglQueryCapability (HAB hab);  
void pglQueryVersion (HAB hab, int *major, int *minor);
```

Visual configuration selection, availability and capability:

```
PVISUALCONFIG pglChooseConfig (HAB hab, int *attribList);  
PVISUALCONFIG * pglQueryConfigs (HAB hab);
```

Controlling Rendering

Manage or query an OpenGL rendering context:

```
HGC pglCreateContext (HAB hab, PVISUALCONFIG pVisualConfig,  
HGC shareList, Bool isDirect);  
Bool pglDestroyContext (HAB hab, HGC hgc);  
Bool pglCopyContext (HAB hab, HGC source, HGC dest, GLuint mask);  
Bool pglMakeCurrent (HAB hab, HGC hgc, HWND hwnd);  
long pglIsIndirect (HAB hab, HGC hgc);
```

HGC **pglGetCurrentContext** (HAB *hab*);

HWND **pglGetCurrentWindow** (HAB *hab*);

Access and release the bitmap of the front buffer:

Bool **pglGrabFrontBitmap** (HAB *hab*, HPS **hps*, HBITMAP **phbitmap*);

Bool **pglReleaseFrontBitmap** (HAB *hab*);

Synchronize execution:

HPS **pglWaitGL** (HAB *hab*);

void **pglWaitPM** (HAB *hab*);

Exchange front and back buffers:

void **pglSwapBuffers** (HAB *hab*, HWND *hwnd*);

Finding a color-index palette:

void **pglSelectColorIndexPalette** (HAB *hab*, HPAL, *hpal*, HGC *hgc*);

Use an OS/2 logical font:

Bool **pglUseFont** (HAB *hab*, HPS *hps*, FATTRS **fontAttribs*,
long *logicalId*, int *first*, int *count*, int *listBase*);

WGL: OpenGL Extension for Microsoft Windows NT and Windows 95

OpenGL rendering is supported on systems that run Microsoft Windows NT and Windows 95. The functions and routines of the Win32 library are necessary to initialize the pixel format and control rendering for OpenGL. Some routines, which are prefixed by `wgl`, extend Win32 so that OpenGL can be fully supported.

For Win32/WGL, the `PIXELFORMATDESCRIPTOR` is the key data structure to maintain pixel format information about the OpenGL window. A variable of data type `PIXELFORMATDESCRIPTOR` keeps track of pixel information, including pixel type (RGBA or color index), single- or double-buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers.

To get more information about WGL, you may want to start with technical articles available through the Microsoft Developer Network at

<http://www.microsoft.com/msdn/>

Initialization

Use `GetVersion()` or the newer `GetVersionEx()` to determine version information. `ChoosePixelFormat()` tries to find a `PIXELFORMATDESCRIPTOR` with specified attributes. If a good match for the requested pixel format is found, then `SetPixelFormat()` should be called to actually use the pixel format. You should select a pixel format in the device context before calling `wglCreateContext()`.

If you want to find out details about a given pixel format, use `DescribePixelFormat()` or, for overlays or underlays, `wglDescribeLayerPlane()`.

Controlling Rendering

Several WGL routines are provided for creating and managing an OpenGL rendering context, rendering to a bitmap, swapping front and back buffers, finding a color palette, and using either bitmap or outline fonts.

Managing an OpenGL Rendering Context

`wglCreateContext()` creates an OpenGL rendering context for drawing on the device in the selected pixel format of the device context. (To create an OpenGL rendering context for overlay or underlay windows, use `wglCreateLayerContext()` instead.) To make a rendering context current, use `wglMakeCurrent()`; `wglGetCurrentContext()` returns the current context. You can also obtain the current device context with `wglGetCurrentDC()`. You can copy some OpenGL state variables from one context to another with `wglCopyContext()` or make two contexts share the same display lists and texture objects with `wglShareLists()`. When you're finished with a particular context, destroy it with `wglDestroyContext()`.

OpenGL Rendering to a Bitmap

Win32 has a few routines to allocate (and deallocate) bitmaps, to which you can render OpenGL directly. `CreateDIBitmap()` creates a device-dependent bitmap (DDB) from a device-independent bitmap (DIB).

`CreateDIBSection()` creates a device-independent bitmap (DIB) that applications can write to directly. When finished with your bitmap, you can use `DeleteObject()` to free it up.

Synchronizing Execution

If you want to combine GDI and OpenGL rendering, be aware there are no equivalents to functions like `glXWaitGL()`, `glXWaitX()`, or `pglWaitGL()` in Win32. Although `glXWaitGL()` has no equivalent in Win32, you can achieve the same effect by calling `glFinish()`, which waits until all pending OpenGL commands are executed, or by calling `GdiFlush()`, which waits until all GDI drawing has completed.

Swapping Buffers

For windows that are double-buffered, the front and back buffers can be exchanged by calling `SwapBuffers()` or `wglSwapLayerBuffers()`; the latter for overlays and underlays.

Finding a Color Palette

To access the color palette for the standard (non-layer) bitplanes, use the standard GDI functions to set the palette entries. For overlay or underlay layers, use `wglRealizeLayerPalette()`, which maps palette entries from a given color-index layer plane into the physical palette or initializes the

palette of an RGBA layer plane. `wglGetLayerPaletteEntries()` is used to query the entries in palettes of layer planes.

Using a Bitmap or Outline Font

WGL has two routines, `wglUseFontBitmaps()` and `wglUseFontOutlines()`, for converting system fonts to use with OpenGL. Both routines build a display list for each requested character from the specified font and font size.

WGL Prototypes

Initialization

Determine version information:

```
BOOL GetVersion ( LPOSVERSIONINFO lpVersionInformation );  
BOOL GetVersionEx ( LPOSVERSIONINFO lpVersionInformation );
```

Pixel format availability, selection, and capability:

```
int ChoosePixelFormat ( HDC hdc,  
CONST PIXELFORMATDESCRIPTOR * pefd );  
BOOL SetPixelFormat ( HDC hdc, int iPixelFormat,  
CONST PIXELFORMATDESCRIPTOR * pefd );  
int DescribePixelFormat ( HDC hdc, int iPixelFormat, UINT nBytes,  
LPIXELFORMATDESCRIPTOR pefd );  
BOOL wglDescribeLayerPlane ( HDC hdc, int iPixelFormat,  
int iLayerPlane, UINT nBytes, LPLAYERPLANEDESCRIPTOR plpd );
```

Controlling Rendering

Manage or query an OpenGL rendering context:

```
HGLRC wglCreateContext ( HDC hdc );  
HGLRC wglCreateLayerContext ( HDC hdc, int iLayerPlane );  
BOOL wglShareLists ( HGLRC hglrc1, HGLRC hglrc2 );  
BOOL wglDeleteContext ( HGLRC hglrc );  
BOOL wglCopyContext ( HGLRC hglrcSource, HGLRC hglrcDest,  
UINT mask );
```

```
BOOL wglMakeCurrent ( HDC hdc, HGLRC hglrc );  
HGLRC wglGetCurrentContext ( VOID );  
HDC wglGetCurrentDC ( VOID );
```

Access and release the bitmap of the front buffer:

```
HBITMAP CreateDIBitmap ( HDC hdc,  
CONST BITMAPINFOHEADER *lpbmih, DWORD fdwInit,  
CONST VOID *lpbInit, CONST BITMAPINFO *lpbmi, UINT fuUsage );  
HBITMAP CreateDIBSection ( HDC hdc, CONST BITMAPINFO *pbmi,  
UINT iUsage, VOID *ppvBits, HANDLE hSection, DWORD dwOffset );  
BOOL DeleteObject ( HGDIOBJ hObject );
```

Exchange front and back buffers:

```
BOOL SwapBuffers ( HDC hdc );  
BOOL wglSwapLayerBuffers ( HDC hdc, UINT fuPlanes );
```

Finding a color palette for overlay or underlay layers:

```
int wglGetLayerPaletteEntries ( HDC hdc, int iLayerPlane, int iStart,  
int cEntries, CONST COLORREF *pcr );  
BOOL wglRealizeLayerPalette ( HDC hdc, int iLayerPlane,  
BOOL bRealize );
```

Use a bitmap or an outline font:

```
BOOL wglUseFontBitmaps ( HDC hdc, DWORD first, DWORD count,  
DWORD listBase );  
BOOL wglUseFontOutlines ( HDC hdc, DWORD first, DWORD count,  
DWORD listBase, FLOAT deviation, FLOAT extrusion, int format,  
LPGLYPHMETRICSFLOAT lpgmf );
```

Basics of GLUT: The OpenGL Utility Toolkit

This appendix describes a subset of Mark Kilgard's OpenGL Utility Toolkit (GLUT), which is fully documented in his book, *OpenGL Programming for the X Window System* (Reading, MA: Addison-Wesley Developers Press, 1996). GLUT has become a popular library for OpenGL programmers, because it standardizes and simplifies window and event management. GLUT has been ported atop a variety of OpenGL implementations, including both the X Window System and Microsoft Windows NT.

This appendix has the following major sections:

- "Initializing and Creating a Window" on page 580
- "Handling Window and Input Events" on page 581
- "Loading the Color Map" on page 583
- "Initializing and Drawing Three-Dimensional Objects" on page 583
- "Managing a Background Process" on page 584
- "Running the Program" on page 585

(See "How to Obtain the Sample Code" on page v for information about how to obtain the source code for GLUT.)

With GLUT, your application structures its event handling to use callback functions. (This method is similar to using the Xt Toolkit, also known as the X Intrinsics, with a widget set.) For example, first you open a window and register callback routines for specific events. Then, you create a main loop without an exit. In that loop, if an event occurs, its registered callback functions are executed. Upon completion of the callback functions, flow of control is returned to the main loop.

Initializing and Creating a Window

Before you can open a window, you must specify its characteristics: Should it be single-buffered or double-buffered? Should it store colors as RGBA values or as color indices? Where should it appear on your display? To specify the answers to these questions, call `glutInit()`, `glutInitDisplayMode()`, `glutInitWindowSize()`, and `glutInitWindowPosition()` before you call `glutCreateWindow()` to open the window.

```
void glutInit(int argc, char **argv);
```

`glutInit()` should be called before any other GLUT routine, because it initializes the GLUT library. `glutInit()` will also process command line options, but the specific options are window system dependent. For the X Window System, `-iconic`, `-geometry`, and `-display` are examples of command line options, processed by `glutInit()`. (The parameters to the `glutInit()` should be the same as those to `main()`.)

```
void glutInitDisplayMode(unsigned int mode);
```

Specifies a display mode (such as RGBA or color-index, or single- or double-buffered) for windows created when `glutCreateWindow()` is called. You can also specify that the window have an associated depth, stencil, and/or accumulation buffer. The *mask* argument is a bitwise ORed combination of `GLUT_RGBA` or `GLUT_INDEX`, `GLUT_SINGLE` or `GLUT_DOUBLE`, and any of the buffer-enabling flags: `GLUT_DEPTH`, `GLUT_STENCIL`, or `GLUT_ACCUM`. For example, for a double-buffered, RGBA-mode window with a depth and stencil buffer, use `GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL`. The default value is `GLUT_RGBA | GLUT_SINGLE` (an RGBA, single-buffered window).

```
void glutInitWindowSize(int width, int height);  
void glutInitWindowPosition(int x, int y);
```

Requests windows created by `glutCreateWindow()` to have an initial size and position. The arguments (*x*, *y*) indicate the location of a corner of the window, relative to the entire display. The *width* and *height* indicate the window's size (in pixels). The initial window size and position are hints and may be overridden by other requests.

```
int glutCreateWindow(char *name);
```

Opens a window with previously set characteristics (display mode, width, height, and so on). The string *name* may appear in the title bar if your window system does that sort of thing. The window is not initially displayed until `glutMainLoop()` is entered, so do not render into the window until then.

The value returned is a unique integer identifier for the window. This identifier can be used for controlling and rendering to multiple windows (each with an OpenGL rendering context) from the same application.

Handling Window and Input Events

After the window is created, but before you enter the main loop, you should register callback functions using the following routines.

```
void glutDisplayFunc(void (*func)(void));
```

Specifies the function that's called whenever the contents of the window need to be redrawn. The contents of the window may need to be redrawn when the window is initially opened, when the window is popped and window damage is exposed, and when `glutPostRedisplay()` is explicitly called.

```
void glutReshapeFunc(void (*func)(int width, int height));
```

Specifies the function that's called whenever the window is resized or moved. The argument *func* is a pointer to a function that expects two arguments, the new width and height of the window. Typically, *func* calls `glViewport()`, so that the display is clipped to the new size, and it redefines the projection matrix so that the aspect ratio of the projected image matches the viewport, avoiding aspect ratio distortion. If `glutReshapeFunc()` isn't called or is deregistered by passing NULL, a default reshape function is called, which calls `glViewport(0, 0, width, height)`.

```
void glutKeyboardFunc(void (*func)(unsigned int key, int x, int y));
```

Specifies the function, *func*, that's called when a key that generates an ASCII character is pressed. The *key* callback parameter is the generated ASCII value. The *x* and *y* callback parameters indicate the location of the mouse (in window-relative coordinates) when the key was pressed.

```
void glutMouseFunc(void (*func)(int button, int state, int x, int y));
```

Specifies the function, *func*, that's called when a mouse button is pressed or released. The *button* callback parameter is one of `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, or `GLUT_RIGHT_BUTTON`. The *state* callback parameter is either `GLUT_UP` or `GLUT_DOWN`, depending upon whether the mouse has been released or pressed. The *x* and *y* callback parameters indicate the location (in window-relative coordinates) of the mouse when the event occurred.

```
void glutMotionFunc(void (*func)(int x, int y));
```

Specifies the function, *func*, that's called when the mouse pointer moves within the window while one or more mouse buttons is pressed. The *x* and *y* callback parameters indicate the location (in window-relative coordinates) of the mouse when the event occurred.

```
void glutPostRedisplay(void);
```

Marks the current window as needing to be redrawn. At the next opportunity, the callback function registered by `glutDisplayFunc()` will be called.

Loading the Color Map

If you're using color-index mode, you might be surprised to discover there's no OpenGL routine to load a color into a color lookup table. This is because the process of loading a color map depends entirely on the window system. GLUT provides a generalized routine to load a single color index with an RGB value, `glutSetColor()`.

```
void glutSetColor(GLint index, GLfloat red, GLfloat green, GLfloat blue);
```

Loads the index in the color map, *index*, with the given *red*, *green*, and *blue* values. These values are normalized to lie in the range [0.0,1.0].

Initializing and Drawing Three-Dimensional Objects

Many sample programs in this guide use three-dimensional models to illustrate various rendering properties. The following drawing routines are included in GLUT to avoid having to reproduce the code to draw these models in each program. The routines render all their graphics in immediate mode. Each three-dimensional model comes in two flavors: wireframe without surface normals, and solid with shading and surface normals. Use the solid version when you're applying lighting. Only the teapot generates texture coordinates.

```
void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);  
void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);
```

```
void glutWireCube(GLdouble size);  
void glutSolidCube(GLdouble size);
```

```
void glutWireTorus(GLdouble innerRadius, GLdouble outerRadius,  
                  GLint nsides, GLint rings);  
void glutSolidTorus(GLdouble innerRadius, GLdouble outerRadius,  
                   GLint nsides, GLint rings);
```

```
void glutWireIcosahedron(void);
void glutSolidIcosahedron(void);
```

```
void glutWireOctahedron(void);
void glutSolidOctahedron(void);
```

```
void glutWireTetrahedron(void);
void glutSolidTetrahedron(void);
```

```
void glutWireDodecahedron(GLdouble radius);
void glutSolidDodecahedron(GLdouble radius);
```

```
void glutWireCone(GLdouble radius, GLdouble height, GLint slices,
                  GLint stacks);
void glutSolidCone(GLdouble radius, GLdouble height, GLint slices,
                  GLint stacks);
```

```
void glutWireTeapot(GLdouble size);
void glutSolidTeapot(GLdouble size);
```

Managing a Background Process

You can specify a function that's to be executed if no other events are pending—for example, when the event loop would otherwise be idle—with `glutIdleFunc()`. This is particularly useful for continuous animation or other background processing.

```
void glutIdleFunc(void (*func)(void));
```

Specifies the function, *func*, to be executed if no other events are pending. If NULL (zero) is passed in, execution of *func* is disabled.

Running the Program

After all the setup is completed, GLUT programs enter an event processing loop, `glutMainLoop()`.

```
void glutMainLoop(void);
```

Enters the GLUT processing loop, never to return. Registered callback functions will be called when the corresponding events instigate them.

Calculating Normal Vectors

This appendix describes how to calculate normal vectors for surfaces. You need to define normals to use the OpenGL lighting facility, which is described in Chapter 5. “Normal Vectors” on page 63 introduces normals and the OpenGL command for specifying them. This appendix goes through the details of calculating them. It has the following major sections:

- “Finding Normals for Analytic Surfaces” on page 588
- “Finding Normals from Polygonal Data” on page 591

Since normals are perpendicular to a surface, you can find the normal at a particular point on a surface by first finding the flat plane that just touches the surface at that point. The normal is the vector that's perpendicular to that plane. On a perfect sphere, for example, the normal at a point on the surface is in the same direction as the vector from the center of the sphere to that point. For other types of surfaces, there are other, better means for determining the normals, depending on how the surface is specified.

Recall that smooth curved surfaces are approximated by a large number of small flat polygons. If the vectors perpendicular to these polygons are used as the surface normals in such an approximation, the surface appears faceted, since the normal direction is discontinuous across the polygonal boundaries. In many cases, however, an exact mathematical description exists for the surface, and true surface normals can be calculated at every point. Using the true normals improves the rendering considerably, as shown in Figure E-1. Even if you don't have a mathematical description, you can do better than the faceted look shown in the figure. The two major sections in this appendix describe how to calculate normal vectors for these two cases:

- “Finding Normals for Analytic Surfaces” on page 588 explains what to do when you have a mathematical description of a surface.
- “Finding Normals from Polygonal Data” on page 591 covers the case when you have only the polygonal data to describe a surface.

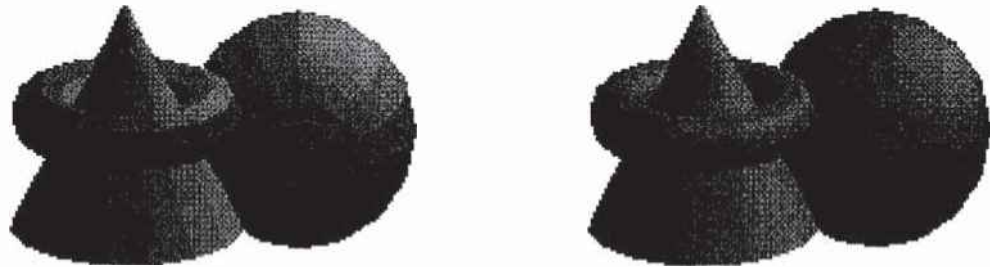


Figure E-1 Rendering with Polygonal Normals vs. True Normals

Finding Normals for Analytic Surfaces

Analytic surfaces are smooth, differentiable surfaces that are described by a mathematical equation (or set of equations). In many cases, the easiest surfaces to find normals for are analytic surfaces for which you have an explicit definition in the following form:

$$V(s,t) = [X(s,t) \ Y(s,t) \ Z(s,t)]$$

where s and t are constrained to be in some domain, and X , Y , and Z are differentiable functions of two variables. To calculate the normal, find

$$\frac{\partial V}{\partial s} \text{ and } \frac{\partial V}{\partial t}$$

which are vectors tangent to the surface in the s and t directions. The cross product

$$\frac{\partial V}{\partial s} \times \frac{\partial V}{\partial t}$$

is perpendicular to both and, hence, to the surface. The following shows how to calculate the cross product of two vectors. (Watch out for the degenerate cases where the cross product has zero length!)

$$\begin{bmatrix} v_x & v_y & v_z \end{bmatrix} \times \begin{bmatrix} w_x & w_y & w_z \end{bmatrix} = \begin{bmatrix} (v_y w_z - w_y v_z) & (w_x v_z - v_x w_z) & (v_x w_y - w_x v_y) \end{bmatrix}$$

You should probably normalize the resulting vector. To normalize a vector $[x \ y \ z]$, calculate its length

$$\text{Length} = \sqrt{x^2 + y^2 + z^2}$$

and divide each component of the vector by the length.

As an example of these calculations, consider the analytic surface

$$V(s,t) = [s^2 \ t^3 \ 3-st]$$

From this we have

$$\frac{\partial V}{\partial s} = [2s \ 0 \ -t], \quad \frac{\partial V}{\partial t} = [0 \ 3t^2 \ -s], \quad \text{and} \quad \frac{\partial V}{\partial s} \times \frac{\partial V}{\partial t} = [-3t^3 \ 2s^2 \ 6st^2]$$

So, for example, when $s=1$ and $t=2$, the corresponding point on the surface is $(1, 8, 1)$, and the vector $(-24, 2, 24)$ is perpendicular to the surface at that point. The length of this vector is 34, so the unit normal vector is $(-24/34, 2/34, 24/34) = (-0.70588, 0.058823, 0.70588)$.

For analytic surfaces that are described implicitly, as $F(x, y, z) = 0$, the problem is harder. In some cases, you can solve for one of the variables, say $z = G(x, y)$, and put it in the explicit form given previously:

$$\mathbf{V}(s, t) = [s \ t \ \mathbf{G}(s, t)]$$

Then continue as described earlier.

If you can't get the surface equation in an explicit form, you might be able to make use of the fact that the normal vector is given by the gradient

$$\nabla F = \left[\frac{\partial F}{\partial x} \ \frac{\partial F}{\partial y} \ \frac{\partial F}{\partial z} \right]$$

evaluated at a particular point (x, y, z) . Calculating the gradient might be easy, but finding a point that lies on the surface can be difficult. As an example of an implicitly defined analytic function, consider the equation of a sphere of radius 1 centered at the origin:

$$x^2 + y^2 + z^2 - 1 = 0$$

This means that

$$F(x, y, z) = x^2 + y^2 + z^2 - 1$$

which can be solved for z to yield

$$z = \pm \sqrt{1 - x^2 - y^2}$$

Thus, normals can be calculated from the explicit form

$$\mathbf{V}(s, t) = [s \ t \ \sqrt{1 - s^2 - t^2}]$$

as described previously.

If you could not solve for z , you could have used the gradient

$$\nabla F = [2x \ 2y \ 2z]$$

as long as you could find a point on the surface. In this case, it's not so hard to find a point—for example, $(2/3, 1/3, 2/3)$ lies on the surface. Using the gradient, the normal at this point is $(4/3, 2/3, 4/3)$. The unit-length normal is $(2/3, 1/3, 2/3)$, which is the same as the point on the surface, as expected.

Finding Normals from Polygonal Data

As mentioned previously, you often want to find normals for surfaces that are described with polygonal data such that the surfaces appear smooth rather than faceted. In most cases, the easiest way for you to do this (though it might not be the most efficient way) is to calculate the normal vectors for each of the polygonal facets and then to average the normals for neighboring facets. Use the averaged normal for the vertex that the neighboring facets have in common. Figure E-2 shows a surface and its polygonal approximation. (Of course, if the polygons represent the exact surface and aren't merely an approximation—if you're drawing a cube or a cut diamond, for example—don't do the averaging. Calculate the normal for each facet as described in the following paragraphs, and use that same normal for each vertex of the facet.)

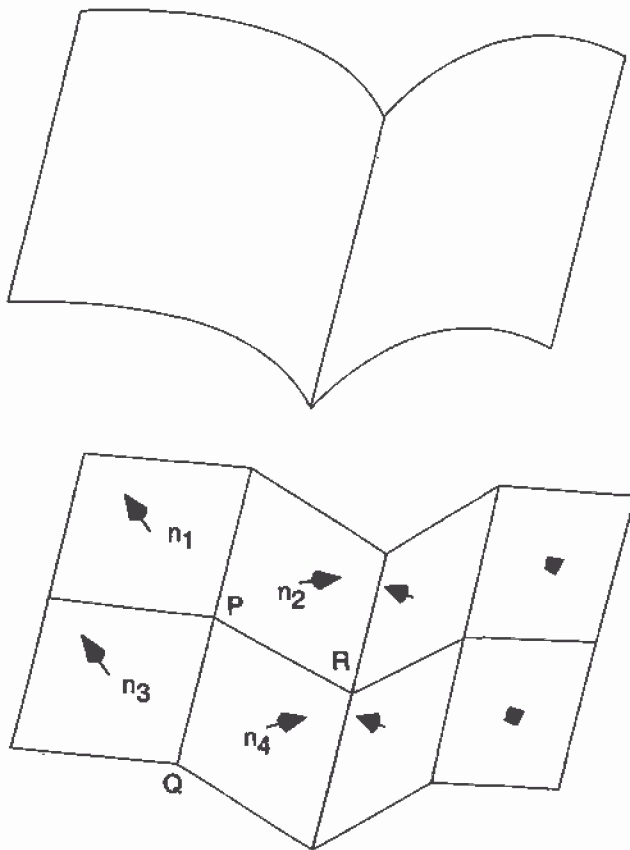


Figure E-2 Averaging Normal Vectors

To find the normal for a flat polygon, take any three vertices v_1 , v_2 , and v_3 of the polygon that do not lie in a straight line. The cross product

$$[v_1 - v_2] \times [v_2 - v_3]$$

is perpendicular to the polygon. (Typically, you want to normalize the resulting vector.) Then you need to average the normals for adjoining facets to avoid giving too much weight to one of them. For instance, in the example shown in Figure E-2, if n_1 , n_2 , n_3 , and n_4 are the normals for the four polygons meeting at point P, calculate $n_1+n_2+n_3+n_4$ and then normalize it. (You can get a better average if you weight the normals by the size of the angles at the shared intersection.) The resulting vector can be used as the normal for point P.

Sometimes, you need to vary this method for particular situations. For instance, at the boundary of a surface (for example, point Q in Figure E-2), you might be able to choose a better normal based on your knowledge of what the surface should look like. Sometimes the best you can do is to average the polygon normals on the boundary as well. Similarly, some models have some smooth parts and some sharp corners (point R is on such an edge in Figure E-2). In this case, the normals on either side of the crease shouldn't be averaged. Instead, polygons on one side of the crease should be drawn with one normal, and polygons on the other side with another.

Homogeneous Coordinates and Transformation Matrices

This appendix presents a brief discussion of homogeneous coordinates. It also lists the form of the transformation matrices used for rotation, scaling, translation, perspective projection, and orthographic projection. These topics are introduced and discussed in Chapter 3. For a more detailed discussion of these subjects, see almost any book on three-dimensional computer graphics—for example, *Computer Graphics: Principles and Practice* by Foley, van Dam, Feiner, and Hughes (Reading, MA: Addison-Wesley, 1990)—or a text on projective geometry—for example, *The Real Projective Plane*, by H. S. M. Coxeter, 2nd ed. (Cambridge: Cambridge University Press, 1961). In the discussion that follows, the term *homogeneous coordinates* always means three-dimensional homogeneous coordinates, although projective geometries exist for all dimensions.

This appendix has the following major sections:

- “Homogeneous Coordinates” on page 594
- “Transformation Matrices” on page 595

Homogeneous Coordinates

OpenGL commands usually deal with two- and three-dimensional vertices, but in fact all are treated internally as three-dimensional homogeneous vertices comprising four coordinates. Every column vector $(x, y, z, w)^T$ represents a homogeneous vertex if at least one of its elements is nonzero. If the real number a is nonzero, then $(x, y, z, w)^T$ and $(ax, ay, az, aw)^T$ represent the same homogeneous vertex. (This is just like fractions: $x/y = (ax)/(ay)$.) A three-dimensional euclidean space point $(x, y, z)^T$ becomes the homogeneous vertex with coordinates $(x, y, z, 1.0)^T$, and the two-dimensional euclidean point $(x, y)^T$ becomes $(x, y, 0.0, 1.0)^T$.

As long as w is nonzero, the homogeneous vertex $(x, y, z, w)^T$ corresponds to the three-dimensional point $(x/w, y/w, z/w)^T$. If $w = 0.0$, it corresponds to no euclidean point, but rather to some idealized "point at infinity." To understand this point at infinity, consider the point $(1, 2, 0, 0)$, and note that the sequence of points $(1, 2, 0, 1)$, $(1, 2, 0, 0.01)$, and $(1, 2.0, 0.0, 0.0001)$, corresponds to the euclidean points $(1, 2)$, $(100, 200)$, and $(10000, 20000)$. This sequence represents points rapidly moving toward infinity along the line $2x = y$. Thus, you can think of $(1, 2, 0, 0)$ as the point at infinity in the direction of that line.

Note: OpenGL might not handle homogeneous clip coordinates with $w < 0$ correctly. To be sure that your code is portable to all OpenGL systems, use only nonnegative w values.

Transforming Vertices

Vertex transformations (such as rotations, translations, scaling, and shearing) and projections (such as perspective and orthographic) can all be represented by applying an appropriate 4×4 matrix to the coordinates representing the vertex. If \mathbf{v} represents a homogeneous vertex and \mathbf{M} is a 4×4 transformation matrix, then $\mathbf{M}\mathbf{v}$ is the image of \mathbf{v} under the transformation by \mathbf{M} . (In computer-graphics applications, the transformations used are usually nonsingular—in other words, the matrix \mathbf{M} can be inverted. This isn't required, but some problems arise with nonsingular transformations.)

After transformation, all transformed vertices are clipped so that x , y , and z are in the range $[-w, w]$ (assuming $w > 0$). Note that this range corresponds in euclidean space to $[-1.0, 1.0]$.

Transforming Normals

Normal vectors aren't transformed in the same way as vertices or position vectors. Mathematically, it's better to think of normal vectors not as vectors, but as planes perpendicular to those vectors. Then, the transformation rules for normal vectors are described by the transformation rules for perpendicular planes.

A homogeneous plane is denoted by the row vector (a, b, c, d) , where at least one of a, b, c , or d is nonzero. If q is a nonzero real number, then (a, b, c, d) and (qa, qb, qc, qd) represent the same plane. A point $(x, y, z, w)^T$ is on the plane (a, b, c, d) if $ax+by+cz+dw=0$. (If $w=1$, this is the standard description of a euclidean plane.) In order for (a, b, c, d) to represent a euclidean plane, at least one of a, b , or c must be nonzero. If they're all zero, then $(0, 0, 0, d)$ represents the "plane at infinity," which contains all the "points at infinity."

If p is a homogeneous plane and v is a homogeneous vertex, then the statement " v lies on plane p " is written mathematically as $pv=0$, where pv is normal matrix multiplication. If M is a nonsingular vertex transformation (that is, a 4×4 matrix that has an inverse M^{-1}), then $pv=0$ is equivalent to $pM^{-1}Mv=0$, so Mv lies on the plane pM^{-1} . Thus, pM^{-1} is the image of the plane under the vertex transformation M .

If you like to think of normal vectors as vectors instead of as the planes perpendicular to them, let v and n be vectors such that v is perpendicular to n . Then, $n^T v = 0$. Thus, for an arbitrary nonsingular transformation M , $n^T M^{-1} Mv = 0$, which means that $n^T M^{-1}$ is the transpose of the transformed normal vector. Thus, the transformed normal vector is $(M^{-1})^T n$. In other words, normal vectors are transformed by the inverse transpose of the transformation that transforms points. Whew!

Transformation Matrices

Although any nonsingular matrix M represents a valid projective transformation, a few special matrices are particularly useful. These matrices are listed in the following subsections.

Translation

The call `glTranslate*(x, y, z)` generates T , where

$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad T^{-1} = \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

The call `glScale*(x, y, z)` generates S , where

$$S = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad S^{-1} = \begin{bmatrix} \frac{1}{x} & 0 & 0 & 0 \\ 0 & \frac{1}{y} & 0 & 0 \\ 0 & 0 & \frac{1}{z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice that S^{-1} is defined only if x , y , and z are all nonzero.

Rotation

The call `glRotate*(a, x, y, z)` generates R as follows:

Let $v = (x, y, z)^T$, and $u = v/\|v\| = (x', y', z')^T$.

Also let

$$S = \begin{bmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{bmatrix} \quad \text{and} \quad M = uu^T + (\cos a)(I - uu^T) + (\sin a)S$$

Then

$$R = \begin{bmatrix} m & m & m & 0 \\ m & m & m & 0 \\ m & m & m & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ where } m \text{ represents elements from } M, \text{ which is a } 3 \times 3 \text{ matrix.}$$

The R matrix is always defined. If $x=y=z=0$, then R is the identity matrix. You can obtain the inverse of R , R^{-1} , by substituting $-a$ for a , or by transposition.

The `glRotate*()` command generates a matrix for rotation about an arbitrary axis. Often, you're rotating about one of the coordinate axes; the corresponding matrices are as follows:

$$\text{glRotate}^*(a, 1, 0, 0): \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos a & -\sin a & 0 \\ 0 & \sin a & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{glRotate}^*(a, 0, 1, 0): \begin{bmatrix} \cos a & 0 & \sin a & 0 \\ 0 & 1 & 0 & 0 \\ -\sin a & 0 & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{glRotate}^*(a, 0, 0, 1): \begin{bmatrix} \cos a & -\sin a & 0 & 0 \\ \sin a & \cos a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As before, the inverses are obtained by transposition.

Perspective Projection

The call `glFrustum(l, r, b, t, n, f)` generates **R**, where

$$\mathbf{R} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{R}^{-1} = \begin{bmatrix} \frac{r-l}{2n} & 0 & 0 & \frac{r+l}{2n} \\ 0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n} \\ 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{-(f-n)}{2fn} & \frac{f+n}{2fn} \end{bmatrix}$$

R is defined as long as $l \neq r$, $t \neq b$, and $n \neq f$.

Orthographic Projection

The call `glOrtho(l, r, b, t, n, f)` generates **R**, where

$$\mathbf{R} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{R}^{-1} = \begin{bmatrix} \frac{r-l}{2} & 0 & 0 & \frac{r+l}{2} \\ 0 & \frac{t-b}{2} & 0 & \frac{t+b}{2} \\ 0 & 0 & \frac{f-n}{-2} & \frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

R is defined as long as $l \neq r$, $t \neq b$, and $n \neq f$.

Programming Tips

This appendix lists some tips and guidelines that you might find useful. Keep in mind that these tips are based on the intentions of the designers of the OpenGL, not on any experience with actual applications and implementations! This appendix has the following major sections:

- “OpenGL Correctness Tips” on page 600
- “OpenGL Performance Tips” on page 602
- “GLX Tips” on page 603

OpenGL Correctness Tips

- Perform error checking often. Call `glGetError()` at least once each time the scene is rendered to make certain error conditions are noticed.
- Do not count on the error behavior of an OpenGL implementation—it might change in a future release of OpenGL. For example, OpenGL 1.1 ignores matrix operations invoked between `glBegin()` and `glEnd()` commands, but a future version might not. Put another way, OpenGL error semantics may change between upward-compatible revisions.
- If you need to collapse all geometry to a single plane, use the projection matrix. If the modelview matrix is used, OpenGL features that operate in eye coordinates (such as lighting and application-defined clipping planes) might fail.
- Do not make extensive changes to a single matrix. For example, do not animate a rotation by continually calling `glRotate*()` with an incremental angle. Rather, use `glLoadIdentity()` to initialize the given matrix for each frame, then call `glRotate*()` with the desired complete angle for that frame.
- Count on multiple passes through a rendering database to generate the same pixel fragments only if this behavior is guaranteed by the invariance rules established for a compliant OpenGL implementation. (See Appendix H for details on the invariance rules.) Otherwise, a different set of fragments might be generated.
- Do not expect errors to be reported while a display list is being defined. The commands within a display list generate errors only when the list is executed.
- Place the near frustum plane as far from the viewpoint as possible to optimize the operation of the depth buffer.
- Call `glFlush()` to force all previous OpenGL commands to be executed. Do not count on `glGet*()` or `glIs*()` to flush the rendering stream. Query commands flush as much of the stream as is required to return valid data but don't guarantee completing all pending rendering commands.
- Turn dithering off when rendering predithered images (for example, when `glCopyPixels()` is called).
- Make use of the full range of the accumulation buffer. For example, if accumulating four images, scale each by one-quarter as it's accumulated.

-
- If exact two-dimensional rasterization is desired, you must carefully specify both the orthographic projection and the vertices of primitives that are to be rasterized. The orthographic projection should be specified with integer coordinates, as shown in the following example:

```
gluOrtho2D(0, width, 0, height);
```

where *width* and *height* are the dimensions of the viewport. Given this projection matrix, polygon vertices and pixel image positions should be placed at integer coordinates to rasterize predictably. For example, `glRecti(0, 0, 1, 1)` reliably fills the lower left pixel of the viewport, and `glRasterPos2i(0, 0)` reliably positions an unzoomed image at the lower left of the viewport. Point vertices, line vertices, and bitmap positions should be placed at half-integer locations, however. For example, a line drawn from $(x1, 0.5)$ to $(x2, 0.5)$ will be reliably rendered along the bottom row of pixels into the viewport, and a point drawn at $(0.5, 0.5)$ will reliably fill the same pixel as `glRecti(0, 0, 1, 1)`.

An optimum compromise that allows all primitives to be specified at integer positions, while still ensuring predictable rasterization, is to translate x and y by 0.375, as shown in the following code fragment. Such a translation keeps polygon and pixel image edges safely away from the centers of pixels, while moving line vertices close enough to the pixel centers.

```
glViewport(0, 0, width, height);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0, width, 0, height);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.375, 0.375, 0.0);
/* render all primitives at integer positions */
```

- Avoid using negative w vertex coordinates and negative q texture coordinates. OpenGL might not clip such coordinates correctly and might make interpolation errors when shading primitives defined by such coordinates.
- Do not assume the precision of operations, based upon the data type of parameters to OpenGL commands. For example, if you are using `glRotated()`, you should not assume that geometric processing pipeline operates with double-precision floating point. It is possible that the parameters to `glRotated()` are converted to a different data type before processing.

OpenGL Performance Tips

- Use `glColorMaterial()` when only a single material property is being varied rapidly (at each vertex, for example). Use `glMaterial()` for infrequent changes, or when more than a single material property is being varied rapidly.
- Use `glLoadIdentity()` to initialize a matrix, rather than loading your own copy of the identity matrix.
- Use specific matrix calls such as `glRotate*()`, `glTranslate*()`, and `glScale*()` rather than composing your own rotation, translation, or scale matrices and calling `glMultMatrix()`.
- Use query functions when your application requires just a few state values for its own computations. If your application requires several state values from the same attribute group, use `glPushAttrib()` and `glPopAttrib()` to save and restore them.
- Use display lists to encapsulate potentially expensive state changes.
- Use display lists to encapsulate the rendering calls of rigid objects that will be drawn repeatedly.
- Use texture objects to encapsulate texture data. Place all the `glTexImage*()` calls (including mipmaps) required to completely specify a texture and the associated `glTexParameter*()` calls (which set texture properties) into a texture object. Bind this texture object to select the texture.
- If the situation allows it, use `gl*TexSubImage()` to replace all or part of an existing texture image rather than the more costly operations of deleting and creating an entire new image.
- If your OpenGL implementation supports a high-performance working set of resident textures, try to make all your textures resident; that is, make them fit into the high-performance texture memory. If necessary, reduce the size or internal format resolution of your textures until they all fit into memory. If such a reduction creates intolerably fuzzy textured objects, you may give some textures lower priority, which will, when push comes to shove, leave them out of the working set.
- Use evaluators even for simple surface tessellations to minimize network bandwidth in client-server environments.
- Provide unit-length normals if it's possible to do so, and avoid the overhead of `GL_NORMALIZE`. Avoid using `glScale*()` when doing

lighting because it almost always requires that `GL_NORMALIZE` be enabled.

- Set `glShadeModel()` to `GL_FLAT` if smooth shading isn't required.
- Use a single `glClear()` call per frame if possible. Do not use `glClear()` to clear small subregions of the buffers; use it only for complete or near-complete clears.
- Use a single call to `glBegin(GL_TRIANGLES)` to draw multiple independent triangles rather than calling `glBegin(GL_TRIANGLES)` multiple times, or calling `glBegin(GL_POLYGON)`. Even if only a single triangle is to be drawn, use `GL_TRIANGLES` rather than `GL_POLYGON`. Use a single call to `glBegin(GL_QUADS)` in the same manner rather than calling `glBegin(GL_POLYGON)` repeatedly. Likewise, use a single call to `glBegin(GL_LINES)` to draw multiple independent line segments rather than calling `glBegin(GL_LINES)` multiple times.
- Some OpenGL implementations benefit from storing vertex data in vertex arrays. Use of vertex arrays reduces function call overhead. Some implementations can improve performance by batch processing or reusing processed vertices.
- In general, use the vector forms of commands to pass precomputed data, and use the scalar forms of commands to pass values that are computed near call time.
- Avoid making redundant mode changes, such as setting the color to the same value between each vertex of a flat-shaded polygon.
- Be sure to disable expensive rasterization and per-fragment operations when drawing or copying images. OpenGL will even apply textures to pixel images if asked to!
- Unless absolutely needed, avoid having different front and back polygon modes.

GLX Tips

- Use `glXWaitGL()` rather than `glFinish()` to force X rendering commands to follow GL rendering commands.
- Likewise, use `glXWaitX()` rather than `XSync()` to force GL rendering commands to follow X rendering commands.

-
- Be careful when using `glXChooseVisual()`, because boolean selections are matched exactly. Since some implementations won't export visuals with all combinations of boolean capabilities, you should call `glXChooseVisual()` several times with different boolean values before you give up. For example, if no single-buffered visual with the required characteristics is available, check for a double-buffered visual with the same capabilities. It might be available, and it's easy to use.

OpenGL Invariance

OpenGL is not a pixel-exact specification. It therefore doesn't guarantee an exact match between images produced by different OpenGL implementations. However, OpenGL does specify exact matches, in some cases, for images produced by the same implementation. This appendix describes the invariance rules that define these cases.

The obvious and most fundamental case is repeatability. A conforming OpenGL implementation generates the same results each time a specific sequence of commands is issued from the same initial conditions. Although such repeatability is useful for testing and verification, it's often not useful to application programmers, because it's difficult to arrange for equivalent initial conditions. For example, rendering a scene twice, the second time after swapping the front and back buffers, doesn't meet this requirement. So repeatability can't be used to guarantee a stable, double-buffered image.

A simple and useful algorithm that counts on invariant execution is erasing a line by redrawing it in the background color. This algorithm works only if rasterizing the line results in the same fragment x,y pairs being generated in both the foreground and background color cases. OpenGL requires that the coordinates of the fragments generated by rasterization be invariant with respect to framebuffer contents, which color buffers are enabled for drawing, the values of matrices other than those on the top of the matrix stacks, the scissor parameters, all writemasks, all clear values, the current color, index, normal, texture coordinates, and edge-flag values, the current raster color, raster index, and raster texture coordinates, and the material properties. It is further required that exactly the same fragments be generated, including the fragment color values, when framebuffer contents, color buffer enables, matrices other than those on the top of the matrix stacks, the scissor parameters, writemasks, or clear values differ.

OpenGL further suggests, but doesn't require, that fragment generation be invariant with respect to the matrix mode, the depths of the matrix stacks, the alpha test parameters (other than alpha test enable), the stencil parameters (other than stencil enable), the depth test parameters (other than depth test enable), the blending parameters (other than enable), the logical operation (but not logical operation enable), and the pixel-storage and pixel-transfer parameters. Because invariance with respect to several enables isn't recommended, you should use other parameters to disable functions when invariant rendering is required. For example, to render invariantly with blending enabled and disabled, set the blending parameters to `GL_ONE` and `GL_ZERO` to disable blending rather than calling `glDisable(GL_BLEND)`. Alpha testing, stencil testing, depth testing, and the logical operation all can be disabled in this manner.

Finally, OpenGL requires that per-fragment arithmetic, such as blending and the depth test, is invariant to all OpenGL state except the state that directly defines it. For example, the only OpenGL parameters that affect how the arithmetic of blending is performed are the source and destination blend parameters and the blend enable parameter. Blending is invariant to all other state changes. This invariance holds for the scissor test, the alpha

test, the stencil test, the depth test, blending, dithering, logical operations, and buffer writemasking.

As a result of all these invariance requirements, OpenGL can guarantee that images rendered into different color buffers, either simultaneously or separately using the same command sequence, are pixel identical. This holds for all the color buffers in the framebuffer or all the color buffers in an off-screen buffer, but it isn't guaranteed between the framebuffer and off-screen buffers.



accumulation buffer

Memory (bitplanes) that is used to accumulate a series of images generated in the color buffer. Using the accumulation buffer may significantly improve the quality of the image, but also take correspondingly longer to render. The accumulation buffer is used for effects such as depth of field, motion blur, and full-scene antialiasing.

aliasing

A rendering technique that assigns to pixels the color of the primitive being rendered, regardless of whether that primitive covers all or only a portion of the pixel's area. This results in jagged edges, or jaggies.

alpha

A fourth color component. The alpha component is never displayed directly and is typically used to control color blending. By convention, OpenGL alpha corresponds to the notion of opacity rather than transparency, meaning that an alpha value of 1.0 implies complete opacity, and an alpha value of 0.0 complete transparency.

ambient

Ambient light is nondirectional and distributed uniformly throughout space. Ambient light falling upon a surface approaches from all directions. The light is reflected from the object independent of surface location and orientation with equal intensity in all directions.

animation

Generating repeated renderings of a scene, with smoothly changing viewpoint and/or object positions, quickly enough so that the illusion of motion is achieved. OpenGL animation is almost always done using double-buffering.

antialiasing

A rendering technique that assigns pixel colors based on the fraction of the pixel's area that's covered by the primitive being rendered. Antialiased rendering reduces or eliminates the jaggies that result from aliased rendering.

application-specific clipping

Clipping of primitives against planes in eye coordinates; the planes are specified by the application using `glClipPlane()`.

attribute group

A set of related state variables, which OpenGL can save or restore together at one time.

back faces

See *faces*.

bit

Binary digit. A state variable having only two possible values: 0 or 1. Binary numbers are constructions of one or more bits.

bitmap

A rectangular array of bits. Also, the primitive rendered by the `glBitmap()` command, which uses its *bitmap* parameter as a mask.

bitplane

A rectangular array of bits mapped one-to-one with pixels. The framebuffer is a stack of bitplanes.

blending

Reduction of two color components to one component, usually as a linear interpolation between the two components.

buffer

A group of bitplanes that store a single component (such as depth or green) or a single index (such as the color index or the stencil index). Sometimes the red, green, blue, and alpha buffers together are referred to as the color buffer, rather than the color buffers.

C

God's programming language.

C++

The object-oriented programming language of a pagan deity.

client

The computer from which OpenGL commands are issued. The computer that issues OpenGL commands can be connected via a network to a different computer that executes the commands, or commands can be issued and executed on the same computer. See also *server*.

client memory

The main memory (where program variables are stored) of the client computer.

clip coordinates

The coordinate system that follows transformation by the projection matrix and precedes perspective division. View-volume clipping is done in clip coordinates, but application-specific clipping is not.

clipping

Elimination of the portion of a geometric primitive that's outside the half-space defined by a clipping plane. Points are simply rejected if outside. The portion of a line or of a polygon that's outside the half-space is eliminated, and additional vertices are generated as necessary to complete the primitive within the clipping half-space. Geometric primitives and the current raster position (when specified) are always clipped against the six half-spaces defined by the left, right, bottom, top, near, and far planes of the view volume. Applications can specify optional application-specific clipping planes to be applied in eye coordinates.

color index

A single value that represents a color by name, rather than by value. OpenGL color indices are treated as continuous values (for example, floating-point numbers), while operations such as interpolation and dithering are performed on them. Color indices stored in the framebuffer are always integer values, however. Floating-point indices are converted to integers by rounding to the nearest integer value.

color-index mode

An OpenGL context is in color-index mode if its color buffers store color indices rather than red, green, blue, and alpha color components.

color map

A table of index-to-RGB mappings that's accessed by the display hardware. Each color index is read from the color buffer, converted to an RGB triple by lookup in the color map, and sent to the monitor.

components

Single, continuous (for example, floating-point) values that represent intensities or quantities. Usually, a component value of zero represents the minimum value or intensity, and a component value of one represents the maximum value or intensity, though other ranges are sometimes used. Because component values are interpreted in a normalized range, they are specified independent of actual resolution. For example, the RGB triple (1, 1, 1) is white, regardless of whether the color buffers store 4, 8, or 12 bits each.

Out-of-range components are typically clamped to the normalized range, not truncated or otherwise interpreted. For example, the RGB triple (1.4, 1.5, 0.9) is clamped to (1.0, 1.0, 0.9) before it's used to update the color buffer. Red, green, blue, alpha, and depth are always treated as components, never as indices.

concave

Not *convex*.

context

A complete set of OpenGL state variables. Note that framebuffer contents are not part of OpenGL state, but that the configuration of the framebuffer is.

convex

A polygon is convex if no straight line in the plane of the polygon intersects the polygon more than twice.

convex hull

The smallest convex region enclosing a specified group of points. In two dimensions, the convex hull is found conceptually by stretching a rubber band around the points so that all of the points lie within the band.

coordinate system

In n -dimensional space, a set of n linearly independent vectors anchored to a point (called the origin). A group of coordinates specifies a point in space (or a vector from the origin) by indicating how far to travel along each vector to reach the point (or tip of the vector).

culling

The process of eliminating a front face or back face of a polygon so that it isn't drawn.

current matrix

A matrix that transforms coordinates in one coordinate system to coordinates of another system. There are three current matrices in OpenGL: the modelview matrix transforms object coordinates (coordinates specified by the programmer) to eye coordinates; the perspective matrix transforms eye coordinates to clip coordinates; the texture matrix transforms specified or generated texture coordinates as described by the matrix. Each current matrix is the top element on a stack of matrices. Each of the three stacks can be manipulated with OpenGL matrix-manipulation commands.

current raster position

A window coordinate position that specifies the placement of an image primitive when it's rasterized. The current raster position and other current raster parameters are updated when `glRasterPos()` is called.

decals

A method of calculating color values during texture application, where the texture colors replace the fragment colors or, if alpha blending is enabled, the texture colors are blended with the fragment colors, using only the alpha value.

depth

Generally refers to the z window coordinate.

depth buffer

Memory that stores the depth value at every pixel. To perform hidden-surface removal, the depth buffer records the depth value of the object that lies closest to the observer at every pixel. The depth value of every new fragment uses the recorded value for depth comparison and must pass the comparison test before being rendered.

depth-cuing

A rendering technique that assigns color based on distance from the viewpoint.

diffuse

Diffuse lighting and reflection accounts for the directionality of a light source. The intensity of light striking a surface varies with the angle between the orientation of the object and the direction of the light source. A diffuse material scatters that light evenly in all directions.

directional light source

See *infinite light source*.

display list

A named list of OpenGL commands. Display lists are always stored on the server, so display lists can be used to reduce network traffic in client-server environments. The contents of a display list may be preprocessed and might therefore execute more efficiently than the same set of OpenGL commands executed in immediate mode. Such preprocessing is especially important for computing intensive commands such as NURBS or polygon tessellation.

dithering

A technique for increasing the perceived range of colors in an image at the cost of spatial resolution. Adjacent pixels are assigned differing color values; when viewed from a distance, these colors seem to blend into a single intermediate color. The technique is similar to the halftoning used in black-and-white publications to achieve shades of gray.

double-buffering

OpenGL contexts with both front and back color buffers are double-buffered. Smooth animation is accomplished by rendering into only the back buffer (which isn't displayed), then causing the front and back buffers to be swapped. See `glutSwapBuffers()` in Appendix D.

edge flag

A Boolean value at a vertex which marks whether that vertex precedes a boundary edge. `glEdgeFlag*` may be used to mark an edge as not on the boundary. When a polygon is drawn in `GL_LINE` mode, only boundary edges are drawn.

element

A single component or index.

emission

The color of an object which is self-illuminating or self-radiating. The intensity of an emissive material is not attributed to any external light source.

evaluated

The OpenGL process of generating object-coordinate vertices and parameters from previously specified Bézier equations.

execute

An OpenGL command is executed when it's called in immediate mode or when the display list that it's a part of is called.

eye coordinates

The coordinate system that follows transformation by the modelview matrix and precedes transformation by the projection matrix. Lighting and application-specific clipping are done in eye coordinates.

faces

The sides of a polygon. Each polygon has two faces: a front face and a back face. Only one face or the other is ever visible in the window. Whether the back or front face is visible is effectively determined after the polygon is projected onto the window. After this projection, if the polygon's edges are directed clockwise, one of the faces is visible; if directed counterclockwise, the other face is visible. Whether clockwise corresponds to front or back (and counterclockwise corresponds to back or front) is determined by the OpenGL programmer.

flat shading

Refers to a primitive colored with a single, constant color across its extent, rather than smoothly interpolated colors across the primitive. See *Gouraud shading*.

fog

A rendering technique that can be used to simulate atmospheric effects such as haze, fog, and smog by fading object colors to a background color based on distance from the viewer. Fog also aids in the perception of distance from the viewer, giving a depth cue.

fonts

Groups of graphical character representations generally used to display strings of text. The characters may be roman letters, mathematical symbols, Asian ideograms, Egyptian hieroglyphics, and so on.

fragment

Fragments are generated by the rasterization of primitives. Each fragment corresponds to a single pixel and includes color, depth, and sometimes texture-coordinate values.

framebuffer

All the buffers of a given window or context. Sometimes includes all the pixel memory of the graphics hardware accelerator.

front faces

See *faces*.

frustum

The view volume warped by perspective division.

gamma correction

A function applied to colors stored in the framebuffer to correct for the nonlinear response of the eye (and sometimes of the monitor) to linear changes in color-intensity values.

geometric model

The object-coordinate vertices and parameters that describe an object. Note that OpenGL doesn't define a syntax for geometric models, but rather a syntax and semantics for the rendering of geometric models.

geometric object

See *geometric model*.

geometric primitive

A point, a line, or a polygon.

Gouraud shading

Smooth interpolation of colors across a polygon or line segment. Colors are assigned at vertices and linearly interpolated across the primitive to produce a relatively smooth variation in color. Also called smooth shading.

group

Each pixel of an image in client memory is represented by a group of one, two, three, or four elements. Thus, in the context of a client memory image, a group and a pixel are the same thing.

half-spaces

A plane divides space into two half-spaces.

hidden-line removal

A technique to determine which portions of a wireframe object should be visible. The lines that comprise the wireframe are considered to be edges of opaque surfaces, which may obscure other edges that are farther away from the viewer.

hidden-surface removal

A technique to determine which portions of an opaque, shaded object should be visible and which portions should be obscured. A test of the depth coordinate, using the depth buffer for storage, is a common method of hidden-surface removal.

homogeneous coordinates

A set of $n+1$ coordinates used to represent points in n -dimensional projective space. Points in projective space can be thought of as points in euclidean space together with some points at infinity. The coordinates are homogeneous because a scaling of each of the coordinates by the same nonzero constant doesn't alter the point to which the coordinates refer. Homogeneous coordinates are useful in the calculations of projective geometry, and thus in computer graphics, where scenes must be projected onto a window.

image

A rectangular array of pixels, either in client memory or in the framebuffer.

image primitive

A bitmap or an image.

immediate mode

Execution of OpenGL commands when they're called, rather than from a display list. No immediate-mode bit exists; the mode in immediate mode refers to use of OpenGL, rather than to a specific bit of OpenGL state.

index

A single value that's interpreted as an absolute value, rather than as a normalized value in a specified range (as is a component). Color indices are the names of colors, which are dereferenced by the display hardware using the color map. Indices are typically masked rather than clamped when out of range. For example, the index 0xf7 is masked to 0x7 when written to a 4-bit buffer (color or stencil). Color indices and stencil indices are always treated as indices, never as components.

indices

Preferred plural of index. (The choice between the plural forms indices or indexes—as well as matrices or matrixes and vertices or vertexes—has engendered much debate between the authors and principal reviewers of this guide. The authors' compromise solution is to use the -ices form but to state clearly for the record that the use of indice [sic], matrice [sic], and vertice [sic] for the singular forms is an abomination.)

infinite light source

A directional source of illumination. The radiating light from an infinite light source strikes all objects as parallel rays.

interpolation

Calculation of values (such as color or depth) for interior pixels, given the values at the boundaries (such as at the vertices of a polygon or a line).

IRIS GL

Silicon Graphics proprietary graphics library, developed from 1982 through 1992. OpenGL was designed with IRIS GL as a starting point.

IRIS Inventor

See *Open Inventor*.

jaggies

Artifacts of aliased rendering. The edges of primitives that are rendered with aliasing are jagged rather than smooth. A near-horizontal aliased line, for example, is rendered as a set of horizontal lines on adjacent pixel rows rather than as a smooth, continuous line.

jittering

A pseudo-random displacement (shaking) of the objects in a scene, used in conjunction with the accumulation buffer to achieve special effects.

lighting

The process of computing the color of a vertex based on current lights, material properties, and lighting-model modes.

line

A straight region of finite width between two vertices. (Unlike mathematical lines, OpenGL lines have finite width and length.) Each segment of a strip of lines is itself a line.

local light source

A source of illumination which has an exact position. The radiating light from a local light source emanates from that position. Other names for a local light source are point light source or positional light source. A spotlight is a special kind of local light source.

logical operation

Boolean mathematical operations between the incoming fragment's RGBA color or color-index values and the RGBA color or color-index values already stored at the corresponding location in the framebuffer. Examples of logical operations include AND, OR, XOR, NAND, and INVERT.

luminance

The perceived brightness of a surface. Often refers to a weighted average of red, green, and blue color values that gives the perceived brightness of the combination.

matrices

Preferred plural of matrix. See *indices*.

matrix

A two-dimensional array of values. OpenGL matrices are all 4×4, though when stored in client memory they're treated as 1×16 single-dimension arrays.

modelview matrix

The 4×4 matrix that transforms points, lines, polygons, and raster positions from object coordinates to eye coordinates.

modulate

A method of calculating color values during texture application, where the texture and the fragment colors are combined.

monitor

The device that displays the image in the framebuffer.

motion blurring

A technique that uses the accumulation buffer to simulate what appears on film when you take a picture of a moving object or when you move the camera while taking a picture of a stationary object. In animations without motion blur, moving objects can appear jerky.

network

A connection between two or more computers that allows each to transfer data to and from the others.

nonconvex

A polygon is nonconvex if there exists a line in the plane of the polygon that intersects the polygon more than twice.

normal

A three-component plane equation that defines the angular orientation, but not position, of a plane or surface.

normalized

To normalize a normal vector, divide each of the components by the square root of the sum of their squares. Then, if the normal is thought of as a vector from the origin to the point (nx', ny', nz') , this vector has unit length.

$$\text{factor} = \text{sqrt}(nx^2 + ny^2 + nz^2)$$

$$nx' = nx / \text{factor}$$

$$ny' = ny / \text{factor}$$

$$nz' = nz / \text{factor}$$

normal vectors

See *normal*.

NURBS

Non-Uniform Rational B-Spline. A common way to specify parametric curves and surfaces. (See GLU NURBS routines in Chapter 12.)

object

An object-coordinate model that's rendered as a collection of primitives.

object coordinates

Coordinate system prior to any OpenGL transformation.

Open Inventor

An object-oriented 3D toolkit, built on top of OpenGL, based on a 3D scene database and user interaction components. It includes objects such as cubes, polygons, text, materials, cameras, lights, trackballs and handle boxes.

orthographic

Nonperspective projection, as in some engineering drawings, with no foreshortening.

parameters

Values passed as arguments to OpenGL commands. Sometimes parameters are passed by reference to an OpenGL command.

perspective division

The division of x , y , and z by w , carried out in clip coordinates.

pixel

Picture element. The bits at location (x, y) of all the bitplanes in the framebuffer constitute the single pixel (x, y) . In an image in client memory, a pixel is one group of elements. In OpenGL window coordinates, each pixel corresponds to a 1.0×1.0 screen area. The coordinates of the lower-left corner of the pixel are x, y are (x, y) , and of the upper-right corner are $(x+1, y+1)$.

point

An exact location in space, which is rendered as a finite-diameter dot.

point light source

See *local light source*.

polygon

A near-planar surface bounded by edges specified by vertices. Each triangle of a triangle mesh is a polygon, as is each quadrilateral of a quadrilateral mesh. The rectangle specified by `glRect*()` is also a polygon.

positional light source

See *local light source*.

primitive

A point, a line, a polygon, a bitmap, or an image. (Note: Not just a point, a line, or a polygon!)

projection matrix

The 4×4 matrix that transforms points, lines, polygons, and raster positions from eye coordinates to clip coordinates.

proxy texture

A placeholder for a texture image, which is used to determine if there are enough resources to support a texture image of a given size and internal format resolution.

quadrilateral

A polygon with four edges.

rasterized

Converted a projected point, line, or polygon, or the pixels of a bitmap or image, to fragments, each corresponding to a pixel in the framebuffer. Note that all primitives are rasterized, not just points, lines, and polygons.

rectangle

A quadrilateral whose alternate edges are parallel to each other in object coordinates. Polygons specified with `glRect*()` are always rectangles; other quadrilaterals might be rectangles.

rendering

Conversion of primitives specified in object coordinates to an image in the framebuffer. Rendering is the primary operation of OpenGL—it's what OpenGL does.

resident texture

A texture image that is cached in special, high-performance texture memory. If an OpenGL implementation does not have special, high-performance texture memory, then all texture images are deemed resident textures.

RGBA

Red, Green, Blue, Alpha.

RGBA mode

An OpenGL context is in RGBA mode if its color buffers store red, green, blue, and alpha color components, rather than color indices.

server

The computer on which OpenGL commands are executed. This might differ from the computer from which commands are issued. See *client*.

shading

The process of interpolating color within the interior of a polygon, or between the vertices of a line, during rasterization.

shininess

The exponent associated with specular reflection and lighting. Shininess controls the degree with which the specular highlight decays.

single-buffering

OpenGL contexts that don't have back color buffers are single-buffered. You can use these contexts for animation, but take care to avoid visually disturbing flashes when rendering.

singular matrix

A matrix that has no inverse. Geometrically, such a matrix represents a transformation that collapses points along at least one line to a single point.

specular

Specular lighting and reflection incorporates reflection off shiny objects and the position of the viewer. Maximum specular reflectance occurs when the angle between the viewer and the direction of the reflected light is zero. A specular material scatters light with greatest intensity in the direction of the reflection, and its brightness decays, based upon the exponential value shininess.

spotlight

A special type of local light source that has a direction (where it points to) as well as a position. A spotlight simulates a cone of light, which may have a fall-off in intensity, based upon distance from the center of the cone.

stencil buffer

Memory (bitplanes) that is used for additional per-fragment testing, along with the depth buffer. The stencil test may be used for masking regions, capping solid geometry, and overlapping translucent polygons.

stereo

Enhanced three-dimensional perception of a rendered image by computing separate images for each eye. Stereo requires special hardware, such as two synchronized monitors or special glasses to alternate viewed frames for each eye. Some implementations of OpenGL support stereo by having both left and right buffers for color data.

stipple

A one- or two-dimensional binary pattern that defeats the generation of fragments where its value is zero. Line stipples are one-dimensional and are applied relative to the start of a line. Polygon stipples are two-dimensional and are applied with a fixed orientation to the window.

tessellation

Reduction of a portion of an analytic surface to a mesh of polygons, or of a portion of an analytic curve to a sequence of lines.

texel

A texture element. A texel is obtained from texture memory and represents the color of the texture to be applied to a corresponding fragment.

textures

One- or two-dimensional images that are used to modify the color of fragments produced by rasterization.

texture mapping

The process of applying an image (the texture) to a primitive. Texture mapping is often used to add realism to a scene. For example, you can apply a picture of a building facade to a polygon representing a wall.

texture matrix

The 4×4 matrix that transforms texture coordinates from the coordinates in which they're specified to the coordinates that are used for interpolation and texture lookup.

texture object

A named cache that stores texture data, such as the image array, associated mipmaps, and associated texture parameter values: width, height, border width, internal format, resolution of components, minification and magnification filters, wrapping modes, border color, and texture priority.

transformations

The warping of spaces. In OpenGL, transformations are limited to projective transformations that include anything that can be represented by a 4×4 matrix. Such transformations include rotations, translations, (nonuniform) scalings along the coordinate axes, perspective transformations, and combinations of these.

triangle

A polygon with three edges. Triangles are always convex.

vertex

A point in three-dimensional space.

vertex array

Where a block of vertex data (vertex coordinates, texture coordinates, surface normals, RGBA colors, color indices, and edge flags) may be stored in an array and then used to specify multiple geometric primitives through the execution of a single OpenGL command.

vertices

Preferred plural of vertex. See *indices*.

viewpoint

The origin of either the eye- or the clip-coordinate system, depending on context. (For example, when discussing lighting, the viewpoint is the origin of the eye-coordinate system. When discussing projection, the viewpoint is the origin of the clip-coordinate system.) With a typical projection matrix, the eye-coordinate and clip-coordinate origins are at the same location.

view volume

The volume in clip coordinates whose coordinates satisfy the three conditions

$$-w \leq x \leq w$$

$$-w \leq y \leq w$$

$$-w \leq z \leq w$$

Geometric primitives that extend outside this volume are clipped.

VRML

VRML stands for Virtual Reality Modeling Language, which is (according to the VRML Mission Statement) “a universal description language for multi-participant simulations.” VRML is specifically designed to allow people to navigate through three-dimensional worlds that are placed on the World Wide Web. The first versions of VRML are subsets of the Open Inventor file format with additions to allow hyperlinking to the Web (to URLs—Universal Resource Locators).

window

A subregion of the framebuffer, usually rectangular, whose pixels all have the same buffer configuration. An OpenGL context renders to a single window at a time.

window-aligned

When referring to line segments or polygon edges, implies that these are parallel to the window boundaries. (In OpenGL, the window is rectangular, with horizontal and vertical edges). When referring to a polygon pattern, implies that the pattern is fixed relative to the window origin.

window coordinates

The coordinate system of a window. It's important to distinguish between the names of pixels, which are discrete, and the window-coordinate system, which is continuous. For example, the pixel at the lower-left corner of a window is pixel (0, 0); the window coordinates of the center of this pixel are (0.5, 0.5, z). Note that window coordinates include a depth, or z, component, and that this component is continuous as well.

wireframe

A representation of an object that contains line segments only. Typically, the line segments indicate polygon edges.

working set

On machines with special hardware that increases texture performance, this is the group of texture objects that are currently resident. The performance of textures within the working set outperforms the textures outside the working set.

X Window System

A window system used by many of the machines on which OpenGL is implemented. GLX is the name of the OpenGL extension to the X Window System. (See Appendix C.)

A

- accumulation buffer, 376, 378, 394–408
 - clearing, 32, 379
 - depth-of-field effect, use for, 402–406
 - examples of use, 394
 - full range for best results, use, 600
 - motion blur, use for, 402
 - sample program with depth-of-field effect, 404
 - sample program with full-scene antialiasing, 397
 - scene antialiasing, use for, 396
- AGL, 566
 - aglChoosePixelFormat(), 566, 569
 - aglCopyContext(), 567, 569
 - aglCreateAGLPixmap(), 567, 569
 - aglCreateContext(), 567, 569
 - aglDestroyAGLPixmap(), 568, 569
 - aglDestroyContext(), 567, 569
 - aglGetConfig(), 566, 569
 - aglGetCurrentContext(), 567, 569
 - aglGetCurrentDrawable(), 567, 569
 - aglGetError(), 568, 569
 - aglListPixelFormats(), 566, 569
 - aglMakeCurrent(), 567, 569
 - aglQueryVersion(), 566, 568
 - aglSetOptions(), 567, 569
 - aglSwapBuffers(), 568, 569
 - aglUpdateCurrent(), 568, 569
 - aglUseFont(), 568, 569
- airbrushing, 528
- Akeley, Kurt, 394
- aliasing, *See* antialiasing
- alpha, 214
 - destination alpha, 236
 - material properties, 197
 - texture image data type, 354
- alpha blending, *See* blending
- alpha test, 384
 - querying current values, 384
 - rendering pipeline stage, 14, 533
- ambient
 - contribution to lighting equation, 207
 - global light, 193, 206
 - light, 173, 174, 182
 - material properties, 175, 197
- animation, 20–24, 600
- antialiasing, 226–239
 - accumulation buffer used for, 395–401
 - characters (by masking), 512
 - characters (by texturing), 523
 - color-index mode, 232
 - coverage values, 227
 - enabling for points or lines, 228
 - enabling for polygons, 236
 - lines, 226, 228–235
 - lines (by texturing), 523
 - points, 228–235, 514
 - polygons, 235
 - RGBA mode, 229
 - sample program in color-index mode, 232
 - sample program in RGBA mode, 229
 - sample program of filled polygons, 236
 - scene, with the accumulation buffer, 396
- architectural applications
 - orthographic parallel projection, use of, 124
- arcs, 428
- aspect ratio
 - perspective projection, 122
 - viewport transformation, 126
- atmospheric effects, *See* fog
- attenuation of light, 183–184
- attribute groups, 78–81
 - client, 78
 - list of, 537–559
 - performance tips, 602
 - server, 78
 - stack depth, obtaining, 79
 - stacks, 78
- auxiliary buffers, 377, 380

B

- back-facing polygons, 56
 - culling, 57
 - material property, specifying, 196
 - two-sided lighting, 194
- background, 29–32
 - color, 29
 - drawing a fixed, 382, 523
- background processing, 584
- backward compatibility
 - tessellation, 426
 - versions, 503
- basis functions, 439, 440
- Bernstein
 - basis, 439
 - polynomial, 443
- Bézier
 - basis, 439, 440
 - curve, 443
 - sample program using mesh for surface, 451
 - sample program which draws curve, 441
 - sample program which draws surface, 448
 - surface, 446
- billboarding, 219, 385
- bitmaps, 278–284
 - display lists cache bitmap data, 257
 - distorting, 509
 - drawing, 283
 - feedback mode, 493
 - fonts, used for, 279, 286
 - imaging pipeline operations, 297
 - ordering of data in, 281
 - origin of, 283
 - sample program, 280
 - sample program that creates a font, 287
 - size of, 281
- bitplanes, 156, 374
 - displayable colors, number of, 158
- blending, 214–223, 392
 - antialiasing polygons, 235
 - coverage calculations for antialiasing, 227
 - destination alpha, 236
 - enabling, 216
 - enabling for antialiasing, 229

- factors (source and destination), 215
 - images, 514
 - ordering polygons before drawing, 222
 - rendering pipeline stage, 14, 533
 - sample program for three-dimensional, 223
 - sample program with blended polygons, 220
 - texture function, 356
 - three dimensions, in, 222
 - uses of, 217
- buffer, *See* framebuffer

C

- C programming language, 8
- CAD/CAM, *See* computer-aided design
- camera analogy, 94–95
 - environment mapping, 370
 - viewport transformations, 125
- capping, *See* computational solid geometry
- characters
 - antialiasing, 523
- circles, 428
- clearing the framebuffer, 29–32, 378–379
 - affected by scissoring, dithering, and masking, 379, 533
 - performance tips, 603
- client-server, *See* networked operation
- clip coordinates, 96, 137
 - feedback mode, 493
- clipping, 125
 - interference regions found using clipping planes, 519
 - overview, 92
 - primitives in rendering pipeline, 12, 531
 - viewing volume, 121
- clipping planes
 - additional clipping planes, 96, 136–139
 - depth-buffer resolution, effect on, 600
 - far, 121–125, 129
 - near, 121–125, 129
 - querying number of additional, 137
 - sample program with additional clipping planes, 138

-
- color
 - alpha values, 214
 - background, 30
 - cube showing blended RGB values, 155
 - current raster color, 285
 - human perception, 153
 - RGBA values for, 33, 154
 - specifying, 32
 - specifying for tessellation, 415
 - specifying in color-index mode, 164
 - specifying in RGBA mode, 163
 - color buffer, 154, 156, 374, 376, 377
 - clearing, 32
 - masking, 381
 - color map, 154, 159
 - loading for antialiasing, 232
 - loading for smooth shading, 167
 - loading, using GLUT, 583
 - size of, 160
 - color-index mode, 159–161
 - changing between RGBA mode and, 162
 - choosing between RGBA mode and, 161
 - coverage calculations for antialiasing, 227
 - dithering, 393
 - layering with writemasks, 381
 - lighting, 209–211
 - lighting calculations in, 210
 - texturing limitations, 321, 329
 - vertex arrays, specifying values with, 68
 - command syntax, 7–9
 - compositing images, 219
 - compositing transformations, 139–146
 - computational solid geometry, 421
 - capping, 390
 - difference of several contours, 421
 - interference regions, 518
 - intersection of two contours, 421
 - union of several contours, 421
 - Computer Graphics: Principles and Practice*, xxi, 157, 593
 - computer-aided design
 - orthographic parallel projection, use of, 124
 - concave polygons
 - GLU tessellation, 410
 - stencil buffer, drawing with the, 516
 - cones, 428, 584
 - improving rendering of, 525
 - constant attenuation, 184
 - contours, 365
 - control points, 438, 442, 446, 455
 - convex polygons, 38
 - Conway, John, 526
 - coordinate systems
 - grand, fixed, 106, 115, 140
 - local, 106, 115, 140, 144
 - simple 2D, 35–36
 - coordinates
 - See clip coordinates, depth coordinates, eye coordinates, homogeneous coordinates, normalized device coordinates, object coordinates, q texture coordinates, texture coordinates, w coordinates, or window coordinates
 - coverage, pixel, 227
 - Coxeter, H. S. M., 593
 - cross product, 118, 589
 - CSG, See computational solid geometry
 - culling, 56–57
 - enabling, 57
 - rendering pipeline stage, 12, 532
 - curves and curved surfaces, 40
 - see also evaluators or NURBS
 - Curves and Surfaces for Computer-Aided Geometric Design*, 439
 - cylinders, 428
- ## D
- data types
 - RGBA color conversion, 163
 - special OpenGL, 8
 - texture data, 328
 - warning about data type conversions, 601
 - decals, 385, 515
 - polygon offset used for, 247
 - texture function, 356
 - depth buffer, 172, 376, 377
 - also see hidden-surface removal
-

- background, using masking for a common, 382
 - blending, use for three-dimensional, 222
 - clearing, 32, 172, 379
 - decals, for, 515
 - Dirichlet domains, for, 525
 - drawing static backgrounds, 523
 - masking, 381
 - near frustum plane effect on resolution, 600
 - pixel data, 295, 303
 - depth coordinates, 97, 128
 - perspective division, 128
 - picking use, 485
 - polygon offset, 247–250
 - rendering pipeline stage for depth-range operations, 12, 532
 - sample program with picking, 486
 - selection hit records, 474
 - depth test, 391
 - also see depth buffer
 - rendering pipeline stage, 14, 533
 - depth-cuing, See fog
 - depth-of-field effect, 402–406
 - sample program, 404
 - destination factor, See blending
 - diffuse
 - contribution to lighting equation, 207
 - light, 174, 182
 - material properties, 175, 197
 - directional light source, 182
 - Dirichlet domains, 524
 - disks, 428
 - display lists, 29, 253
 - changing mode settings, 275
 - compiling, 262
 - creating, 259
 - deleting, 267
 - disadvantages, 259, 265
 - editing limitations, 256
 - error handling, 261, 600
 - executing, 259, 265
 - executing multiple, 267
 - font creation, 268, 285
 - hierarchical, 265
 - immediate mode, mixing with, 265
 - indices for, obtaining, 262
 - naming, 262
 - nesting, 265
 - nesting limit, querying, 266
 - networked operation, 264
 - performance tips, 257, 602
 - querying use of an index, 267
 - rendering pipeline stage, 11
 - sample program creating a font, 269
 - sample program for creating, 253, 259
 - sharing among rendering contexts, 563, 575
 - state variables saved and restored, 274
 - tessellation, use with, 426
 - uses for, 257, 275
 - vertex-array data, 264
 - what can be stored in, 263
 - distorted images, 509
 - texture images, 359
 - dithering, 158–159, 392, 600
 - and clearing, 379
 - rendering pipeline stage, 14, 533
 - dot product
 - lighting calculations, use in, 207
 - double-buffering, 22–24
 - automatic glFlush(), 35
 - changing between single-buffering and, 162
 - object selection using the back buffer, 508
 - querying its presence, 377
 - sample program, 24
 - drawing
 - clearing the window, 30
 - forcing completion of, 33
 - icosahedron, 83
 - points, 42
 - polygons, 42, 55
 - preparing for, 29
 - rectangles, 40
 - spheres, cylinders, and disks, 428–436
 - drawing pixel data, See pixel data
 - Duff, Tom, 219
- ## E
- edge flags, 62–63
 - tessellated polygons generate, 414
 - vertex arrays, specifying values with, 68

emission, 175, 198, 206

enabling

alpha test, 384

antialiasing of points or lines, 228

antialiasing polygons, 236

blending, 216

color material properties mode, 201

culling, 57

depth test, 391

dithering, 159, 392

evaluators, 443, 447

fog, 240

lighting, 195

line stippling, 51

logical operations, 393

normal vectors for evaluated surfaces, automatic
generation of, 447, 455

polygon offset, 247

polygon stippling, 58

stencil test, 386

texture coordinate generation, 369

texturing, 322, 326

unit length normal vectors ensured, 65

endianness, 300

environment mapping, 369

error handling, 501–503

error string description, 503

recommended usage, 600

evaluators, 440–454

basis functions, 439, 443

evenly spaced values, 445, 449

one-dimensional, 440

rendering pipeline stage, 11

sample program using mesh for 2D Bézier
surface, 451

sample program which draws 1D Bézier curve,
441

sample program which draws 2D Bézier surface,
448

sample program which generates texture
coordinates, 452

tessellation usage, 602

texture coordinates, generating, 452

two-dimensional, 446, 447

event management, using GLUT, 19

example programs, See programs

extensions

vendor-specific, 505

eye coordinates, 96, 137

texture coordinate generation, 364, 369

F

fade effect, 507

Farin, Gerald E., 439

feedback, 491–498

array contents, 497

pass-through markers, 494

querying current rendering mode, 472

returned data, 493

sample program, 495

steps to perform, 492

tessellation, obtaining vertex data after, 426

Feiner, Steven K., xxi, 593

field of view, 100

calculate, using trigonometry to, 130

filtering, 344–346

mipmapped textures, 338–344, 346

texture border colors, 361

flat shading, 165

flight simulation

fog, use of, 239

flushing, 33, 600

fog, 239–247

blending factors, 243

color-index mode, 244

density, 244

enabling, 240

equations, 243

hints, 240

RGBA mode, 244

sample program in color-index mode, 245

sample program in RGBA mode, 240

Foley, James D., xxi, 157, 593

fonts, 285–289

antialiased characters (by masking), 512

antialiased characters (by texturing), 523

bitmapped, 286

creating with display lists, 268

drawing, 284

- drawing as bitmaps, 279
- multi-byte, 286
- same program, 287
- sample program using multiple display lists, 269
- X fonts, using, 564
- Foran, Jim, 372
- foreshortening, perspective, 120
- fragments, 156, 374
 - alpha test, 384
 - blending, 215
 - depth test, 391
 - rendering pipeline operations, 13, 533
 - scissor test, 383
 - tests, 383–393
 - texture functions, 356
- framebuffer, 156, 375
 - capacity per pixel, 376
 - clearing, 378–379
 - copying pixel data within, 290, 295, 296
 - enabling for reading, 380
 - enabling for writing, 380
 - minimum configuration with the X Window System, 376
 - querying color resolution, 156
 - reading pixel data from, 290, 292
 - writing pixel data to, 290, 294
- front-facing polygons, 56
 - specifying material property for, 196
 - two-sided lighting, 194
- frustum, 120
- ftp (file-transfer protocol) site
 - GLUT source code, xxii
 - GLX specification, 562
 - OpenGL Programming Guide, xxii
- Fundamentals of Computer Aided Geometric Design*, 439
- Fundamentals of Three-Dimensional Computer Graphics*, 318

- G**

- Game of Life, 526
- gamma correction, 157

- Gardner, Martin, 526
- geometric primitives, 37–48, 530–532
 - performance when specifying, 603
 - rendering pipeline stage, 12
- geosciences
 - use of texturing in applications, 364
- giraffe, 160
- glAccum(), 395
- glAlphaFunc(), 384
- glAreTexturesResident(), 352
- glArrayElement(), 71
 - legal between glBegin() and glEnd(), 46
- Glassner, Andrew S., xxi
- glBegin(), 42, 43, 414
 - restrictions, 45
- glBindTexture(), 326, 348
- glBitmap(), 279, 283
 - feedback mode, 493
 - fonts, used for, 286
 - imaging pipeline operations, 297
 - pixel-storage modes effect, 299
- glBlendFunc(), 216
- glCallList(), 256, 259, 265
 - legal between glBegin() and glEnd(), 46
- glCallLists(), 268
 - fonts, use for, 285
 - legal between glBegin() and glEnd(), 46
 - sample program, 287
- glClear(), 30, 31, 379, 533
 - depth buffer, clearing the, 172
- glClearAccum(), 32, 379
- glClearColor(), 30, 31, 379
- glClearDepth(), 31, 379
- glClearIndex(), 32, 165, 379
 - fog, use with, 245
- glClearStencil(), 32, 379
- glClipPlane(), 137
- glColor*(), 33, 163
 - legal between glBegin() and glEnd(), 46
- glColorMask(), 379, 381
- glColorMaterial(), 201
 - performance tips, 602
- glColorPointer(), 68

glCopyPixels(), 290, 295
 alternative uses, 527
 dithering, turn off, 600
 feedback mode, 493
glReadBuffer() effect, 380
 imaging pipeline operations, 296
 pixel-transfer modes effect, 302

glCopyTexImage1D(), 336
glReadBuffer() effect, 380
 pixel-transfer modes effect, 302

glCopyTexImage2D(), 329
glReadBuffer() effect, 380
 pixel-transfer modes effect, 302

glCopyTexSubImage1D(), 337, 337
glReadBuffer() effect, 380
 pixel-transfer modes effect, 302

glCopyTexSubImage2D(), 335
glReadBuffer() effect, 380
 pixel-transfer modes effect, 302

glCullFace(), 57

glDeleteLists(), 267, 286

glDeleteTextures(), 351

glDepthFunc(), 391

glDepthMask(), 381
 blending opaque and translucent objects, 223

glDepthRange(), 128
gluUnProject(), relationship to, 147

glDisable(), 10, 48

glDisableClientState(), 68

glDrawArrays(), 74

glDrawBuffer(), 295, 380

glDrawElements(), 72

glDrawPixels(), 290, 294, 387, 524
 alternative uses, 527
 feedback mode, 493
 pixel-storage modes effect, 299
 pixel-transfer modes effect, 302

glEdgeFlag*(), 63
 legal between **glBegin()** and **glEnd()**, 46

glEdgeFlagPointer(), 68

glEnable(), 48, 178
 also see enabling

glEnableClientState(), 46, 67

glEnd(), 42, 43, 414
 restrictions, 45

glEndList(), 256, 259, 263

glEvalCoord*(), 445, 447
 legal between **glBegin()** and **glEnd()**, 46
 used instead of **glVertex*()**, 440, 443

glEvalMesh*(), 445, 449

glEvalPoint*()
 legal between **glBegin()** and **glEnd()**, 46

glFeedbackBuffer(), 492
glRenderMode(), use with, 472

glFinish(), 35

glFlush(), 34, 35, 600

glFog*(), 243

glFrontFace(), 56

glFrustum(), 101, 121, 121, 533

glGenLists(), 256, 262
 fonts, use for, 286

glGenTextures(), 326, 347

glGetBooleanv(), 10, 49, 537
 double-buffering support, querying, 377
 stereo support, querying, 377

glGetClipPlane(), 536

glGetDoublev(), 10, 49, 537

glGetError(), 10, 502, 536

glGetFloatv(), 10, 49, 537
 line width attributes, obtaining, 51
 point size attributes, obtaining, 50

glGetIntegerv(), 10, 49, 537
 alpha test information, obtaining, 384
 attribute stack depth, obtaining, 79
 clipping planes, obtaining number of
 additional, 137
 color resolution, obtaining, 156
 display list nesting limit, obtaining, 266
 matrix stack depth, obtaining, 135
 maximum texture size, obtaining, 330
 name stack depth, obtaining, 473
 pixel map information, obtaining, 304
 rendering mode, obtaining current, 472
 stencil-related values, obtaining, 386

glGetLight*(), 10, 536

glGetMap*(), 536

glGetMaterial*(), 536
glGetPixelMap*(), 536
glGetPointerv(), 10, 49, 537
glGetPolygonStipple(), 10, 536
glGetString(), 503, 536
glGetTexEnv*(), 536
glGetTexGen*(), 536
glGetTexImage(), 536
 pixel-storage modes effect, 299
 pixel-transfer modes effect, 302
glGetTexLevelParameter*(), 331, 536
glGetTexParameter*(), 536
 texture residency, obtaining, 351
glHint(), 228
 fog use, 240
 texture use, 326
glIndex*(), 164
 fog, use with, 245
 legal between **glBegin()** and **glEnd()**, 46
glIndexMask(), 379, 381
glIndexPointer(), 68
glInitNames(), 471, 472, 473
glInterleavedArrays(), 76
glIsEnabled(), 10, 48, 537
glIsList(), 267
glIsTexture(), 347
glLight*(), 178, 180, 181, 186
glLightModel*(), 193
glLineStipple(), 51
glLineWidth(), 50
glListBase(), 267
 fonts, use for, 286
 sample program, 287
glLoadIdentity(), 101, 103, 112, 533
 performance tips, 602
 viewing transformations, use before, 99
glLoadMatrix*(), 102, 104, 104, 533
glLoadName(), 472, 474
glLogicOp(), 394
glMap*(), 442, 443, 446
glMapGrid*(), 445, 449
glMaterial*(), 179, 196
 legal between **glBegin()** and **glEnd()**, 46
 performance tips, 602
glMatrixMode(), 101, 103
 use with matrix stacks, 133
glMultMatrix*(), 102, 104, 533
 performance tips, 602
glNewList(), 256, 259, 262
glNormal*(), 64
 legal between **glBegin()** and **glEnd()**, 46
glNormalPointer(), 68
glOrtho(), 124, 533
 picking matrix use, 479
glPassThrough(), 492, 494
glPixelMap*(), 304
glPixelStore*(), 299
 cannot be stored in display lists, 264
 polygon stippling, 58
 texture image data, effect on, 328, 329, 332, 335, 336
glPixelTransfer*(), 302, 524
 texture image data, effect on, 328, 329, 332, 335, 336
glPixelZoom(), 305, 509
glPointSize(), 50
glPolygonMode(), 56
 antialiasing, effect on, 235
 polygon offset, use with, 247
glPolygonOffset(), 248
glPolygonStipple(), 58
 pixel-storage modes effect, 299
glPopAttrib(), 10, 79, 274, 537
glPopClientAttrib(), 10, 81, 537
glPopMatrix(), 133, 143, 189, 274
 restore orientation of coordinate systems, 146
 selection, use with, 471
glPopName(), 472, 473
glPrioritizeTextures(), 353
glPushAttrib(), 10, 79, 274, 537
glPushClientAttrib(), 10, 81, 537
glPushMatrix(), 133, 143, 189, 274
 save orientation of coordinate systems, 146
 selection, use with, 471

glPushName(), 471, 472, 473
glRasterPos*(), 279, 282
 images, for positioning, 290
 selection hits, can cause, 474
glReadBuffer(), 295, 380
glReadPixels(), 290, 292
 glReadBuffer() effect, 380
 pixel-storage modes effect, 299
 pixel-transfer modes effect, 302
glRect*(), 40
glRenderMode(), 471, 472, 474, 492
glRotate*(), 109, 140, 143, 533
 performance tips, 602
glScale*(), 99, 110, 143, 533
 performance tips, 602
glScissor(), 383
glSelectBuffer(), 471, 472
 display lists, cannot be stored in, 264
glShadeModel(), 165
glStencilFunc(), 385
glStencilMask(), 381
glStencilOp(), 386
glTexCoord*(), 326, 358
 legal between glBegin() and glEnd(), 46
glTexCoordPointer(), 68
glTexEnv*(), 326, 354
glTexGen*(), 364
 environment mapping, 370
glTexImage1D(), 335
 pixel-storage modes effect, 299
 pixel-transfer modes effect, 302
glTexImage2D(), 326, 327
 pixel-storage modes effect, 299
 pixel-transfer modes effect, 302
 specifying mipmaps, 339
glTexParameter*(), 326, 363
 specifying filtering methods, 345
glTexSubImage1D(), 336
 pixel-storage modes effect, 299
 pixel-transfer modes effect, 302
glTexSubImage2D(), 332
 pixel-storage modes effect, 299
 pixel-transfer modes effect, 302
glTranslate*(), 108, 140, 143, 533
 performance tips, 602
GLU, 2, 14, 410
 drawing spheres, cylinders, and disks, 428–436
 error string description, 503
 obsolete routines
 gluBeginPolygon(), 427
 gluEndPolygon(), 427
 gluNextContour(), 427
 quadrics, 428–436
 tessellation, 39, 410–428
 version numbers, obtaining, 504
gluBeginCurve(), 455, 464
gluBeginSurface(), 455, 463
gluBeginTrim(), 465
gluCylinder(), 429, 431
gluDeleteNurbsRenderer(), 460
gluDeleteQuadric(), 429, 429
gluDeleteTess(), 426, 427
gluDisk(), 429, 432
gluEndCurve(), 455, 464
gluEndSurface(), 455, 463
gluEndTrim(), 465
gluErrorString(), 429, 462, 503
 polygon tessellation, 414
gluGetNurbsProperty(), 461, 536
gluGetString(), 505, 536
gluGetTessProperty(), 422, 536
gluLoadSamplingMatrices(), 461
gluLookAt(), 97, 99, 116, 140
gluNewNurbsRenderer(), 455, 459
gluNewQuadric(), 428, 429
gluNewTess(), 412, 427
gluNurbsCallback(), 455, 462
gluNurbsCurve(), 455, 464
gluNurbsProperty(), 455, 460
gluNurbsSurface(), 455, 463
gluOrtho2D(), 125, 601
 resized windows, use with, 36
gluPartialDisk(), 429, 432
gluPerspective(), 101, 123, 140
 picking matrix use, 479

gluPickMatrix(), 479
 gluProject(), 150
 gluPwlCurve(), 465
 gluQuadricCallback(), 429, 429
 gluQuadricDrawStyle(), 428, 430
 gluQuadricNormals(), 428, 430
 gluQuadricOrientation(), 428, 430
 gluQuadricTexture(), 429, 431
 gluScaleImage(), 329
 gluSphere(), 429, 431
 GLUT, 15, 579–585

- basic functions, 16–20
- event management, 19
- glutCreateWindow(), 17, 581
- glutDisplayFunc(), 17, 581
- glutIdleFunc(), 20, 584
- glutInit(), 16, 580
- glutInitDisplayMode(), 16, 580
- glutInitWindowPosition(), 17, 581
- glutInitWindowSize(), 17, 581
- glutKeyboardFunc(), 19, 582
- glutMainLoop(), 17, 585
- glutMotionFunc(), 19, 582
- glutMouseFunc(), 19, 582
- glutPostRedisplay(), 17, 256, 582
- glutReshapeFunc(), 19, 582
 - simple example, 35
- glutSetColor(), 16, 165, 210, 583
 - smooth shading, use for, 167
- glutSolidCone(), 584
- glutSolidCube(), 20, 583
- glutSolidDodecahedron(), 584
- glutSolidIcosahedron(), 584
- glutSolidOctahedron(), 584
- glutSolidSphere(), 20, 583
- glutSolidTeapot(), 584
- glutSolidTetrahedron(), 584
- glutSolidTorus(), 583
- glutSwapBuffers(), 23
- glutWireCone(), 584
- glutWireCube(), 20, 583
- glutWireDodecahedron(), 584
- glutWireIcosahedron(), 584
- glutWireOctahedron(), 584
- glutWireSphere(), 20, 140, 583
- glutWireTeapot(), 584
- glutWireTetrahedron(), 584
- glutWireTorus(), 583
 - sample program introducing GLUT, 18
 - window management, 16, 35
- gluTessBeginContour(), 423
- gluTessBeginPolygon(), 423
- gluTessCallback(), 412, 423, 427
- gluTessEndContour(), 423
- gluTessEndPolygon(), 423
- gluTessNormal(), 422, 422, 426
- gluTessProperty(), 417, 423
- gluTessVertex(), 423, 427
- gluUnProject(), 147, 150
- glVertex*(), 41
 - legal between glBegin() and glEnd(), 46
 - using glEvalCoord*() instead, 440
- glVertexPointer(), 46, 68
- glViewport(), 102, 126
 - using with resized windows, 36
- GLX, 14, 562
 - ftp site for GLX specification, 562
 - glXChooseVisual(), 563, 604
 - glXCopyContext(), 563
 - glXCreateContext(), 563
 - glXCreateGLXPixmap(), 563
 - glXDestroyContext(), 563
 - glXDestroyGLXPixmap(), 563
 - glXGetClientString(), 562
 - glXGetConfig(), 376, 563
 - glXGetCurrentContext(), 563
 - glXGetCurrentDisplay(), 563
 - glXGetCurrentDrawable(), 563
 - glXIsDirect(), 563
 - glXMakeCurrent(), 563
 - glXQueryExtension(), 562
 - glXQueryExtensionsString(), 562
 - glXQueryServerString(), 562
 - glXQueryVersion(), 562
 - glXSwapBuffers(), 23, 564
 - glXUseXFont(), 564
 - glXWaitGL(), 564
 - performance tips, 603

glXWaitX(), 564
performance tips, 603
Gouraud shading, See smooth shading

H

Haeberli, Paul, 372, 394
haze, See fog
header file, 15
hidden-line removal, 521
polygon offset used for, 247
hidden-surface removal, 171–173, 391
hierarchical models, 132, 265
picking, 483–485
highlights, See specular
hints, 228
fog, 240
perspective correction, 228, 326
hits (selection), See selection (hit records)
holes in polygons, 38, 518
homogeneous coordinates, 37, 594
Hoschek, Josef, 439
Hughes, John F., xxi, 593

I

IBM OS/2 Presentation Manager to OpenGL
Interface, see PGL
icosahedron, drawing, 83
identity matrix, 99, 103, 112, 602
illumination, See lighting
images, 278, 289–295
also see pixel data
blending, 514
compositing, 215
distorted, 509
imaging pipeline, 291, 296–308
interpolating between, 514
magnifying or reducing, 305
nonrectangular, 219
projecting, 523

sample code which draws an image, 294
sample program which draws, copies, and
zooms an image, 306
scaling and rotating, 523
sources of, 290
superimposing, 515
transposing, 528
warping, 523
imaging pipeline, See images (imaging pipeline)
immediate mode, 29, 252
display lists, mixing with, 265
infinite light source, 182
input events
handling, using GLUT, 19
intensity
texture image data type, 354
*Interactive Inspection of Solids: Cross-sections and
Interferences*, 518
interference regions, 518
interleaved arrays, 75
interpolating
color values and texture coordinates, 228, 357
invariance
of an OpenGL implementation, 600, 605
Inventor, see Open Inventor

J

jaggies, 226
jittering, 396, 401, 407
accFrustum() routine, 396
accPerspective() routine, 396
sample code to jitter projection transformations,
397
sample program with orthographic projection,
401

K

Kilgard, Mark, xxii, 15, 562, 579
Korobkin, Carl, 372

L

Lasser, Dieter, 439
layers, drawing, 511
Life, Game of, 526
light sources, 180–192
 ambient light, 174, 182
 contribution to lighting equation, 206
 diffuse light, 174, 182
 directional, 182
 display lists cache values, 258
 infinite light source, 182
 local light source, 182
 maximum number of sources, 178
 moving along with the viewpoint, 191
 moving light sources, 187–192
 multiple light sources, 186
 performance tips, 178
 positional, 182
 rendering pipeline stage, 12, 531
 RGBA values, 175
 sample program that moves the light source, 189
 specifying a light source, 178
 specular light, 174
 spotlights, 184–186
 stationary, 187
lighting
 also see light sources, material properties
 ambient light, 173
 approximation of the real world, 173
 attenuation, 183–184
 calculations in color-index mode, 210
 color-index mode, 209–211
 default values, using, 179
 display lists cache values, 258
 enabling, 178, 179
 enabling and disabling, 195
 equation that calculates lighting, 205
 global ambient light, 193, 206
 lighting model, 192–195
 lighting model, specifying a, 179
 rendering pipeline stage, 12, 531
 sample program introducing lighting, 176
 steps to perform, 176

 two-sided materials, 194
 viewer, local or infinite, 194
line segment, 38
linear attenuation, 184
lines, 38
 antialiasing, 228–235, 523
 connected closed loop, specifying, 43, 44
 connected strip, specifying, 43, 44
 feedback mode, 493
 querying line width, 51
 sample program with wide, stippled lines, 53
 specifying, 43, 44
 stippling, 51
 tessellated polygons decomposed into, 414
 width, 50
local light source, 182
logical operations
 rendering pipeline stage, 14, 533
 transposing images, using for, 528
lookup table, See color map
luminance, 292, 314
 pixel data formats for, 293, 298
 texture image data type, 354

M

magnifying images, 305
masking, 381
 antialiasing characters, 513
 layers, drawing, 511
 rendering pipeline stage, 14, 533
material properties, 179, 195–204
 ambient, 175, 197
 changing a single parameter with `glColorMaterial()`, 201
 changing material properties, 199
 diffuse, 175, 197
 display lists cache values, 258
 emission, 175, 198, 206
 enabling color material properties mode, 201
 performance when changing, 602
 rendering pipeline stage, 12, 531
 RGBA values, 176

-
- sample program which changes material properties, 199
 - sample program which uses `glColorMaterial()`, 202
 - shininess, 198
 - specular, 175, 198
 - two-sided lighting, 194
 - matrix
 - also see matrix stack
 - choosing which matrix is current, 103
 - column-major ordering, 104
 - current, 99
 - danger of extensive changes, 600
 - display lists cache matrix operations, 257
 - identity, 99, 103, 112, 602
 - loading, 104
 - modelview, 96, 103
 - multiplying matrices, 104
 - NURBS, specifying for sampling, 461
 - orthographic parallel projection, 598
 - perspective projection, 598
 - projection, 101, 103
 - rotation, 596
 - row-major ordering, 104
 - scaling, 596
 - texture, 371
 - transformation pipeline, 96
 - transformations of homogeneous coordinates, 594
 - translation, 596
 - matrix stack, 132–136
 - choosing which matrix stack is current, 133
 - current matrix stack, 533
 - modelview, 135
 - popping, 133
 - projection, 135
 - pushing, 133
 - querying stack depth, 135
 - texture, 371
 - Megahed, Abe, 518
 - Microsoft
 - Microsoft Win32, See Win32
 - Microsoft Windows, 14
 - Microsoft Windows 95, 574
 - Microsoft Windows NT, xxii
 - Microsoft Windows to OpenGL interface, See WGL
 - mipmapping, 338–344
 - minification filters, 346
 - texture objects for mipmaps, 350
 - mirroring objects, See scaling
 - modeling transformations, 99, 104, 108–113
 - camera analogy, 94
 - connection to viewing transformations, 100
 - example, 111
 - rotation, 109
 - rotation matrix, 596
 - sample program, 112
 - scaling, 110
 - scaling matrix, 596
 - translation, 108
 - translation matrix, 596
 - models
 - rendering wireframe and solid, 20, 583
 - modelview matrix, 96, 103
 - arbitrary clipping planes, effect on, 137
 - stack, 135
 - motion blur, 402
 - stippling, with, 507
 - motion, See animation
 - movie clips, 527
 - multiple layers
 - displaying with overlap, 511
- ## N
- name stack, 471–475
 - creating, 472
 - initializing, 472
 - loading, 472
 - multiple names, 483–485
 - popping, 472
 - pushing, 472
 - querying maximum depth, 473
 - networked operation, 34–35
 - attribute groups, saving and restoring, 78
 - display lists, 264
 - versions, 504
 - Non-Uniform Rational B-Splines, see NURBS

nonplanar polygons, 39

normal vectors, 63–65, 178

- calculating, 588
- calculating for analytic surfaces, 588
- calculating for polygonal data, 591
- calculating length, 65
- cross product, calculating normalized, 85
- enabling automatic unit length division, 65
- inverse matrix generated, 533
- matrix transformations, 96
- normalized, 65
- NURBS, generating for, 463
- quadrics, generated for, 430
- rendering pipeline stage, 12, 531
- specifying, 64
- tessellation, specifying for, 415
- transformations, 595
- unit length optimizes performance, 603
- vertex arrays, specifying values with, 68

normal, See normal vectors

normalized device coordinates, 96

NURB Curves and Surfaces (book title), 439

NURBS, 455–468

- creating a NURBS curve or surface, 462–464
- creating a NURBS object, 459
- culling, 460
- deleting a NURBS object, 460
- display list use, 256
- error handling, 462
- method of display (lines or filled polygons), 460
- normal vectors, generating, 463
- properties, controlling NURBS, 460
- querying property value, 461
- references, 439
- sample program which draws a lit NURBS surface, 456
- sample program with a trimmed surface, 467
- sampling precision, 460
- source for matrices, 461
- steps to use, 455
- texture coordinate generation, 463
- trimming, 464–468

NURBS Book, The, 439

NURBS for Curve and Surface Design, 439

O

object coordinates, 96

- texture coordinate generation, 364

objects, See models

opacity, 215

Open Inventor, 3, 15

OpenGL Extension to the X Window System, see GLX

OpenGL Programming for the X Window System, xxii, 15, 16, 562, 579

OpenGL Reference Manual, xxi, 529, 536, 562

OpenGL Utility Library, see GLU

OpenGL Utility Toolkit, see GLUT

orthographic parallel projection, 101, 124–125

- jittering, 400
- matrix, 598
- specifying with integer coordinates, 601

outlined polygons, 55, 63

- polygon offset solution, 247

overlapping objects, 518

P

painting, 215, 218, 528

partial disks, 428

pass-through markers, 494

performance tips

- clearing the window, 32
- display lists, 256, 257
- flat shading, 603
- flushing the pipeline, 34
- fog, 240
- GLX tips, 603
- hints, 228
- light source attenuation, effect of, 184
- light sources, effect of additional, 178
- list of general tips, 602
- material properties, changing, 602
- NURBS and display lists, 256
- pixel data alignment, 301
- pixel data, drawing, 314
- polygon restrictions, 39

perspective projection matrix, 598
shadows created with, 520

projection transformations, 100, 120–125
camera lens analogy, 94
collapsing geometry to a single plane, 600
jittering, 397, 400
orthographic parallel, 101, 124–125, 601
perspective, 120–123
picking, 479
texturing effects, 372
two-dimensional, 125

proxy textures, 330

Q

q texture coordinates, 372
avoiding negative values, 601

quadratic attenuation, 184

quadrics, 428–436
creating an object, 429
destroying an object, 429
drawing as points, lines, and filled polygons, 430
error handling, 429
normal vectors, generating, 430
orientation, 430
quadratic equation, 428
sample program, 433
steps to use, 428
texture coordinates, generating, 431

quadrilateral
specifying, 43, 45
strip, specifying, 43, 45

R

raster position, 282
after drawing a bitmap, 283
current, 282, 533
current raster color, 285
current, obtaining the, 282
transformation of, 282

rasterization, 156, 374

exact, two-dimensional, 601
rendering pipeline stage, 13

reading pixel data, See pixel data

Real Projective Plane, The, 593

rectangles
specifying, 40

reducing images, 305

reflecting objects, See scaling

reflection, See material properties

reflective objects, See environment mapping

refresh, screen, 21

removing hidden surfaces, See hidden-surface removal

repeatability, 606

resident textures, 332, 351
management strategies, 352
querying residence status, 351

RGBA mode, 157
changing between color-index mode and, 162
choosing between color-index mode and, 161
coverage calculations for antialiasing, 227
data type conversion, 163
light source colors, 175
lighting calculations in, 205
material property values, 176
vertex arrays, specifying values with, 68

robot arm example, 143–146

Rogers, David, 428

Rossignac, Jarek, 518

rotating images, 523

rotation, 109
matrix, 596

S

sample programs, See programs

scaling, 110
matrix, 596

scaling images, 523

Schneider, Bengt-Olaf, 518

Scientific American, 526

scissor test, 383

and clearing, 379
 rendering pipeline stage, 14, 533
 Segal, Mark, 372
 selection, 470–491
 back buffer for, using the, 508
 hit records, 474
 programming tips, 489
 querying current rendering mode, 472
 rendering pipeline stage, 532
 sample program, 475
 steps to perform, 471
 sweep selection, 490
 shading
 flat, 165
 performance tips, 603
 sample program with smooth shading, 166
 smooth, 165
 specifying shading model, 165
 shadows, 205, 406, 520
 shininess, 198
 also see environment mapping
 silhouette edges, 82
 smoke, See fog
 smooth shading, 165
 solar system example, 140–143
 source factor, See blending
 specular
 contribution to lighting equation, 208
 light, 174
 material properties, 175, 198
 spheres, 428, 583
 split-screen
 multiple viewports, 126
 spotlights, See light sources
 state machine, 9–10
 state variables, 48
 attribute groups, 78–81
 display list execution, effect of, 273
 enable and disable states, 48
 list of, 537–559
 performance of storing and restoring, 602
 querying, 49
 stencil buffer, 376, 377
 clearing, 32, 379
 concave polygons, for drawing, 516
 decals, for, 515
 Dirichlet domains, for, 525
 Game of Life, for the, 526
 hidden-line removal, 522
 masking, 381
 pixel data, 293, 303
 stencil test, 385–391
 examples of using, 387
 interference regions found using clipping
 planes, 519
 querying stencil parameters, 386
 rendering pipeline stage, 14, 533
 sample program, 387
 stereo, 377, 380
 querying its presence, 377
 stippling
 display lists cache stipple patterns, 258
 enabling line stippling, 51
 enabling polygon stippling, 58
 fade effect, use for, 507
 line pattern reset, 52, 493, 497
 lines, 51
 polygons, 57
 sample program with line stipple, 53
 sample program with polygon stippling, 60
 stencil test, use of, 391
 translucency, use to simulate, 506
 stitching, 247
 stretching objects, See scaling
 stride
 vertex arrays, 70, 76
 subdivision, 81–89
 generalized, 88
 icosahedron example, 86
 recursive, 88
 subimages, 332–335, 336
 superimposing images, 515
 surface normals, See normal vectors
 surfaces, See evaluators or NURBS
 swapping buffers, See double-buffering
 syntax, See command syntax

T

Terminator 2, 370

tessellation, 39, 410–428

- backward compatibility with obsolete routines, 426

- begin and end callback routines, 414

- callback routines, 412–417

- combine callback routine, 414, 416

- computational solid geometry, winding rules used for, 421

- contours, specifying, 423

- converting code to use the GLU 1.2 tessellator, 427

- creating an object, 412

- decomposition into geometric primitives, 414

- deleting objects, 426

- display list use, 256

- edge flag generation, 414

- error handling, 414

- evaluators used to perform, 602

- interior and exterior, determining, 418–422

- intersecting contours combined, 414, 416

- performance tips, 426

- polygons, specifying, 423

- properties, 417–422

- reuse of objects, 412, 426

- reversing winding direction, 422

- sample code, 414, 415, 424

- user-specified data, 417

- vertices, specifying, 415, 423

- winding rules, 418–422

texels, 14, 318

text, see characters

texture coordinates, 326, 357–371

- assigning manually, 357

- avoiding negative *q* values, 601

- clamping, 360–363

- computing manually, 358

- enabling automatic generation of, 369

- environment mapping, automatic generation for, 370

- evaluators, generated by, 452

- generating automatically, 364–371

- NURBS, generating for, 463

- q* coordinate, 372

- quadrics, generated for, 431

- reference planes, specifying, 364

- rendering pipeline stage, 12, 531

- repeating, 360–363

- sample program with texture coordinate generation, 365

- tessellation, specifying for, 415

- vertex arrays, specifying values with, 68

- wrapping modes, 360–363

texture functions, 354–357

- blend, 356

- blending color, 356

- decal, 326, 356

- fragment operations, 356

- modulate, 356

- pixel-transfer modes effect, 354

- replace, 356

- texture internal format, interaction with, 354

texture images

- alpha data, 354

- borders, 337, 361

- components, 327

- data types, 328

- distorting, 359

- framebuffer as a source of, 329, 334, 336

- imaging pipeline operations, 297

- intensity data, 354

- internal format, 327

- luminance data, 354

- mipmaps, 338–344

- one-dimensional, 335–337

- performance affected by internal format, 328

- performance of texture subimages, 602

- power of 2 size restriction, 329

- proxy textures, 330

- querying maximum size, 330

- residence status, 351

- resident textures, 332, 351

- resident textures, management strategies of, 352

- sample program with mipmaps, 340

- sample program with subimages, 333

- specifying, 326–338

- subimages, 332–335, 336

- working set of textures, 332, 346, 351

texture mapping, see texturing

- texture matrix, 371
 - rendering pipeline stage, 531
- texture objects, 326, 346–351
 - binding, 348
 - creating, 348
 - data which can be stored in, 348
 - deleting, 351
 - fragmentation of texture memory, 353
 - least-recently used (LRU) strategy, 353
 - mipmaps, 350
 - naming, 347
 - performance tips, 346, 602
 - priority, 352
 - rendering pipeline, 13, 532
 - sample program, 323
 - sample program with multiple texture objects, 348
 - sharing among rendering contexts, 563, 575
 - steps to perform, 346
 - using, 348
- texturing
 - also see texture coordinates, texture functions, texture images, texture matrix, and texture objects
 - antialiasing characters, 523
 - antialiasing lines, 523
 - blending, 219
 - border colors, treatment of, 361
 - color-index mode limitations, 321, 329
 - creating contours, 365
 - decals with alpha testing, 385
 - display lists cache texture data, 258
 - enabling, 322, 326
 - environment mapping, 369
 - filtering, 344–346
 - image transformations, 523
 - mipmapping, 338–344, 346
 - perspective correction hint, 326
 - rendering pipeline stage, 13, 532
 - sample program, 323
 - sample program with evaluated, Bézier surface, 452
 - sample program with mipmapping, 340
 - sample program with texture coordinate generation, 365
 - sample uses for, 523
 - simulating shadows or spotlights, 372
 - steps to perform, 321
 - what's new in release 1.1, 321
- 3D Computer Graphics: A User's Guide for Artists and Designers*, xxi
- 3D models, rendering, 20, 583
- Tiller, Wayne, 439
- tips, programming, 599
 - also see performance tips
 - error handling, 600
 - selection and picking, 489
 - transformations, 129
- transformations
 - also see modeling transformations, projection transformations, viewing transformations, and viewport transformations
 - combining multiple, 139–146
 - display lists cache transformations, 257
 - general-purpose commands, 102
 - matrices, 595–598
 - mimicking the geometric processing pipeline, 149
 - modeling, 104, 108–113
 - ordering correctly, 105–108
 - overview, 92
 - performance tips, 602
 - projection, 100, 120–125
 - reversing the geometric processing pipeline, 147
 - sample program, 98
 - sample program combining modeling transformations, 141, 144
 - sample program for modeling transformations, 112
 - sample program showing reversal of transformation pipeline, 148
 - troubleshooting, 129–131
 - units, 123
 - viewing, 104, 113–118
 - viewport, 102, 125–128
- translation, 108
 - matrix, 596
- translucent objects, 215, 506
 - stencil test, creating with the, 390
- transparent objects, 215
 - creating with the alpha test, 385
- transposing images, 528

triangle
fan, specifying, 43, 45
specifying, 43, 45
strip, specifying, 43, 45
tessellated polygons decomposed into, 414

trimming
curves and curved surfaces, 464–468
sample program, 467

two-sided lighting, 194

U

up-vector, 99

Utility Library, OpenGL, see GLU

Utility Toolkit, OpenGL, see GLUT

V

van Dam, Andries, xxi, 157, 593

van Widenfelt, Rolf, 372

vendor-specific extensions, 505

versions, 503–505
GLU, 504

vertex, 37
also see vertex arrays
evaluators, generating with, 440
feedback mode, 493
per-vertex operations pipeline stage, 12, 531
specifying, 41
tessellation, specifying for, 415, 423
transformation pipeline, 96

vertex arrays, 65–77
dereference a list of array elements, 72
dereference a sequence of array elements, 74
dereference a single element, 71
disabling, 68
display list use, 264
enabling, 67
interleaved arrays, 75
interleaved arrays, specifying, 76
performance tips, 603
querying, 537
reuse of vertices, 74

sample program, 69
specifying data, 68
steps to use, 66
stride between data, 70, 76

video
fake, 527
flipping an image with `glPixelZoom()`, 306
textured images, 332

viewing
camera analogy, 94–95

viewing transformations, 99, 104, 113–118
connection to modeling transformations, 100
default position, 99
different methods, 118
pilot view, 119
polar view, 119
tripod analogy, 94
up-vector, 99

viewing volume, 121
clipping, 125, 136
jittering, 397, 400

viewpoint
lighting, for, 194

viewport transformations, 97, 102, 125–128
photograph analogy, 94
rendering pipeline stage, 12, 532

visual simulation
fog, use of, 239

Voronoi polygons, 524

W

w coordinates, 37, 96, 102
avoiding negative values, 601
lighting, use with, 183
perspective division, 128, 532

warping images, 523

Watt, Alan, 318

web sites
IBM OS/2 software and documentation, 570
Microsoft Developer Network, 574
Silicon Graphics' OpenGL, xxii
Template Graphics Software, 566

WGL, 14, 574

- wglCopyContext(), 575, 576
- wglCreateContext(), 574, 575, 576
- wglCreateLayerContext(), 575, 576
- wglDeleteContext(), 576
- wglDescribeLayerPlane(), 574, 576
- wglDestroyContext(), 575
- wglGetCurrentContext(), 575, 577
- wglGetCurrentDC(), 575, 577
- wglGetLayerPaletteEntries(), 576, 577
- wglMakeCurrent(), 575, 577
- wglRealizeLayerPalette(), 575, 577
- wglShareLists(), 575, 576
- wglSwapLayerBuffers(), 575, 577
- wglUseFontBitmaps(), 576, 577
- wglUseFontOutlines(), 576, 577

Williams, Lance, 338

Win32

- ChoosePixelFormat(), 574, 576
- CreateDIBitmap(), 575, 577
- CreateDIBSection(), 575, 577
- DeleteObject(), 575, 577
- DescribePixelFormat(), 574, 576
- GetVersion(), 574, 576
- GetVersionEx(), 574, 576
- SetPixelFormat(), 574, 576
- SwapBuffers(), 575, 577

winding rules, 418–422

- computational solid geometry, used for, 421
- reversing winding direction, 422

window coordinates, 97, 126

- feedback mode, 493
- polygon offset, 248

window management

- glViewport() called, when window resized, 126
- using GLUT, 16, 35

working set of textures, 332, 346, 351

- fragmentation of texture memory, 353

writemask, See masking (buffers)

writing pixel data, See pixel data (drawing)

X

X Window System, 14, 562

- client-server rendering, 5
- minimum framebuffer configuration, 376
- X Visual, 162, 562

Z

z buffer, See depth buffer

z coordinates, See depth coordinates

zooming images, 305

- filtered, 528

T385

.N435 *OpenGL® programming guide*

1996 *The official guide to learning
OpenGL*

OpenGL[®] Programming Guide Second Edition

The Official Guide to Learning OpenGL, Version 1.1

OpenGL is a powerful software interface for graphics hardware that allows graphics programmers to produce high-quality color images of 3D objects. The functions in the OpenGL library enable programmers to build geometric models, view models interactively in 3D space, control color and lighting, manipulate pixels, and perform such tasks as alpha blending, antialiasing, creating atmospheric effects, and texture mapping.

The *OpenGL Programming Guide, Second Edition*, shows how to create graphics programs, many of which highlight features of the latest OpenGL release. Assuming users have a background in C programming, the book discusses the architecture and functions of OpenGL, Version 1.1.

The second edition contains the following additions and improvements:


- coverage of the new features of OpenGL, Version 1.1, including all texturing changes, vertex arrays, polygon offset, and RGBA logical operations
- the incorporation of the OpenGL Utility Toolkit, GLUT, in all programming examples
- an overview of the OpenGL rendering pipeline and state machine
- enhanced coverage of polygon tessellation, quadric surfaces, pixel operations, and error handling
- more performance tips
- a greatly expanded index

The OpenGL Technical Library provides tutorial and reference books for OpenGL. The library enables programmers to gain a practical understanding of OpenGL and show them how to unlock its full potential.

The OpenGL Technical Library is developed under the auspices of the Architecture Review Board (ARB), an industry consortium responsible for guiding the evolution of OpenGL and related technologies. The OpenGL ARB is composed of industry leaders such as Digital Equipment Corporation, Evans & Sutherland, Hewlett-Packard, IBM, Intel, Intergraph, Microsoft, Sun Microsystems, and Silicon Graphics.

The *OpenGL Programming Guide, Second Edition*, was written by **Mason Woo, Jackie Neider, and Tom Davis.**

Cover design by Jean Seal
Cover image by

 **ADDISON-W**
Addison-Wesley
is an imprint of
Find A-W Devel-
Web at <http://w>



OPENGL PROG GD 2EB
NEIDER, J
03GRAP

90000

ADDISON

Aug/97 169

-2 1

0-201-46138-2

\$40.95

* AP