Hardware Support for Multitasking Graphics

William Cowan, Christopher Wein*, Marceli Wein†, Kellogg S. Booth§*

**Department of Computer Science, University of Waterloo, Waterloo, Ontario, N2L 3G1.*
†Institute for Information Technology, National Research Council of Canada, Ottawa, Ontario, K1A 0R6.
§Department of Computer Science, University of British Columbia, Vancouver, British Columbia, V6T 1W5.

Abstract

Significant improvements in the performance of many window systems are possible if tasks are allowed to write directly to the display instead of having to synchronize with a server. Such operation depends on the ability of tasks to perform atomic transactions with the graphics subsystem, a capability that depends on the provision of suitable hardware in the interface to the graphics subsystem. This paper describes the design and construction of an interface that makes atomic transactions possible for multiple tasks by extending the sender id mechanism into the address space of the host. New methods of structuring multitasking graphics software that are made possible by the interface are described, along with similar hardware features that could further improve the performance of graphics programs in multitasking environments.

1. Motivation and Introduction

The search for higher performance through parallelism in computer graphics has usually been concentrated on rendering, the time-consuming process by which data structures are turned into pixels. There the search for parallel algorithms has been very successful [1]. Additionally, there are successful designs for hardware that make it possible to take full advantage of the parallelism of some algorithms [2]. This parallelism, which commonly occurs at very small granularity, is usually conceived as being internal to the graphics system itself, well below the level at which user programs operate. The purpose of the present paper is to discuss hardware support for parallelism at large granularity. A typical example occurs in the psychology workstation [3], which has been the focus of several years development in the Computer Graphics Laboratory at the University of Waterloo. An issue central to this project is the necessity of establishing precise timing for the delivery of an image and the user's response to it, making access to the graphics hardware with a minimum of overhead an important design goal. In the psychology workstation an application program consists of a collection of tasks running on one or more processors under the Harmony operating system [4]. The number of tasks is relatively small, usually in the neighbourhood of twenty. Each plays a distinct role, so the granularity is large. When several tasks wish to access the graphics hardware, synchronization is needed, which can be computationally expensive if the tasks perform their graphics operations in small chunks. This paper presents one solution that was implemented and discusses how hardware might be designed to provide efficient graphics support for such a system. Virtually all new high-performance workstations, such as those built by SGI, Stardent, and HP/Apollo, support multitasking in one form or another. Thus, solutions to the problems this paper addresses will be increasingly important for the next generation of graphics accelerators as performance bottlenecks are identified and cleared by putting additional functionality into hardware.

The relationship of multitask programs to graphics capabilities is not a new problem. Large granularity multitasking is useful for matching the multiple threads of user activity with-multiple threads in the computation to achieve a program that more naturally mimicks the user's conceptual model of a system [5, 6, 7]. It is also useful for improving throughput by overlapping update and refresh in the frame buffer [8], eliminating performance bottlenecks by allowing flexible scheduling of tasks based on real-time constraints imposed by the parameters of human real-time performance [3], and separating rendering activity from screen management activity in a window system [9]. In each case, the introduction of multitasking brings with it the problem of synchronizing access to the underlying graphics hardware which, if not resolved efficiently, will sabotage the entire design by creating intolerable execution delays.

Three popular systems illustrate the range of software techniques that are currently used for synchronizing access to graphics hardware. The Macintosh, prior to System 7 [10], solves the problem by executing all graphics primitives within a single thread of control. Thus, non-atomic access to the graphics hardware, such as the need to assign a graphics context prior to drawing a line that uses it, is left in the hands of the application programmer. The X window system [11], on the other hand, was designed to allow multiple tasks independent access to the graphics hardware. It solves the synchronization problem by having all access to the hardware performed within the thread of control of the server. Application tasks send requests to the server, which serializes the requests and manipulates the hardware with appropriate preservation of state. NeWS adopts an intermediate course [12]. All graphics operations are executed within the server, but multiple threads of control are allowed inside the server. In fact, a user program can be written so that it executes completely within the server. The implementation within the server, however, is not pre-emptive so that each thread of control is able to manipulate graphics state with confidence that it will not be interrupted. The above solutions all exist to allow software to interact gracefully with hardware that provides no special support for multitasking. Each one has a deficiency, however. The Macintosh does not allow pre-emptive multitasking; the others do not allow the user program to control the time at which graphics operations are executed. Thus, none is suitable for real-time multitasking graphics, which is needed in the psychology workstation.

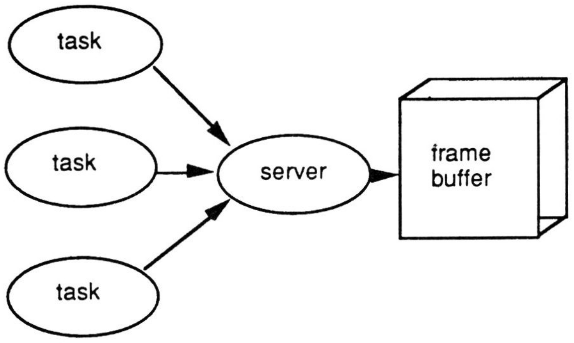


Figure 1. Several tasks accessing a graphics system synchronized by a server.

Such support can be provided if the graphics hardware is sufficiently primitive, and if allocation of the physical frame buffer to tasks is static [13]. Static allocation is important, because it creates a need for explicit synchronization: A task must acquire a grant to write on a set of physical pixels and the grant is guaranteed not to be revoked without explicit synchonization. For example, the task might receive a request to stop writing, reply with an acknowledgement, then do no writing until a new grant with a new set of pixels is given. Allocation strategies are important in the implementation of window systems, so they are discussed later in the paper. To see how to provide multitask support with static allocation, suppose the frame buffer resides completely within the address space of user processes and contains no internal state. Then an administrator coordinates access to the frame buffer, responding to requests for physical pixels by determining which of the pixels are free and replying with suitable grants.

Having the frame buffer within the address space of the user is important because it allows the interface to have no internal state: a single bus transaction provides read or write access atomically. Frame buffers with more complicated interfaces, using DMA or address registers, require several bus transactions per transfer and consequently have internal state. Then every graphics transaction must be explicitly synchronized. Similarly frame buffers with internal state that affects graphics transactions, such as write masks, require synchronization of each transaction. Other aspects of internal state of the frame buffer, such as colour look-up tables (LUTs), must be standardized by the administrator. The initial configuration of the psychology workstation used a 'dumb' frame buffer with static allocation to provide direct access to the frame buffer for individual tasks. This direct access is very important because graphics transactions must have timing latencies of the order of milliseconds in a system that requires about a millisecond to synchronize. Thus, synchronizing every graphics transaction is not feasible.

Unfortunately the graphics performance of the system thus configured is severely limited by the frame buffer, by its access time and by its inability to do any graphics processing. Most enhancements to the hardware performance reduce access bandwidth at the cost of introducing internal state into the graphics system. Unfortunately, the internal state of such systems makes synchronization of accesses essential and the loss caused by synchronization overhead greatly outweighs all other benefits. This problem applies to all methods of synchronizing access to the system, even controlling the address space occupied by the graphics system using memory mapping hardware [14]. The cost of a context switch unavoidably involves changing all state referring to the graphics system. A different solution presented itself in the form of an almost surplus Ikonas/Adage RDS 3000 display system, a design almost a decade old, but incorporating several features designed to support multiple processors, features that are found on almost no modern systems. These features contain two flaws, however. They are designed for multiprocessing, not for multitasking, and the interface treats the host as a single-processor, single-task machine. To use the RDS 3000 a new interface had to be designed, since Ikonas/Adage provided only interfaces to Digital Equipment Unibus and Qbus and the psychology workstations use VME-bus. In this design the challenge was to create an interface capable of making these multitasking features available to Harmony tasks.

The remainder of this paper describes a solution to this challenge and the insight it offers into extending hardware support for large granularity multitasking graphics. The next section describes the objectives of the interface design, relating the principles of multitasking to the RDS 3000 architecture. The third section discusses the design and construction of the interface. The paper closes with a discussion of the capabilities gained by this new interface and those aspects of multitasking graphics it is unable to support. These lacunae suggest new hardware features, likely to be needed by graphics systems that are expected to support more advanced software.

2. Design Objectives

The overall objective of the design is to create an interface that allows maximum parallelism, thus minimizing queues inside servers and the necessity for time-consuming synchronization. The optimization is discussed in the context of features provided by the RDS 3000, which is a raster display system, but it can be considered from a general perspective, from which principles applicable to graphics systems of any type can be deduced.

A graphics system appears logically as a data structure that is accessed by two processes, one being the host system which determines the contents of the data structure, the other being the video pipeline which uses it to create pixels on a display surface. In a vector display system this structure can be the display list; in a raster system it can be the image memory; in a system with a geometry pipeline it can be the data structure that feeds the pipeline. Most graphics systems contain a large amount of internal state, which can be divided into two parts. The first part affects only transactions between the host and the data structure, an example being the write masks [15]. The other affects the interpretation of data in the video pipeline, an example being the colour LUTs. Although a task writing to the graphics system needs to know both parts of the state to get predictable results, the two parts of the internal state have significantly different synchronization requirements. To see why, imagine that a writing task has the ability to tag its transactions. If the hardware that interprets the host state is able to interpret the tag it can apply its state in a way that is specific to the writing task. Thus, different tasks can utilize different state without any need for synchronization. However, if the tag is not stored in the main data structure it is unavailable in the video pipeline and hardware that interprets video state cannot apply it in a task specific way.

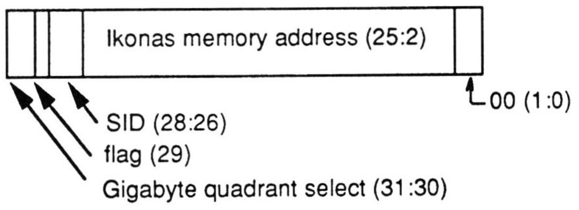
This process is easy to understand in the RDS 3000. Transactions on the processor bus of the system can be tagged by a 3-bit sender id (SID), which selects the write mask used for read/write operations into the image memory. No provision is made for storing the SID in image memory or for propagating it down the video pipeline. (It would be more correct to say 'almost no' since the programmer can set aside parts of the image to be used as SID in the video pipeline. This possibility is addressed in the final section.) Tasks accessing the frame buffer can use the SID capability to create atomic access, using host state specific to the accessing task, state that was set on earlier transactions. Contrarily, state in the video pipeline does not have this capability. It must be synchronized by a server, as is commonly done for the colour LUTs: a task requests the use of an entry from the server and can use it only after acknowledgement from the server. Note that the considerations discussed here apply to whatever point in the chain from host to video screen where SID information is no longer retained, a point that can be determined by software if the hardware is sufficiently flexible.

Thus, in the RDS 3000 the SID provides atomic access to the image memory through the write masks. Two tasks using different write masks are able to access image memory simultaneously because the interface hardware contains multiple write masks and uses the SID for deciding which one to use. How SIDs are set is thus a critical issue. Two different methods are used. The first method, used by the original host interface and by the BPS, a high performance graphics processor on the processor bus, makes available a special register containing the SID, which is settable by the processor. Every subsequent bus transaction from that processor is accompanied by the SID. This method allows different processors on the processor bus atomic access to the image memory, but it does not allow different tasks on a processor atomic access. Thus, it is well-suited to early versions of the BPS, which were not interruptible, but not to later ones, which could be interrupted by other processors. Similarly it is well-suited to hosts running operating systems, like Unix, that do all access through a single device driver, but it is unable to support hosts on which multiple tasks access the image memory simultaneously. The second method, used by the MPC, a MC68000-based computer used to control peripherals, maps the address space of the processor bus into its local bus through a large set of small windows, each with its own SID. Thus, different tasks running on the MPC can use different SIDs, allowing them atomic access to image memory, a design that is well-suited to an interruptible system for controlling interactive devices. Access to window registers must, of course, be synchronized by an administrator, and synchronization should occur infrequently to minimize overhead. Unfortunately, the windows are small, 2K Ikonas words corresponding to as little as one line of image memory, and there are only 512 of them with the result that frequent synchronization is likely to be necessary for many application programs. This limitation is a consequence of the smallness of the 68000 address space, 16 Mbytes, compared to the 64 Mbyte address space of the processor bus.

In creating a new host interface the use of multiple SIDs to create atomic access to the image memory for several tasks simultaneously is an essential feature. Ideally, a task should be able to access the whole processor address space with its own SID. In particular, features such as mapping registers that create state in the interface itself must be replicated enough times that they are used atomically when an access is made to the processor bus. Since the SID can be considered logically as extra address specification the host bus must be able to address 64Mbytes times the number of SIDs made available. This condition can be fulfilled in a straightforward way on a host with a large enough address space, especially since the RDS 3000 is designed on the expectation that it will be a slave, comunicating to the host only via interrupts.

3. Design Results

The primary objective of the interface design is to provide the host computer with atomic access to as many of the resources of the RDS 3000 as possible. The Unibus/Qbus interface released by Ikonas/Adage did not have this capability, preventing the host from taking full advantage of the multitasking features of the Ikonas. This design was sensible since the interface was designed for use in time-sharing systems, which require access to the graphics system to be synchronized by a driver/handler. Complicating the interface design was the inability of the Unibus/Qbus of that era to address more than 256 Kbytes, which is very small compared to the 64Mbyte address space of the RDS 3000 processor bus. To overcome this limitation, that interface accesses the processor bus through a set of registers that contain the address on the processor bus, the SID and the function code, a set of signals on the processor bus indicating the nature of the transaction, read/write, pixel/word, high/low resolution and so on. Command and status registers are also provided, along with pointers to Unibus data. Both write through and DMA input/output are supported. To perform one memory operation each of the registers is set by a separate Unibus cycle, violating the atomicity required for multitasking operation. (This problem could have been overcome by allocating a set of registers for each SID. For several reasons this approach makes little sense. There is limited address space in the more or less statically allocated I/O page of the Unibus architecture; interface hardware would be needed to synchronize simultaneous requests; and the extra facilities offered would lie outside the expected programming paradigm.)



Ikonas memory address (25:2)
00 (1:0)
SID (28:26)
flag (29)
Gigabyte quadrant select (31:30)

Figure 2. Interpretation of the 32 bit VME address word when used to access the Ikonas graphic system.

Interface design is simpler with the VMEbus, because its 4 Gbyte address space easily encompasses the address space of the RDS 3000 processor bus. The approach chosen was to reserve 30 bits of address space (one quarter of the VMEbus address space) encoding the function code, SID and address within those 30 bits. Twenty-six bits must be allocated to the RDS 3000 address because the VMEbus is byte addressed while the RDS 3000 is long word addressed. This addressing choice forces all VME transactions with the interface to be of type LONG and to be longword aligned [16]. The remaining 4 bits, 26–29, must encode the function code and sender id. However, four bits are insufficient to encode these values directly owing to the number of different function codes. To solve this problem, only the SID is broadcast as part of the Ikonas address and each SID has a function code register located on the interface. During a transaction, the interface uses the broadcast SID to obtain the right function code from the register and communicates it to the RDS 3000. To allow setting of the function codes the remaining bit is used as a flag to indicate that a task is accessing the on-board function code register corresponding to the given SID. This function code lookup method adds overhead to update the function code, but it maintains atomicity between SIDs. Furthermore, most accesses set the function code once and execute many transactions using that function code.

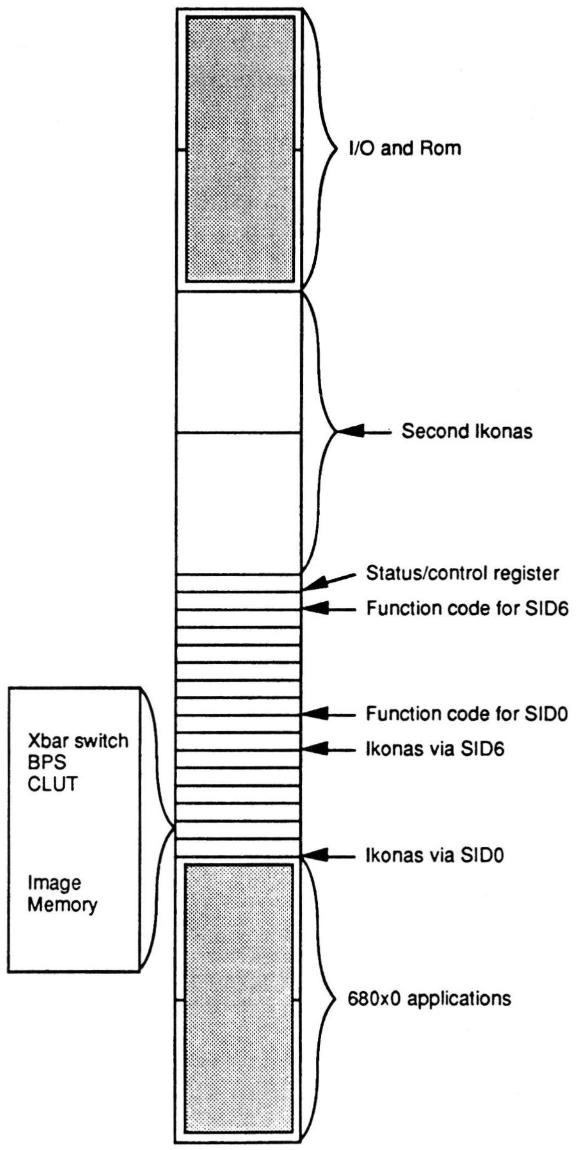


Figure 3. Address space of the VME bus, showing the portions used by the seven Ikonas images, the seven function code registers and the status/control register.

Of the eight possible SIDs, one is reserved for the interface control register, leaving seven for VMEbus tasks. The interface control register provides functions that are controlled via the processor bus, such as BPS operation and RDS 3000 generated interrupts. Thus, an address is constructed as follows: Bits 0-1 are zero (because all transactions are aligned and of type LONG), bits 2-25 contain the address to be put on the processor bus of the RDS 3000, bits 26-28 contain the SID, bit 29 is the flag indicating memory or register access and bits 30-31 indicate which gigabyte (of four) the RDS 3000 occupies. Of the four gigabytes in the VMEbus address space, only 1 and 2 are available since user code and data reside in gigabyte 0 and system code resides in gigabyte 3. Therefore, a maximum of two independent RDS 3000 systemss can reside within the VMEbus address space.

Two implementation methods are possible. Each produces a subsystem consisting of two cards; one,the host interface card, residing in the VME chassis and accepting VMEbus cycles, and the other, the graphics interface card, residing in the RDS 3000 chassis and generating RDS 3000 bus cycles. The two cards communicate by a protocol that controls the movement of data across the physical connection between the two cards. The first option is to design and build a new graphics interface card along with the host interface card. This option allows simultaneous transmission of all address, data and control signals, ensuring the best possible performance at the cost of high design effort. The second option is to build only the host interface card, connecting to the existing graphics interface card and emulating the card-to-card protocol used by the Unibus interface subsystem. This method takes advantage of existing hardware and eliminates the need for RDS 3000 prototype cards, but provides slower operation because all signals must be multiplexed over a 16 bit interface bus. Preliminary calculations indicated that the speed of this option is adequate, so it was chosen. The correctness of this decision was born out by subsequent performance measurements, reported below.

The existing Unibus interface provides a write through protocol: the host writes to a set of registers describing the function it wishes to execute, then writes the data through a pair of 16 bit registers. The VMEbus interface emulates this functionality by accepting a VMEbus cycle. Then, while holding the VMEbus, it sequences all signals over the interface bus at the maximum rate. When the graphics interface acknowledges the completion of the operation, the VMEbus cycle is terminated. All information required to perform a complete RDS 3000 memory operation is contained directly or indirectly in the signals presented by a single VMEbus cycle. Atomicity of RDS 3000 bus operation is thus preserved.

In addition to supporting read/write cycles, the interface provides a control/status register that allows the host to control other features of the RDS 3000. Interrupts, both processor and video, are transferred directly from the processor bus to the VMEbus. They are auto-vectored and are cleared when the control/status register, which indicates the source of the interrupt, is read. This register also allows the host to control execution of the Ikonas bit slice processor and to reject memory updates that occur outside the video blanking interval. It also has a bit which, when set, resets the internal state of the interface.

The VMEbus interface is designed as a 16 state algorithmic state machine (ASM) which handles both the VMEbus protocol of the VMEbus and the interface protocol. The ASM is implemented with three PALs (programmable array logic) to increase flexibility and to reduce the number of parts. Currently, the ASM is clocked at a 4 Mhz using a PAL logic family with a propagation delay of 15ns. At this speed the interface is capable of a maximum transfer rate of 14.4 Mbits per second, close to the maximum speed of the VMEbus in the psychology workstation, which has no cards capable of block mode transfers. Without any redesign the interface could be clocked as fast as 16 MHz, which would accomodate the full bandwidth of the VMEbus with block mode transfers. Faster parts, wider PALs and shorter cables should permit a miximum clock rate of about 25 MHz, with a 80 Mbit per second transfer rate that would essentially saturate the RDS 3000 bus on read accesses. Thus, the choice of a multiplexed interface design does not limit interface performance.

4. Discussion

The 4 Gbyte address space of the full VMEbus is large enough to make it possible to construct an interface that allows atomic transactions with the image memory using the SID mechanism. The interface is fast enough to use the full bandwidth of the VMEbus, so that it is saturated only by block transfers or by several processors accessing it simultaneously. Block transfers cannot be sustained by a single processor, because it must stop to calculate new data. Thus, the full interface bandwidth can be used only when several processors are using the graphics system simultaneously. Without the SID mechanism such usage would be impossible and the response time of graphics operations would be limited not by hardware but by the software overhead of tasks synchronizing with one another or with a server. The remainder of this section discusses a variety of possibilities opened up by this hardware development. It begins by discussing new software capabilities that are possible using this interface, then describes features that cannot be supported, concentrating particularly in the area of high performance window systems. It concludes by mentioning a variety of new hardware features needed if graphics systems are to have the flexibility to support this software.

The present interface was designed specifically to support direct frame buffer access by tasks that use static areas of the display surface. The software interface involves an administrator during initialization. A task that intends to write to the frame buffer requests an area from the administrator. The bounds of the area, which may be affected by other areas, are replied to the task, along with a unique SID. The task then uses this SID when writing to the frame buffer and is guaranteed that its write cycles are atomic. This scheme is used in the context of video state, consisting of colour LUTs, cross bar switch, window and viewport registers, and the like, that is initialized to standard values by the administrator and left untouched by the application tasks. Of these resources one is potentially shareable, the colour

LUTs. Entries in the tables can be shared among tasks, with the administrator handling the allocation. Many different strategies are possible, similar to the set offered by X [17]. Experience shows, however, that animated programs find colour table allocation to be a major problem in X [18]. Another possibility exists given the particular hardware of the RDS 3000, which has four 256 entry colour LUTs and a cross-bar switch in the video data path. Suppose three bits of image memory are reserved for storage of the SID of whatever task wrote a particular pixel. These three bits, propagated down the video pipeline, are routed by the cross-bar switch so that two are used to select among the four LUTs and the third is used as the highest order bit indexing into the selected table. Effectively the LUTs then become eight 128 entry tables, each independently dedicated to a SID, so that they may be accessed directly by the task that owns the appropriate SID. A similar feature recently appeared in hardware specifically designed to support X [19], allowing each of sixteen windows to have its own LUT.

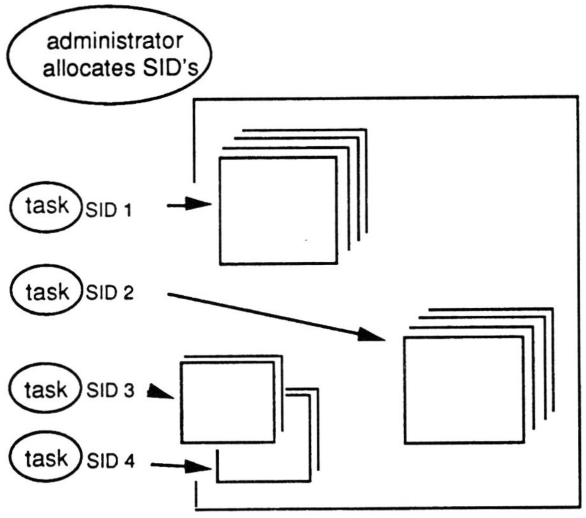


Figure 4. Several tasks writing to distinct parts of the frame buffer, using their SIDs for independent access to various bit-planes and pixels.

The multiple LUT technique, which is made possible by the flexibility of the RDS 3000 hardware, points out an area where small amounts of additional hardware produce significant extensions of functionality. The solution that allows tasks to use independent LUTs requires a fixed setting of the cross-bar switch. The bits dedicated to SIDs are necessarily fixed, but the other bits need not be. If they were to be replicated, with one set per SID, then each task could set the low bits of the cross-bar independently. Then all the graphic techniques made possible by LUTs and cross-bars [15] could be practiced independently by any task using the graphics system.

The BPS, which is used when high speed operation of relatively simple tasks like scan conversion is required, shows another aspect of SID usage. In normal use high level graphics commands are given to the BPS by application programs, then carried out by the BPS (the high performance graphics processor) which accesses the image memory. If the BPS is given the SID of the requesting task it is able to write to image memory using state set up by the requester, and to interleave writing with access by other tasks without any need for synchronization. This technique requires a synchronous use of the BPS by the requesting task, which is the normal usage pattern in single task software.

The above description shows the possibilities for multitask operation with static screen allocation, the application for which the interface was designed. Dynamic screen allocation is also possible. When the window manager receives a request to move a window it must synchronize with tasks that own affected windows, waiting until they acknowledge the request. Then data structures indicating the window position can be updated and damage reports sent out so that tasks can redraw their windows. Although this procedure involves a round of synchronization not needed in a system like X, the faster redraw, which involves no communication or synchronization, is likely to give superior performance.

At this point it is possible to see that current hardware limits the performance of the software described above. Eight SIDs are certain to be insufficient. To make use of them multiple instantiations of the cross-bar switch and LUTs are needed. Multiple LUTs are, in fact, already available as a single VLSI component from Brooktree. And because the BPS can control its SID, multiple BPSs can also improve system throughput, although bandwidth limitations on the processor bus may prevent effective utilization of a large number of them.

There remains the problem of providing independent pan, scroll and zoom and independent cursors. The former are important components of systems designed to offer effective animation [15]. The latter are certain to be important when coordinated windows are controlled by different tasks, for example, in multiple views of three-dimensional scenes. Such a development requires significant change in the design of the video controller/sequencer, so that it fetches pixel values based on independent origin, zoom and cursor registers, indexed by SID. When this type of hardware innovation is available tasks can write to image memory using window-relative coordinates, with the window manager independently controlling origin registers to change the position of windows on the screen.

The final bottleneck is communication from the graphics system to the host. Current systems make this a very narrow path, with a single interrupt line connected to each interrupt source in the graphics system. The interface is cumbersome. For example, suppose several tasks wish to update their parts of the screen during the next vertical retrace. Communication times are too slow for one task to catch the retrace interrupt and send messages indicating its occurrence to the tasks that want to perform updates. Consequently the task that catches the interrupt must maintain a queue of work orders, executing them without any context switches when the interrupt occurs. Whether or not it obtains a SID along with the work order and uses it when accessing the graphics, serious synchronization problems exist. Thus, a better system of distributing interrupts is needed. The location monitor is the feature equivalent to the SID. It allows interrupts to be distributed individually to multiple recipients, and is an obvious feature of a more functional interface. Unlike interrupts that involve volatile data, which must be delivered to a single recipient since the interrupt acknowledgement must be tied to turning off the interrupt, ones like vertical retrace interrupts can be safely delivered to multiple recipients and turned off by hardware. The provision in the interface of a flexible facility for delivering interrupts to specific tasks using location monitors completes the functions needed for efficient two way communication between individual tasks and the graphics resources they control.

5. Conclusion

In a large granularity system the sender id easily generalizes into an effective mechanism that supports tight interaction between individual tasks and the graphics resources they control. Future development of hardware and software along this direction is likely to pay significant dividends in terms of enhanced graphics functionality in multi-window systems. In fact, the architecture issues discussed in this paper are relevant to any graphics system based on the concept of symmetric processing. For example, the original design of the Titan workstation by Ardent, which was carried over to the Stardent system, uses symmetric multiprocessing where the graphics processor is merely one of the system processors. This architecture leads naturally to coarse-grain parallelism in which the issues discussed in this paper are relevant.

Finally, it is interesting to note that the interface construction discussed in this paper and the further enhancements discussed in this last section are easy to carry out in the RDS 3000, a decade old design. Its flexibility, modularity and openness make it possible to design and substitute components in a way that cannot be done with newer designs offering higher graphics performance at the cost of closed hardware. Systems like the RDS 3000 may be unsuitable for the production graphics that makes up the vast bulk of the market, but they are the life blood of laboratories that conduct research into new techniques for combining computation and graphics.

6. Acknowledgements

This research was partially supported by grants from the Natural Sciences and Engineerring Research Council and from the Information Technology Research Centre, a Centre of Excellence supported by the Province of Ontario. One of the authors (WC) wishes to thank Kevin Schlueter for informative discussions concerning the structure of current window servers, particularly X and NeWS. We would also like to thank Nick England for moral and other support as we have worked at getting extra mileage from our RDS 3000 systems.

7. References

[1] C. Ferguson, *Parallel Algorithms for Computer Graphics*, UCSC–90–08, Computer Research Laboratory Technical Report, University of California, Santa Cruz, 1990.

[2] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs and L. Israel, 'Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories', *Proceedings of SIGGRAPH '89*, published as *Computer Graphics*, 23(3), 111-120, 1989.

[3] K. S. Booth, L. R. Bartram, W. B. Cowan, J. D. Morrison, and P. P. Tanner, 'A system for conducting experiments concerning human factors in interactive graphics', *Graphics Interface '88*, 34–42, 1988.

[4] W. M. Gentleman, S. A. MacKay, D. A. Stewart and M. Wein, *Using the Harmony Operating System: Release 3.0*, ERA-377, NRCC No. 30081, National Research Council of Canada: Ottawa, 1989.

[5] J. C. Beatty, R. J. Beach, K. S. Booth, E. L. Fiume and D. A. Plebon, 'The message is the medium: multiprocess structuring of an interactive paint program', *Ninth Annual Conference on Computer Graphics and Interactive Techniques*, 277-287, 1982.

[6] D. R. Forsey, *Transposing Harmony*, M. Math. Thesis, Department of Computer Science, University of Waterloo, 1985.

[7] K. S. Booth, W. B. Cowan and D. R. Forsey, 'Multitasking support in a graphics workstation', *1st International Conference on Computer Workstations*, 82-89, 1985.

[8] K. S. Booth, D. R. Forsey and A. W. Paeth, 'Hardware assistance for z-buffer visible surface algorithms', *IEEE Computer Graphics and Applications*, 6, 31-39, 1986.

[9] J. V. Kelley, K. S. Booth and M. Wein, 'Design experience with a multiprocessor window system architecture', *Graphics Interface '89*, 62-69, 1989.

[10] At the time of writing System 7 for the Macintosh is rumored to have possibly pre-emptive multitasking capability.

[11] R. W. Scheifler and J. Gettys, 'The X window system', *ACM Transactions on Graphics*, 5, 79–109, 1986.

[12] J. J. Gosling, D. S. H. Rosenthal and M. Arden, *The NeWS Book*, Springer-Verlag: New York, 1989.

[13] P. P. Tanner, K. S. Booth and B. M. Fowler, 'Experience with graphics support for a multiprocessor workstation', *Parallel Processing for Computer Vision and Display International Conference,* Leeds, 1988. Appears in *Parallel Processing for Computer Vision and Display*, edited by P. M. Dew, R. A. Earnshaw and T. R. Heywood, Reading: Addison-Wesley, 1989, pp. 298–307.

[14] D. Rhoden and C. Wilcox, Hardware acceleration for Window systems, *Proceedings of SIGGRAPH '89,* published as *Computer Graphics,* 22, 247–253.

[15] K. S. Booth and S. A. MacKay, 'Techniques for Frame Buffer animation', *Graphics Interface '82,* 213-220, 1982.

[16] *VMEbus Specification Manual,* Micrology pbt: Tempe, 1985.

[17] O. Jones, *Introduction to the X Window System,* Prentice-Hall: New York, 1989.

[18] F. K. T. Sun, W. Cowan and K. S. Booth, 'Understanding visual effects in a windowed environment', *Graphics Interface '90,* 100–107, 1990.

[19] D. Voorhies, D. Kirk and O. Lathrop, 'Virtual Graphics', *Proceedings of SIGGRAPH '88,* published as *Computer Graphics*, 23, 61–67.