# The X Window System

Robert W. Scheifler[1]
MIT Laboratory for Computer Science

Jim Gettys[2]
Digital Equipment Corporation
MIT Project Athena

July 1986
Revised October 1986[3]

## Abstract

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, that 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 [**Information Systems**]: User/Machine Systems - *human factors*; I.3.2 [**Computer Graphics**]: Graphic 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: window systems, window managers, virtual terminals

---

[1]545 Technology Square, Cambridge, MA 02139.

[2]Project Athena, MIT, Cambridge, MA 02139.

[3]To appear in Transactions on Graphics #63, Special Issue on User Interface Software, 

## 1. Introduction

The X Window System (or simply X

developed at MIT has achieved fairly widespread popularity recently, particularly in the Unix[4] 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 mandating 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 stream-based inter-process 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 contemporary window systems, and in general is limited by display hardware rather than network communication. For example, 19500 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 the simultaneous need for a window system from two separate groups at MIT. In the summer of 1984, the Argus system [13] at the Laboratory for Computer Science needed a debugging environment for multi-process distributed applications, and a window system seemed the only viable solution. Project Athena [3] 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 [11] and VAX hardware, but it was clear at the outset that other architectures and displays had to be supported. In particular, equal numbers of 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 [2], as an alternative to VGTS [10, 18] for the V system [4]. 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 [21]) done 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 (due 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 [8, 12, 25, 28, 30, 31, 32, 33, 29]. The asynchronous communication protocol used in X is a significant improvement over the synchronous protocol used in W,

---

[4]Unix is a trademark of AT&T Bell Laboratories.

but is very similar to that used in Andrew [7, 16]. 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 transparently tailor desktop management.

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[5] 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*[6] 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 on 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.

---

[5]Version 10.

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

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 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 multi-plane display with color monitor, driven by a high-performance graphics co-processor. 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 importantly, 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 [9] 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. The two machines should not have to have the same architecture or operating system.

   There are numerous examples of why this 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 therefore is 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 [20] 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. Rather than 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, rather than just providing a fixed menu facility.

The system should be designed in such a way that it is possible to implement management policy both 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 *directing* 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 between 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 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 image, cannot be imposed as the only form of graphics interface. Such models generally fail to provide adequate support for some important class of applications, and different user communities tend to have strong opinions about which model is "best". High-level models are extremely important to provide, but they should be built in layers on top of the base window system.

   Support for 3-D graphics is not listed as a requirement, but this is not to say it is unimportant. We simply have not considered 3-D graphics, due to lack of expertise and lack of time.

9. The system should be extensible.

   For example, the core system may not support 3-D graphics, but it should be possible to extend the system with such support. The extension mechanism should allow communities to extend the system non-cooperatively, yet allow such independent extensions to be merged gracefully.

We believe that a window system must satisfy these requirements to be a viable standard in an environment of high-performance workstations and mainframes connected via high-performance local area networks. X satisfies most of these requirements, but currently fails to satisfy a few due to practical considerations of staffing and time constraints: the design and much of the implementation of the base window system was to be handled solely by the first author; it was important to get a working system up fairly quickly; and the immediate applications only required relatively simple text and graphics support. As a result, X is not designed to handle high-end color displays or to deal with input devices other than a keyboard and mouse; some support for high-quality text and graphics is missing; X only provides support for one class of management policy; and no provision has been made for extensions. As discussed in Section 10, these and other problems are being addressed in a redesign of X.

## 3. System Model

The X window system is based on a client-server model; this model follows naturally from requirements two and three in the previous section. For each physical display, there is a controlling server. A client application and a server communicate over a reliable duplex (8-bit) byte stream. A simple block stream protocol is layered on top of the byte stream. If the client and server are on the same machine, the stream is typically based on a local inter-process communication (IPC) mechanism, and otherwise a network connection is established between the pair. Requiring nothing more than a reliable duplex byte stream (without urgent data) for communication makes X usable in many environments. For example, the X protocol can be used over TCP [21], DECnet [36], and Chaos [15].

Multiple clients can have connections open to a server simultaneously, and a client can have connections open to multiple servers simultaneously. The essential tasks of the server are to multiplex requests from clients to the display, and demultiplex keyboard and mouse input back to the appropriate clients. Typically, the server is implemented as a single sequential process, using round-robin scheduling among the clients, and this centralized control trivially solves many synchronization problems; however, a multi-process server has also been implemented. Although one might place the server in the kernel of the operating system in an attempt to increase performance, a user-level server process is vastly easier to debug and maintain, and performance under Unix in fact does not seem to suffer. Similar performance results have been obtained in Andrew [7]. Various tricks are used in both clients and server to optimize performance, principally by minimizing the number of operating system calls [6].

The server encapsulates the base window system. It provides the fundamental resources and mechanisms, and the hooks required to implement various user interfaces. All device dependencies are encapsulated by the server; the communication protocol between clients and the server is device independent. By placing all device dependencies on one end of a network connection, applications are truly device independent. The addition of a new display type simply requires the addition of a new server implementation; no application changes are required. Of course, the server itself is designed as device independent code layered on top of a device dependent core, so only the "back end" of the server need be reimplemented for each new display.[7]

---

[7]A back end has been implemented using a programming interface to X itself, such that a complete "recursive" X server executes inside a window of another X server.

**Figure 2-1:** System Structure

### 3.1. Network Considerations

It is extremely important for the server to be robust with respect to client failures. The server, and the network protocol, must be designed so that the server never trusts clients to provide correct data. As a corollary, the protocol must be designed in such a way that, if the server ever has to wait for a response from a client, it must be possible to continue servicing other clients. Without this property, a buggy client or a network failure could easily cause the entire display to freeze up.

Byte ordering is a standard problem in network communication: when a 16-bit or 32-bit quantity is transmitted over an 8-bit byte stream, is the most significant byte transmitted first (big-endian byte order) or is the least significant byte transmitted first (little-endian byte order)? Some machines with byte-addressable memory use big-endian order internally, and others use little-endian order. If a single order is chosen for network communication, some machines will suffer the overhead of swapping bytes, even when communicating with a machine using the same internal byte order. Such an approach also means that both parties in the communication must worry about byte order.

The X protocol uses a different approach. The server is designed to accept both big-endian and little-endian connections. For example, using TCP this is accomplished by having the server listen on two distinct ports; little-endian clients connect to the server on one port, and big-endian clients connect on the other. Clients always transmit and receive in their native byte order. The server alone is responsible for byte swapping, and byte swapping only occurs between dissimilar architectures. This eliminates the byte swapping overhead in the most common situations, and greatly simplifies the building of client-side interface libraries in various programming languages. X is not unique in its use of this trick; the current VGTS implementation uses the same trick, and similar protocol optimizations have been used in various network-based applications.

Another potential problem in protocol design is word alignment. In particular, some architectures require 16-bit quantities to be aligned on 16-bit boundaries and 32-bit quantities to be aligned on 32-bit boundaries in memory. To allow efficient implementations of the protocol across a spectrum of 16-bit and 32-bit architectures, the protocol is defined to consist of blocks that are always multiples of 32 bits, and each 16-bit and 32-bit quantity within a block is aligned on 16-bit and 32-bit boundaries, respectively.

X is designed to operate in an environment where the inter-process communication round-trip time is between 5 and 50 milliseconds, both for local and for network communication. We also assume that data transmission rates are comparable to display rates; for example, to transmit and display 5000 characters per second, a data rate of approximately 50Kb (kilobits per second) will be needed, and to transmit and display 20000 characters per second, a data rate of approximately 200Kb will be needed. Networks and protocol implementations with these characteristics are now quite commonplace. For example, workstations running Berkeley Unix, connected via 10Mb (megabits per second) local area networks, typically have round-trip times of 15 to 30 milliseconds, and data rates of 500Kb to 1Mb.

The round-trip time is important in determining the form of the communication protocol. The most common communication will be text and graphics requests sent from a client to the server. Examples of individual requests might be to draw a string of text or to draw a line. Such requests could be sent either synchronously, in which case the client sends a request only after receiving a reply from the server to the previous request, or they could be sent asynchronously, without the server generating any replies. However, since the requests are sent over a reliable stream, they are guaranteed to arrive, and arrive in order, so replies from the server to graphics requests serve no useful purpose. Moreover, with round-trip times over 5 milliseconds, output to the display must be asynchronous, or it will be impossible to drive high-speed displays adequately. For example, at 80 characters per request and a 25 millisecond round-trip time, only 3200 characters per second can be drawn synchronously, whereas many hardware devices are capable of displaying between 5000 and 30000 characters per second.

Similarly, polling the server for keyboard and mouse input would be unacceptable in many applications, particularly those written in sequential languages. For example, an application attempting to provide real-time response to input has to poll periodically for input during screen updates. For an application with a single thread of control, this effectively results in synchronous output, and consequent performance loss. Hence, input must be generated asynchronously by the server, so that applications need at most perform local polling.

The round-trip time is also important in determining what user interfaces can be supported without embedding them directly in the server. The most important concern is whether remote, application-level mouse tracking is feasible. By *tracking*, we do not mean maintaining the cursor image on the screen as the user moves the mouse; that function is performed autonomously by the X server, often directly in hardware. Rather, applications track the mouse by animating some other image on the screen in real time as the mouse moves. For round-trip times under 50 milliseconds, tracking is perfectly reasonable, driven either by motion events generated by the server or by continuous polling from the application. With a refresh occurring up to 30 times every second, remote tracking is demonstrably "instantaneous" with mouse motion.

For tracking to be effective, however, relatively little time can be spent updating the display at each movement, so typically only relatively small changes can be made to the screen while tracking. This is certainly the case for common operations, such as rubber banding window outlines and highlighting menu items. It might be argued that the ability to run application-specific code in the server is required for acceptable hand-eye coordination during complex tracking. For example, NeWS [27] provides such a mechanism in a novel way. However, we are not convinced there are sufficient benefits to justify such complexity. Complex tracking typically is bound up intimately with application-specific data structures and knowledge representations, and such information is used by the "back end" of the application as well as the "front end". In a distributed system it is folly to believe that applications will download large front ends into a server; communication round-trip times are a reality that cannot be escaped.

### 3.2. Resources

The basic resources provided by the server are windows, fonts, mouse cursors, and off-screen images; later sections describe each of these. Clients request creation of a resource by supplying appropriate parameters (such as the name of the font); the server allocates the resource and returns a 31-bit unique identifier used to represent it. The use and interpretation of a resource identifier is independent of any network connection. Any client that knows (or guesses) the identifier for a resource can use and manipulate the resource freely, even if it was created by another client. This capability is required to allow window managers to be written independently of applications, and to allow multi-process applications to manipulate shared resources. However, to avoid problems associated with clients that fail to clean up their resources at termination (which is all too common in operating systems where users can unilaterally abort processes), the maximum lifetime of a resource is always tied to the connection over which it was created. Thus, when a client terminates, all of the resources it created are destroyed automatically.

Access control is performed only when a client attempts to establish a connection to the server; once the connection is established the client can freely manipulate any resource. Since accidental manipulation of some other client's resource is extremely unlikely (both in theory and in practice), we believe introducing access control on a per-resource basis would only serve to decrease performance, not to significantly increase security or robustness. The current access control mechanism is based simply on host network addresses, as this information is provided by most network stream protocols, and there seems to be no widely used or even widely available user-level authentication mechanism. Host-based access control has proven to be marginally acceptable in a workstation environment, but is rather unacceptable for time-shared machines.[8]

Each client-generated protocol request is a simple data block consisting of an opcode, some number of fixed-length parameters, and possibly a variable-length parameter. For example, to display text in a window, the fixed-length parameters include the drawing color and the identifiers for the window and the font, and the variable-length parameter is the string of characters. All operations on a resource explicitly contain the identifier of the resource as a parameter. In this way, an application can multiplex use of many windows over a single network connection. This multiplexing makes it easy for the client to control the time-order of updates to multiple windows. Similarly, each input event generated by the server contains the identifier of the window in which the event occurred. Multiplexing over a single stream allows the client to act on events from multiple windows in correct time order; timestamps alone are inadequate without strong guarantees from the stream mechanism.

Numerous Unix-based window systems [14, 16, 17, 19, 28, 30, 33] use file or channel descriptors to represent windows; window creation involves an interaction with the operating system, which results in the creation of such a descriptor. Typically, this means the window cannot be named (and hence cannot be shared) by programs running on different machines, and perhaps not even by programs running on the same machine. More serious, there is often a severe restriction on the number of active descriptors a process may have: 20 on older systems and usually 64 on newer systems. The use of 50 or more windows (albeit nested inside a single top-level window) is quite common in X applications. The use of a single connection, over which an arbitrary number of windows can be multiplexed, is clearly a better approach.

## 4. Window Hierarchy

The server supports an arbitrarily branching hierarchy of rectangular windows. At the top is the *root* window, which covers the entire screen. The *top-level* windows of applications are created as subwindows of the root window. The window hierarchy models the now-familiar "stacks of papers" desktop. For a given window, its subwindows can be stacked in any order, with arbitrary overlaps. When window W1 partially or completely covers window W2, we say that W1 *obscures* W2. This relationship is not restricted to siblings; if W1 obscures W2, then W1 may also obscure subwindows of W2. A window also obscures its parent. Window hierarchies never interleave; if window W1 obscures sibling window W2, then subwindows of W2 never obscure W1 or subwindows of W1. A window is not restricted in size or placement by the boundaries of its parent, but a window is always visibly clipped by its parent: portions

---

[8] It is interesting that *professors* at MIT have argued vociferously to disable all access control.

of the window that extend outside the boundaries of the parent are never displayed, and do not obscure other windows. Finally, a window can be either *mapped* or *unmapped*. An unmapped window is never visible on the screen; a mapped window can only be visible if all of its ancestors are also mapped.

Output to a leaf window (one with no subwindows) is always clipped to the visible portions of the window; drawing on such a window never draws into obscuring windows. Output to a window that contains subwindows can be performed in two modes. In *clipped* mode the output is clipped normally by all obscuring windows (including subwindows), but in *draw-through* mode the output is not clipped by subwindows. For example, draw-through mode is used on the root window during window management, tracking the mouse with the outline of a window to indicate how the window is to be moved or resized. If clipped mode were used instead, the entire outline would not be visible.

The coordinate system is defined with the X axis horizontal and the Y axis vertical. Each window has its own coordinate system, with the origin at the upper left corner of the window. Having per-window coordinate systems is crucial, particularly for top-level windows; applications are almost always designed to be insensitive to their position on the screen, and having to worry about race conditions when moving windows would be a disaster. The coordinate system is discrete: each pixel in the window corresponds to a single unit in the coordinate system, with coordinates centered on the pixels, and all coordinates are expressed as integers in the protocol. We believe fractional coordinates are not required at the protocol level for the raster graphics provided in X (see section 6), although they may be required for high-end color graphics, such as anti-aliasing. The aspect ratio of the screen is not masked by the protocol, since we believe that most displays have a one to one aspect ratio; in this regard X is arguably device dependent.

Although the coordinate system is discrete at the protocol level, continuous or alternate-origin coordinate systems certainly can be used at the application level, but client-side libraries must eventually translate to the discrete coordinates defined by the protocol. In this way, we can ignore the many variations in floating-point (or even fixed-point) formats among architectures. Further, the coordinates can be expressed in the protocol as 16-bit quantities, which can be manipulated efficiently in virtually every machine/display architecture, and which minimizes the number of data bytes transmitted over the network. The use of 16-bit quantities does have a drawback, in that some applications (particularly CAD tools) like to perform zoom operations simply by scaling coordinates and redrawing, relying on the window system to clip appropriately. Since scaling quickly overflows 16 bits, additional clipping must be performed explicitly by such applications.

A window can optionally have a *border*, a shaded outer frame maintained explicitly by the X server. The origin of the window's coordinate system is inside the border, and output to the window is clipped automatically so as not to extend into the border. The presence of borders slightly complicates the semantics of the window system; for simplicity we will ignore them in the remainder of this paper.

The basic operations on window structure are straightforward. An unmapped window is created by specifying the parent window, the position within the parent of the upper left corner of the new window, and the width and height (in coordinate units) of the new window. A window can be destroyed, in which case all windows below it in the hierarchy are also destroyed. A window can be mapped and unmapped, without changing its position. A window can be moved and resized, including being moved and resized simultaneously. A window can also be "depthwise" raised to the top or lowered to the bottom the stack with respect to its siblings, without changing its coordinate position. Currently mapping or configuring a window forces the window to be raised. This restriction appeared to simplify the server implementation, but also happened to match the basic management interface we expected to build. This restriction will be eliminated in the next version.

The windows described above are the usual *opaque* windows. X also provides *transparent* windows. A transparent window is always invisible on the screen, and does not obscure output to, or visibility of, other windows. Output to a transparent window is clipped to that window, but is actually drawn on the parent window. Thus, for output, a transparent window is simply a clipping rectangle that can be applied to restrict output within a (parent) window. Input processing for transparent and opaque windows is identical, as described in Section 8. In Section 10 we will argue that most uses of transparent windows are better satisfied with other mechanisms. Therefore, for simplicity, we will ignore transparent windows in the rest of this paper.

The X server is designed explicitly to make windows inexpensive. Our goal was to make it reasonable to use windows for such things as individual menu items, buttons, even individual items in forms and spreadsheets. As such, the server must deal efficiently with hundreds (though not necessarily thousands) of windows on the screen simultaneously. Experience with X has shown that many implementors find this capability extremely useful.

## 5. Color

The screen is viewed as two dimensional, with an N-bit *pixel* value stored at each coordinate. The number of bits in a pixel value, and how a value translates into a color, depends on the hardware. X is designed to support two types of hardware: monochrome and pseudo-color. A monochrome display has one bit per pixel, and the two values translate into black and white. Pseudo-color displays typically have between four and twelve bits per pixel; the pixel value is used as an index into a color map, yielding red, green, and blue intensities. The color map can be changed dynamically, so that a given pixel value can represent different colors over time. Gray-scale is viewed as a degenerate case of pseudo-color.

We desire a design matching most display hardware, while abstracting differences in such a way that programmers do not have to double or triple-code their applications to cover the spectrum. We also want multiple applications to coexist within a single color map, so that applications always show true color on the screen. To allow this, and to keep applications device independent, pixel values should not be coded explicitly into applications. Instead, the server must be responsible for managing the color map, and color map allocation must be expressed in hardware-independent terms.

All graphics operations in X are expressed in terms of pixel values. For example, to draw a line, one specifies not only the coordinates of the end-points but the pixel value with which to draw the line. (Logic functions and plane-select masks are also specified, as described in Section 6.) On a monochrome display, the only two pixel values are zero and one, which are (somewhat arbitrarily) defined to be black and white, respectively. On a pseudo-color display, pixel values zero and one are pre-allocated by the server, for use as "black" and "white", so that monochrome applications display correctly on color displays. Of course, the actual colors need not be black and white, but can be set by the user.

There are two ways for a client to obtain pixel values. In the simplest request, the client specifies red, green, and blue color values, and the server allocates an arbitrary pixel value and sets the color map so the pixel value represents the closest color the hardware can provide. The color map entry for this pixel value cannot be changed by the client, so if some other client requests an equivalent color, the server is free to respond with the same pixel value. Such sharing is important in maximizing use of the color map. To isolate applications from variations in color representation among displays (due, for example, to the standard of illumination used for calibration), the server provides a color database which clients can use to translate string names of colors into red, green, and blue values tailored for the particular display.

The second request allocates writable map entries. This mechanism was designed explicitly for X; we are not aware of a comparable mechanism in any other window system. The client specifies two numbers, $C$ and $P$, with $C$ positive and $P$ non-negative; the request can be expressed as "allocate $C$ colors and $P$ planes". The total number of pixel values allocated by the server is $C*2^P$. The values passed back to the client consist of $C$ base pixel values, and a plane mask containing $P$ bits. None of the base pixel values have any one bits in common with the plane mask, and the complete set of allocated pixel values is obtained by combining all possible combinations of one bits from the plane mask with each of the base pixel values. The client can optionally require the $P$ planes to be contiguous, in which case all $P$ bits in the plane mask will be contiguous.

There are three common uses of this second request. One is simply to allocate a number of "unrelated" pixel values; in this case, $P$ will be zero. A second use is in imaging applications, where it is convenient to be able to perform simple arithmetic on pixel values. In this case, a contiguous block of pixel values is allocated by setting $C$ to one and $P$ to the log (base 2) of the number of pixel values required, and requesting contiguous allocation. Arithmetic on the pixel values then requires at most some additional shift and mask operations.

A third form of allocation arises in applications that want some form of overlay graphics, such as highlighting or outlining regions. Here the requirement is to be able to draw and then erase graphics without disturbing existing window contents. For example, suppose an application typically uses four colors, but needs to be able to overlay a rectangle outline in a fifth color. An allocation request with C set to four and P set to one results in two groups of four pixel values. The four base pixel values are assigned the four normal colors, and the four alternate pixel values are all assigned the fifth color. Overlay graphics can then be drawn by restricting output (see the next section) to the single bit plane specified in the mask returned by the color allocation. Turning bits in this plane on (to ones) changes the image to the fifth color, and turning them off reverts the image to its original color.

## 6. Graphics and Text

Graphics operations are often the most complex part of any window system, simply because so many different effects and variations are required to satisfy a wide range of applications. In this section we sketch the operations provided in X, so that the basic level of graphics support can be understood. The operations are essentially a subset of the Digital Workstation Graphics Architecture; the VS100 display [11] implements this architecture for 1-bit pixel values. The set of operations purposely was kept simple, in order to maximize portability.

Graphics operations in X are expressed in terms of relatively high-level concepts, such as lines, rectangles, curves, and fonts. This is in contrast to systems in which the basic primitives are to read and write individual pixels. Basing applications on pixel-level primitives works well when display memory can be mapped into the application's address space for direct manipulation. However, both display hardware and operating systems exist for which such direct access is not possible, and emulating pixel-level manipulations in such an environment results in extremely poor performance. Expressing operations at a higher level avoids such device dependencies, and also avoids potential problems with network bandwidth. With high-level operations, a protocol request transmitted as a small number of bits over the network typically affects ten to one hundred times as many pixels on the screen.

## 6.1. Images

Two forms of off-screen images are supported in X: bitmaps and pixmaps. A bitmap is a single plane (bit) rectangle. A pixmap is an N-plane (pixel) rectangle, where *N* is the number of bits per pixel used by the particular display. A bitmap or pixmap can be created by transmitting all of the bits to the server; a pixmap can also be created by copying a rectangular region of a window. Bitmaps and pixmaps of arbitrary size can be created. Transmitting very large (or deep) images over a network connection can be quite slow; however, the ability to make use of shared memory in conjunction with the IPC mechanism would help enormously when the client and server are on the same machine.

The primary use of bitmaps is as masks (clipping regions). Several graphics requests allow a bitmap to be used as a clipping region [35]. Bitmaps are also used to construct cursors, as described in Section 8. Pixmaps are used to store frequently drawn images, and as temporary backing-store for pop-up menus (as described in Section 8). However, the principal use of pixmaps is as tiles, that is, as patterns which are replicated in two dimensions to cover a region. Since there are often hardware restrictions as to what tile shapes can be replicated efficiently, guaranteed shapes are not defined by the X protocol. An application can query the server to determine what shapes are supported, although to date most applications simply assume 16 by 16 tiles are supported. A better semantics is to support arbitrary shapes, but allow applications to query as to which shapes are most efficient.

The tiling origin used in X is almost always the origin of the destination window. That is, if enough tiles were laid out, one tile would have its upper left corner at the upper left corner of the window. In this way, the contents of the window are independent of the window’s position on the screen, and the window can be moved transparently to the application.

Servers vary widely in the amount of off-screen memory provided. For example, some servers limit off-screen memory to that accessible directly to the graphics processor (typically one to three times the size of screen memory), and fonts and other resources are allocated from this same pool. Other servers utilize their entire virtual address space for off-screen memory. Since off-screen memory for images is finite, an explicit part of the X protocol is the possibility that bitmap or pixmap creation can fail. Depending on the intended use of the image, the application may or may not be able to cope with the failure. For example, if the image was being stored simply to speed up redisplay, the application can always transmit the image directly each time (see below). If the image was to be a temporary backing-store for a window, the application can fall back on normal exposure processing (as described in Section 7). Servers should be constructed in such a way as to virtually guarantee sufficient memory (e.g., by caching images) for creating at least small tiles and cursors, although this is not true in current implementations.

## 6.2. Graphics

All graphics and text requests include a logic function and a plane-select mask (an integer with the same number of bits as a pixel value) to modify the operation. All sixteen logic functions are provided. Given a source and destination pixel, the function is computed bitwise on corresponding bits of the pixels, but only on bits specified in the plane-select mask. Thus the result pixel is computed as

((source FUNC destination) AND mask) OR (destination AND (NOT mask))

The most common operation is simply replacing the destination with the source in all planes.

The simplest graphics request takes a single source pixel value and combines it with every pixel in a rectangular region of a window. Typically this is used to fill a region with a color, but by varying the logic function or masks, other effects can be achieved. A second request takes a tile, effectively constructs a tiled rectangular source with it, and then combines the source with a rectangular region of a window.

An arbitrary image can be displayed directly, without first being stored off-screen. For monochrome images, the full contents of a bitmap are transmitted, along with a pair of pixel values; the image is displayed in a region of a window with those two colors. For color images, the full contents of a pixmap can be transmitted and displayed. In order to avoid inordinate buffer space in the server, very large images must be broken into sections on the client side and displayed in separate requests.

The CopyArea request allows one region of a window to be moved to (or combined with) another region of the same window. This is the usual *bitblt*, or "bit block transfer" operation. The source and destination are given as rectangular regions of the window; the two regions have the same dimensions. The operation is such that overlap of the source and destination does not affect the result.

X provides a complex primitive for line drawing. It provides for arbitrary combinations of straight and curved segments, defining both open and closed shapes. Lines can be *solid*, by drawing with a single source pixel value, *dashed*, by alternately drawing with a single source pixel value and not drawing, and *patterned*, by alternately drawing with two source pixel values. Lines are drawn with a rectangular brush. Clients can query the server to determine what brush shapes are supported; a better semantics would be to support arbitrary shapes, but allow applications to query as to which shapes are most efficient.

A final request allows an arbitrary closed shape (such as could be specified in the line drawing request) to be filled with either a single source pixel value or a tile. For self-intersecting shapes, the even-odd rule is used: a point is inside the shape if an infinite ray with the point as origin crosses the path an odd number of times.

## 6.3. Text

For high-performance text, X provides direct support for bitmap fonts. A font consists of up to 256 bitmaps; each bitmap in a font has the same height but can vary in width. To allow server-specific font representations, clients "create" fonts by specifying a name rather than by downloading bitmap images into the server. An application can use an arbitrary number of fonts, but (as with all resources) font allocation can fail for lack of memory. A reasonably implemented server should support an essentially unbounded number of fonts (e.g., by caching), but some existing server implementations are deficient in this respect. Unlike Andrew [7], no heuristics are applied by the server when resolving a name to a font; specific communities or applications may demand a variety of heuristics, and as such they belong outside the base window system. Also unlike Andrew, the X server is not free to dynamically substitute one font for another; we do not believe such behavior is necessary or appropriate.

A string of text can be displayed using a font either as a mask or as a source. Using a font as a mask, the foreground (the one bits in the bitmap) of each character is drawn with a single source pixel value. Using a font as a source, the entire image of each character is drawn, using a pair of pixel values. Source font output is provided specifically for applications using fixed-width fonts in emulating traditional terminals.

To support "cut and paste" operations between applications, the server provides a number of buffers into which a client can read and write an arbitrary string of bytes. (This mechanism was adopted from Andrew.) Although these buffers are used principally for text strings, the server imposes no interpretation on the data, so cooperating applications can use the buffers to exchange such things as resource identifiers and images.

## 7. Exposures

Given that output to obscured windows is possible, the issue of *exposure* must be addressed. When all (or a piece) of an obscured window again becomes visible (for example, as the result of the window being raised), is the client or the server responsible for restoring the contents of the window? In X, it is the responsibility of the client. When a region of a window becomes exposed, the server sends an asynchronous event to the client, specifying the window and the region that has been exposed; the rest is up to the application. A trivial application might simply redraw the entire window; a more sophisticated application would only redraw the exposed region.

Why is the client responsible? Because X imposes no structure on, or relationships between, graphics operations from a client, there are only two basic mechanisms by which the server might restore window contents: by maintaining display lists, and by maintaining off-screen images. In the first approach, the server essentially retains a list of all output requests performed on the window. When a region of the window becomes exposed, the server either re-executes all requests to the entire window, or only re-executes requests that affect the region while clipping the output to that region. In the alternative approach, when a window becomes obscured the server saves the obscured region (or perhaps the entire window) in off-screen memory. All subsequent output requests are executed not only to the visible regions of the window, but to the off-screen image as well. When an obscured region becomes visible again, the off-screen copy is simply restored.

We believe neither server-based approach is acceptable. With display lists, the server is unlikely to have any reasonable notion of when later output requests nullify earlier ones. Either the display list becomes unmanageably long, and a refresh that should appear nearly instantaneous instead appears as a slow-motion replay, or the server spends a significant length of time pruning the display list, and normal-case performance is considerably reduced. One problem with the off-screen image approach is (virtual) memory consumption: on a 1024 by 1024 8-plane display, just one full-screen image requires one megabyte of storage, and multiple overlapping windows could easily require many times that amount. Another problem is that the cost of the implementation can be prohibitive. Consider, for example, the QDSS display [34], which has a graphics co-processor. In the QDSS, display memory is inaccessible to the host processor. In addition, the co-processor cannot perform operations in host memory, and has relatively little off-screen memory of its own. The only viable way to maintain off-screen images for displays like the QDSS may be to emulate the co-processor in software. It can easily take tens of thousands of lines of code to emulate a co-processor, and such emulation may execute orders of magnitude slower than the co-processor.

Our belief is that many applications can take advantage of their own information structures to facilitate rapid redisplay, without the expense of maintaining a distinct display structure or backing-store in the client or the server, and often with even better performance. (Sapphire [17] permits client refresh for this reason.) For example, a text editor can redisplay directly from the source, and a VLSI editor can redisplay directly from the layout and component definitions. Many applications will be built on top of high-level graphics libraries that automatically maintain the data structures necessary to implement rapid redisplay. For example, the structured display file mechanism in VGTS could be supported in a client library. Of course, pushing the responsibility back on the application may not simplify matters, particularly when retrofitting old systems to a new environment. For example, the current GKS design does not provide adequate hooks for automatic, system-generated refresh of application windows, nor does it provide an adequate mechanism for forcing refresh back on the application.

Relying on client-controlled refresh also derives from window management philosophy. Our belief is that applications cannot be written with fixed top-level window sizes built in. Rather, they must function correctly with almost any size, and continue to function correctly as windows are dynamically resized. This is necessary if applications are to be usable on a variety of displays under a variety of window management policies. (Of course, an application may need a minimum size to function reasonably, and may prefer the width or height to be a multiple of some number; X allows the client to attach a resize hint to each window to inform window managers of this.) Our belief is that most applications, for one reason or another, will already have code for performing a complete redisplay of the window, and that it is usually straightforward to modify this code to deal with partial exposures. Similar arguments were used in the design of both Andrew and Mex, and experience has confirmed their decision [7, 23].

This is not to argue that the server should never maintain window contents, only that it should not be *required* to maintain contents. For complex imaging and graphics applications, efficient maintenance by the server may be critical for acceptable performance of window management functions. There is nothing inherent in the X protocol that precludes the server from maintaining window contents and not generating exposure events. In the next version of X, windows will have several attributes to advise the server as to when and how contents should be maintained.

In X, clients are never informed of what regions are obscured, only of what regions have become visible. Thus, clients have insufficient information to try and optimize output by only drawing to visible regions. However, we feel this is justified on two grounds. First, realistically, users seldom stack windows such that the active ones are obscured, so there is little point in complicating applications to optimize this case. More importantly, allowing applications to restrict output to only visible regions would conflict with the desire to have the server maintain obscured regions automatically when possible.

An interesting complication with the CopyArea request (described in Section 6) arises, having decided on client refresh. If part of the source region of the CopyArea is obscured, then not all of the destination region can be updated properly, and the client must be notified (with an exposure event) so that it can correct the problem. Since output requests are asynchronous, care must be taken by the application to handle exposure events when using CopyArea. In particular, if a region is exposed and an event sent by the server, a subsequent CopyArea may move all or part of the region before the event is actually received by the application. Several simple algorithms have been designed to deal with this situation, but we will not present them here.

Client refresh raises a visual problem in a network environment. When a region of a window becomes exposed, what contents should the server initially place in that window? In a local, tightly-coupled environment, it might be perfectly reasonable to leave the contents unaltered, because the client can almost instantaneously begin to refresh the region. In a network environment however (and even in a local system where processes can get "swapped out" and take considerable time to swap back in), inevitable delays can lead to visually confusing results. For example, the user may move a window, and see two images of the window on the screen for a significant length of time, or resize a window and see no immediate change in the appearance of the screen.

To avoid such anomalies in X, clients must define a *background* for every window. The background can be a single color, or it can be a tiling pattern. Whenever a region of a window is exposed, the server immediately paints the region with the background. Users therefore see window shapes immediately, even if the "contents" are slow to arrive. Of course, many application windows have some notion of a background anyway, so having the server initialize with a background seldom results in extraneous redisplay. In fact, many non-leaf windows typically contain nothing but a background, and having the server paint that background frees the applications from performing any redisplay at all to those windows.

Although we believe client-generated refresh is acceptable most of the time, it does not always perform well with momentary pop-up menus, where speed is at a premium. To avoid potentially expensive refresh when a menu is removed from the screen, a client can explicitly copy the region to be covered by the menu into off-screen memory (within the server) before mapping the menu window. A special unmap request is used to remove the menu: it unmaps the window without affecting the contents of the screen or generating exposure events. The original contents are then copied back onto the screen. In addition, the client usually *grabs* the server for the entire sequence, using a request which freezes all other clients until a corresponding ungrab request is issued (or the grabbing client terminates). Without this, concurrent output from other clients to regions obscured by the menu would be lost. Although freezing other clients is in general a poor idea, it seems acceptable for momentary menus.

## 8. Input

We now turn to a discussion of input events, but first we briefly describe the support for mouse cursors. Clients can define arbitrary shapes for use as mouse cursors. A cursor is defined by a source bitmap, a pair of pixel values with which to display the bitmap, a mask bitmap which defines the precise shape of the image, and a coordinate within the source bitmap which defines the "center" or "hot spot" of the cursor. Cursors of arbitrary size can be constructed, although only a portion of the cursor may be displayed on some hardware. Clients can query the server to determine what cursor sizes are supported, but existing applications typically just assume a 16 by 16 image can always be displayed. Cursors also can be constructed from character images in fonts; this provides a simple form of named indirection, allowing custom tailoring to each display without having to modify the applications.

A window is said to *contain* the mouse if the hot spot of the cursor is within a visible portion of the window or one of its subwindows. The mouse is said to be *in* a window if the window contains the mouse but no subwindow contains the mouse. Every window can have a mouse cursor defined for it. The server automatically displays the cursor of whatever window the mouse is currently in; if the window has no cursor defined, the server displays the cursor of the closest ancestor with a cursor defined.

Input is associated with windows. Input to a given window is controlled by a single client, which need not be the client that created the window. Events are classified into various types, and the controlling client selects which types are of interest to it. Only events matching in type with this selection are sent to the client. When an input event is generated for a window and the controlling client has not selected that type, the server *propagates* the event to the closest ancestor window for which some client has selected the type, and sends the event to that client instead. Every event includes the window that had the event type selected; this window is called the *event window*. If the event has been propagated, the event also includes the next window down in the hierarchy between the event window and the original window on which the event was generated.

## 8.1. The Keyboard

For the keyboard, a client can selectively receive events on the press or release of a key. Keyboard events are not reported in terms of ASCII character codes; instead, each key is assigned a unique code, and client software must translate these codes into the appropriate characters. The mapping from keycaps to keycodes is intended to be "universal" and predefined; a given keycap has the same keycode on all keyboards. Applications generally have been written to read a "keymap file" from the user’s home directory, so that users can remap the keyboard as they see fit.

The use of coded keys is secondary to the ability to detect both up and down transitions on the keyboard. For example, a common trick in window systems is for mouse button operations to be affected by keyboard *modifiers* such as the Shift, Control, and Meta keys. A useful feature of the Genera [31] system is the use of a "mouse documentation line", which changes dynamically as modifiers are pressed and released, indicating the function of the mouse buttons. A base window system must provide this capability. Transitions are not only useful on modifiers; various applications for systems other than X have been designed to use "chords" (groups of keys pressed simultaneously), and again the window system should support them.

The keyboard is always *attached* to some window (typically the root window or a top-level window); we call this window the *focus* window. A request can be used (usually by the input manager) to attach the keyboard to any window. The window that receives keyboard input depends on both the mouse position and the focus window. If the mouse is in some descendant of the focus window, that descendant receives the input. If the mouse is not in a descendant of the focus window, then the focus window receives the input, even if the mouse is outside the focus window. For applications that wish to have the mouse state modify the effect of keyboard input, a keyboard event contains the mouse coordinates, both relative to the event window and global to the screen, as well as the state of the mouse buttons.

To provide a reasonable user interface, keyboard events also contain the state of the most common modifier keys: Shift, ShiftLock, Control, and Meta. Without this information, anomalous behavior can result. If the user switches windows while modifier keys are down, the new client must somehow determine which modifiers are down. Placing the modifier state in the keyboard events solves such problems, and also has another benefit: most clients do not have to maintain their own shadow of the modifier state, and so often can completely ignore key release events. However, there is a conflict between this server-maintained state and client-maintained keyboard mappings. In particular, clients cannot use non-standard keys as modifiers, or use chords without the possibility of anomalies such as described above. We believe the correct solution (not yet supported in X) is for the server to maintain a bit mask reflecting the full state of the keyboard, and to allow clients to read this mask. An application using chords or non-standard modifiers would request the server to send this mask automatically whenever the mouse entered the application's window.

### 8.2. The Mouse

The X protocol is (somewhat arbitrarily) designed for mice with up to three buttons. An application can selectively receive events on the press or release of each button. Each event contains the current mouse coordinates (both local to the window and global to the screen), the current state of all buttons and modifier keys, and a timestamp which can be used, for example, to decide when a succession of clicks constitutes a double or triple click. An application can also choose to receive mouse motion events, either whenever the mouse is in the window, or only when particular buttons have also been pressed. The application cannot control the granularity of the reporting, nor is any minimum granularity guaranteed. In fact, typical server implementations make an effort to compact motion events, to minimize system overhead and wired memory in device drivers. As such, X may not serve adequately for fine-grained tracking, such as in fast moving free-hand drawing applications.

Even with motion compaction, servers can generate considerable numbers of motion events. If an application attempts to respond in real time to every event, it can easily get far behind relative to the actual position of the mouse. Instead, many applications simply treat motion events as hints. When a motion event is received, the event is simply discarded, and the client then explicitly queries the server for the current mouse position. In waiting for the reply, more motion events may be received; these are also discarded. The client then reacts based on the queried mouse position. The advantage of this scheme over continuously polling the mouse position is that no CPU time is consumed while the mouse is stationary.

Clients can also receive an event each time the mouse enters or leaves a window. This can be particularly useful in implementing menus. For example, each menu item can be placed in a separate subwindow of the overall menu window. When the mouse enters a subwindow, the item is highlighted in some fashion (e.g., by inverting the video sense), and when the mouse leaves the window the item is restored to normal. Implementing a menu in this manner requires considerably less CPU overhead than continuous polling of the mouse, and also less overhead than using motion events, since most motion events would be within windows and thus uninteresting.

Due to the nature of overlapping windows, and because continuous tracking by the server is not guaranteed, the mouse may appear to move instantaneously between any pair of windows on the screen. Certainly the window the mouse was in should be notified of the mouse leaving, and the window the mouse is now in should be notified of the mouse entering. However, all of the windows "in between" in the hierarchy may also be interested in the transition. This is useful in simplifying the structure of some applications, and is necessary in implementing certain kinds of window managers and input managers. Thus, when the mouse moves from window A to window B, with window W as their closest (least) common ancestor, all ancestors of A below W also receive leave events, and all ancestors of B below W receive enter events.

Except for mouse motion events, it might be argued that events are infrequent enough that the server should always send all events to the client, and eliminate the complexity of selecting events. However, some applications are written with interrupt-driven input; events are received asynchronously, and cause the current computation to be suspended so that the input can be processed. For example, a text editor might use interrupt-driven input, with the normal computation being redisplay of the window. The receipt of extraneous input events (for example, key release events) can cause noticeable "hiccups" in such redisplay.

## 9. Input and Window Management

There are two basic modes of keyboard management: *real-estate* and *listener*. In real-estate mode, the keyboard "follows" the mouse; keyboard input is directed to whatever window the mouse is in. In listener mode, keyboard input is directed to a specific window, independent of the mouse position. Some systems provide only real-estate mode [1, 28], some only listener mode [8, 17, 19, 23, 30, 31], and Andrew [7] provides both, although the mode cannot be changed during a session. Both modes are supported in X, and the mode can be changed dynamically. Real-estate mode is the default behavior, with the root window as the focus window, as described in the previous section. An input manager can also make some other (typically top-level) window the focus window, yielding listener mode. Note however, that in listener mode in X, the client controlling the focus window can still get real-estate behavior for subwindows, if desired; this capability has proven useful in several applications.

The primary function of a window manager is reconfiguration: restacking, resizing, and repositioning top-level windows. The configuration of nested windows is assumed to be application-specific, and under control of the applications. There are two broad categories of window managers: manual and automatic. A manual window manager is "passive", and simply provides an interface to allow the user to manipulate the desktop; windows can be resized and reorganized at will. The initial size and position of a window typically (but not always) is under user or application control. Automatic window managers are "active", and operate for the most part without human interaction; size and position at window creation, and reconfiguration at window destruction, are chosen by the system. Automatic managers typically tile the screen with windows, such that no two windows overlap, automatically adjusting the layout as windows are created and destroyed. Andrew [7], Star [24], and Cedar [32] provide automatic management, plus limited manual reconfiguration capability.

Existing window managers for X are manual. Automatic management that is transparent to applications cannot be accomplished reasonably in X; future support for automatic management is discussed in Section 10. In the current X design, clients are responsible for initially sizing and placing their top-level windows, not window managers. In this way, applications continue to work when no window manager is present. Typically, the user either specifies geometry information in the application command line, or uses the mouse to sweep out a rectangle on the screen. (For the latter, the application grabs the mouse, as described below.)

### 9.1. Mouse-Driven Management

Existing managers are primarily mouse-driven, and are based on the ability to "steal" events. Specifically, a manager (or any other client) can *grab* a mouse button in combination with a set of modifier keys, with the following effect. Whenever the modifier keys are down and the button is pressed, the event is reported to the grabbing client, regardless of what window the mouse is in. All mouse-related events continue to be sent to that client until the button is released. As part of the grab, the client also specifies a mouse cursor to be used for the duration of the grab, and a window to be used as the event window. A manager specifies the root window as the event window when grabbing buttons; with the event propagation semantics described in Section 8, the grabbed events contain not only the global mouse coordinates, but also the top-level application window (if any) containing the mouse. This is sufficient information to manipulate top-level windows.

Using this button-grab mechanism, several different management interfaces have been built, including a "programmable" interface [5] allowing the user to assign individual commands or user-defined menus of commands to any number of button/modifier combinations. For example, a button click (press and release without intervening motion) might be interpreted as a command to raise or lower a window, or to attach the keyboard; a press/motion/release sequence might be interpreted as a command to move a window to a new position; or a button press might cause a menu to pop up, with the selection indicated by the mouse position at the release of the button. By allowing both specific commands and menus to be bound to buttons, a range of interfaces can be constructed to satisfy both "expert" and "novice" users.

Another form of manager simply displays a static menu bar along the top of the screen, with items for such operations as moving a window and attaching the keyboard. The menu is used in combination with a mouse-grab primitive, with which a client can unilaterally grab the mouse and then later explicitly release it; during such a mouse-grab, events are redirected to the grabbing client, just as for button-grabs. When the user clicks on a menu bar item with any button, the manager unilaterally grabs the mouse. The user then uses the mouse to execute the specific command. For example, having clicked on the "move" item, the user indicates the window to move by placing the mouse in the window and pressing a button, then indicates the new position by moving the mouse and releasing the button. The manager then releases the mouse.

### 9.2. Icons

One important "resizing" operation performed by a window manager is transforming a window into a small icon and back again. In X, icons are merely windows. Transforming a window into an icon simply involves unmapping the window and mapping its associated icon. The association between a window and its icon is maintained in the server, rather than the window manager, and either the application or the manager can provide the icon. In this way, the manager can provide a default icon form for most clients, but clients can provide their own if desired, possibly with dynamic rather than static contents. The client is still insulated from management policy, even if it provides the icon: the manager is responsible for positioning, mapping, and unmapping the icon, and the client is responsible only for displaying the contents.

The icon state is maintained in the server not only to allow clients to provide icons, but to avoid the loss of state if the window manager should terminate abnormally. When a window manager terminates, any windows it has created are destroyed, including icon windows. With knowledge of icons, the server can detect when an icon is destroyed, and automatically remap the associated client window. Without this, abnormal termination of the window manager would result in "lost" windows.

### 9.3. Race Conditions

There are many race conditions that must be dealt with in input and window management, due to the asynchronous nature of event handling. For example, if a manager attempts to grab the mouse in response to a press of a button, the mouse-grab request might not reach the server until after the button is released, and intervening mouse events would be missed. Or, if the user clicks on a window to attach the keyboard there, and then immediately begins typing, the first few keystrokes might occur before the manager actually responds to the click and the server actually moves the keyboard focus. A final example is a simple interface in which clicking on a window lowers it. Given a stack of three windows, the user might rapidly click twice in the same spot, expecting the top two windows to be lowered. Unless the first click is sent to the manager and the resulting request to lower is processed by the server before the second click takes place, the event window for the second click will be the same as for the first click, and the manager will lower the first window twice.

A work-around for the last example, used by existing managers, is to ignore the event window reported in most events. Instead, the global mouse coordinates reported in the event are used in a follow-up query request to determine which top-level window now contains that coordinate. However, not all race conditions have acceptable solutions within the current X design. For a general solution, it must be possible for the manager to synchronize operations explicitly with event processing in the server. For example, a manager might specify that, at the press of a button, event processing in the server should cease until an explicit acknowledgment is received from the manager.

## 10. Future

Based on critiques from numerous universities and commercial firms, a fairly extensive evaluation and redesign of the X protocol has been underway since May 1986. Our desire is to define a "core" protocol that can serve as a standard for window system construction over the next several years. We expect to present the rationale for this new design in the very near future, once it has been validated by at least a preliminary implementation. In this section, we highlight the major protocol changes.

### 10.1. Resource Allocation

Since the server is responsible for assigning identifiers to resources, each resource allocation currently requires a round-trip time to perform. For applications that allocate many resources, this causes a considerable start-up delay. For example, a multi-pane menu might consist of dozens of windows, numerous fonts, and several different mouse cursors, leading to a delay of one second or longer.

In retrospect, this is the most significant defect in the design of X. To get around these delays, programming interfaces have been augmented to provide "batch mode" operations. If several resources must be created, but there are no inter-dependencies among the allocation requests, all of the requests are sent in a batch, and then all of the replies are received. This effectively reduces the delay to a single round-trip time.

A better solution to this problem is to make clients generate the identifiers. When the client establishes a connection to the server, it is given a specific subrange from which it can allocate. This change will significantly improve start-up times without affecting applications, as identifiers can be generated inside low-level libraries without changing programming interfaces.

### 10.2. Transparent Windows

One use of transparent windows is as clipping regions. However, they are unsatisfactory for this purpose because every coordinate in a graphics request must be translated by the client from the "real" window's origin to the transparent window's origin. A better approach to clipping regions is to allow clients to create clipping regions and attach them to all graphics requests. As noted in Section 6, X currently allows a clipping region in the form of a bitmap to be attached to a few graphics requests. Allowing a clipping region, specified either as a bitmap or a list of rectangles, to be attached to all graphics requests provides a more uniform mechanism.

The major use of transparent windows to date is actually as inexpensive opaque windows. In the current server implementation, transparent windows can be created and transformed significantly faster than opaque windows. Because of this, transparent windows are often used when opaque windows would otherwise be adequate. We believe a new implementation of the server will improve the performance of opaque windows to the point that this will no longer be necessary.

With explicit clipping regions added for graphics, and the performance advantages of transparent windows reduced, the only remaining use of transparent windows is for input (and cursor) control. Various applications want relatively fine-grained input control, and such control must not affect graphics output. Close control of cursor images and mouse motion events seems particularly important. However, the vast majority of the time control naturally is associated with normal window boundaries, so it would be unwise to divorce input control completely from windows. As such, the new protocol provides "input-only" windows, which act like normal windows for the purposes of input and cursor control, but which cannot be used as a source or destination in graphics requests, and which are completely invisible as far as output is concerned.

### 10.3. Color

X originally was not designed to deal with direct-color displays. Direct-color displays typically have between 12 and 36 bits per pixel; the pixel value consists of three subfields, which are used as indexes into three independent color maps: one for red intensities, one for green, and one for blue. Some direct-color displays also have a fourth subfield, sometimes referred to as "z-channel" information, used to control attributes such as blending or chroma keying. We now understand how to incorporate direct-color displays without z-channel information into X, in such a way that the differences between direct-color and pseudo-color color maps need not be apparent to the application, yet still allowing all of the usual color map tricks to played.

At present there is only one color map for all applications, and color applications fail when this map gets full. Although dozens of applications typically can be run under X within a single 8-bit pseudo-color map, a single map is clearly unacceptable when dealing with small color maps, or with multiple applications (e.g., CAD tools) that need large portions of the color map. The solution is to support multiple virtual color maps, still permitting applications to coexist within any map, but allowing the possibility that not all applications show true color simultaneously. This also matches next-generation displays, which actually support multiple color maps in hardware [37].

### 10.4. Graphics

Perhaps the biggest mistake in the graphics area was failing to support fonts with kerning (side bearings). For example, a relatively complete emulation of the Andrew programming interface was built for X, but Andrew applications depend heavily on kerned fonts. There are other deficiencies that will be corrected. For example, large glyph-sets (e.g., Japanese) will be supported, as well as stippling (using a clip mask constructed by tiling a region with a bitmap). The notions of line width, join style, and end style found in PostScript [22] are usually preferred to brush shapes for line drawing, and will be supported.

In an attempt to support a wide range of devices, the exact path followed for lines and filled shapes was originally left undefined in X (the class of curve was not even specified). Different devices use slightly different algorithms to draw straight lines, and it seemed better to have high performance with minor variation than to have uniformity with poor performance. Relatively few devices support curve drawing in hardware, but some support it in firmware, and again performance seemed more important than accuracy. In retrospect, however, allowing such device dependent behavior was a poor decision. The vast majority of applications draw lines aligned on an axis, and speed and precision are not an issue. The applications that do require complex shapes also require predictable results, so precise specifications are important.

A notable feature missing in X is the ability to perform graphics operations off screen. The reasons for this are essentially the same as those presented when discussing exposures in Section 7. In particular, not all graphics co-processors can operate on host memory, and emulating such processors can be expensive. However, application builders have demanded this capability, and the demand appears to be sufficient leverage to convince server implementors to provide the capability. Off-screen graphics will be possible in the new protocol, although the amount of off-screen memory and its performance characteristics may vary widely. In addition, the protocol is being extended to allow the manipulation of both images and windows of varying depths. For example, a server might support depths of 1, 4, 8, 12, and 24 bits. This allows imaging applications to transmit data more compactly, allows for more efficient memory utilization in the server, and provides a match with next-generation display hardware.

A common debate in graphics systems is whether and where to have state. Should parameters such as logic function, plane mask, source pixel value or tile, tiling origin, font, line width and style, and clipping region be explicit in every request or collected into a state object? The current X protocol is stateless, for the following reasons: both state and stateless programming interfaces can be built easily on top of the protocol; the currently supported graphics requests have just few enough parameters that they can be represented compactly; and the initial set of displays we were interested in (and the implementations we had in mind for them) would not benefit from the addition of state. However, we now believe that a state-based protocol is generally superior, as it handles complex graphics gracefully and allows significantly faster implementations on some displays.

### 10.5. Management

An obvious interface style presently not supported in X is the ability to use the keyboard for management commands. To allow this, a key-grab mechanism, akin to the button-grab mechanism described in Section 9, will be provided. To allow such styles as using the first button click in a window to attach the keyboard, both button-grabs and key-grabs have been extended to apply to specific sub-hierarchies, rather than always to the entire screen. To handle the kinds of race conditions described in

Section 9, a general event synchronization mechanism has been incorporated into the grab mechanisms.

To support automatic window management, a manager must be able to intercept certain management requests from clients (such as mapping or moving a window) before they are executed by the server, and to be notified about others (such as unmapping a window) after they are executed. In addition, some managers want to provide uniform title bars and border decorations automatically. To allow this, it is useful to be able to "splice" hierarchies: to move a window from one parent to another. To allow input managers and window managers to be implemented as separate applications, the ability for multiple clients to select events on the same window is being added. For example, both a window manager and an input manager might be interested in the unmapping or destruction of a window.

### 10.6. Extensibility

The information that input and window managers might desire from applications is quite varied, and it would be a mistake to try and define a fixed set. Similarly, the information paths between applications (e.g., in support of "cut and paste") need to be flexible. To this end, we are adding a Lisp-ish property list [26] mechanism to windows, and the event mechanism is being augmented to provide a simple form of inter-client communication.

The new X protocol explicitly continues to avoid certain areas, such as 3-D graphics and anti-aliasing. However, a general mechanism has been designed to allow extension libraries to be included in a server. The intention is that all servers implement the "core" protocol, but each server can provide arbitrary extensions. If an extension becomes widely accepted by the X community, it can be adopted as part of the core. Each extension library is assigned a global name, and an application can query the server at run-time to determine if a particular extension is present. Request opcodes and event types are allocated dynamically, so that applications need not be modified to execute in each new environment.

## 11. Summary

The X Window 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, that 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.

To date, the X design and implementation effort has focused on the base window system, as described in this paper, and in essential applications and programming interfaces. The design of the network protocol, the design and implementation of device-independent layer of server, and the implementation of several applications and a prototype window manager, were carried out by the first author. The design and implementation of the C programming interface, the implementation of major portions of several applications, and the coordination of efforts within Project Athena and Digital, were carried out by the second author. In addition, many other persons from Project Athena, the Laboratory for Computer Science, and institutions outside MIT have contributed software.

Necessary applications such as window managers and VT100 and Tektronics 4014 terminal emulators have been created, and numerous existing applications, such as text editors and VLSI layout systems, have been ported to the X environment. Although several different menu packages have been implemented, we are only now beginning to see a rich library of tools (scroll bars, frames, panels, more menus, etc.) to facilitate the rapid construction of high-quality user interfaces. Tool building is taking place at many sites, and several universities are now attempting to unify window systems work with X as a base, so that such tools can be shared.

The use of X has grown far beyond anything we had imagined. Digital has incorporated X into a commercial product, and other manufacturers are following suit. With the appearance of such products, and the release of complete X sources on the Berkeley 4.3 Unix distribution tapes, it is no longer feasible to track all X use and development. Existing applications written in C are known to have been ported to seven machine architectures of more than twelve manufacturers, and the C server to six machine architectures and more than sixteen display architectures. In most cases the code is running under Unix, but other operating systems are also involved. In addition, relatively complete server implementations exist in two Lisp dialects. Apart from designing the system to be portable, a large part of this success is due to MIT's decision to distribute X sources without any licensing restrictions, and the willingness of people in both educational and commercial institutions to contribute code without restrictions.

### Acknowledgments

Our thanks go to the many people who have contributed to the success of X. Particular thanks go to those who have made significant contributions to the non-proprietary implementation: Paul Asente (Stanford University), Scott Bates (Brown University), Mike Braca (Brown), Dave Bundy (Brown), Dave Carver (Digital), Tony Della Fera (Digital), Mike Gancarz (Digital), James Gosling (Sun Microsystems), Doug Mink (Smithsonian Astrophysical Observatory), Bob McNamara (Digital), Ron Newman (MIT), Ram Rao (Digital), Dave Rosenthal (Sun), Dan Stone (Brown), Stephen Sutphen (University of Alberta), and Mark Vandevoorde (MIT).

Special thanks go to Digital Equipment Corporation. A redesign of the protocol and a reimplementation of the server to deal with color and to increase performance was made possible with funding (in the form of hardware) from Digital. To their credit, all of the resulting device-independent code remained the property of MIT.

# References


**1.** *???* Apollo, ???

**2.** Asente, P. W Reference Manual. Stanford University, 1984. internal document.

**3.** Balkovich, E., Lerman, S., and Parmelee, R. "Computing in Higher Education: The Athena Experience". *Communications of the ACM 28*, 11 (Nov. 1985).

**4.** Cheriton, D. "The V Kernel: A Software Base for Distributed Systems". *IEEE Software 1*, 2 (April 1984).

**5.** Gancarz, M. UWM: A User Interface for X Windows. Summer Conference Proceedings, Atlanta, USENIX Association, 1986.

**6.** Gettys, J. Problems Implementing Window Systems in Unix. Winter Conference Proceedings, Denver, USENIX Association, 1986.

**7.** Gosling, J. and Rosenthal, D. A Window-Manager for Bitmapped Displays and Unix. In *Methodology of Window-Managers*, F. R. A. Hopgood et al, Eds., Springer-Verlag, 1986.

**8.** Hawley, M. J., and Leffler, S. J. Windows for Unix at Lucasfilm. Summer Conference Proceedings, Portland, USENIX Association, 1985.

**9.** *Information Processing: Graphical Kernel System (GKS) - Functional Description.* DIS 7942, International Standards Organisation, 1982.

**10.** Lantz, K.A. and Nowicki, W. I. "Structured Graphics for Distributed Systems". *ACM Transactions on Graphics 3*, 1 (Jan. 1984).

**11.** Levy, H. "VAXstation: A General-Purpose Raster Graphics Architecture". *ACM Transactions on Graphics 3*, 1 (Jan. 1984).

**12.** Lipkie, D. E., Evans, S. R., Newlin, J. K., and Weissman, R. L. "Star Grapics: An Object-Oriented Implementation". *Computer Graphics 16*, 3 (July 1982).

**13.** Liskov, B. and Scheifler, R. "Guardians and Actions: Linquistic Support for Robust, Distributed Programs". *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983).

**14.** *???* MassComp, Inc., ???

**15.** Moon, D. Chaosnet. AI Memo 628, MIT Artificial Intelligence Laboratory, June, 1981.

**16.** Morris, J., et al. "Andrew: A Distributed Personal Computing Environment". *Communications of the ACM 29*, 3 (March 1986).

**17.** Myers, B. Issues in Window Management Design and Implementation. In *Methodology of Window-Managers*, F. R. A. Hopgood et al, Eds., Springer-Verlag, 1986.

**18.** Nowicki, W. *Partitioning of Function in a Distributed Graphics System*. Ph.D. Th., Stanford University, Stanford, CA, 1985.

**19.** *???* ICL, ???

**20.** Pike, R. "The Blit: A Multiplexed Graphics Terminal". *AT&T Bell Laboratories Technical Journal 63*, 8 (Oct. 1984).

**21.** Postel, J. Transmission Control Protocol. RFC 793, USC/Information Sciences Institute, Sept., 1981.

**22.** Adobe Systems Inc.. *PostScript Lanuage Reference Manual.* Addison-Wesley, 1985.

**23.** Rhodes, R., Haeberli, P, and Hickman, K. Mex - A Window Manager for the IRIS. Summer Conference Proceedings, Portland, USENIX Association, 1985.

**24.** Smith, D. C., et al. The Star User Interface: An Overview. Proceedings of the National Computer Conference, Houston, June, 1982.

**25.** Stallman, R., Moon, D., and Weinreb, D. Lisp Machine Window System Manual. MIT Artificial Intelligence Laboratory, Aug., 1983.

**26.** Steele, G. L.. *Common Lisp: The Language.* Digital Press, 1984.

**27.** *NeWS Preliminary Technical Overview.* Sun Microsystems, Inc., 1986.

**28.** *Programmer's Reference Manual for SunWindows.* Sun Microsystems, Inc., 1985.

**29.** Sweet, R. "Mesa Programming Environment". *ACM SIGPLAN Notices 20*, 7 (July 1985).

**30.** Sweetman, D. A Modular Window System for Unix. In *Methodology of Window-Managers*, F. R. A. Hopgood et al, Eds., Springer-Verlag, 1986.

**31.** *Programming the User Interface.* Symbolics, Inc., 1986.

**32.** Teitelman, W. The Cedar Programming Environment: A Midterm Report and Examination. CSL 83-11, Xerox PARC, June, 1984.

**33.** Trammel, R. D. A Capability Based Hierarchic Architecture for Unix Window Management. Summer Conference Proceedings, Portland, USENIX Association, 1985.

**34.** *VCB02 Video Subsystem Technical Manual.* Educational Services, Digital Equipment Corporation, 1986. Order Number AZ-GLGAB-MN.

**35.** Warnock, J., and Wyatt, D. K. "A Device Independent Graphics Imaging Model for Use with Raster Devices". *Computer Graphics 16*, 3 (1982).

**36.** Wecker, S. "DNA: The Digital Network Architecture". *IEEE Transactions on Communications CIM-28*, 4 (April 1980).

**37.** Wilkes, A. J., et. al. "The Rainbow Workstation". *The Computer Journal 27*, 2 (May 1984).

# Table of Contents

# List of Figures