Design Experience with a Multiprocessor Window System Architecture

J. V. Kelley[1]
K. S. Booth

Computer Graphics Laboratory
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

M. Wein

Laboratory for Intelligent Systems
National Research Council of Canada
Ottawa, Ontario, Canada K1A 0R6

Abstract

We discuss the design and implementation of a multiprocessor window system architecture. The implementation is based on the X Window System (X) and runs under Harmony, a multiprocessor, multitasking, real-time operating system. A number of bottlenecks has been identified that limit the use of parallelism in the implementation. Some of these can be eliminated by changes to the implementation, but many are inherent in the definition of the X protocol. The principal contribution of this paper is an analysis of a multiprocessor workstation architecture that has evolved from our experience with X and other graphics support systems in the multiprocessor Harmony environment. The new design is intended to permit full realization of multiprocessor graphics support in a windowing environment by explicitly separating the screen management functions from the graphics rendering functions of a window system.

Résumé

Nous discuterons dans cet article du développement d'un système de fenêtrage multiprocesseur. Le système utilise l'environnement *X window* fonctionnent sur le système d'exploitation *Harmony* et supporte l'ensemble du protocole X. Un certain nombre de problèmes limitant le parallélisme du système furent identifiés. Certain d'entre eux peuvent être éliminés par de simples modifications. D'autres sont inhérent à la definition du protocole X. La contribution principale de cet article consiste en une analyse de l'architecture des stations de travail multiprocesseur qui est basée sur notre expérience avec le protocole X et les systèmes graphiques dans un environnement multiprocesseur comme Harmony. Le nouveau système permettera le développement de systèmes graphiques multiprocesseur dans un environnement de fenêtrage grâce à la division des tâches entre les foncions de gestion des fenêtres et les fonctions de rendu.

Keywords: Multiprocessor, Parallelism, Synchronization, Window, X Window System.

[1]Present address: Laboratory for Intelligent Systems, National Research Council of Canada, Ottawa, Ontario, Canada K1A 0R6.

Introduction

There are two bottlenecks in network-oriented, server-based window systems. The first is caused by the heavy traffic of low-level graphics primitives across the network. The trend towards sending higher-level abstractions is a step towards alleviating this bottleneck. The second bottleneck is caused by the lack of parallelism in the support software that manages the window system.

We describe the implementation, analysis and design for a multiprocessor-based window server that addresses the second limitation, which is inherent in a network-oriented server. Both the implementation and the design identify problems in resource sharing, resource locking, and distribution of functions across multiple processors [3]. We discuss these problems and identify characteristics of frame buffer architecture that are necessary to support multiprocessing in a workstation with a network-oriented window system.

The next section reviews the basic components of window systems and introduces the problems that we address in our implementation and design. Subsequent sections present a more detailed discussion of the Harmony implementation of the X Window System (X) and the new multiprocessor design (the Harmony operating system and X are described below). We conclude with a set of observations and recommendations for multiprocessor workstation systems.

Window Systems

A *window system* is a collection of software that controls a computer's graphical display and provides a base upon which application programs can be written. This is analogous to the role of an operating system, because both systems present an extended virtual machine to applications while at the same time protecting and controlling access to the hardware. The window system provides applications with independent virtual display surfaces called *windows*.

Computer systems that support the concurrent execution of multiple applications by a single user are essential. Performing multiple activities simultaneously is natural for people and can enhance productivity if supported adequately in the user interface. Studies have shown that conscious thought deals with

concepts stored in "short-term" memory [1] and that the capacity of short-term memory is limited [13]. Maintaining the state of an application visually relieves the burden placed on short-term memory by turning the display screen into a "visual cache." This becomes especially important when the user is managing several applications which are executing concurrently because the visual cache eases the cognitive burden of switching context, allowing the user to focus on his problems. The trend towards concurrency therefore provides the *raison d'être* for window systems, which allow the sharing among multiple concurrently executing applications of a computer display and other devices that interact with the user. The window system allows the applications to display output independently of each other, and allows the user to rapidly switch his focus from one application to another.

The functional components of most window systems can be divided into four categories that are briefly summarized below, but are described in more detail elsewhere [8,9,10]. The components of a window system are organized in a hierarchical structure, each building upon the functions provided by the lower levels. From lowest to highest are the hardware, the window agent, the dialogue manager, and the window manager. Most window systems allow applications to access any of these levels, in some cases even to the extent of permitting direct access to the hardware.

Server-based and Kernel-based Systems

Before describing the components, we will examine the two overall approaches that have been used to build window systems, the *server-based* approach and the *kernel-based* approach [12]. In a server-based system, one or more of the software components are placed in a user-level server task. Applications access window system functions by sending requests to the server using the interprocess communication facilities provided by the operating system. An increasingly important advantage of server-based systems is that they are easily extended in a distributed computing environment to allow a client and server to execute across a network, so that an application running on a supercomputer can easily display output on an inexpensive workstation.

In kernel-based systems the innermost components are embedded in the kernel of the operating system. Applications perform window system functions using libraries built on top of the kernel primitives. In theory all the components of the window system can be placed in the kernel, but this approach is rarely used because user-level code is much easier to build, modify, and maintain than kernel code.

The primary functional difference between the two approaches is the manner in which synchronization of access to the display hardware[2] is achieved. Kernel-based systems usually require that locking primitives be invoked to access the frame buffer. If the locking primitives are invoked for individual graphics operations, such as drawing a simple line, the cost of the locking operations may dominate the cost of rendering an image. But the alternative, invoking locking primitives around groups of requests, is error-prone and presents an additional, unwanted concern in higher-level graphics routines. With server-based systems, applications have no direct access to the frame buffer. The server is responsible for synchronizing application requests, normally by processing them in serial order. In effect, this gives the client a lock on the frame buffer for the duration of the request.

Window System Components

The *window agent* is responsible for managing the hardware. The window agent multiplexes devices between applications and may provide them with a device independent interface. Display multiplexing is accomplished by providing windows to applications. The window agent is responsible for mapping the windows onto the physical display. The window agent provides an imaging model for rendering into windows. The model may vary from low-level pixel manipulation to high-level, device independent structured graphics. In the past, the model chosen was determined primarily by the facilities of the underlying hardware. The current trend, however, is to provide high-level abstractions that can be used with most existing hardware.

The *dialogue manager* synthesizes the low-level I/O primitives provided by the window agent into interaction techniques, providing a "dialogue" between the user and the application. For example, it might interpret a mouse button press as a request to bring up a menu, which it would do, then wait for a selection to be made from the menu by another mouse button press. The dialogue manager would then pass on the selection event to the application, which would never see the primitive mouse events.

The *window manager* is responsible for managing the multiple simultaneous user-application dialogues that may be taking place. Communication between the user and window manager is through a "meta-dialogue," so named because it is used to manipulate other user-application dialogues. While the window agent provides the mechanism for sharing devices, the window manager sets the sharing policy.

The Harmony X Server

The implementation of a window system for the Harmony multiprocessor, multitasking, real-time operating system [4,5,6] was motivated by both research interests and practical considerations. If multiple tasks are allowed to access the frame buffer then a mechanism to coordinate access is essential. A window system normally includes a graphics library, which could be used to support the development of Harmony applications. The X Window System [7,15,16] was chosen for several reasons: X is designed to be easily ported to a variety of hardware and operating system configurations; the source to a sample X implementation and a wide variety of X applications is freely redistributable; many institutions are involved in the on-going development of X-based tools; and X follows the server-based model, which is more appropriate for Harmony than the kernel-based model. It is a portable, network-transparent window system developed at the Massachusetts Institute of Technology (MIT). The Harmony X server implements the X11 protocol. Subsequent discussion pertains to that version of the protocol.

X follows the client-server model (see Figure 1). Fundamentally, X is a protocol for communication between appli-

[2]Often a *frame buffer*, but other types of display hardware can be used.

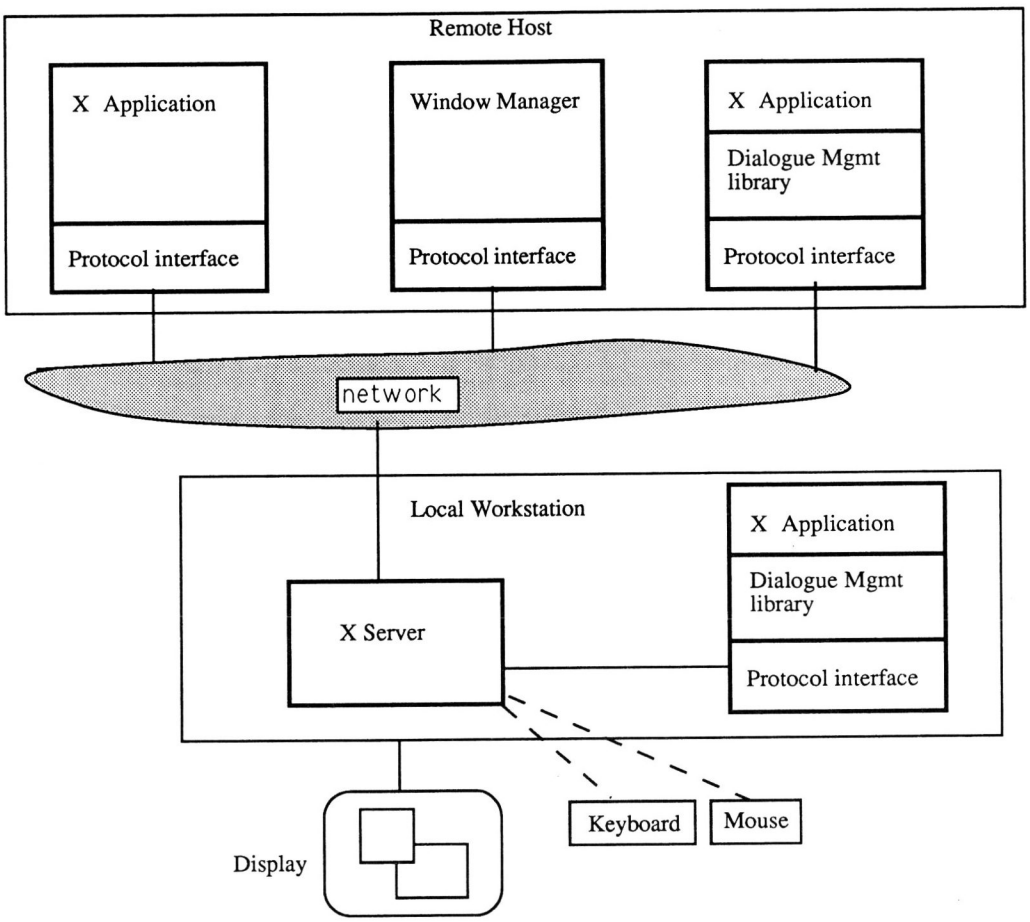

Figure 1. X Window System Model

cations and display servers. The implementation of applications or servers may be done in any fashion, as long as the communication conforms to the X protocol specification. An X server performs the functions of the window agent, managing a display and various input devices, typically a keyboard and mouse.

An X *screen* is a two dimensional array of *pixels* (the frame buffer). Each pixel is an N-bit value, where N is the number of bit planes in the screen. A pixel value is translated into a colour value using a *colourmap*. A colourmap is a set of entries defining colour values. X provides several classes of colourmap to match current display technology.

X provides a fairly diverse set of graphics operations that range from individual pixel manipulations to rendering arcs, polygons, and fonts. The use of higher-level constructs is preferred to minimize the load placed on the communication channel between client and server. Also, the higher-level constructs offer server implementors the ability to choose the most efficient mechanism provided by the display hardware for carrying out a particular request type.

The Harmony X Server is based upon the MIT sample X server, with perhaps 75 percent of that code being unmodified. The remaining 25 percent is specific to the Matrox frame buffer board or the NRC/Dy-4 implementation of Harmony. Minor problems with the implementation arose from the differences in the environments provided by UNIX and Harmony.

The performance of the current implementation is in many respects adequate for traditional workstation use. However, the implementation will fail to meet the needs of many applications that place real-time constraints on graphics output and user interaction. To some extent the performance problems can be solved by improving the implementation. The manner by which Harmony applications normally meet real-time constraints is by distributing the work load across multiple processors. The X Server could use this technique, but to do so would require a major redesign of the internal structure of the sample X server to perform synchronization between components executing as independent tasks on multiple processors. Even this may not be adequate, however. Fundamental limitations of the X design that seem to discourage this approach are discussed in the next section.

Multiprocessor Window System Design

The primary function of a window system is to multiplex a display amongst several processes. This requires that display accesses be synchronized. For example, suppose one application

moves a window so as to obscure a window belonging to another application. This second application, an independent process, may be simultaneously drawing into its window. If these actions are not synchronized, the second application may end up drawing into the window of the first application. The method by which display access is synchronized is of crucial importance in determining whether the window system will be acceptable on a multiprocessor system. This paper concentrates on issues related to output. We do not address the issue of how input should be handled. That issue is orthogonal to the output mechanism, although similar problems arise.

Conventional Systems

Both of the conventional window system models, kernel-based and server-based, have deficiencies that make them undesirable in a multiprocessor environment. Kernel-based systems synchronize display access by allowing processes to perform a system call which grants exclusive access to a region of the display (see Figure 2). The kernel contains a data structure indicating display ownership that must be accessed sequentially, thus introducing a bottleneck. If the display locking operation is invoked around every graphical function, then significant overhead will be introduced. But if locking is performed around groups of functions, other processes may end up blocked for extended time periods.This issue becomes especially critical if real-time response is expected of the window system. Locking on a multiprocessor must involve some form of interprocessor communication, which makes the locking operation even more expensive than on a uniprocessor system.

Server-based systems synchronize display access by granting a single process, the server, ownership of the display (see Figure 3). Client-server communication introduces some overhead, but the overhead can be alleviated by choosing a suitable protocol. Often, the server will synchronize access by simply processing requests in serial order. This method fails to utilize the full power available on a multiprocessor system. Alternatively, the server can distribute requests to worker processes running on different processors. This suffers from some of the disadvantages of the kernel-based system because the server must determine how the requests will interact before dispatching them to the workers. In any case, all requests must pass through the server for processing, so a bottleneck can result on the server's processor.

The bottlenecks inherent in the kernel-based and server-based window systems result from the centralization of information required to synchronize display output. The solution to this problem is to distribute the synchronization information.

A New Multiprocessor Window System Model

The hardware and window agent components of a new multiprocessor window system model are shown in Figure 4. The window agent is broken into two distinct components, the *display manager* and the *renderers*. This partitioning of function is based upon the logical independence of window manipulation from rendering.

The display manager is responsible for distributing synchronization information to renderers so that renderers share the display without conflict. The display manager accepts requests to create, delete, translate, stack, and alter the size of windows. The source of these requests is left unspecified, but might be a window manager, the renderers, or the applications. The synchronization information distributed by the display manager takes the form of access grants to the display and revocations of those grants. An access grant corresponds to the notion of a window exposure found in most window systems. The display manager grants exclusive ownership of a portion of the display to a renderer when a region of the renderer's window is visible on the

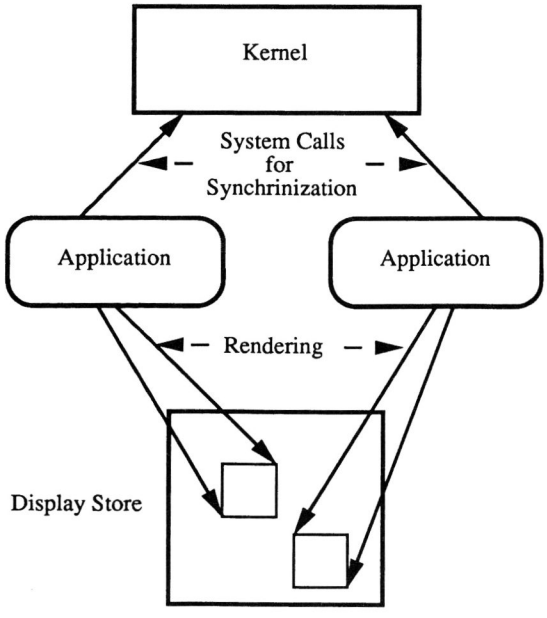

Figure 2. Kernel-based Window System Model

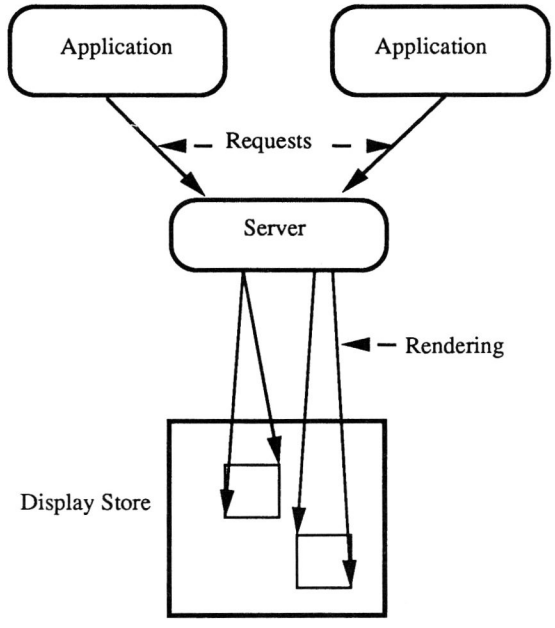

Figure 3. Server-based Window System Model

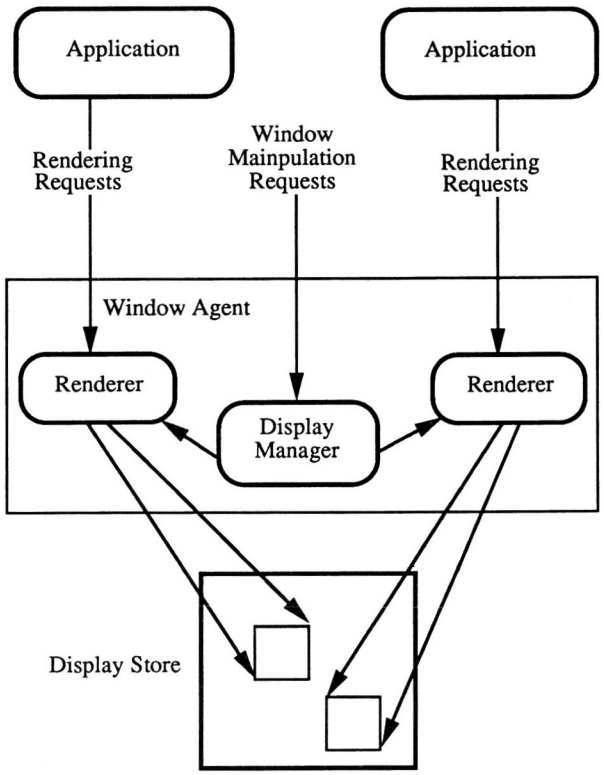

Figure 4. Multiprocessor Window System Model

display. If the window is destroyed or obscured by another window, some or all of the access to the display may be revoked. The display manager never touches the contents of the display. The contents are only manipulated by renderers.

Renderers are processes (or *teams* of processes) which have created a window and are manipulating its contents. Renderers may be servers performing the manipulations on behalf of client processes. Because a window may not be fully visible, a renderer is responsible for maintaining the contents of the off-screen regions of the window. A renderer is responsible for clipping its output to regions of the display it has been allocated by the display manager.

It is important that renderers respond to requests from the display manager quickly, as otherwise a visually confusing image may be presented to the user. Therefore, renderers should keep their atomic actions as small as possible. One option available to a renderer would be to hand rendering requests to worker processes, so the renderer can continue to listen for requests from the display manager. If a request is received to revoke access to an area a worker is rendering into, the renderer could destroy the worker process and relinquish access. Subsequently, however, the renderer must recover from the incomplete operation performed by the destroyed worker.

Synchronization through the Display Manager

The display manager must maintain a data structure indicating the current organization of windows in the display. In general, a window need not be rectangular, but rectangular windows lead to the simplest implementation. If rectangular windows are assumed, the data structures and algorithms described by Pike [14] can be used. The data structures used by renderers may vary a great deal from one type of renderer to the next. The requirement that a renderer restrict its output to regions of the frame buffer to which it has access means that renderers must retain a list of clipping regions. The remaining data structures depend primarily on the method the renderer uses to perform image retention, if it implements retention. A text renderer might simply retain an array of characters. Other renderers would require more elaborate mechanisms.

The display manager is responsible for synchronizing all window manipulation functions, including window creation, deletion, translation (changing the location of a window), scaling (changing the size of a window), and stacking (changing the priority of a window with respect to other windows). The two principal effects of any window manipulation are that some visible regions of windows become obscured and some obscured regions become visible. Window manipulation is thus performed in two stages.

In the first stage, the display manager revokes access to regions of the display that will be obscured by the manipulation. The display manager cannot arbitrarily revoke access, but instead must request that a renderer relinquish its access to the display. Those renderers whose access has been revoked must take whatever action is necessary to preserve the contents of their window and acknowledge the request. All the renderers that must process such requests may do so in parallel.

The second set of requests to be sent by the display manager grants display access to renderers whose windows are exposed. A simple implementation might wait until all the requests revoking access have been acknowledged before sending the first request granting access. However, the display manager can grant access to a region as soon as the manager has exclusive ownership of that region. That is, granting access to a region can proceed when the previous owners of the region have given up that ownership by acknowledging the request revoking access. Thus, each grant request can be viewed as being blocked on a set of revoke requests, and can proceed when all the revoke requests have been acknowledged. A renderer which is granted access to a region of the display should immediately render some image into that region. This image should be the best possible based upon the image retention performed by the renderer. If no image is available, a background should be rendered to erase the region's previous contents.

These two phases of synchronization apply to all window manipulations, but scaling and translation require an additional step. Scaling a window requires that the renderer be informed of the window's new dimensions. This step can be taken before the revoke requests are dispatched. The interpretation of a change in a window's dimensions is left to the renderers. The window may be treated as a *viewport* or as a *graphics window*.[3] If a window is treated as a viewport, then a change in the size of the

[3]The term *graphics window* is used to refer to the standard graphics definition of window to distinguish it from the window system definition.

window results in the display of the same region of the world at a different scale. If a window is treated as a graphics window, a change in the size of a window results in a different region of the world being displayed at the same scale. A renderer may choose to implement either interpretation.

Translating a window requires that the renderer be informed of the window's new location. One method of accomplishing the translation is to revoke access to the display for all visible areas of the window to be translated so that the renderer retains the window's image, then inform the renderer of the new location, and finally grant access to the renderer to regions of the display on which the translated window is visible. This method is inefficient, however, as it may require copying the image twice and storing the image temporarily. A better method is to revoke access only to regions of the window to be translated that will be obscured after the translation. The translation message sent by the display manager to the renderers can then imply the granting of access to the display at the translated position for all remaining visible areas of the window, while simultaneously revoking access to the old position. The renderer is thus given the option of simply copying the image from one part of the display to another.

Multiple window manipulations may take place simultaneously. Because the display manager is merely an arbiter of display ownership, it may be orchestrating the manipulation of one window when another request arrives. Provided that the second manipulation does not conflict with the first, some or all of the granting and revoking of display access can proceed in parallel and independently of the actions required by the first manipulation. In some cases when manipulations do intersect, certain optimizations may be possible. Consider, for example, the situation when window A is translated to a new location. Normally, this manipulation would result in the the granting of access to the display to the renderer for window B which is underlying window A. However, if window C is manipulated to occupy the location vacated by window A before window B has been granted access to the display, then access can be granted to window C instead, bypassing window B entirely.

Remaining Bottlenecks

Some limitations remain. The display hardware may be an unavoidable bottleneck. It is important that the display support concurrent access by multiple processes or processors. The Matrox frame buffer boards in use in the prototype implementation support atomic access to any pixel (byte) of the board from multiple processors, but not atomic sub-pixel access. As a result, it is possible to partition the frame buffer into independent regions in X and Y but not into separate bitplanes. Support for multiple write masks could alleviate this problem. In fact, the fault does not lie solely with the Matrox boards but also with the processor boards. The processors could support the atomic read-modify-write cycle allowed by the VME bus, but do not. If the multiprocessor window system model is to be fully exploited, the hardware must be designed with the requirements of the model in mind. A more complete discussion of hardware problems is provided in [3].

The other bottleneck that still remains is the display manager, because it must process any attempt to manipulate windows. However, window manipulation tends to occur infrequently relative to rendering. This property is especially true of systems that provide only user-driven window management. In contrast to traditional window systems which grant clients access to the display only for a small number of operations at a time, requiring clients to perform the same synchronization procedure over and over again for their subsequent requests, the model proposed here grants clients access to the display until the window configuration changes, thereby taking advantage of the static nature of the configuration.

Cursors

Most window systems provide cursors limited to a few colours and to a certain maximum size. There is almost always only a single cursor available. So only a single pointing device will interact with the user. Although one device is sufficient to meet the needs of most traditional applications, more flexibility on the part of the window system would lead to a richer environment. The protocol by which movement of the cursor is controlled is important, but because we are describing only the output components of the window system we will show only how cursor images may be effected.

Cursors would, in most cases, be provided by processes dedicated to that purpose. In many cases hardware may provide the functionality required for cursors. However, it is likely that more cursors than the hardware makes available may eventually be required, or cursors with features not provided by the hardware may be needed. As a result, a "soft" cursor will be necessary. The only mechanism defined by the multiprocessor window system model for the implementation of such a cursor is to have a renderer perform that function. This method provides the required functionality, because the cursor can be arbitrarily shaped and coloured, but at the cost of introducing computational overhead. The overhead arises from the violation of the assumption that the window configuration is relatively static, because the cursor is realized using a moving window.

Translating the cursor window requires at least two transactions between the display manager and other processes: an initial request to the display manager to translate the window and a message from the display manager to the cursor renderer indicating that the cursor window should be moved. If the cursor window intersects other windows, then grant and revoke requests will be generated as well. Given the possibility of an arbitrarily large cursor and any number of windows, it is not possible to place a bound on the time required to move the cursor. The point at which the cursor will cease to provide the interactive response required of it depends upon the system in use. Whether the additional functionality provided by a "soft" cursor outweighs the cost is a decision that has to be made on a per implementation basis.

Renderer Flexibility

The benefits of the multiprocessor window system model are most evident on a multiprocessor system, but some benefit is derived on a uniprocessor system due to the division of the window agent into two components. Even on a uniprocessor, rendering protocols and image retention models can be tailored to the needs of an application by creating a new renderer type. For

example, one renderer might emulate a character display be receiving ASCII text and ANSI escape sequences, while a second renderer acts as a PostScript interpreter, and a third acts as a PHIGS engine. The abstractions the renderers provide to their clients can thus vary from low-level pixel manipulations to high-level device independent graphics.

Moreover, a further advantage of the multiprocessor window system model is that the renderers can be added and removed dynamically while the remainder of the window system continues to function. Because they have fixed imaging methods, traditional window systems can add new functionality only by building on top of existing methods, which introduces extra overhead. The multiprocessor window system model allows a new renderer to have direct access to the hardware if it desires, or it can use the rendering facilities already provided by another renderer. Other window systems, notably TheWA [10], also have the ability to support multiple rendering protocols.

Because the scheduling of the processing of rendering requests is not specified by the multiprocessor window system model, flexible process priority management mechanisms may be exploited in operating systems that provide them. An operating system that supports preemptive scheduling, for example, makes it possible for one renderer to interrupt another that is processing a request. This contrasts with the approach of some server-based window systems which define their own scheduling policy. For example, the sample X server from MIT will process a maximum of ten requests from one client before checking for requests from other clients. In a NeWS server [12], rendering is performed by lightweight processes (tasks) scheduled in a non-preemptive fashion. Thus it is impossible for one task to interrupt another so a task which never voluntarily blocks can prevent all other tasks from running.

A prototype implementation of a subset of these facilities is described elsewhere [3].

Conclusions

Conventional window systems have been designed with uniprocessor architectures in mind, resulting in single-threaded structures. The X Window System is one example, and its implementation under Harmony has made its deficiencies apparent. The design of *any* software system should include maximizing the potential parallelism possible [2]. It is much more difficult to extract the parallelism from a sequential process than it is to perform potentially parallel actions in sequence. Hopefully the emergence of multiprocessor systems will influence the design of window systems, such that one will take advantage of the parallelism that is becoming possible.

The multiprocessor window system design presented here is a first attempt to remedy the limitations of conventional window systems by explicitly identifying those functions that may be performed in parallel. Decomposition of the window agent into two components, the display manager and the renderers, and distribution of synchronization information to the renderers allows rendering to different windows to proceed independently until the display is accessed. Because display access typically represents only a small portion of total rendering time, this potential bottleneck will not be felt for a small number of processors. Thus the multiprocessor window system model removes a significant bottleneck present in kernel-based and server-based window systems.

References

[1] Rudolf Arnheim. *Visual Thinking*. University of California Press, Berkeley, 1971.

[2] K. S. Booth, W. M. Gentleman, and J. Schaeffer. Anthropomorphic programming. Technical Report CS-82-47, University of Waterloo, Waterloo, Ontario, Canada, May 1982.

[3] Kellogg S. Booth, Barry M. Fowler, and Peter P. Tanner. Experience with graphics support for a multiprocessor workstation. In *Proceedings of Parallel Processing for Computer Vision and Display International Conference*, January, 1988.

[4] W. M. Gentleman. Message passing between sequential processes: The reply primitive and the administrator concept. *Software Practice and Experience*, **11**(5):435-466, May 1981.

[5] W. M. Gentleman. Using the Harmony operating system. Technical Report DEE NRCC/ERB-966, National Research Council of Canada, May 1985.

[6] W. M. Gentleman, M. Wein, S. A. MacKay, D. A. Stewart, R. K. Parr, and D. Green. Harmony, an operating system for embedded industrial multiprocessor applications. In *Proceedings of Compint '87*, November 1987.

[7] Jim Gettys. Problems implementing window systems in UNIX. In *Summer Conference Proceedings*, USENIX Association, 1986.

[8] James A. Gosling. Partitioning of function in window systems. In F. R. A. Hopgood et al., editor, *Methodology of Window Management*, pages 101-106. Springer-Verlag, 1986.

[9] Keith A. Lantz. Multi-process structuring of user interface software. *Computer Graphics*, **21**(2):124-130, April 1987.

[10] Keith A. Lantz, Joseph I. Pallas, and Michael A. Slocum. TheWA beyond traditional window systems. Draft of January 21, 1987.

[11] Keith A. Lantz, Peter P. Tanner, Carl Binding, Andrew Dwelly, and Kuan-Tsae Huang. Reference models, window systems, and concurrency. *Computer Graphics*, **21**(2):87-97, April 1987.

[12] Sun Microsystems. NeWS preliminary technical overview. Sun Microsystems, Inc., Mountain View, California, October 1986.

[13] George Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. In *The Psychology of Communication*. Basic Books, New York, 1967.

[14] Rob Pike. Graphics in overlapping bitmap layers. In *Proceedings of ACM/SIGGRAPH '83*, published as *Computer Graphics*, **17**(3):331-356, July 1983.

[15] Robert W. Scheifler. X Window System protocol, Version 11 Release 1 edition, 1987.

[16] Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, **5**(2):79-109, April 1986.