A Window Architecture Providing Predictable Temporal Performance

Philippe F. Bertrand, William Cowan, Marceli Wein

Department of Computer Science,
University of Waterloo, Waterloo, Ontario, N2L 3G1
519–888–4534
e-mail: [PFBertrand | WBCowan | MWein]@watcgl.UWaterloo.ca

ABSTRACT

Programs running on high performance workstations are beginning to require the ability to perform precise temporal synchronization for applications that range from process and system visualization to video games like flight simulators. Unfortunately, the architecture of operating systems and graphics makes this capability very hard to achieve. This paper describes the architecture and implementation of *SLwindows*, a window system that provides all the usual facilities of overlapping windows in addition to real-time performance controllable from within the individual applications. It does so by making use of graphics hardware that duplicates context within the graphics system, allowing the implementation of independent graphics data paths that run from each application all the way to the digital-to-analogue converters. The window system written to take advantage of this hardware allows individual application tasks direct access to all the graphics hardware, giving them real-time control of the graphical output. In doing so it eliminates much of the time-consuming overhead that exists in current window systems, such as damage repair. The implementation was performed using a decade-old graphics system, combined with custom-built hardware. But the capabilities of commercial graphics subsystems are improving in ways that make widespread implementation of similar systems possible in the near future. They will be essential as application programs begin to use facilities like multi-media that require tight control of temporal synchronization among a variety of output streams.

KEY WORDS: window systems, graphics context, multimedia, hardware and software architecture, workstation, temporal synchronization.

INTRODUCTION

A variety of applications are poorly served by hardware and software architectures offered in the current generation of high performance workstations. Of specific interest in this paper are applications that use a class of features relying on temporal synchronization. Examples include animation, coordination of events in different windows, and synchronization with audio output. Even more important, because the human user operates in real-time any application with a user interface is potentially a real-time program. Thus, all application environments should offer the programmer access to real-time control of the computation.

A class of applications that has recently increased awareness of the need for good temporal performance is navigation in virtual reality. The image must be presented at a constant rate, with negligible jitter, forcing the delay from hand-controller input through the application to the display to be within a narrow band [2, 21]. Otherwise the desired perception is completely lost to the user. Many other applications have similar requirements because they link hand-eye coordination to a real-time application. Such systems must provide tight feedback, low latency and a quality of response not found in contemporary graphics workstations. The requirements are well documented in the literature of avionics and industrial systems. Wickens [21] gives a good overview. Awareness of these issues can also be found in the graphics community [1, 19].

A related class of applications makes use of multimedia presentation, requiring jitter-free presentation of visual material and synchronization among several threads, possibly including audio and tactile feedback. For such applications the temporal synchronization demands are as great or greater than they are for virtual reality.

This paper discusses some of the shortcomings of existing systems, presents *SLwindows*, a novel window system architecture providing good temporal performance, and reports on a prototype

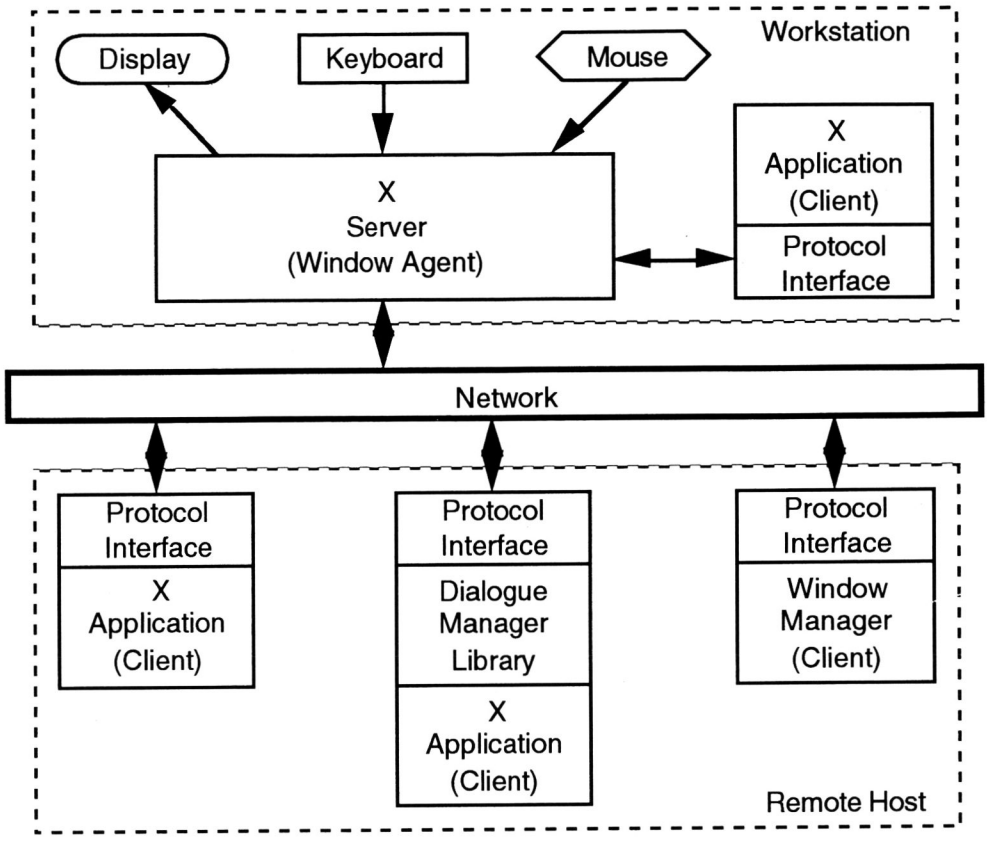


Figure 1. Relationship of System Components in the X Window System.

implementation of the architecture demonstrating its feasibility.

The thesis of this paper is that architectural improvements in the workstation window system and the underlying graphics hardware are more effective than waiting for increases in computing power and in graphics throughput. Adequate performance is available today using modest CPU bandwidth, and temporal control can be provided in addition to raw speed: any player of video games knows that too soon is often worse than too late.

PROPERTIES OF EXISTING SYSTEMS

Time-critical applications are increasingly emerging from embedded systems to run on interactive systems that range from high performance workstations to mass-market computers. Universally these platforms run under the control of an operating system through which application programs control the system hardware. The more elaborate the operating system the more the application program is isolated from the underlying hardware and the less temporal control is available. In this respect operating systems of mass-market computers are converging with Unix, which dominates the high performance market. OS/2 [13] is a typical example. In such environments a highly tuned application can be frustrated by the unreliable temporal response offered by the operating system.

This problem is often compounded by the window system, which inserts another hard-to-control layer between the application and the hardware. A good example is the X Window System [16], which currently occupies a dominant position in the workstation market. It has evolved with a strong emphasis for support of network oriented applications, permitting a window to display the output of applications that run either locally or remotely. The resulting loose coupling of application and display inhibits temporal control of application output. Thus, while X is strong for flexibility and generality, it is weak for temporal fidelity.

The X window system is based on the UIMS model discussed by Lantz et al. [15]. At its centre is a monolithic server (Figure 1). All accesses to the graphics hardware go through the server which synchronizes accesses and controls graphics context. The hardware can be very fast. Therefore a single task driving one window can achieve excellent performance. In addition, software techniques can be used to enhance performance

by reducing the fidelity of rendering for moving objects [20]. But, because the window server manages the graphics pipeline and handles all windows on the screen it creates a performance bottleneck when multiple windows are active. Window-specific state information exists in data structures controlled by the X Server and all synchronization occurs there. Server traffic includes window updates, damage reports and colour table management in addition to handling mouse and keyboard events.

For example, X presents the user with a set of nominally independent windows but they actually interact because they share resources such as hardware, data structures and graphics context. Within the server requests are handled so as to reduce the context switching overhead, degrading the accuracy with which an application's temporal behaviour can be controlled[1]. One way to eliminate the undesirable coupling between windows is the elimination global state information, which needs to be accessed through a concurrency control mechanism [8].

Colour look-up tables (LUTs) are also a problem, since they are shared among all applications. Two strategies are possible. A static allocation method makes the entire LUT available to all applications by setting in a standard set of pre-defined colours. This method eliminates server overhead, but makes it impossible to realize effects like look-up table animation [5, 17]. A dynamic allocation method, on the other hand, has the server allocate entries as application tasks demand them. Thus, a task can, in principle, demand enough entries for its animation, then, after writing the appropriate patterns into pixel memory, issue commands that change entries to effect the animation.

X offers both dynamic and static methods, but its implementation of dynamic allocation is inadequate in a way that is typical of server-based systems. The commands issued by the application must result in precisely timed changes to the LUTs or the animation is jerky. But network activity, system demands and competition from other applications makes precise timing impossible to achieve [19]. Furthermore, since entries are allocated at run time there is no guarantee that entries will be available when the program runs: they may already have been acquired by other applications. Thus, even jerky animation may be impossible to provide.

Better animation can be provided by treating the LUTs as a local resource, allocating them statically to a window, and then letting the window access them directly. Then, as long as the application can synchronize itself with a real-time clock it can synchronize the LUT updates, ensuring smooth animation. Such an architecture requires that pixels in the frame buffer be grouped into window-specific blocks, and that the grouping be preserved in video processing.

ARCHITECTURE OF THE SYSTEM

The design of the window system *SLwindows* and the constructed prototype presented here were developed to support applications with temporal fidelity and to overcome the difficulties discussed above.

The SLwindows system was developed as part of an ongoing project that has produced a psychology workstation for administering vision and perception experiments, a task requiring real-time image display and response collection [3]. These activities require temporal performance predictable to within a few milliseconds. Ensuring the reliability of a program's temporal performance turned out to be difficult, and led to the development of software probes for such systems. SLwindows is an outgrowth of that activity.

The psychology workstation project uses a programming model in which an application is structured as a set of tasks running on a tightly coupled multi-processor and communicating through message passing. The application runs under the Harmony real-time multiprocessing operating system [12]. A typical system consists of two to six single-board CPUs plus peripherals. Prudent allocation of tasks to processors, combined with a suitable choice of priorities, ensures real-time response of critical tasks.

Most applications developed in the psychology workstation project take over the entire screen, without the benefit of a window system. A graphics server task is used to synchronize screen accesses from the tasks that comprise the application. Thus, parallelism was achieved at the application level but not in the graphics processor.

The first step towards a window system for time dependent applications was the port of the X Window system to the multiprocessor architecture [14]. This exercise showed that X could indeed run on a multi-processor but demonstrated the principal weakness of X's architecture. State information is so intimately shared among different parts of the server that it was essentially impossible to increase throughput by taking advantage of the possibility for parallel execution of different functions. Clearly, a higher

[1] The 'temporal behaviour' of an application is difficult to control. For example, to achieve temporal synchronization displayed events must occur at precise times, as specified by the real-time clock to which all temporal events make reference. Such control of temporal behaviour, in which events must be protected equally from earliness and lateness must be distinguished from good temporal performance, which can be produced by high throughput alone, without the necessity for real-time synchronization.

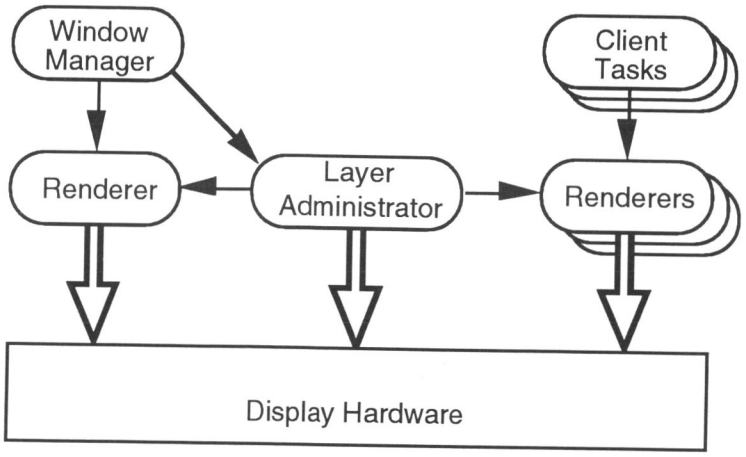


Figure 2. SLwindows' Task Structure

level of parallelism requires graphics hardware that is consistent with multiple servers.

The design and implementation presented here includes innovation in graphics hardware capabilities and in window system architecture. The overriding goal is to introduce parallelism from the application all the way to the graphics hardware, using parallel paths through the frame buffer and logically independent look-up tables. Because the system is supported by a multiprocessor multitasking operating system the application tasks can execute independently, unimpeded by other activities and synchronized to a real-time clock.

The underlying frame buffer hardware provides independent paths into the frame buffer for different applications and eliminates as much global state information as possible [7]. The stateless nature of the interface removes the need for shared data structures, for interlocking, concurrency control and for expensive context switching.

The second hardware component that made this project possible is a video processor developed to demonstrate a frame buffer architecture capable of storing reflectance values instead of RGB triples [9]. SLwindows uses the reflectance board as a dynamic cross-bar switch that is able to steer pixel data at video rates.

DESIGN OF SLWINDOWS

Based on the issues discussed above, the design of the SLwindows must meet several objectives. The system architecture must improve concurrency[2] in both software and hardware. To maintain this concurrency and reduce the overhead on graphics operations, concurrent modules must have direct access to the hardware. In addition to the improved concurrency, it is important to reduce or eliminate damage reports, a major source of overhead in window systems.

Hardware Requirements

To achieve desired performance, the graphics hardware must support multiple independent data paths. These paths may include components as complex as geometry engines and as simple as image pixels. The output of each of data path must be directed to a specific virtual frame buffer. The virtual frame buffer associated with a path should be configurable.

These virtual frame buffers, or *layers*, are distinct portions of a physical frame buffer that can be protected from access by inappropriate paths, using write masks, for example. A mechanism must be provided to identify the layer to be displayed at each pixel. Because layers occupy distinct physical memory they are immune to damage caused by window overlapping.

To achieve independence, each layer must have its own colour lookup table. For these colour tables to be properly accessed during the screen refresh, the mechanism that identifies the layer to be displayed at a pixel must also specify which colour table is to be used for the pixel.

Software Structure

To improve concurrency and remove the bottleneck in traditional designs, the window agent functionality [15] of the traditional single server is distributed into a layer administrator and many renderers (Figure 2). The layer administrator

[2] Concurrent means logically parallel but not necessarily truly parallel. True parallelism, on the other hand, implies concurrency. Thus, concurrency is necessary but not sufficient for true parallelism.

provides the mechanism for sharing the display: mainly the configuration, allocation, and manipulation of layers. Each client accesses the display hardware through its own renderer.

Like the window manager in the X window system, the SLwindows window manager is a client task that identifies itself to the system as the task responsible for window policy. Since it is a client task, it also accesses the display hardware through a renderer. When the window manager initializes, it requests a renderer from the layer administrator, with a layer for the background and a layer for emergency messages. The emergency message layer is allocated immediately to make sure it is always available.

The window creation process begins with a client request to the window manager. The window manager in turn requests a renderer and two layers from the layer administrator. The renderer and one of the layers is for the client area while the second layer is given to the window manager's renderer to draw the window frame[3]. All subsequent requests for window manipulation, such as movement, resizing or reordering, are translated by the window manager into requests to the layer administrator, asking it to handle both the client layer and the underlying frame layer. The layer administrator services these requests by modifying hardware state information to which it alone has access, without needing access to either the client or frame layers.

When a new renderer and layer are requested, the layer administrator creates a layer and allocates a colour lookup table. A pipeline is initialized to use the new layer and colour table, after which the new renderer is created and granted access to these resources. The layer administrator is the only task permitted to reconfigure the state of the hardware. The configuration includes the size, positioning and ordering of layers.

The renderer provides access to a layer and manages the associated colour table. Once created, the renderer has full and direct access to its layer through its own data path. Drawing and changes to colour table entries are performed without any communication with other renderers or with the layer administrator. In this respect, the renderer behaves as a client's personal X server, customized to serve the client.

This distributed approach results in a system that has no central server. It is for this reason, this server-less window system is called SLwindows

Internal Communications

Communication between different components of the SLwindows system occurs only in response to user input. This input is usually single threaded because the user has a single source of focal attention, but can be multi-threaded without violation of the SLwindows model. User requests to resize, move or change the ordering of windows are assumed to occur infrequently, so they can be processed less urgently than animation requests from applications. As mentioned above, user requests are first processed by the window manager which then makes the appropriate requests to the layer administrator. The layer administrator then communicates with only the affected renderers: the window manager's renderer and the renderer of the client that owns the affected window. All other renderers are uninvolved in this communication activity.

The layer administrator only needs to notify renderers of resize events. Changes in layer depth and placement are handled directly by the layer administrator, which simply modifies the appropriate hardware state. The window manager and client renderers of the affected window(s) continue oblivious to the changes. Hence, there are no damage reports due to changes in the overlapping of windows.

Resize events are signalled to the affected renderers because they affect screen drawing. For example, if a renderer treats its layer as a viewport, a resize event requires image rescaling. If it treats its layer as a window (one-to-one window to viewport mapping), the event may signal that more of the image is exposed, possibly requiring extra drawing.

The Window Manager's Renderer

At first glance, concurrency appears lost because the window manager's renderer owns all the layers used for frames. But a careful inspection of interaction requirements indicates that concurrency is unimportant or even detrimental in the window manager. User-originated window actions, like moving or resizing, require focal attention and must be controlled serially by the user, with the speed of the user's actions closely integrated with the system response. For similar reasons, application-originated actions similar to those originated by a user, which could occur concurrently, should be rare, because it would be impossible for a user to control a system that produced frequent reactions resulting from internal conditions and not user input.

The LID buffer provides another interesting interactive feature, since it can be used to identify the window to which an input event is directed, with speed and precision that does not depend on window shape.

IMPLEMENTATION OF SLWINDOWS

SLwindows was implemented on the multi-processor psychology workstation [6], using an Adage/Ikonas RDS–3000 Display System. This

[3] The frame includes the border, title bar, and sizing boxes.

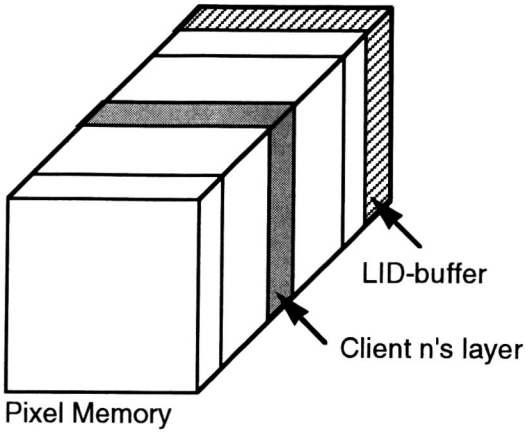


Figure 3. Partitioning of Frame Buffer Into Layers

system provides great flexibility for prototyping because of its open bus architecture. The systems used for SLwindows are configured with 1024x1024x32 bits of image memory and a BPS32 graphics microprocessor.

Concurrent data paths are provided by linking the RDS–3000 display system to the multiprocessor application host using a custom-built interface [7]. The individual paths are identified by their sender identification (SID) and are accessed by specifying the SID during an access to the RDS–3000. The interface is designed so that accesses are completely atomic. Each SID has a write mask for controlling access to image memory on a bitplane by bitplane basis.

Implementation of Layers and Video Processing

Each layer is comprised of a set of bitplanes (Figure 3) which are protected by setting the writemasks of each renderer to allow writing to only the layers it owns. Each renderer is also given a block of the RDS–3000's 1024 entry colour lookup table for use as its colour table.

A special layer owned by the layer administrator, called the *LID-buffer* , is used to identify the layer that is visible for each pixel (Figure 3). The LID-buffer is used pixel-by-pixel as an index to the layer that should be displayed for a each pixel (Figure 4). When a pixel is fetched from the frame buffer it contains the bits for every layer in the system, including the bits of the LID-buffer and the layer that is visible at the pixel. A custom designed board, normally used for reflectance calculations [9], is used as a dynamic cross-bar switch, using the contents of the LID-buffer to steer only the bits of the visible layer to the colour hardware. The visible layer's LID is transformed by the dynamic cross-bar into the base index of the layer's colour table. This base index combined with the layer's data form a ten bit index into the RDS-3000's colour lookup table.

General Hardware Limitations

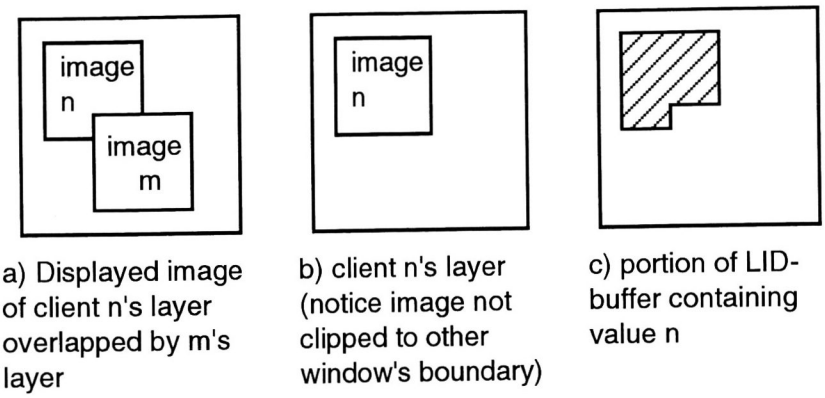
The RDS–3000 and its interface provide only eight distinct SIDs. Since the layer administrator requires one to access the hardware and the window manager's renderer requires another, a maximum of six client windows may be open at a given time. Although it is evident that a commercial system must provide more than six windows, this number is sufficient to prove the concept.

Since the reflectance board was not designed to be used as a dynamic cross-bar, it has some limitations. The restricted flexibility requires extra bitplanes to be assigned to the LID-buffer reducing the bitplanes available for client and frame data. The resulting prototype system may have as many as six four-colour (2-bit) client windows (with two colour frames) and one four colour alert layer for emergency messages. Client windows are restricted to a maximum of 128 colours (7-bits).

Move Layer Event

In order to make the movement of layers transparent to the renderers when layers are implemented as a collection of bitplanes, the hardware must support the pan and scrolling of individual bitplanes. Unfortunately, this feature is not available in the RDS–3000[4]. Therefore, contrary to the theoretical design, the prototype system must notify the renderer when its layer is moved. The renderer then corrects its x,y origin and invokes the graphics microprocessor to

[4] It existed in early image processing frame buffers but disappeared during the 1970s and early 1980s.



a) Displayed image of client n's layer overlapped by m's layer

b) client n's layer (notice image not clipped to other window's boundary)

c) portion of LID-buffer containing value n

Figure 4. Operation of the LID-buffer

perform a fast bitcopy of its frame buffer image to the new location. Since the graphics microprocessor assumes the SID of the client for whom it is currently working, the bitcopy uses the writemask mechanism to avoid corrupting other layers.

Special Renderers

One attractive feature of this architecture is the possibility of supplying special purpose renderers for application tasks that have special rendering requirements. For example, a special animation renderer was implemented to economize on responses to move events. This renderer relies on the frequency of animation changes to deal with the damage caused when a window is moved. Instead of wasting bandwidth copying an image that will be redrawn as part of the animation, the renderer simply updates its origin and waits to draw the next frame of the animation on behalf of the client.

In addition to the normal and animation renderers, two other special renderers were implemented. The live video renderer is a normal renderer augmented with operations to change the destination and size of input from video capture hardware in accordance to its layer's position and size, enabling the display of live video. The window manager renderer was designed to share a single colour table block among all of its layers. This was necessary due to partitioning restrictions imposed by our version of a dynamic cross-bar.

COMMENTS ON THE IMPLEMENTATION

The prototype hardware and software were implemented using a decade-old graphics system. While it is obsolete as a high-performance display system, its open architecture lends itself extremely well to reconfiguring and to prototyping projects such as SLwindows. Some of the desirable characteristics are the presence of a programmable graphics processor, a clean and high performance backplane bus, reconfigurable video chain and programmable video timing generator.

The prototype implementation is judged to be a success, in that it provides full system functionality on a system with modest CPU bandwidth: four to six MC68020s clocked at 12.5 MHz. For example, it is able to present live video in a moving window without noticeable artifacts, and to provide continuous control of four simultaneous smooth animations on a single frame buffer. In addition, because each renderer's colour table is private, colour table animation with precise temporal control is easy to implement. Yet, the prototype implementation suffers from a variety of shortcomings that show how the hardware should be improved. Simulation of a dynamic cross-bar using the reflectance hardware consumed several image planes that would otherwise have been available for window contents. The resulting shortage of image planes produced allocation problems such as pixel bit fragmentation, along with poor dynamic window creation and deletion. Thus, future implementations require a true dynamic cross-bar switch.

The graphics performance of the system was found to be limited by the throughput of the MC68020 CPUs running applications. This shows that when server and damage overhead is reduced systems need a larger share of their computational bandwidth in the application CPU, and a smaller share in the graphics processor. Another solution might place each renderer on a specialized graphics processor instead of on the application CPU, at the probable cost of complicating the overall graphics hardware, and possibly the real-time operating system.

Finally, the prototype offered poor support for double buffered graphics. The double buffer implementation uses two layers owned by the same renderer. Whenever rendering to a buffer is complete the renderer requests the layer

administrator to swap the buffers. This operation would ideally be performed by the renderer itself, without any synchronization overhead. A second cross bar switch, not necessarily dynamic could be placed in the video path to enable this functionality. Each renderer would then own sets of input and output bits in this cross bar in addition to its layers.

A useful result of our prototype is the identification of a skeleton multimedia application that could be the basis of a future benchmark. The application should drive three imaging windows at a uniform rate of frame update and a high degree of temporal synchronization, within one frame of about 17 milliseconds. An ancillary task is processing sound information, that must be reproduced at a jitter-free rate within the acceptable limits for *wow* and *flutter*, while synchronized to the images. Hardware to produce a suitable audio signal is currently being constructed in our lab.

Clearly, there has to be sufficient computing power to generate the information. In our architecture that computing power is achieved by partitioning the application into light-weight tasks and allocating them to a set of tightly coupled processors. An obvious partitioning allocates tasks supporting each window to a unique processor and decoupling the performance of each window from each of the other windows. If each window is 200 by 300 pixels the required data rate is about 2M pixels per second per window, uninterrupted. This rate is close to the throughput that is possible for our obsolete hardware, and easily within the capacity of more modern graphics systems.

The image and sound streams only interact for temporal synchronization, and there is almost no implicit interaction through resource competition. In fact the streams do not interact with one another, but interact individually with a system-wide real-time clock. A system clock with a five millisecond resolution is sufficiently accurate to maintain the required synchronization tolerance.

In applications where elaborate 3D hand controllers are required, total system delay needs to be less than approximately 40 milliseconds, from the hand controller to the change in the displayed image. One can use the modular nature of the architecture and support such devices on separate processors as long as care is exercised in defining tasks and message paths so that no unnecessary delay is introduced. Bajura et al. [2] suggests that predictive control can reduce the apparent delay even further.

CONCLUSION

The design and implementation of a prototype have demonstrated a novel window architecture that provides predictable temporal performance and can meet the requirements of temporal fidelity in such applications as multimedia, navigation through virtual worlds and animation.

The implementation of the demonstrated prototype is limited by the capabilities of the available hardware. However, we may hope that improvements in display subsystems will provide the support for more realistic implementations. For example, SGI systems currently provide a window ID that allows a window manager to specify the visible region of a window, simplifying the writing of pixels by an application [18]. This provides some of the functionality of the write masks in the RDS–3000 system but needs to be duplicated when the bits are read out into the video pipeline to provide support for implementations like SLwindows. Another characteristic of the design is a large pixel depth: 32 bits is satisfactory for a demonstration prototype, but inadequate for a realistic system. Currently 32 bitplanes is at the high end of workstation graphics, but this parameter is currently increasing. A recent product announcement [11] described a graphics engine that supports a pixel depth of 128 bits, including some type of dynamic cross-bar capability. Clearly, advances in technology are removing some of the limitations in the demonstrated prototype. As each limitation is removed the barrier to removing the others becomes lower, so it is reasonable to expect that workstations of the immediate future will be able to support window systems that have the advantages demonstrated in the SLwindows prototype implementation.

ACKNOWLEDGEMENTS

This project could not have been undertaken without hardware designed constructed and debugged by Chris Wein and Ed Dengler. Once again, we would like to thank Nick England for providing us with the 'Ikonasaurus', a graphics system whose flexibility and configurability makes it still usable for research long after every one of its components is obsolete.

REFERENCES

1. Abi-Ezzi, S. Graphics Software Architecture for the Future (Panel, Chair: A. van Dam). Computer Graphics (SIGGRAPH'92), vol. 26(2), (1992), pp. 389–390.
2. Bajura, M., H. Fuchs and R. Ohbuchi. Merging Virtual Objects with the Real World. Computer Graphics (SIGGRAPH'92), vol. 26(2), (1992), pp. 203–210.
3. Bartram L. R., K. S. Booth and W. B. Cowan. Issues in the Design of Workstations for Psychology Experimentation. The 1989 IFIP WG 5.10 International Working Conference on Workstations for

Experiments (Lowell, MA, July, 1989), pp. 164–172. Springer-Verlag, 1993.

4. Bertrand, P. F. A Server-less Window System for Multi- tasking, Multi-processor Systems. M. Math Thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, 1993.
5. Booth, K. S. and S. A. MacKay. Techniques for Frame Buffer Animation. Proceedings of Graphics Interface'82, Toronto, Ontario, (1982), pp. 213–220.
6. Booth, K. S., W. B. Cowan and D. R. Forsey. Multitasking Support in a Graphics Workstation. Proceedings of the 1st International Conference on Computer Workstations, (1985), pp. 82–89.
7. Cowan, W. B., C. J. Wein, M. Wein and K. S. Booth. Hardware Support for Multitasking Graphics. Proceedings of Graphics Interface'91, Calgary, Alberta, (1991), vol. 199–206.
8. Cowan, W. B., M. Wein and P. F. Bertrand. A Window System Architecture to Support Consistency and Context. Unpublished manuscript, 1992.
9. Dengler, E. A. and W. B. Cowan. A New Frame Buffer Architecture Capable of Storing Reflectance. Research Report CS–92–25, Department of Computer Science, University of Waterloo, Waterloo, Ontario, 1992.
10. Evans & Suterland and Sun Microsystems. Product Update: The Fastest Graphics in the World. Press Release, Salt Lake City, UT: Evans & Sutherland, 1992.
11. Fisher, S. S., M. McGreevy, J. Humphries and W. Robinett. Virtual Environment Display System. Proceedings 1986 Workshop on Interactive 3D Graphics, Chapel Hill, NC, (1986), pp. 77–87.
12. Gentleman, W. M., S. A. MacKay, D. A. Stewart, and M. Wein. Using the Harmony Operating System, Release 3.0, NRC/ERA–377. Ottawa, Ontario: National Research Council of Canada, 1986.
13. IBM. OS/2 System Programming Guide. International Business Machines Corporation, Armonk, NY. Pub. No. 10G6494, 1992.
14. Kelley, J. V., K. S. Booth and M. Wein. Design Experience with a Multiprocessor Window System Architecture. Proceedings of Graphics Interface'89, London, Ontario, (1988), pp. 62-69.
15. Lantz, K. A., P. Tanner, C. Binding, K.-T. Huang and A. Dwelly. Reference Models, Window Systems, and Concurrency. Computer Graphics (ACM SIGGRAPH Workshop on Software Tools for User Interface Management), vol. 21(2), (1987) pp. 87–101.
16. Scheiffler, R. W. and J. Gettys. The X Window System. ACM Transactions on Graphics, vol. 5, (1986), pp. 79-109.
17. Shoup, R. G. Color Table Animation. Computer Graphics (SIGGRAPH'79), vol. 13(2), (1979), pp. 8–13.
18. Silicon Graphics. IRIS Indigo Family Technical Report (Version 1.0), 1991.
19. Sun, F. K., W. B. Cowan, and K. S. Booth. Understanding Visual Effects in Windowed Environment. Proceedings of Graphics Interface'90, Halifax, Nova Scotia, (1990), pp. 100–105.
20. Turk, G. Re-Tiling Polygonal Surfaces. Computer Graphics (SIGGRAPH'92), vol. 26(2), (1992), pp. 55–64.
21. Wickens, C. D. Engineering Psychology and Human Performance (2nd ed.). New York: Harper Collins, 1992.