The X Window System
Version 11*

James Gettys†, Philip L. Karlton‡, and Scott McGregor§

Digital Equipment Corporation
Silicon Graphics Computer Systems

CRL 90/08 10 December 1990

Abstract

The X Window System¶ has become widely accepted by many manufacturers. X provides network transparent access to display servers, allowing local and remote client programs to access a user's display. X is used on high performance workstation displays as well as terminals, and client programs run on everything from micro to super computers.

This paper describes the tradeoffs and basic design decisions made during the design of X Version 11. We presume familiarity with the paper describing X Version 10.

Keywords: X Window System, interactive human-computer interface system, distributed systems.

¶The X Window System is a Massachusetts Institute of Technology trademark.

*This paper will appear in a special issue of *Software Practice and Experience.*

†Digital Equipment Corporation, Cambridge Research Lab, One Kendall Square, Bldg. 700, Cambridge, MA 02139, U.S.A. jg@crl.dec.com

‡Silicon Graphics Computer Systems, Systems Software Division, 2011 N. Shoreline Boulevard, Mountain View, CA 94039-7311, U.S.A. karlton@sgi.com

§Formerly at Digital Equipment Corporation; now at The Santa Cruz Operation, 400 Encinal Street, P.O. Box 1900, Santa Cruz, CA 95061, U.S.A. mcgregor@sco.com

The X Window System

Version 11

JAMES GETTYS
Digital Equipment Corporation, Cambridge Research Laboratory
One Kendall Square, Bld 700
Cambridge, MA 02139, U.S.A.
jg@crl.dec.com

PHILIP L. KARLTON
Silicon Graphics Computer Systems, Systems Software Division, 2011 N. Shoreline Boulevard
Mountain View, CA 94039-7311, U.S.A.
karlton@sgi.com

SCOTT MCGREGOR[1]
Digital Equipment Corporation
mcgregor@sco.com

SUMMARY

The X Window System[2] has become widely accepted by many manufacturers. X provides network transparent access to display servers, allowing local and remote client programs to access a user's display. X is used on high performance workstation displays as well as terminals, and client programs run on everything from micro to super computers.

This paper describes the tradeoffs and basic design decisions made during the design of X Version 11. We presume familiarity with the paper describing X Version 10[1].

KEY WORDS X Window System Interactive Human-computer interface system Distributed Systems

[1] Now at The Santa Cruz Operation, 400 Encinal Street, P.O. Box 1900, Santa Cruz, California 95061

[2] The X Window System is a trademark of the Massachusetts Institute of Technology.

Contents

Introduction

The X Window System has become widely used over the last several years.

After the release of Version 10 of X (hereafter called X10) from MIT in the fall of 1985, and its release as a product under Ultrix[1] the following January, many people at several corporations and universities requested extensions to X to support their favorite application or hardware. X10 had been limited to what several people could implement, and these limitations were typically well known (and even documented). For example, X10 could only support displays with no more than 16 bits/pixel, and this limitation was inherent in many areas of the wire protocol.

Design of Version 11 started late in the spring of 1986. A larger set of contributors added expertise and experience in areas where it had been lacking in X10. The design was circulated for public comment and review that summer. A sample implementation of the protocol design started that fall, and became available in the fall of 1987. Feedback from reviewers and alpha and beta testers resulted in significant design changes to the core protocol, and deletions of some functions which were found to be poorly designed. (These functions were left to future extensions to X11). A brief history of events is available in The X Window System[2].

The X11 protocol was designed with little idea of how it would be implemented and was fully specified before the implementation began. It is of course true, however, that if we did not understand how to implement something in a reasonable amount of time and effort (since timeliness was critical) we did not add it to the design; for example, non-rectangular windows have been added as an extension since the original release. We did not understand at the time how easy they would be to implement, and therefore explicitly rejected them during the design meeting. The specification was changed during alpha and beta test as we learned from the implementation; often errors in the specification or design flaws were uncovered as the implementation proceeded. We are very skeptical of systems that have never been implemented before widespread adoption; similarly, systems that have not been carefully specified before implementation begins are also suspect.

Somewhat after the base window system work, other groups began design of a number of toolkits and window managers for X.[4,30] Feedback from these groups was particularly important to the base window system design.

In this paper, X10 and X11 are used to refer to specific X versions, when we must distinguish between them.

[1] Ultrix is a trademark of Digital Equipment Corporation.

Goals of Version 11 Design

"The only thing worse than generalizing from a single example is generalizing from no examples at all." - Phil Karlton

We realized at the first design meeting that it was impossible to design a protocol that was both upward compatible with X10 and able to support the different kinds of display hardware we could foresee. Additionally, we wished to support many window management policies that X10 could not support, such as window "decoration" and tiling. This fact and the long "wish list" required a complete redesign of X. The major goals for X11 included:

- Redesign of the basic protocol encoding to increase efficiency and reduce server round trips
- Support for deep frame buffers (more than 16 bits per pixel)
- Support for a wider variety of color maps
- Clean basic pixel-oriented graphics
- Text capabilities for WYSIWYG editors
- Clean up and generalization of facilities (a laundry list from X10)
- Extensible protocol
- Facilities for clients to exchange information (cut and paste of arbitrary information)

At the first design meeting, we decided that at initial release the X11 core protocol would NOT support:

- Three dimensional graphics
- Sub-pixel graphics and anti-aliasing
- World coordinate systems
- Video, speech, high-fidelity audio and related topics

Some capabilities were originally included in the core protocol, but either during design or implementation were put off for future work. These were:

- Alternate input devices, now a proposed extension
- Rotated text and graphics, now left to either a PostScript* extension in some vendor's implementations, or to a possible future extension to graphics contexts

3-D graphics, while important, would have delayed Version 11 by at least a year, and an interesting set of current applications do not require it. Political controversy over the "correct" 3D graphics design has delayed 3D greatly, and might have prevented X11 altogether. Instead, we decided that a general extension facility was a better strategy to allow addition of 3D and other facilities at a later date. History (seen from four years later, with the 3D[23] extension yet to be standardized) shows that it was a correct decision.

*PostScript® is a trademark of Adobe Systems, Incorporated

The graphics model needed to be clean but minimal. At current and near-term future display resolutions, we felt it was essential (from experience with some CAD applications) that applications be able to know precisely which pixels are modified by a graphics operation. We therefore deliberately rejected graphics that are independent of screen resolution.

The long list of potential uses of X we could foresee, as well as those we could not foresee, meant that the system must be easily extended to support other imaging models (3D, PostScript[1]), and other basic facilities (video, input devices).

Lessons Learned from Previous X Versions

Experience during X's development drove home the cost of round trip messages between clients and the X server. With X11, steps were taken to avoid such interactions. These were:

- The initial connection to the server fetches commonly-used information about the server and screens.
- Resource ID's (for example, windows and pixmap ID's) are allocated by the client.
- A client can use pixels for window borders and backgrounds, rather than being restricted to pixmaps.

After establishing a connection, every X10 program immediately needed information about the screen (commonly, the screen's size), causing one or more round trip queries to the server, and worse, these queries often were embedded in library functions and executed more than once since Xlib did not remember the information. In X11 all information about the physical characteristics of the screens of a display is sent immediately when a connection is established, and maintained by the X11 library, and is thereafter available locally to applications.

Resource ID's were allocated in the server when resources were created in X10, and returned to the client as part of a synchronous request. The first toolkits built for X10[2,3] exposed how expensive synchronous resource creation could be, as applications began to create many windows and other resources. In X11, as part of the connection setup message, a client is allocated a range of resource ID's, allowing a client to allocate a resource ID directly and makes all resource creation requests non-blocking calls.

X10 also required pixmaps for all backgrounds and borders of windows, even when they were solid colors, again resulting in more round trips for window creation and excessive consumption of server resources.

Clients external to the base system provide much of the user interface to the window system and is possibly the most unusual part of X. Previous versions of X had shown us how valuable

[1] PostScript is a trademark of Adobe Systems Incorporated

external window management was, though certain styles of window management (for example, tiling or decorating window managers) could not be built. X11 provides the facilities to build these styles of window management.

X11 introduces a new resource type: the GC (short for Graphics Context), which encapsulates the state required for graphics operations. Previous versions of X had relied on state-free graphics derived from the Digital VAXstation 100* graphics system: every graphics request provided a complete list of parameters. Extending X10's model to include the more sophisticated graphics we needed would have required greatly expanding the number of arguments to graphics requests, incurring transport overhead and interface complexity.

A state-based graphics model reduces the resource validation overhead of graphics requests, as well as the transport overhead (by reducing the size of a graphics request) and complexity of the interface. We considered making the window and graphics state part of the connection, or the graphics context part of the window. These options would have reduced the overhead of many painting calls still further, but would have complicated applications in multi-threaded environments or enlarged the window structure greatly. X11 therefore specifies both a window and a GC in every painting request, which is quite compact and allows for easy multi-threaded use of a single connection to the server.

All data transmitted by the protocol is kept naturally aligned for 32 bit architectures as in X10, making it easier to port the server and X library. It also increases efficiency since even architectures which allow access to unaligned data (like the VAX*) may exact substantial performance penalties. X10 had a fixed size basic request which resulted in unused bytes being transmitted from clients to the server for many requests. X11 uses variable length requests, sending no more bytes than necessary for the request (given the 4 byte rounding requirement). The minimum request size in X11 therefore is four bytes, in contrast to X10's 24 bytes, resulting in substantial speedup in X11. Transport was seldom a performance bottleneck in X10, but with faster servers and X11's more complete graphics model the fixed overhead would have become more significant.

The X10 wire protocol was also awkward to interpret. The server had to look inside of a fixed length request to see if there was additional data for the request, making it difficult to know when an entire request had be received. Request dispatching is the main loop of the X server, performance here is critical. X11's wire protocol encoding avoids the problem. Sixteen bits of each request (in a fixed location) specifies the length (in 32 bit quantities) of the entire request, so no knowledge of the protocol is required to know when a request is complete, and

* VAX and VAXstation are trademarks of Digital Equipment Corporation.

allows the protocol to be extended easily, as the dispatcher needs no knowledge of specific X request formats.

Display Support

Supporting current and future display hardware was X11's most important design goal. X10 expected at most one colormap on a single screen, in which the entire pixel of at most 16 bits indexed into a table to get values to drive the CRT. X11 supports up to 32 bit pixels, multiple interpretations of pixel data (called visuals, discussed in further detail below), multiple colormaps and multiple screens, including hardware with true color support, which often has 24 bits/pixel. Many displays have more sophisticated colormap hardware as well. They may allow clients to regard them both as pseudo-color devices (writable colormap) and as true color displays (8 bits each of red, green and blue), and sometimes include more than one lookup table, which may or may not be available simultaneously. Multiple screens may be driven by a single server, and different screens may have quite different capabilities.

While the core X11 protocol does not define 3D requests, we knew that a 3D extension would be necessary and that X must be usable on very high-end 3D displays, some of which have as many as 200 bits per pixel (though they do not use more than 32 bits/pixel for storage of color information, using the additional bits for double buffering, Z and/or alpha channel information).

Some display hardware allows multiple interpretations of pixel data, often on a per-pixel basis. For example, a few bits of each pixel may determine which colormap is used when interpreting each pixel and the colormaps may be of completely different types. X11 carefully separates the specification of how a pixel is interpreted from the pixel's color data. These different ways of dealing with pixel interpretation on a screen are called visuals. Some visual types may only be usable at certain depths of the display. For example, a 24 bit true color display might also support an 8 bit deep pseudo-color colormap. For each screen of the display, there is a list of valid visual types supported at different depths of the screen; because default visual types are defined for each screen, most simple applications need not deal with the complexity of many visual types.

Visual types are *StaticGray*, *GrayScale*, *StaticColor*, *TrueColor*, *PseudoColor* and *DirectColor*. The following concepts may serve to make the explanation of visual types clearer. The screen can be color or gray scale and can have a colormap that is writable or read-only, and the colormap can be a single table or 3 tables (one for red, green and blue), leading to the following diagram:

	Color R/O	Color R/W	Gray Scale R/O	Gray Scale R/W
Undecomposed Colormap	Static Color	Pseudo Color	Static Gray	Gray Scale
Decomposed Colormap	True Color	Direct Color		

Conceptually, as each pixel is read out of video memory for display on the screen, goes through a look-up stage by indexing into a colormap. Colormaps can be manipulated arbitrarily on some hardware, in limited ways on other hardware, and not at all on other hardware (A colormap that provides the identity transform and is read-only may not actually exist in hardware). The visual types affect the colormap and the RGB values in the following ways:

- For *PseudoColor*, a pixel value indexes a colormap to produce independent RGB values, and the red, green and blue values can be individually changed.
- *GrayScale* is treated the same way as PseudoColor except that the primary color that drives the screen is undefined. Thus, the client must always store the same value for red, green, and blue in the colormaps.
- For *DirectColor*, a pixel value is decomposed into separate RGB subfields, and each subfield separately indexes the colormap for the corresponding value. The RGB values can be changed dynamically.
- *TrueColor* is treated the same way as DirectColor except that the colormap has predefined, read-only RGB values. These RGB values are server-dependent but provide linear or near-linear ramps in each primary.
- *StaticColor* is treated the same way as PseudoColor except that the colormap has predefined, read-only, server-dependent RGB values.
- *StaticGray* is treated the same way as StaticColor except that the RGB values are equal for any single pixel value, thus resulting in shades of gray. A monochrome display can be thought of as a StaticGray visual with a two-entry colormap.

A single X11 server (sometimes called a display in X terminology) may support several screens that share a set of input devices. Screens can be of entirely different types: one screen might be monochrome and a second screen color, for example. X11 allows off screen pixmaps and backing store support, which may be limited by a given hardware implementation to finite resources. However, the X11 core protocol does not allow moving windows from one screen to another and operations between drawables on different screens; we judged the complexity too great to require this ability. There is no way to guarantee that an operation could be performed between screens, because the resources required can not be determined in advance. If two physical screens are similar enough, a server implementors may provide a single double-width (or double-height) screen rather than two separate screens, or an extension can be defined to allow this operation between similar enough screens.

X supports two basic forms of color map entries. Read-only color map entries may be shared between clients; read/write entries are typically allocated for exclusive use of a single client. X encourages applications to share colors out of a default color map. X11 introduces multiple color maps for applications that cannot live sharing color resources with other applications. If an allocation in the default map fails, a client can copy the entries it allocated to a new color map of the same visual type and free those entries out of the original color map. An additional allocation primitive was added for allocation in a segmented color map.

Windows

X10 supported transparent and normal windows. X11 introduces window classes, in part to support extensions, though this capability has not yet been used. The X11 core protocol supports two window classes: *input-output* and *input-only* windows. Input-only windows differ from X10 transparent windows in that graphics output is explicitly disallowed. Input-only windows can have cursors and behave as normal windows for input dispatch operations, but do not clip the windows behind them which differs somewhat from X10 transparent windows. For example, menus can be implemented as a set of input only windows with highlighting keyed off of window enter/exit events; this approach worked very well in X10.

Borders and backgrounds of windows in X11 can be specified using either a pixel value or a pixmap. In X10, all window backgrounds and borders had to be specified as pixmaps implying that the pixmaps must exist. The most common case observed, however, was that backgrounds or borders are solid colors, so the overhead of creating these pixmaps was a waste and slowed application startup. In X11 windows need not have a background, and the border width can be zero (allowing applications that do not want them to ignore borders) allowing applications which know they will paint a window entirely on exposure to avoid the background fill operation, which is useless and may cause flicker. Backgrounds can also be inherited from the parent window, and if their parent window's background is a pixmap, the background will be correctly painted with the same tile origin.

When windows are resized, two issues emerge: what to do with the contents of the window, and what to do with any subwindows of the window. X10 always threw away the contents of windows and left subwindows in their original location. As a result, subwindows were often in the wrong location when their parents were resized causing applications to repaint twice, once in the wrong location, followed by a repaint in the correct location after repositioning subwindows. Applications had to page in the code to repaint all subwindows, causing more paging activity, which exacerbated performance problems.

To solve this problem present in X10 and many other window system, X11 introduces the concepts of *bit gravity* and *window gravity* hints. Bit gravity allows applications to indicate where the existing contents of a window should be after resize, and was invented by Bob Ayers and Phil Karlton as part of the Xerox Development Environment[10]. Window gravity, original to X11, allows applications to indicate where subwindows should be moved to when their parent is resized. Gravities can be any of *center, north, south, east, west, northeast, northwest, southeast, and southwest*. An application can also ask that window contents always be thrown away and the background repainted (*ForgetGravity*) or remain at the same location relative to the root window (*StaticGravity*), which, for example, is useful for cropping images when dragging a window border. An application can ask for subwindows to be unmapped (*UnmapGravity*) when their parent is resized.

X has always taken and continues to take the position that clients are responsible for the contents of windows. X10 implementations never maintained backing store. Backing store is difficult to provide on some hardware and is usually slower than asking applications to regenerate the contents of windows, even on displays as simple as frame buffers. To provide backing store is impossible in the general case: even virtual memory will run out eventually, and X encourages the use of many windows. It was also a requirement that the same protocol be useful to a low-end (possibly non-paging) X server as well as high-end workstations, implying that backing store and save unders could only be hints, and not required, due to the memory requirements backing store and save-unders impose.

X11 applications can now provide *backing store* or *save-under* hints to the window system server (which may be ignored). The backing store hint informs the server that the window is computationally difficult to regenerate, and that preserving the contents of the window when it is obscured would be beneficial. The save-under hint informs the window system that saving the pixels under the window may be beneficial (for example, for pop-up menus).

As the name implies, the server may or may not honor hints and may stop honoring the hints at any time. Implementations which honor these requests may reduce the number of exposure events sent to clients and may prevent inactive applications from having to be paged in when windows are moved, but comes at a cost in memory itself.

Since backing store memory available for a window can be very precious, particularly on a deep display, there are also window attributes that inform the server which planes of the screen need to be saved in backing store, along with a pixel value base so that the window can be regenerated when exposed. Most current implementations and clients do not yet take advantage of this optimization, though we expect that some will in the future, particularly on deep displays with applications performing plane oriented colormap allocations. Only the planes of the window actually in use by the application need be saved.

Windows also have a *property list*, where arbitrary data can be stored associated with a window, discussed in a later section.

Each window has a color map associated with it; clients can change the color map of a window as long as the visual type allows it.

At the time of the protocol design meetings, we only had the X10 server design to go by. As the list of new functionality grew long, we decided to be conservative and only support rectangular shape windows in the core X11 protocol. In the X10 server implementation, non-rectangular windows would have been quite difficult to implement. We knew from experience that rectangular windows were sufficient for essentially all applications. There are a significant number of applications, however, which need to confine graphics output to non-rectangular regions. As a result, the core X11 protocol supports clip regions in GC's.

More recently, arbitrary shaped windows windows were added to the X11 sample server implementation as an extension; initial implementation time was approximately four hours. Had we realized the implementation would be that easy, we would have specified non-rectangular windows in the core protocol. In retrospect, this is understandable: the requirement for region-based graphics had forced the sample server's clipping algorithms in the sample server to also be region-based. We expect that non-rectangular windows are mostly useful for certain kinds of input control, and for situations in which application writers want to ensure that arbitrary areas on the screen are at the same stacking order when manipulated; they have also been used to implement rounded buttons.

X10 always raised a window to the top of the stack when the window was manipulated which was quite irritating under some circumstances. X11 does not change the window's stacking order when a window is resized or moved but provides separate primitives for controlling the relative stacking order of windows.

X windows include a border, which can be either a pixel or a pixmap. Borders are somewhat contrary to the general philosophy of windows in X, where they are very light-weight objects by comparison with windows in many other systems.

We seriously discussed removing window borders and backgrounds from X11. Given backgrounds, one could synthesize borders by two nested windows. Backgrounds have indeed proved themselves before and since, but borders are more contentious. Against borders are the observation that borders increase window size, increase server complexity, and complicate coordinate systems. But many windows need some visible edge and borders are therefore a simple treatment of a common case. This permits users to see windows enough to "mouse ahead" even if the application is paged out or otherwise busy. Removing borders would have made the conversion to X11 more difficult. Any memory savings in the size of window structures would be outweighed by the additional window structures in the server presuming borders are implemented as two windows (so windows would be visible for mouse-ahead). In the end, we decided to retain window borders, and we continue arguing among ourselves (sometimes heatedly).

Atoms, Property Lists and Selections

X10 had protocol requests for associating a few specific pieces of data with a window. Applications and window managers used this data to communicate with each other. It was clear that there was insufficient information to support all possible window managers.

X10 had a collection of requests which would attach some window management information (name, desired geometry, etc.) to windows, or allow information to be stored on windows to allow interchange of data between applications. Any finite number of requests would have been inadequate as new window managers were written. Further, applications needed their own information channels between one another.

The ad-hoc collection of facilities was ripe for replacement by the much more general facilities found in X11. In addition, un-typed cut buffers are really inadequate for data exchange between applications. Several of us were familiar with the selection mechanism first developed at Xerox, which allows polymorphic exchange of type information; we very much wanted this kind of facility for X11, in addition to cut buffers (for backward compatibility). No other mechanisms were seriously considered. X11 has a single simple, general, mechanism called Properties for associating data with windows; with the selection mechanisms much more powerful general communication can take place.

Though is possible to define an arbitrary inter-client communication protocol on top of the mechanisms that X11 provides, it is not an appropriate use of these facilities. The X protocol is not an efficient RPC mechanism or a good way to transfer a 10 megabyte file. Much better, general facilities such as NCS[4] are becoming available and should be used. The intent of the facilities X11 provides is to enable X applications to interchange relatively small volumes of typed data conveniently. We suggest that applications exchanging large volumes of data arrange a rendezvous using X, but then use other mechanisms more suited and more efficient for large data transport. It is a failing of the ICCCM that this point of view is not encouraged more.

Atoms

The inter-client communication facilities in X11 use *Atoms*. At the conceptual level, Atoms are unique names. They can be thought of as a bundle of bytes, like strings, but shorter. We felt that passing sequences of bytes across the wire would be too costly. Further, implementation is easier if events as they appear "on the wire" have a fixed size (in fact, 32 bytes), and since some events contain Atoms a fixed-size representation for them was needed. The X11 protocol request InternAtom registers a byte sequence with the server, which returns a unique 29-bit value[1] (unique over the server's lifetime) that maps to the byte sequence. The inverse operator is also available (GetAtomName).

The protocol specifies a number of predefined Atoms. They are an implementation trick to avoid the cost of Interning many atoms that are expected to be used during the startup phase of all applications.

Toolkits or the X language libraries should cache atom-name mappings and call InternAtom only when required; in addition, they may want to batch InternAtom calls in the library, to reduce server round trips.. This has proved to be a significant performance problem; such caching in Xlib is very likely to appear in release 5 of the MIT X distribution. The common lisp (CLX[5]) interface, for instance, makes no distinction between predefined atoms and other atoms; all atoms are viewed as symbols at the interface. However, a CLX implementation typically keeps a symbol/atom cache, which it initializes with the predefined atoms.

Properties

Each window has an associated property list which may be empty. The server itself makes no semantic use of the value of or the existence of properties. It does, of course, mediate their use. Property lists are somewhat like those used in LISP. A property consists of a name, a type, a data format, and some data. Each property is named by an atom, and its type is named by an additional atom. At most one property of any given name, independent of type, can be associated with a window. Properties can be changed, read, and deleted after they are created. The list of properties on a window can be listed. Data stored on window properties is reclaimed when the window is destroyed, not when the creator/editor of the property goes away.

When the value of a property changes or is deleted, an event of type PropertyChange is sent to any interested client, permitting clients to monitor changes to properties (for example, the name of windows, or other window manager hints).

[1] All ID's in X are 29 bits (the high order three bits must be zero) which eases their use in tagged language environments such as lisp and CLU, where a few bits may be used for basic type information.

It would have been desirable to be able to describe an arbitrary data structure within a property. Unfortunately, that would have necessitated inventing yet another mechanism, actually a language, for record declaration. Coupled with the need for byte swapping when interchanging data between some machine architectures, this seemed like too much complexity for us to undertake. Instead all of the data within a single property is an array with elements of width 8 bits, 16 bits or 32 bits. Client routines must explicitly pack and unpack their data structures into and out of properties. (There are several "built in" properties along with their associated types, and Xlib does the packing and unpacking).

It was an explicit design goal of X11 to specify mechanism rather than policy. As a result, a client that converses with the server may operate "correctly" in isolation but not coexist properly with others sharing the same server. The Inter-Client Communication Conventions Manual (ICCCM[2, 36]), adopted as a standard after X11's initial release, defines a set of conventions and standard properties to allow clients to cooperate in areas of selections, cut buffers, window management, session management and other resources. As the X community grows in size and experience, we expect that additional "standard" property types will be defined.

Selections

X10 only supported cut and paste buffers. X11 provides a much more general mechanism, called a selection, which was first implemented in window systems at Xerox. A *selection* can be thought of as an indirectly addressed property value with dynamic type. That is the property is not stored in the server, but is maintained by some client (the *owner*). A selection is global in nature: it belongs to the user (though maintained by clients), rather than being private to a particular window subhierarchy or a particular set of clients. There can be an arbitrary number of selections, each named by an atom.

When a client asks for the value of a selection, it specifies a *selection target type*. The target type can be used to control the transmitted representation of the value. For example, if the selection is an image, then the target type might specify whether the image should be sent in XYFormat or ZFormat. The target type can also be used to control the class of values transmitted; e.g., asking for the "looks" (fonts, line spacing, indentation, etc.) of a paragraph selection, rather than the text of the paragraph.

The discussion below is only an outline; for full details, see the ICCCM.

Two clients exchange data using selections in a sequence which starts when a client asserts ownership by setting the owner to a window that it created (typically because the user selected some data he wants to use in some other client). The other client can then request the selection be converted to some *target* (which may not be the same as the desired type) typically when the user copies the selection into the application.

The selection owner converts its representation of the selected material to the type dictated indirectly through the target specified by the requester, and stores it as a property on the window specified in the request. (Typically it is one of the requester's windows.) The owner then sends a SelectionNotify event to the requester letting it know that the conversion is completed.

If there is no owner of that selection the server sends SelectionNotify with property=None, meaning that the requester can expect an answer in a reasonable time frame, since either the server will respond or the conforming owner will respond. For values that are too large to be stored in the server, handshaking is used to copy smaller fragments.

By using the time stamp from the last event from the user race conditions can be prevented, since getting ownership of a selection or requesting conversion happens as the result of a user action.

There can be an arbitrary number of selections, each named by an atom. To conform with the inter-client conventions, however, clients need deal with only three selections: *Primary, Secondary* and *Clipboard.* Other selections may be used freely for private communication among related groups of clients. Clipboards present a completely different style of user interface, using the same primitives. We will illustrate the use of selections here by discussing Primary and Secondary selections only; see the ICCCM for details of Clipboard.

The selection named by the atom *Primary* is used for all commands which take only a single argument. It is the principal means of communication between clients which use the selection mechanism.

The selection named by the atom *Secondary* is used:

- As the second argument to commands taking two arguments, for example "exchange primary and secondary selections".
- As a means of obtaining data when there is a primary selection, and the user does not wish to disturb it.

The selection named by the atom *Clipboard* is used to hold data being transferred between clients, normally being "cut" or "copied" and then "pasted."

Primary

Many commands take but a single argument; the user can merely make a selection and then execute the command that applies to it. It is important that the command invoked in some application not care the selection is in the same application, implying that the selection must be global. Since the selection is usually highlighted in some manner, it is necessary for the old selection to be un-highlighted whenever a new selection is made. X11 automatically sends events to clients to inform them when they lose a selection.

Not all functions want to know the same thing about the selection. Some might want the text (the copy command); some might be more interested in the font or face of the selection (the look-like-that command); etc. To avoid the need to enumerate all the possible selection targets, the X11 is built with an open ended set of target types. Clients have to be prepared for the possibility that the owner will be unable to convert the selection to the target requested. At a minimum, most owners can deliver the selection as text.

A simple example: you receive a mail message containing the name of a file and you decide you would like to view it in some file display application. Selecting the file name and invoking the "Load" function in the application should be sufficient, rather than requiring the user to bring up a file selection dialog box and copy the file name into the text field of it and then invoke the open function. If there is no appropriate selection when a load function was invoked, then a file selection dialog box would be opened.

A slightly more complicated example: you have a debugger window and a generic file display/editing window. Imagine are looking at some file in the displayer, hunting for the bug. You select some word in the file, move the mouse into the debugger window and invoke the "Set Breakpoint" command button. The debugger and the file displayer are not in the same address space, (they might not even be on the same machine.), and don't know much about each other. They have agreed, however, on the semantics of some selection targets. The debugger then asks the current selection "What is the value as a 'FileName'?" and gets back "/udir/karlton/hacks/selection.c"; it asks for the value of the selection as a 'LineNumber'; gets back 93. The debugger can then set the break point at the correct point.

Similar kinds of cooperation would make using a mail system easier. If most objects had a notion of an author/owner, then when a mail composition tool, for instance, were invoked, it could get the author and initialize the form correctly. There is no need to build a mailer into a bulletin board reader, when the mailer can ask the bulletin board reader for the information it needs when it needs it. The information might include the author/owner of a message being replied to, the name and position of the file containing the bulletin board information, the information of the message itself, in whatever mutually agreeable form is convenient, and so on.

Secondary

A single global selection is insufficient in some user interface models, which may need the notion of the "current" selection yet also need to be able to communicate other information between applications, using the same request-reply model of the primary selection.

For example the user may want to swap the value of two items on the screen. He can make primary and secondary selections and then press a swap button.

Another case is when the user is entering new text (in some user interfaces, the current selection is reduced to merely be the location of the caret when the user is entering new text) and sees some desirable text some place else just waiting to be grabbed and inserted. One user interface for accomplishing this task is to make a secondary selection in some manner (for example, holding down a shift key while using the mouse buttons) and then letting up on the shift key when the proper secondary-selection has been made. Note that in this case that the primary selection is unaffected by the manipulation.

Selections are fundamentally different from cut buffers, since cut buffers have no way for clients to negotiate the type of the interchanged information. On a limited operating system environment such as MS-DOS* or MAC/OS, only a single application can run at a time, so it is not possible for two applications to negotiate the "greatest common multiple" data representation in common between applications. Instead, such systems must take a much more pessimistic view of data interchange, which has sometimes been promoted as a good idea, when it is merely expediency caused by the limited execution environment.

Selections do increase the burden on application writers to some extent, but represent a major improvement for users of the system. Much of the additional work required is generally encapsulated in the toolkit the application writer uses, so it is not usually a great burden on writers of most applications.

Events

X events inform the client of external events, caused either directly by the user (mouse, key/button, enter/leave events) or as a side effect of a request by a client (expose, resize, map, etc.). X11 has many more event types that X10 did; most of these are for support of toolkits and window managers, and typical programmers do not have to deal with them directly.

Device-related events propagate from the source window to ancestor windows until some client application has selected that event type or until the event is explicitly discarded. The X server generally sends an event to a client application only if a client has specifically asked to be informed of that event type.

Event reporting is enabled (with a few exceptions) by the window's event flag attribute, which can be set when a window is created or changed by explicit requests. X10 allowed only a single client to select for input. X11 allows more than one client to select most types of events (in effect, the event flag attribute is per client, per window, rather than just per window), solving various problems that the restriction caused in X10 (For example, X11 lets an applica-

*MS-DOS® is a registered trademark of Microsoft Corporation

tion capture all keystrokes to a display). Applications can suppress event propagation by use of the *Do Not Propagate* event mask, to prevent the server from generating unneeded events from child windows.

In contrast to X10, X11 allows multiple clients to select for the same events; the events will be broadcast to all clients who have asked. Only a single client can select for redirected events, however.

The table below summarizes X11's event types:

Event type:	Cause
MotionNotify	Pointer moved
EnterNotify, LeaveNotify	Pointer entered or left a window
KeyPress, KeyRelease ButtonPress, ButtonRelease	Keyboard key or pointer button state changed
FocusIn, FocusOut	Keyboard input focus state changed
KeymapNotify	Keyboard state at EnterNotify, FocusIn events
ColormapNotify	Colormap changed or (un)installed
Expose	Window exposed
GraphicsExpose, NoExpose	Exposure due to CopyArea
CirculateNotify	Window circulated
ConfigureNotify	Window reconfigured
GravityNotify	Change due to window gravity
MapNotify, UnmapNotify	Window (un)mapped
ReparentNotify	Window reparented
VisibilityNotify	Window visibility changed
CreateNotify, DestroyNotify	Window created or destroyed
ResizeRequest, CirculateRequest, ConfigureRequest MapRequest	Window state change requested
ClientMessage	Message sent from a (possibly different) client
PropertyNotify, SelectionClear SelectionRequest, SelectionNotify	Selection related events
MappingNotify	Keyboard keysym mapping changed

A few events are always sent to all clients. MappingNotify events are always sent to clients, and GraphicsExpose and NoExpose events are sent as a result of CopyPlane and CopyArea requests unless the client suppresses them by setting the graphics exposures in the GC to False.

Most events contain a small number of fixed fields: the event type, the serial number of the last request executed for this client, a bit which indicates whether the event is synthetic (i.e. was sent by another client), and a window ID. The window ID in the fixed part of the event is chosen to be that most useful for toolkit dispatching. The X11 library for C also adds a pointer to the display from which the event was received. Additional data specific to the event type is sent in each event.

Many events contain a timestamp indicating when the event occurred. There are several more events which should have contained a timestamp; the oversight may be rectified in a future (upward compatible) revision of the protocol. Timestamps are used to resolve race conditions that occur when multiple clients interact with a single server.

Clients normally keep track of their windows in some detail, since it is very expensive to query the server for information about windows. It is generally a mistake for a client to keep detailed geometry information about its window tree based on server events. Such a local database will often be out of date; many race conditions occur when windows are resized by external window managers, and it effectively makes a client synchronous with the server, which kills performance. In general, toolkits should assume that they have control of subwindows in their application, and should use structure notification events to know when their top-level windows have been resized. Of course, a toolkit must still ask the server for notification of focus changes, window crossing, and other such events.

Window Management Functions

X has always separated window management policy from the base window system. Typically, a single client, external to the X server provides the user interface for manipulating existing windows, using the facilities the server provides. These clients are called "window managers."

A number of different window managers were written for X10. These typically worked by clicking on a window with one or more modifier keys, or by clicking on a window provided by the window manager and then on the window to be manipulated. Some of these window managers are great for expert users. They typically require two hands to use, however, and are relatively unapproachable because there are no visible clues to help the user learn how to use the system. Experience with X10, however, had convinced us that external window managers are not only feasible, but highly desirable.

The X10 protocol precluded several important styles of user interface. On the Xerox Star[14], the Apple Macintosh*, and many other window systems, most operations to move, resize, raise,

* Macintosh is a trademark licensed to Apple Computer, Inc. Apple® and the Apple logo are registered trademarks of Apple Computer, Inc.

or lower an application window are accessed by mouse operations on window "decoration," i.e. clicking on the title or other visible appendages at the edge of the window. Some other systems like, the Andrew[7] system provide a tiled user interface in which application windows can not overlap. Some general mechanisms had to be found to allow these and other window managers.

In X10, the server immediately performed all requests that a window be mapped, resized, moved or raised. X11 adds facilities for window managers (or toolkits) to override any operation that would affect the placement of a window. When any such request is received an event is sent (the request is *redirected*) to the client that has requested such control. The controlling client can then enforce whatever placement policy it wants. Since pop-ups must respond immediately for good interactive feel, there is no time to involve a window manager. Each window therefore has an additional attribute called *override-redirect* which if set overrides redirection. Since there is no guarantee that a MapWindow request actually results in the window being mapped, redirection requires clients to be event-driven for repainting other than pop-up windows.

The redirect facilities in concert with the *reparent* operation allow a window manager to decorate windows: it can create a frame for a window, which may include title bars, resize boxes, and other visible cues to the user, and reparent the client's window into the frame, before the client's window is ever mapped to the screen. Events inside the frame but not inside the application's window will be dispatched to the window manager.

By convention, clients store "hints" on their windows to inform the window manager of their preference for size or location, as discussed above in the section on properties. Some window managers may choose not to honor these hints (for example a window manager which imposes a tiling user interface). Correct X clients are expected to do the best they can with whatever screen real-estate the window manager has provided, rather than fight the window manager.

Child windows are normally destroyed when their parents are destroyed, which may occur either directly via a DestroyWindow request or when a connection fails. A window manager can arrange, however, that if it should exit (possibly due to a failure on its part), and its frame windows are therefore destroyed, that other client's windows will be properly reparented and remapped, using the *save-set* facility in X11, providing for a more robust environment in the face of inevitable window manager failures. By adding to the window manager's save set the managed windows, these windows will be reparented when the window manager's connection closes. Reparenting window managers should use this facility to avoid accidental destruction of clients if the window manager dies.

The close down mode of a connection determines if resources outlive the connection that created them. For example, a user could use this to examine the window of a dead application, which would normally have been destroyed before the user could diagnose the problem.

Screen real-estate is not the only scarce resource that window managers need to be able to control. By keeping track of the creation of colormaps and their assignment to windows, a window manager can set policy on which colormaps are active at a given moment, based on the current input focus, pointer location, or other factors. In some user interface styles, keyboard input is sent to the window under the pointer; in others it is set to a specific window. Each of these has its advantages and disadvantages, and users often have very strong feelings on which they prefer. X11 contains facilities that allow a window manager (and therefore ultimately the user, through his choice of window manager) to control input focus.

X normally sends events to clients immediately, which is the correct default behavior, but it does not provide sufficient control over input processing when an application lags behind an experienced user. For example, some user interface styles may want the first click on a window to set the keyboard input focus, and also still deliver the event to the application. "Mouse ahead" on menus should work properly (even though the menu's window may not have been mapped yet). Many of these user interfaces depend upon synchronous delivery of events.

The basic mechanism controlling event delivery in X11 is called a "grab." There are two kinds of grabs: active and passive. When mouse buttons or keyboard keys are grabbed, events will be sent to the grabbing client rather than the client who would normally have received them. An active grab occurs when a single client grabs the keyboard and/or pointer explicitly.

A passive grab occurs when clients grab a particular keyboard key or pointer button in a window, and the grab will activate when the key or button is actually pressed (you might think of a passive grab as a trap, waiting to be sprung). If the keyboard or pointer is in synchronous mode, no further events are processed until the grabbing client allows them. The keyboard or pointer is considered *frozen* during this interval, and the X11 server postpones processing of subsequent events until told. The event that triggered the grab can also be replayed. Delivery of pointer and keyboard events can be controlled independently. Note that the logical state of a device (as seen by client applications) may lag the physical state if device event processing is frozen. Grabs activate on an outside in sequence (the largest containing window with a grab set), whereas normal event propagation is inside out (the event goes to the smallest enclosing window).

Various events may be generated as a side effect of a grab. For example, a FocusOut event will be sent to interested clients when a keyboard grab activates. The pointer cursor can also be confined to a window during a grab; for example, you can implement scroll bars in which the cursor will not leave the scroll area until the pointer button is released.

Passive grabs are convenient for implementing reliable pop-up menus. For example, you can guarantee that the pop-up is mapped before the up pointer button event occurs by grabbing a button requesting synchronous behavior. The down event will trigger the grab and freeze further event processing until you have the chance to map the pop-up window (typically with override-redirect set on the window). You can then allow further event processing, and in this example the up event would then be correctly processed relative to the pop-up window. See the companion paper by Gajewska et al.[22] for more detail and mistakes in the design of grabs.

The X11 server maintains the time when the input focus was last changed, when the keyboard was last grabbed, when the pointer was last grabbed, or when a selection was last changed. Your application may be slow reacting to an event (it may have been paged out, or be at the other end of a slow network link, for example). You often need some way to specify that your request should not be performed if another application has in the meanwhile taken control of the keyboard, pointer, or selection. By providing the timestamp from the event in the request for one of these operations, you can arrange that grabs and other requests not take effect if someone else has performed an operation in the meanwhile. This problem exists because multiple clients send requests to the server, and these requests are issued (but may not arrive!) in the logical sequence that user actions occur in. So much as possible, clients use timestamps from user events to let the server know the time-ordering the requests are logically in.

In addition to the general facilities discussed above, there are a set of less basic, but still important facilities discussed below.

On occasion a client may need to make a sequence of requests atomic, (for example, when editing a shared property) by grabbing the entire server. Server grabs should not span interactions with a user. Keyboard keys can also be grabbed like pointer keys, allowing the implementation of various sorts of help facilities which may want to bind such help to a keyboard key.

Window managers may also want to stop execution of a client. The KillClient request kills a client by naming one of their resources (typically a window); the server will then close the connection to that client without warning. The closing of the server connection informs the client that it should exit.

Window managers need to control which color maps are currently in use on the screen; for example, they may want to install colormaps based on which window owns the input focus. At any given time, some set of color maps may be installed simultaneously; the number of color maps is hardware dependent. X11 provides calls to install or uninstall color maps. Clients (typically window managers) can also ask to be informed when the color map of a window is changed, or when color maps are installed or uninstalled.

Clients may need to inform other clients of significant events. X11 events can be synthesized and sent to a client. The server adds a bit to the event to indicate the event was synthetic (for security reasons); the event is otherwise untouched.

Graphics

Previous X versions used a state free graphics interface derived from the VS100. A state free graphics interface could not gracefully extend to the level of functionality we needed, which included functionality from other window and graphics packages. We therefore completely redesigned the graphics interface. The hardest decisions were what to leave out; our primary goal was to create a design that fixed the more serious omissions of X10 and could be implemented within a reasonable amount of time. We settled on three other general design goals: simplicity, performance, and the definition of a precise drawing model. In general, we took Butler Lampson's sage advice[8]: Interfaces should be simple, they should be complete, and they should admit a sufficiently small and fast implementation. Do one thing well, don't generalize, get it right, don't hide power, and leave it to the client.

Though it was tempting to provide grandiose functionality, our goal of simplicity lead us to exclude 3D graphics, world coordinate systems (including sub-pixel positioning), and anything more than just basic image and text manipulation functions. Some of these were hard choices; for example, without sub-pixel positioning it's not possible to produce the best rendition on color or gray-scale displays.

We capitalized on experience from several rendering models in X11 including X10[1], PostScript[11], MS-Windows[12], the MacIntosh[13], Xerox workstations[8,9,10,14], and 3D CAD/CAM. From PostScript we took the line and font models. X10 and Xerox window systems gave us the rasterops and basic integer coordinates. The 3D graphics world gave us the concept of adjacent objects not overlapping pixels. While it may seem like from these sentences we combined everything, we carefully left out the PHIGS[15]/GKS[16] imaging model of display lists. With retrospect, some support for scaling of coordinate systems and 90, 180 and 270 degree rotations in the core protocol might have been a good idea and not too hard to implement; the current design is awkward to implement world coordinate systems in client code. This could be added by extending the GC in a clean fashion. 3D functionality was left to a later date.

Performance is the most important aspect of any graphics system. Many operations are required to display an application's graphic user interface, and the difference between fast and slow rendering determines whether techniques such as dragging images or editing of complex text and graphics are feasible. One performance technique that works quite well is to observe that applications that draw a line or a rectangle tend to draw another with high probability. By designing many of the common drawing primitives to operate on an array of objects instead of a single object, saves both network protocol overhead and set-up costs in the low-level drawing software and hardware. We call these "poly" routines and apply them to points, lines, rectangles, arcs, and text. Another performance technique allows for machine-specific renderings that might differ from the official semantics. Some graphics functions are so frequently used and performance so critical that there are special interfaces that overlapped with others in functionality. With hindsight, we should have made all graphics requests poly requests, including polygon, which is fertile ground for an X extension.

The X protocol is based on a bi-directional stream. Requests that generate a reply from the server can also be batched or streamed, if the client is willing to do the necessary book keeping. Therefore a library interface can generate multiple X requests and handle the resulting replies in a single server round trip. This ability to batch requests was exploited more heavily in X10 Xlib, where many more basic requests (for example, CreateWindow) required replies. As new facilities in X11 are now being more heavily exercised, we expect to add some additional library interfaces as needed for performance problems; at the time of this writing, for example, performance measurement shows InternAtom is causing many unnecessary round trips to the server on application startup which can be avoided both by caching in the Xlib and by such batching interfaces.

Some of the X10 functions needed attention, such as XDraw. XDraw drew a connected set of lines and curves, but didn't specify the algorithm used for curves other than they had to pass through the vertices! XDraw in particular prompted the religion around precise definition of algorithms. X11 defines exactly what pixels each graphics operation affects, for two reasons: it makes testing possible, and it makes applications more portable across different X implementations. Most graphic applications carefully craft their screen appearance -- lines just touch circles, menu outlines have exact pixel relationships with other screen areas, etc. When an application is run on different systems, it would hurt our portability goals if the application developer had to recode the user interface to restore the proper screen appearance. MS-Windows[1] and PostScript[11] solve the portability problem by providing a standard implementation, thereby achieving identical results across different vendors' platforms. Alternatively, the goal of the sample server in X11 was to provide a generic implementation with the expectation that vendors would rewrite significant portions of the server to get better performance or added functionality, requiring that the X11 protocol specification precisely define the rendering semantics

[1] MS-Windows[12] is a Trademark of Microsoft Incorporated

for each screen operation. Recently a test suite is being created that helps verify server conformance with the specification.

Graphics Operations

In addition to routines for displaying text and clearing and copying areas, X11 provides drawing routines for points, lines, rectangles, arcs, and polygons (but unfortunately, no poly-polygon call, which would have been very useful). Xlib provides two interfaces to these requests: for single objects and for an array of objects. The protocol provides the array form only. Xlib, however, combines adjacent single calls (if possible) into a single call on the "poly" routines, doing wonders for graphics benchmark performance for less than optimal programs, and helps real programs as well.

We decided not to include any curve functions in X11 other than elliptical arcs. This decision was hard because we all wanted splines, yet we could not reach consensus on which family of splines to include. Clients who want curves are encouraged to decompose the curves on the client side into connected line segments, or to lobby for a future spline extension.

The filled polygon algorithm we chose in X11 differs from systems such as the MacIntosh. The Macintosh's QuickDraw and most other PC graphics packages connect the polygon vertices with Bresenham lines and fill all the pixels up to and including the edges. We wanted a model that would support compositing functions and extend gracefully to 3D.

How abutting shapes paint present a very interesting issue, which can be summarized in two questions:

- Are there any gaps?
- Are there any overlaps?

In X11, the answers are no and no.

We believe that drawing two adjacent polygons should touch every pixel exactly once. For example, a pie chart using X11 would draw every pixel once where QuickDraw would draw a number of the edge pixels multiple times. Drawing a pixel multiple times isn't a serious issue for business graphics (although some glitches show up when using XOR), but it is fatal for color compositing, especially when a surface is divided into many small polygons (for example, by many CAD applications). The X11 protocol specification defines a filled polygon to include the pixels that are inside the mathematical bounds of line segments joining the vertices. A pixel is considered in the interior if the edge passes through the center and the pixel is to the right or below of the edge.

The X11 model for "wide" lines (i.e. with a width greater than a single pixel) is similar to the model for polygons. The line is defined by a path with a center on the line segment between the two endpoints, drawn with a line segment brush perpendicular to the line. This model generalizes to arcs, where the brush is perpendicular to the slope of the arc.

The X11 design is bit awkward with respect to lines. Our mathematical design is slower than the more popular Bresenham algorithm for single pixel lines, so we wanted to allow Bresenham for single pixels and the more general algorithm for wider lines. Our compromise is to borrow a trick from PostScript and create the notion of a zero-width line, but use it for performance rather than to draw a minimum width as PostScript does. A zero width line is drawn as fast as the server implementation can, and has loose guarantees about the exact pixels that will be affected. A width-one line is drawn per the general algorithm and will correctly abut lines of other widths. The client has the choice of performance or precision. While this compromise lacks elegance, it seems to be effective.

Among mistakes, wide arcs stand out. The problem with the specification showed up during the implementation of the test suite code, when it was already too late to amend the specification. The original server implementation didn't implement “wide" arcs to specification. The correct implementation produces "lumpy" or varying width curves, as a side affect of the wide line definition. The subsequently published polygonal pen algorithms described by John Hobby[17], suggest possible solutions to this problem. In addition, it is impossible to generate symmetric arcs or circles, since it is not possible to specify the center of the arc to a half pixel center, again showing the problems with our unwillingness to compromise on integer pixel representations.

X11 provides relatively minimal (but sufficient) facilities for imaging. Basic primitives include transferring a pixmap to the server and retrieving data from a window or pixmap. The core protocol does not attempt to support any compression or image processing algorithms. These requests have one important difference from the rest of the X11 protocol, since the server's byte order, alignment and padding of image data is imposed on the client (elsewhere in the protocol, the server is required to adjust to the client), so that data can move between clients and servers without copying. Under most circumstances the X11 library hides byte and bit order from a client and will adjusts the data appropriately when it is transferred to or from the server. Given the volume of image data, matching screen characteristics is critical for performance of some applications.

Graphics Contexts

In X10, each Xlib graphics routine took a list of parameters that completely determined the output semantics, and the parameters were transmitted on each request to the server. In X11, we introduced the notion of "state" or "context" by which graphics routines take some of their parameters from the protocol request and others from a graphics context or "GC" specified by the request.

The use of GCs allows many infrequently changed parameters to be removed from the call, reducing request size and therefore transport cost. GC's also map well to hardware. For example, hardware that has registers that must be loaded with color information for drawing operations only need to be loaded when the GC is explicitly changed. In X10, the registers either had to be loaded on each output request, or each parameter had to be compared with the previous request to determine if a change was needed. Some restraint is necessary though; if parameters that typically change on every graphics call are included in the context, performance will suffer because the client will have to make two requests -- one to change the context and another to display the graphics primitive.

It was difficult to choose exactly which information should be part of the context and which should be parameters, but we settled on twenty-three GC attributes described below:

function	Also known as rasterop, function determines how bits from a source are logically combined with the destination.
plane_mask	Allows graphics operations to be restricted so that they affect a subset of the display planes on a color or multi-bit-per-pixel display.
foreground background	The color values used to draw the foreground and background components of graphic objects. For example, a double-dashed line alternates between the GC foreground and background colors.
line_width line_style cap_style join_style	These attributes determine how lines are drawn, describing the width (in pixels), the style (e.g. solid or dashed), and how the lines are terminated or joined. The cap and join styles are very similar to PostScript.
fill_style fill_rule	The first attribute affects all line, and text requests, determining whether the object is drawn with the solid colors or with patterns selected from the stipple or tile attributes (see below). The fill rule affects self-intersecting polygons, choosing winding rule or even-odd computation of the polygon's interior.
arc_mode	Arcs can be filled as chords or pie-slices.

tile stipple	A tile is a pixmap that is replicated across any object drawn, whereas a stipple is used as a mask for the foreground and background colors. For example, text could be output normally with a solid color, tiled for a pattern of arbitrary colors, or stippled to get a partial coloring (e.g. 50% gray).
ts_x_origin ts_y_origin	An origin can be set within a tile or stipple so that an application can control the alignment with other objects. For example, the origin might be set so that the fill will align with a back ground pattern.
font	The font used by text operations.
subwindow_mode	Determines whether child windows are excluded from the region a GC can draw into.
graphics_exposures	Requests GraphicsExpose and NoExpose events when performing a CopyPlane or CopyArea request
clip_mask clip_x_origin clip_y_origin	Each GC may be further restricted by the client to draw in a subset region of a window, defined by a list of rectangles or a pixmap. The clip origin may be set to align the clip mask relative to the destination drawable.
dashes dash_offset	The dash list and offset are much like PostScript in allowing the client to define arbitrary line dash patterns and control the initial alignment.

Here is an example of what an X11 graphics call would have looked like had we not introduced GC's:

```
XFillRectangle (display, drawable, x, y, width, height, function, planemask, foreground,
        background, tile_or_stipple, ts_x_origin, ts_y_origin, clip_x_origin, clip_y_origin,
        clipmask)
```

and then with the addition of GCs:

```
XFillRectangle (display, drawable, gc, x, y, width, height);
```

The decisions to include certain attributes in the GC were difficult. For example, some applications change the font on nearly every text call or the color on every rectangle call. In the end, we decided to take the conservative approach of including attributes in the GC on the grounds that setting GC attributes is relatively inexpensive, and that programs would be able to cache a number of GCs when switching between a small number of states. Applications can also keep several lists of different colors, and send them all at once. At least one person is experimenting with an extension to allow color changes on a per-line basis (which can be implemented so that applications are both upward and downward compatible).

Early in the design process, we were concerned about clients that might want to read back the contents of a GC. We envisioned that the server implementors would take advantage of being explicitly notified on state changes and to convert these into machine specific forms (especially for clip regions). The combination of requiring the server to keep the client-supplied form, the complexity of some of the data structures (e.g. clip masks), the performance consideration of a server round-trip to read the attributes, and the inclusion of requests to copy GC's, allowing them to be saved and restored, led us to exclude the ability to read the attributes back from a GC (in the server). (Besides, the application just wrote that value; it can remember!) We have, however, included a write-back cache as part of Xlib, which allows GC attributes to be queried and solves the round-trip performance problems. The drawback to caching GC attributes in Xlib is that multiple clients sharing a GC cannot reliably query values without explicit communication and synchronization. In practice, it seems very rare that applications want to share GC's.

The number of attributes in a GC means that they are fairly large data structures (the original server implementation used 408 bytes per GC). Our expectation was that the number of GCs an application would use would be relatively small, averaging one or two per application. We seem to have underestimated the laziness of application writers (or the utility of having lots of GCs to toolkit implementors), since instrumented servers show as many as hundreds of GCs in use. The R4 release reduced the GC size to 116 bytes.

Text Painting and Font Support

X10 has very primitive font support inherited from the Digital VAXstation 100. It is inadequate for WYSIWYG editors, and does not support fonts of more than 256 glyphs, essential for internationalization. One of the major goals of X11 was therefore better font support. X has always taken the view that a character in a client application is merely an index into a table of glyphs; it has never implied any control or font shift semantics. Any scheme by which multifont text is encoded into a string by shift or escape schemes is left to a client program and not legislated by the X server.

X11 supports fonts of one or two byte characters. The general character and font metrics are similar to PostScript. We decided against including multi-plane (gray-scale) fonts since they are best used with sub-pixel positioning. We also decided against arbitrary rotation and scaling of characters; our belief was that outline font technology was unsatisfactory at today's monitor resolution (at that time); improvements in algorithms for outline to bitmap font conversion have occurred since X11 was designed, so there is now hope for solutions in this area. After argument, we decided not to support four quadrant character rotation, believing a general rotation extension or a PostScript extension to X was a better solution. We probably should have included 90 degree rotation as it is very much easier than general rotation.

Each font has a set of metrics describing basic information of the font: its interline spacing, what characters are defined, etc. Each glyph in the font has left and right bearing information, character width, and ascent and descent information. Each font also has associated with it a set of font properties, describing other information: how to superscript or subscript the font, spacing information, underlining and strikeout information, weight, italic information, etc. The font property list is extensible to define arbitrary other properties about a font that may be needed.

The most general form of text painting requests allows for changing font and adding space between every pair of characters printed. A full line of text in a WYSIWYG editor can be displayed in a single request. Glyphs in a font are normally thought of as masks, where only bits in the glyph that are a '1' are painted. In addition to these text requests, there are requests which render the background of a character, rather than just the foreground (called "image text"), which is useful for terminal emulators to avoid flicker.

It is surprisingly difficult to devise a scheme for naming fonts in a systematic way. The core X11 protocol is quite silent on font naming, since we did not have a systematic scheme at the time the protocol was being defined. It does provide a wild-card facility to enable matching font names with some more general pattern. The X font naming convention since has been defined in an auxiliary document, in the X Logical Font Description Conventions[2] (XLFD). The XLFD defines systematic ways of naming fonts to take into account point size, font face, italic, foundry and the many other characteristics associated with a font, and it is now part of the X11 standard.

Fonts are named independently of the host file system. X11 does however, acknowledge that there may be more than one place (directory) to find new fonts. Fonts are found according to a search path, which defines which order fonts will be found in. The search path is global to the X11 server, which is arguably wrong; it should probably have been specific to each client.

Cursors are usually stored in a font in X11, rather than as bit maps in client applications (though it is still possible to form a cursor from bitmaps). By storing cursors in fonts, cursors can be named independently of their size, so that different cursor hardware can be supported easily. The same glyphs in a cursor font might be different on a display with 64x64 cursor hardware than a 16x16 cursor hardware.

Extensions

Extensions were planned for from the start of the X11 design process. An extension is a set of related requests, events and errors. A client can query the server to find out what extensions exist, and what their names are. For example, the X Shape extension defines additional event types to inform clients when their non-rectangular windows change shape. It adds nine additional requests for operating on windows defined in terms of regions, and another event type.

A client queries an extension by name. Once an extension has been initialized, the server returns a major operation code for the extension, which the library remembers, and the server then dispatches requests with that major op-code to the extension. The extension will dispatch off of the minor op code. By reasonable use of the extension name space, arbitrary numbers of extensions are possible. 128 op-codes are reserved for extensions (restricting a client to using no more than 128 extensions at one time, which seems safe!), and each extension can have an unlimited number of requests, as the contents of a request are not interpreted by the server.

The C language X library is written in such a way that an extension need not require any explicit initialization, to ensure that extensions are first class citizens. Extensions can add information to any of the data structures that the library maintains, and library extensions can register call-backs so they will be called whenever these data structures are created and destroyed. At this writing, extensions have been approved for non-rectangular windows. Specifications for alternate input devices, multi-buffering and 3D extensions are out for public review. Some vendors have implemented Display PostScript[1] as an extension, and there are non-standard extensions for shared memory transport. Work is also under way for live video, scaled outline fonts, image processing and other extensions.

Keyboards

“Any problem in computer science can be solved by an extra level of indirection."

- Roger Needham

Keyboards vary immensely, even within the products of a single manufacturer, because of history and the requirements of different languages. The great diversity of keyboards was not

[1] Display PostScript[34] is a trademark of Adobe Systems Incorporated.

appreciated in previous versions of X, since we had only dealt with DEC and IBM keyboards, which were both very similar and had not changed over several generations of hardware, and we had not thought about internationalization problems. Other manufacturers exhibit much more variation and even within a single manufacturer's keyboards there are wide national variants in the placement of key symbols. X can accommodate all these variations. It does, however, take the position that all key transitions should be observable, to allow applications to take advantage of chording user interfaces. Some older hardware does not meet this requirement, and such applications may not run on such obsolete hardware. We highly recommend that manufacturers of X workstations and terminals build correct keyboard hardware.

In previous versions of X, when a keyboard key changed state, a *keycode* was sent to the appropriate client along with the state of the modifier keys (shift, lock, control, etc.). Keyboards had to emulate the Digital LK201 as best they could. (The IBM RT/PC and Digital keyboards were almost identical.) For a given keycode, the X library presumed that a given set of symbols were on the key (for example, that the "[{" characters are paired on the same key). The original X11 specification still reflected the X10 design, as we intended to introduce additional keycodes for other combinations found on other keyboards. After complaints from manufacturers and users, we realized the old design would not solve the problem (due to the multiplicative combinations of keys and national character sets) and would have resulted in a large registration problem of keycodes, which would have been an administrative nightmare.

X11 introduces the notion of *keysyms*, which are codes for symbols engraved on keys. The server supplies clients with a list of keysyms for each keycode the keyboard generates. Keycodes are now completely arbitrary quantities, and are most usually hardware scan codes of the manufacturer. They are used to index into the keysym list to find which keysyms are on the given key. It is left to a client (by using X library routines or its favorite toolkit) to choose how to interpret the list in combination with modifier key information By using ISO standards for printing characters and defining an additional character set for symbols only found on keyboards, we were able generate a list of keysyms which includes almost any symbol found on any keyboard in the world. We attempted to eliminate duplicates between ISO character sets to encourage application portability. The server's keysym lists can be changed, allowing general keyboard reconfiguration. X11 allows eight modifier keys (modifier keys are keys like shift, control, alt, meta, etc.), and there are requests which specify what keys should be modifiers as well. X11 adds events which allow clients to track the entire state of the keyboard if needed, and the server notifies all clients when keyboard mappings are changed.

Distributed Systems Architecture

A useful distributed system architecture consists of protocols for many services, of which X is only one. Unfortunately, some of the other needed major protocols have not yet been standardized, for example authentication protocols. X11 contains provisions in the connection setup handshake of the protocol for clients and servers to exchange arbitrary authorization information; we hope eventually to take advantage of authentication services to provide a more graceful environment, when authentication systems like Kerberos[18] are generally available. In the meanwhile, the host based access control of previous versions of X11 remains as the fallback solution for the problem. As in previous versions of X, once connections are established, a client can perform arbitrary operations on its own or other client's resources.

We have resisted adding capabilities to X which are not related to screen management, despite occasional pleas. For example, people have asked that X implement a general remote procedure call system; their (perfectly rational) contention is that X already provides (as one of the first ubiquitous network protocols) an existing communications path between applications often on quite different machine architectures, sometimes with different network protocols (for example, DECnet and TCP/IP). While in the short run this might be expedient, it would only result in long term chaos.

Similarly, many people have asked for an audio X extension; we believe this is better provided by a separate audio server, along the lines proposed by the VOX effort, both to keep complexity isolated and particularly because of the real time requirements of audio.[19]

Lessons Learned and Results

X10, while useful and demonstrating good ideas, was limited by what a very small group (3 people working on device independent code) could implement in less than a year. It was not an adequate long term base for applications. Its very success generated a severe problem; if X10 were not replaced quickly enough, the large number of existing applications would prevent incompatible change. As a result, we were under extreme time pressure. We were more conservative in our goals than we would have been under other circumstances, and the initial sample X11 server implementation was written with an eye to schedule and portability rather than ultimate performance (though performance was always the most important consideration when protocol design decisions were being made). For the design and initial implementation of X11 we had approximately 15 months, and six people worked on the X11 sample server and library (see acknowledgments below).

The performance of X11 implementations is almost always limited by the graphics subsystem (or the quality of the device dependent code). Except for GetImage and PutImage, transport cost is usually a small fraction of the cost of the graphics operations themselves. A number of changes were made between X10 and X11 which can greatly improve performance for some combinations of applications, transport, and display hardware; these include:

- The wire protocol uses variable length requests, and is therefore much more compact, minimizing transport costs for all requests
- The length field at the beginning of the request permits a simpler, faster protocol dispatcher
- GC's allow implementations using graphics engines with graphics state to avoid unneeded loads of the graphics engine. GC's also shorten the size of graphics requests which helps frame buffer implementations as well.
- By allowing many graphics operations to be sent in a single call, the poly form of graphics requests reduces protocol transport and dispatch overhead for many applications. The X11 Xlib library exploits poly requests to merge many single library calls in "dusty deck" applications into a single poly graphics request.
- Client images which match screen format can be moved to the server without any computation required, and for implementations which use shared memory the data does not have to be copied extra times on its way to a frame buffer.

The first two points above are best proved by an example. On a Digital VS2000 the simplest request (essentially a no-op) in X10 took approximately 750 microseconds (including all client, server, and transport overhead). The X11 No Operation request is approximately three times as fast (240 microseconds) on the same hardware, though part of the speed increase is improvements in compiler technology. The graphics performance for some applications has improved by a much larger factor due to poly graphics calls, while permitting applications good if not optimal performance when using a much simpler programming style A fair estimate of X11's speed advantage over X10 is about a factor of two for protocol processing, and much more for graphics operations.

It is hard to quantify the savings due to GC's. To encode all the information contained in a GC affecting each graphics operation would have required a very large request size, or the number of basic graphics requests (and complexity of the programming interface) would have had to have be very large. GC's also allow the server to save many resource ID validations. It does pay to draw using the same GC for as long as possible before changing the GC or the contents of the GC, to reduce the overhead of state changes.

There is no doubt that X11 is a larger and more complex window system. There are two parts to this: the complexity of the programming model and the size of the server implementation

We do not expect that all facilities in X11 are used by all applications; in fact, many facilities are used by relatively few applications, (often only by window managers,) and are not of interest to general application programmers. Toolkits hide most low level X11 facilities from most application programmers, and we can expect window manager writers to be few and far between. Only this year, however, are appropriate introductory texts appearing that explain the use of the common toolkits for applications programmers which has been a major problem in the past because many people had trouble figuring out which X11 facilities were useful to them.

One major contributor to the code size is X11's much richer graphics model; another is its somewhat richer event facilities. The code size is not much of a problem since only the code for the graphics operations that are actually used need be in memory on most implementations.

Dynamic memory usage has been a serious problem. Window data structures are referenced very frequently when performing window operations and therefore cannot be paged out without dire performance consequences. All running applications contribute to the size of window structure data. The initial sample server was implemented under extreme time pressure, and comparing its initial releases with the carefully tuned X10 server is unfair. However, much of the same sort of tuning has now been done for the X11 sample server implementation, which appeared in of X11 R4. The size of the R4 server's window structure is 132 bytes, compared to X10's 130 bytes. This is nearly a factor of three improvement over the previous release of the sample server, and again returns X's windows to being "cheap," though not (as some people have thought) free. Window manipulation performance has also gone up by a large factor. Window's data are also much more likely to be in one data structure, improving locality.

GC's, however, are new to X11 and represent some amount of additional memory usage. They are not as heavily used as windows are (though more heavily than we had anticipated during the design process). Recent work has reduced their size by nearly a factor of four. Of course, servers which implement the optional features of backing store and save-unders may use large amounts of additional memory, dwarfing the memory used by window and GC structures. The largest contributor to memory usage, however, has been the appearance of very serious applications for X11, a problem we are most happy to have; some of the existing toolkits, however, have had memory usage problems similar to the server, and could use serious dieting and in some cases redesign.

In order to achieve reasonable performance, the X protocol since its beginning has been a streaming protocol, rather than using strict remote procedure call. The performance differential is so large (some simple measurements put it at a factor of 30) that no other choice was possible. This makes precise error reporting more difficult. This is mitigated somewhat by two factors: 1) the protocol and library keep track of a request sequence number, which is reported with errors, making it somewhat possible for a programmer to backtrack from the point at which the error is reported to where it actually was generated, and 2) the X11 Xlib library supports a run time flag that can be set by a debugger (or from the application) to force the library to run synchronously, causing errors to be reported at the point they occur in the application (though at large performance loss).

Some people question the decision that backing store and save unders are hints in X11, wishing that they would have been required. We see no reason to regret this decision. For example, on deep displays (some of which have as many as 96 bits/pixel) backing store is very expensive to provide, and generally much slower than the application regenerating the contents on demand, which applications must be able to handle both for initial display and when resized by the user. Even on one or 8 bit displays, this is usually true, since restoring from backing store involves both a read and write memory cycle to copy from backing store to the screen, rather than clearing the window and writing the contents (all writes, which are generally fastest in frame buffer designs). It also allows for much smaller server implementations, on limited systems such as PC's and X terminals. Hints permit servers to do as much as possible on behalf of clients, given their (sometimes severe) resource limitations.

During design of X11, we tried to ensure that the core X11 specification was complete enough for a large class of workstation applications. We have succeeded, and we believe that this approach is much preferable to some systems in which applications writers must constantly query whether a given feature is available or not. We occasionally find applications which presume that they can always get all the resources (principally memory) and fail as a result, but this is generally quite rare. It is still a bit soon to tell if the extension mechanism will have its intended effect of allowing graceful evolution of X, or whether the weight of out of date implementations not providing later standard extensions will result in many applications avoiding the use of all extensions, even though they may ease certain applications greatly.

X11 does not permit the read back of all information that may have been stored in the server (for example, the X11 protocol does not permit querying the GC state). This makes modularity somewhat harder to accomplish. There are several reasons for this deliberate omission. Some of this information (for example, tiles and stipples) may have been freed and there may very well be no resource ID existing for the resource. There are potential solutions to this problem but they are somewhat ugly. More importantly, X10 had taught us that library queries to the server are so expensive that for best performance, an X application should never rely upon the X server for information the application once had; applications should remember what all information they need later, rather than expecting the server to remember it for them. None the less, complaints about this have been loud.

As one might expect, there are design flaws exposed by experience. For example, some events lack time stamps, which makes it impossible to avoid certain race conditions. One can draw the moral that it may be better to err on the side of sending too much information, rather than too little; we did not see why the timestamps would be needed at the time, and it was only later we realized we needed timestamps in essentially all events. Passive grabs are not as general as they should be. Most (but not all) of these problems could be fixed with an upward compatible protocol revision or by using unused fields in event structures; it may be necessary to add extensions as well.

In some areas, the protocol could be extended without requiring explicit extension or awareness by the client or server of protocol versions. The fact that all requests have a known length would allow additional arguments to be passed (in fact, this technique has been used to add information to window properties needed by the ICCCM, though this is not a protocol change) and a server could ignore extra data it does not expect. The X11 Xlib library implementation actually discards any unexpected data on replies, for example, though this is not required by the specification and might be considered an error. Additional bits can be allocated in various masks as well, for example. To date, we have been very reluctant to attempt such upward compatible changes, particularly as X goes through the standardization process. Further thought in this area might have paid dividends.

We still argue among ourselves about having preserved window borders from X10. One can argue that they treat a very common case with a simple solution and which saves memory; but they complicate coordinate systems transformations and server internals, which have to compensate for border width..

Most protocol library stubs in the X library are highly stylized, and a stub generator would have reduced the work required and be especially helpful to extension writers. Schedule constraints made a stub generator impossible.

Future Topics

Early specifications of X11 included support for other input devices (for example, trackballs, button-boxes, tablets, etc.). During the implementation and alpha testing of the sample X11 implementation, we decided that the design was seriously flawed and we did not understand the problem as well as we had thought. Rather than living with a bad design, we removed all support for devices except the mouse, pending support to an eventual protocol extension[35]. As of this writing, the input extension is in technical review for possible adoption as a standard X extension.

The early X11 designs included support for rotated (in 90 degree increments) text painting. During the implementation, we realized that the design was badly flawed and therefore removed it. The best solution would be an extension to the GC allowing a client to rotate the coordinate system, uniformly applied across all graphics operations. Pressure of time did not allow us to add rotated text to the design, particularly since we realized the problem relatively late in the implementation of the sample server after much of the machine independent graphics code had already been implemented. PostScript extensions, of course, support such text operations.

Some kinds of input event handling are difficult to do in X, as more elaborate decisions may need to be made than can be done with the fully predefined X protocol. NeWS[20] encourages clients to load PostScript code into its server. We believe that loading code into the server is not as useful as NeWS proponents claim (particularly for toolkit use, where our observation is that toolkits are much more bound up in applications than with the screen) and that it ultimately leads to a fragile system. Still, there is no doubt that it does have potential for input handling. We strongly believe, however, that PostScript is a very poor choice for an extension language. We believe Scheme[21], for example, as the extension language would be a much better choice, due to its much better semantics and generality. Further work should be done in this area.

It is not clear that wide-dashed-rounded lines and ellipses were worth the effort. We suffered from a schizophrenia between the integer pixel model and the PostScript imaging model for lines, which really only works well with sub-pixel coordinates. Subsequently, we discovered that PostScript actually implemented something slightly different than a circular pen, in order to avoid lumpy lines. A polygonal pen model for lines would have given us nicer-looking lines and curves possibly at the cost of some additional programming complexity. Circular pens give quite ugly results at screen resolution and are computationally very difficult for arcs. Even reasonably decent implementations of the X11 specification took four releases of X11 and serious implementation work, and wide arcs are still very slow.

We probably should have given more thought to memory requirements in the server (though release 4 has cured many of the memory use problems, it is clear than X11 will never use less memory than X10 did). Some more time and thought would have made forwarding the X11 protocol (very desirable for cooperative work applications) much easier.

Acknowledgments

It is impossible to thank here everyone who has contributed to X. The numbers of people and organizations involved at this date are huge and very gratifying to the authors. Other papers in this volume acknowledge contributors in other areas. We must, however, acknowledge specifically those who contributed to the core X11 design and sample server implementation, without which there would be no X Version 11.

Robert W. Scheifler acted as X protocol architect. His duties as X Consortium director prevented his contribution to this paper; without him, there would be no X window system. Other contributors to the X11 protocol are: Dave Carver (Digital HPW, now at MIT / Project Athena); Jim Gettys (Digital and MIT/Project Athena, now at Digital CRL), Branco Gerovac (Digital HPW), Phil Karlton (then at Digital WSL, now at SGI), Scott McGregor (Digital WSL, now at SCO), Ram Rao (Digital UEG), David Rosenthal (Sun Microsystems), and Dave Winchell (Digital UEG).

Jim Gettys acted as X library architect, and implementation was done by Ron Newman (MIT/Project Athena, now at Lotus), and Jim Gettys with assistance from Tom Benson (Digital VMS) and Jackie Greenfield (Digital VMS).

The invited reviewers who provided useful input include: Andrew Cherenson (U.C. Berkeley), Burns Fisher (Digital VMS), Dan Garfinkel (HP), Leo Hourvitz (NeXT), Brock Krizan (HP), David Laidlaw (Stellar), Dave Mellinger (Interleaf), Ron Newman (MIT, now at Lotus), John Ousterhout (U.C. Berkeley), Andrew Palay (ITC CMU), Ralph Swick (MIT/Project Athena and Digital), Craig Taylor (Sun Microsystems), and Jeffery Vroom (Stellar).

The sample server was implemented by Phil Karlton (Digital WSL, now at SGI), Susan Angebrannt (Digital WSL), Raymond Drewry (Digital WSE), and Todd Newman (Digital WSE), who also provided good input into the protocol design.

Joel McCormack, Keith Packard, and Bob Scheifler finally proved that X11 really is more efficient than X10.

Without electronic mail X's development would have taken much longer and been poorer; so our thanks must go to DARPA for its support of the Internet. Only three face to face meetings of the designers were held during initial X11 development to resolve issues not closed by electronic mail.

Thanks to Ken Lee (Digital WSL) for his X bibliography.

X Version 11 is a result of contributors all over the world. We thank all those many people who helped make X what it is.

References

1. Scheifler, Robert W, and James Gettys, 'The X Window System,' *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 79-109, April, 1986.
2. Scheifler, Robert W., and James Gettys, *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM and XLFD,* Digital Press, Bedford MA, 1990, 2nd Edition, ISBN 0-13-972050-2.

3. Rao, Ram and S. Wallace, ‘The X Toolkit,’ in *Proceedings of the Summer 1987 USENIX Conference,* pp. 117-130, USENIX Association, Berkeley CA.
4. Linton, Mark A, John M. Vlissides, and Paul R. Calder, ‘Composing User Interfaces with InterViews,’ *IEEE Computer*, vol. 22, no. 2, pp. 8-22, February, 1989. Early InterViews implementations ran on X10.
5. Dineen, Terence, Paul J. Leach, Nathaniel W. Mishkin, Joseph N. Pato and Geoffrey L. Wyant, ‘The Network Computing Architecture and System: and Environment for Developing Distributed Applications,’ in *Proceedings of the Summer 1987 USENIX Conference*, pp. 117-130, USENIX Association, Berkeley CA.
6. Cessna, Keith et al. *CLX Common Lisp X Interface*, MIT X Consortium, Cambridge, MA., 1989.
7. Morris, James H. et al., ‘Andrew: A Distributed Personal Computing Environment’, *Communications of the ACM*, vol. 29, no. 3, March 1986, pp. 185-201.
8. Lampson, Butler, Personal Distributed Computing, ‘The Alto and Ethernet Software’, in *A History of Personal Workstations*, Ed. A. Goldberg, ACM Press 1988, pp. 293-335.
9. Thacker, Charles, Personal Distributed Computing, The Alto and Ethernet Hardware, in *A History of Personal Workstations*, Ed. A. Goldberg, ACM Press 1988, pp. 265-289.
10. Xerox Corporation, *The ViewPoint*™ *Programmers Manual*, 1986, 1988. XDE is the predecessor to ViewPoint, by approximately five years.
11. *PostScript*® *Language Reference Manual*, Adobe Systems, Incorporated, Addison-Wesley Publishing Company, ISBN 0-201-10174-2, 1985.
12. Petzold, Charles, *Programming Windows*, 2nd ed., 1990, Microsoft Press, ISBN 1-55615-264-7.
13. Chernicoff, Stephen, *MacIntosh Revealed: Vol. 1 - Unlocking the Toolbox, 2nd Ed.*, Hayden Books, Indianapolis IN. 1988, Chapt. 4-6 and 8.
14. Smith, D.C. et al., ‘The Star User Interface, An Overview,’ in *Proc. AFIPS Conf.*, pp.515-528, 1982.
15. *Programmer’s Hierarchical Interactive Graphics System (PHIGS)*, International Proposed Draft Standard ISO 9592-1:1988(E), International Standards Organization, Geneva, Oct. 1987.
16. *Graphical Kernel System for Three Dimensions (GKS-3D)*, ISO/DIS 8805, International Standards Organization, Geneva, Apr. 1987.
17. Hobby, John D., ‘Rasterizing Curves of Constant Width,’ *Journal of the ACM*, vol. 36, no. 2, April 1989, pp. 209-229.
18. Steiner, Jennifer G., Clifford Neuman and Jeffry I. Schiller, ‘Kerberos: An Authentication Service for Open Network Systems,’ *Proceedings of the Winter, 1988 USENIX Conference*,

pp. 191-202, USENIX Association, Berkeley, CA. Kerberos is a trademark of Massachusetts Institute of Technology.

19. B. Arons, Carl Binding, Keith Lantz, and Chris Schmandt, 'The VOX Audio Server.' in *2nd IEEE Computer Society International Multimedia Communications Workshop*, IEEE Communications Society, April 1989.
20. Gosling, James, David S.H. Rosenthal, and Michelle Arden, *The NeWS Book*, *An Introduction to the Network/extensible Window System*, Springer-Verlag, ISBN 0-387-96915-2.
21. Rees, Johnathan and William Clinger et. al., 'Revised^3 Report on the Algorithmic Language Scheme', *ACM SIGPLAN Notices*, vol. 21, no. 12, pp. 37-79, December 1986.
22. Gajewska, Hania, Mark Manasse, Joel McCormack, 'Why X Is Not Our Ideal Window System,' *Software - Practice and Experience*, this volume.
23. Rost, Randi J., Jeffery D. Friedberg, and Peter L. Nishimoto, 'PEX: A Network-Transparent 3D Graphics System,' *IEEE Computer Graphics and applications*, vol. 9, no. 4, pp. 14-25.
24. Young, Douglas A. *X Window System: Programming and Applications with Xt*, OSF/Motif Edition, Prentice-hall, 1990. ISBN 0-13-497074-8.
25. Hopgood, F. R. A., *Methodology of Window Management*, Springer Verlag, New York, 1986.
26. Jones, Oliver, *Introduction to the X Window System*, Prentice-Hall, Englewood Cliffs, New Jersey, 07632, 1989, ISBN 0-13-499997-5.
27. McCormack, Joel and Paul Asente, 'An Overview of the X Toolkit," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, pp. 46-55, October, 1988.
28. Rosenthal, David S., A Simple X11 Client program, or, How Hard Can It Really Be to Write 'Hello, World'?,' *Proceedings of the Winter, 1988 USENIX Conference*, pp. 229-235, USENIX Association, Berkeley, CA.
29. Johnson, Eric and Kevin Reichard, *X Window Applications Programming*, MIS: Press, ISBN 1-55828-016-2.
30. Asente, Paul and Ralph Swick, *X Window System Toolkit, The Complete Programmer's Guide and Specification*, Digital Press, Bedford MA, 1990, ISBN 0-13-972191-6.
31. Open Software Foundation, *OSF/Motif Series* (5 volumes), Prentice Hall, 1990. ISBN 0-13-640491-X, 13-640525-8, 13-640517-7, 13-640509-6, 13-640483-9.
32. Rosenthal, David S. H., 'Window Exchange,' *UNIX Review*, vol. 7 no 12, pp. 58-64.
33. Sun Microsystems, *OPEN LOOK Graphical User Interface Series*, Addison-Wesley, 1990, ISBN 0-201-52365-5, ISBN 0-201-42364-7.
34. Holzgang, D.A., Display PostScript® Programming, Addison Wesley, Reading MA, 1990, ISBN 0-201-51814-7.

35. Patrick, Mark, and George Sachs, *X11 Input Extension Library Specification*, MIT X Consortium, December 1989.
36. Widner, Glenn, *The X11 Inter-Client Communications Conventions Manual*, this volume.