The X Window System

ROBERT W. SCHEIFLER
MIT Laboratory for Computer Science
and
JIM GETTYS
Digital Equipment Corporation and MIT Project Athena

An overview of the X Window System is presented, focusing on the system substrate and the low-level facilities provided to build applications and to manage the desktop. The system provides high-performance, high-level, device-independent graphics. A hierarchy of resizable, overlapping windows allows a wide variety of application and user interfaces to be built easily. Network-transparent access to the display provides an important degree of functional separation, without significantly affecting performance, which is crucial to building applications for a distributed environment. To a reasonable extent, desktop management can be custom-tailored to individual environments, without modifying the base system and typically without affecting applications.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.4.4 [**Operating Systems**]: Communication Management—*network communication*; *terminal management*; H.1.2 [**Models and Principles**]: User/Machine Systems—*human factors*; I.3.2 [**Computer Graphics**]: Graphics Systems—*distributed/network graphics*; I.3.4 [**Computer Graphics**]: Graphics Utilities—*graphics packages*; *software support*; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*device independence*; *interaction techniques*

General Terms: Design, Experimentation, Human Factors, Standardization

Additional Key Words and Phrases: Virtual terminals, window managers, window systems

1. INTRODUCTION

The X Window System (or simply X) developed at MIT has achieved fairly widespread popularity recently, particularly in the UNIX[1] community. In this paper we present an overview of X, focusing on the system substrate and the low-level facilities provided to build applications and to manage the desktop. In X, this base window system provides high-performance graphics to a hierarchy of resizable windows. Rather than mandate a particular user interface, X provides primitives to support several policies and styles. Unlike most window systems, the base system in X is defined by a *network protocol*: asynchronous

[1]UNIX is a trademark of AT&T Bell Laboratories.

Authors' addresses: R. W. Scheifler, 545 Technology Square, Cambridge, MA 02139; J. Gettys, Project Athena, MIT, Cambridge, MA 02139.

stream-based interprocess communication replaces the traditional procedure call or kernel call interface. An application can utilize windows on any display in a network in a device-independent, network-transparent fashion. Interposing a network connection greatly enhances the utility of the window system, without significantly affecting performance. The performance of existing X implementations is comparable to that of contemporary window systems and, in general, is limited by display hardware rather than network communication. For example, 19,500 characters per second and 3500 short vectors per second are possible on Digital Equipment Corporation's VAXStation-II/GPX, both locally and over a local-area network, and these figures are very close to the limits of the display hardware.

X is the result of two separate groups at MIT having a simultaneous need for a window system. In the summer of 1984, the Argus system [16] at the Laboratory for Computer Science needed a debugging environment for multiprocess distributed applications, and a window system seemed the only viable solution. Project Athena [4] was faced with dozens, and eventually thousands, of workstations with bitmap displays and needed a window system to make the displays useful. Both groups were starting with the Digital VS100 display [14] and VAX hardware, but it was clear at the outset that other architectures and displays had to be supported. In particular, IBM workstations with bitmap displays of unknown type were expected eventually within Project Athena. Portability was therefore a goal from the start. Although all of the initial implementation work was for Berkeley UNIX, it was clear that the network protocol should not depend on aspects of the operating system.

The name X derives from the lineage of the system. At Stanford University, Paul Asente and Brian Reid had begun work on the W window system [3] as an alternative to VGTS [13, 22] for the V system [5]. Both VGTS and W allow network-transparent access to the display, using the synchronous V communication mechanism. Both systems provide "text" windows for ASCII terminal emulation. VGTS provides graphics windows driven by fairly high-level object definitions from a structured display file; W provides graphics windows based on a simple display-list mechanism, with limited functionality. We acquired a UNIX-based version of W for the VS100 (with synchronous communication over TCP [24] produced by Asente and Chris Kent at Digital's Western Research Laboratory. From just a few days of experimentation, it was clear that a network-transparent hierarchical window system was desirable, but that restricting the system to any fixed set of application-specific modes was completely inadequate. It was also clear that, although synchronous communication was perhaps acceptable in the V system (owing to very fast networking primitives), it was completely inadequate in most other operating environments. X is our "reaction" to W. The X window hierarchy comes directly from W, although numerous systems have been built with hierarchy in at least some form [11, 15, 18, 28, 30, 32–36]. The asynchronous communication protocol used in X is a significant improvement over the synchronous protocol used in W, but is very similar to that used in Andrew [10, 20]. X differs from all of these systems in the degree to which both graphics functions and "system" functions are pushed back (across the network) as application functions, and in the ability to tailor desktop management transparently.

The next section presents several high-level requirements that we believe a window system must satisfy to be a viable standard in a network environment, and indicates where the design of X fails to meet some of these requirements. In Section 3 we describe the overall X system model and the effect of network-based communication on that model. Section 4 describes the structure of windows, and the primitives for manipulating that structure. Section 5 explains the color model used in X, and Section 6 presents the text and graphics facilities. Section 7 discusses the issues of window exposure and refresh, and their resolution in X. Section 8 deals with input event handling. In Section 9 we describe the mechanisms for desktop management.

This paper describes the version[2] of X that is currently in widespread use. The design of this version is inadequate in several respects. With our experience to date, and encouraged by the number of universities and manufacturers taking a serious interest in X, we have designed a new version that should satisfy a significantly wider community. Section 10 discusses a number of problems with the current X design and gives a general idea of what changes are contemplated.

2. REQUIREMENTS

A window system contains many interfaces. A *programming* interface is a library of routines and types provided in a programming language for interacting with the window system. Both low-level (e.g., line drawing) and high-level (e.g., menus) interfaces are typically provided. An *application* interface is the mechanical interaction with the user and the visual appearance that is specific to the application. A *management* interface is the mechanical interaction with the user, dealing with overall control of the desktop and the input devices. The management interface defines how applications are arranged and rearranged on the screen, and how the user switches between applications; an individual application interface defines how information is presented and manipulated within that application. The *user* interface is the sum total of all application and management interfaces.

Besides applications, we distinguish three major components of a window system. The *window manager*[3] implements the desktop portion of the management interface; it controls the size and placement of application windows, and also may control application window attributes, such as titles and borders. The *input manager* implements the remainder of the management interface; it controls which applications see input from which devices (e.g., keyboard and mouse). The *base window system* is the substrate on which applications, window managers, and input managers are built.

In this paper we are concerned with the base window system of X, with the facilities it provides to build applications and managers. The following requirements for the base window system crystallized during the design of X (a few were not formulated until late in the design process):

1. *The system should be implementable on a variety of displays.* The system should work with nearly any bitmap display and a variety of input devices. Our design focused on workstation-class display technology likely to be available in a

[2] Version 10.

[3] Some people use this term for what we call the base window system; that is not the meaning here.

university environment over the next few years. At one end of the spectrum is a simple frame buffer and monochrome monitor, driven directly by the host CPU with no additional hardware support. At the other end of the spectrum is a multiplane display with color monitor, driven by a high-performance graphics coprocessor. Input devices, such as keyboards, mice, tablets, joysticks, light pens, and touch screens, should be supported.

2. *Applications must be device independent.* There are several aspects to device independence. Most important, it must not be necessary to rewrite, recompile, or even relink an application for each new hardware display. Nearly as important, every graphics function defined by the system should work on virtually every supported display; the alternative, which is to use GKS-style inquire operations [12] to determine the set of implemented functions at run time, leads to tedious case analysis in every application and to inconsistent user interfaces. A third aspect of device independence is that, as far as possible, applications should not need dual control paths to work on both monochrome and color displays.

3. *The system must be network transparent.* An application running on one machine must be able to utilize a display on some other machine, nor should it be necessary for the two machines to have the same architecture or operating system.

There are numerous examples of why this is important: a compute-intensive VLSI design program executing on a mainframe, but displaying results on a workstation; an application distributed over several stand-alone processors, but interacting with a user at a workstation; a professor running a program on one workstation, presenting results simultaneously on all student workstations.

In a network environment, there are certain to be applications that must run on particular machines or architectures. Examples include proprietary software, applications depending on specific architectural properties, and programs manipulating large databases. Such applications still should be accessible to all users. In a truly heterogeneous environment, not all programming languages and programming systems are supported on all machines, and it is very undesirable to have to write an interactive front end in multiple languages in order to make the application generally available. With network-transparent access, this is not necessary; a single front end written in the same language as the application suffices.

One might think that remote display will be extremely infrequent, and that performance is therefore much less important than for local display. Experience at MIT, however, indicates that many users routinely make use of the remote display capabilities in X, and that the performance of remote display is quite important. The desktop display, although physically connected to a single computer, is used as a true *network virtual terminal*; indeed, the idea of an X server (see the next section) built into a Blit-like terminal [23] is an intriguing one.

4. *The system must support multiple applications displaying concurrently.* For example, it should be possible to display a clock with a sweep second hand in one window, while simultaneously editing a file in another window.

5. *The system should be capable of supporting many different application and management interfaces.* No single user interface is "best"; different communities have radically different ideas about user interfaces. Even within a single community, "experts" and "novices" place different demands on an interface. Instead of mandating a particular user interface, the base window system should support a wide range of interfaces.

To achieve this, the system must provide *hooks* (mechanism) rather than *religion* (policy). For example, since menu styles and semantics vary dramatically among different user interfaces, the base window system must provide primitives from which menus can be built, instead of just providing a fixed menu facility.

The system should be designed in such a way that it is possible to implement management policy in a way that is external to the base window system and external to applications. Applications should be largely independent of management policy and mechanism; applications should *react to* management decisions, rather than *direct* those decisions. For example, an application needs to be informed when one of its windows is resized and should react by reformatting the information displayed, but involvement of the application should not be required in order for the user to change the size. Making applications management independent, as well as device independent, facilitates the sharing of applications among diverse cultures.

6. *The system must support overlapping windows, including output to partially obscured windows.* This is in some sense a by-product of the previous requirement, but it is important enough to merit explicit statement. Not all user interfaces allow windows to overlap arbitrarily. However, even interfaces that do not allow application windows to overlap typically provide some form of pop-up menu that overlaps application windows. If such menus are built from windows, then support for overlapping windows must exist.

7. *The system should support a hierarchy of resizable windows, and an application should be able to use many windows at once.* Subwindows provide a clean, powerful mechanism for exporting much of the basic system machinery back to the application for direct use. Many applications make use of their own window-like abstractions; some even implement what is essentially another window system, nested within the "real" window system. It is important to support arbitrary levels of nesting. What is viewed as a single window at one abstraction level may well require multiple subwindows at a lower level. By providing a true window hierarchy, application windows can be implemented as true windows within the system, freeing the application from duplicating machinery such as clipping and input control.

8. *The system should provide high-performance, high-quality support for text, 2-D synthetic graphics, and imaging.* The base window system must provide "immediate" or "transparent" graphics: The application describes the image precisely, and the system does not attempt to second-guess the application. The use of high-level models, whereby the application describes *what* it wants in terms of fairly abstract objects and the system determines *how* best to render the

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.