Computer Graphics, Volume 22, Number 4, August 1988

Virtual Graphics

Douglas Voorhies, David Kirk, Olin Lathrop

Apollo Computer Inc.
330 Billerica Road
Chelmsford, Mass. 01824

ABSTRACT

Graphics can be implemented as a virtual system resource. This abstraction appears to each application on a multiprocessing workstation as a dedicated rendering and display pipeline. A variety of simple mechanisms support the simultaneous display of different types of images and eliminate the need for low-level device driver software. They permit applications to embed graphics instructions directly in their code. The abstraction allows for cleaner software design, higher performance, and effective concurrent use of the display by several applications.

INTRODUCTION

There is a conflict between the multiple-process orientation of workstations and the single-task orientation of graphics hardware. Timesliced CPUs with virtual memory support many concurrent processes [2,9], and increasingly, those processes demand fast, sophisticated graphics for high interactivity and for complex data visualization. Moreover, windowing systems are raising the users' expectations for concurrency [10].

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1988 ACM-0-89791-275-6/88/008/0247 $00.75

Previously, graphics hardware has not been structured to match this environment. (See Figure 1). Usually it is coupled to the CPU by an ad-hoc protocol. Although current graphics hardware connects to a high-bandwidth internal bus, its interface still presents the old paradigm of a dedicated graphics display serving one application from the far end of a communications line [5]. Despite the graphics hardware holding considerable per-process drawing state information, there is rarely any state management design discipline or effective swapping mechanism present [7]. Consequently, layers of graphics systems software are needed to manage access and context. Additional low-level software complexity arises from the need to throttle the CPU's request rate so as not to exceed the graphics hardware's input capacity. Finally, a single screen-wide display mode such as a color lookup table or pixel format satisfies only one class of applications at a time. Thus, the single-task orientation of graphics hardware has left the meta-rendering problem of updating and displaying multiple images unaddressed.

CPU/graphics interfaces are ad-hoc because they fall between two cultures. To the CPU and OS, graphics is a peripheral, and to graphics, the CPU and memory are a command source. Ad-hoc I/O interfaces are usually "dressed up" by intervening software for elegance and for device independence. While device independence is a requirement in many cases, applications are increasingly seeking the maximum performance from expensive hardware. For them, the intervening software is an obstacle. As we have learned, with more careful integration, this software can be eliminated.

In this paper we develop an approach which makes current user-interface and graphics rendering technology (both hardware and software) more effective on a workstation.

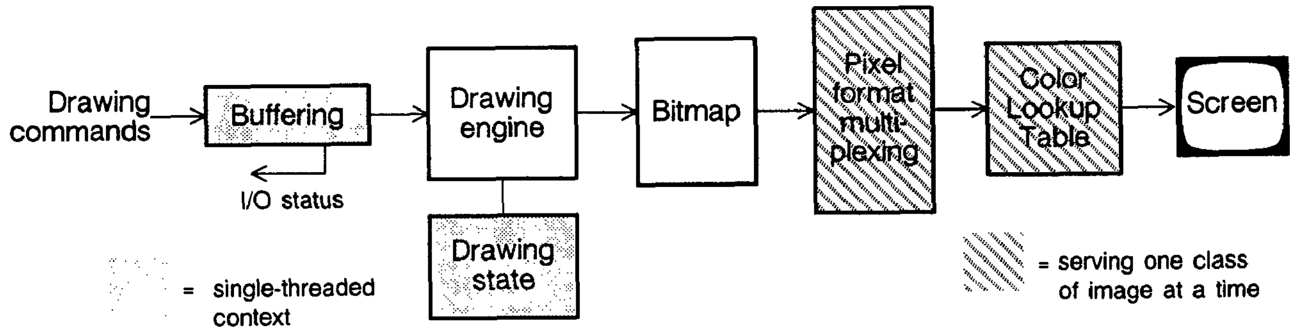

Figure 1

We offer an effective abstraction, "virtual graphics", which cleans up the ad-hoc hardware/software interface and tackles the problems of both time-slicing drawing hardware and simultaneously displaying varied images.

"VIRTUAL GRAPHICS"

By "virtual graphics" we mean every application has the illusion of owning a dedicated rendering co-processor. Each screen window is independently displayed. Applications need not deal with sharing access, switching contexts, modulating command flows, I/O status checking, screen-wide pixel formats, or sharing color lookup tables.

"Virtual" means that implementation details are hidden beneath an abstract interface, allowing higher levels to be simpler [3]. Virtual graphics hides the single-threaded nature of the rendering and display hardware. Because the interface is simplified, multiple threads of graphics rendering can be handled efficiently. Since each applications' windows are updated more smoothly, the benefits extend to the user as well.

Although windowing systems, such as X-windows [14] or Apollo's Display Manager, have already begun to address the screen allocation problem, they do so by making window shape and size variable at display time. This, in turn, places demands on the drawing and display hardware to support window-oriented update and display.

To create the appearance of a dedicated resource to multiple requestors, the hardware must either have sufficient **parallelism** to actually perform all work simultaneously, or be able to rapidly **switch** between tasks. The success of both CPU timesharing and the virtual memory page sharing have proven rapid switching of an expensive resource to be effective [11]. Since CPUs and high-end graphics hardware cost roughly the same, concurrency through rapid switching is the most practical solution.

The key to managing state is to switch low-level contexts rapidly. It is very difficult to emulate fast task switching at a high abstraction level if lower levels cannot exchange their state quickly. Useful techniques include the simple switching of the context (by moving a pointer) or the complete swapping of the context (by a saving and restoring copy).

IMPLEMENTATION

At Apollo we have implemented a high-end graphics system for the DN10000. It offers virtual graphics by combining six simple mechanisms (see Figure 2):

1. The integration of graphics via a co-processor interface
2. The exception-based detection of graphics ownership violations
3. The efficient swapping of drawing context
4. The exception-based management of command flow rate
5. The clipping of drawing to window boundaries
6. The switching of pixel formats and lookup tables per pixel

These mechanisms are not very powerful separately, but together they remove the barriers between the applications and the graphics hardware. Applications become unaware

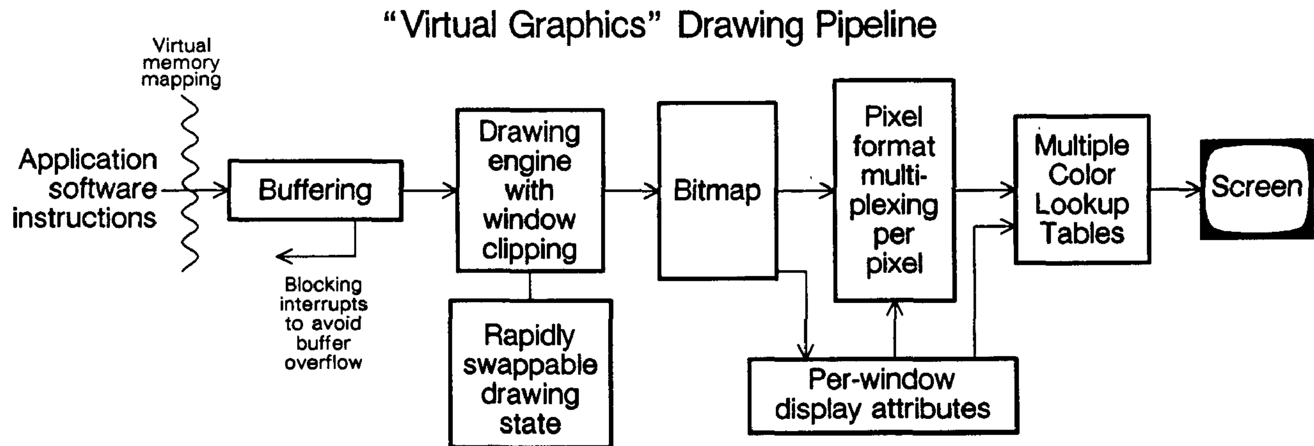

Figure 2

that they are sharing the hardware device, or its drawing context, or the screen.

Co-processor interface

The graphics hardware is mapped into the application's memory space. Memory-mapping allows control to be cleanly presented in higher-level languages. It permits the application to use *Store* instructions to set all drawing parameters and initiate all drawing operations. Drawing is simply done by initializing all relevant parameters and then requesting an operation.

Graphics is a co-processor since the *Store* subsumes both an 8-bit graphics opcode and 32-bit data operand in its address and data. The parameters and operations are selected by 8 low address bits, and the data is the 32-bit operand. By packing all the commands into one page, the cost of setting up the mapping is minimized.

Exception-based detection of graphics ownership violations

State ownership and graphics access are built upon virtual memory access management mechanisms. Both the graphics drawing commands and the bitmap itself are mapped into the physical memory space. By allowing only one requesting process at a time to map the *graphics command page*, access to the graphics becomes single-threaded. Even in a shared-memory and multiple CPU system, access remains exclusive. Whenever the *graphics command page* is mapped, that process's context is presumed to be active in the drawing hardware.

The fault handler grants access to processes following a "fairness" policy. A graphics access fault differs from page faults in that the requesting process does not necessarily get access as soon as possible. The access policy software permits only one requesting process to "own" the graphics; the others wait their turn. In this respect it resembles the "fair" allocation of CPU timeslices to processes.

The bitmap may also be mapped into the process's virtual address space. In order to insure a consistent view of the bitmap by both direct reads and writes as well as by the drawing hardware, an interlock is needed. Here again, virtual memory mapping can detect both direct bitmap access as well as command page access. Only one or the other is mapped at a time. Execution of a direct bitmap read/write is delayed until all drawing has completed, and a request for drawing is assumed to signal the end of direct bitmap access.

Fast drawing context swapping in hardware

Drawing context can be swapped quickly by a VLSI state machine within the graphics hardware. In our implementation, the state is large because this rendering hardware is highly parallel and strives for unusually high image quality. The per-process drawing state is kept in 144 hardware registers (576 bytes) during use. When swapping, the live state (524 bytes) is exchanged with a copy in a local

graphics context RAM. The remaining per-process state (52 bytes) stays in that RAM while in use, and so may be switched by updating a reference pointer.

This local RAM has space for six graphics contexts, including the active one; these copies function as a cache over additional ones in main memory. The state machine can perform a swap to and from the local RAM in 16 μsec., which is less time than a CPU process swap takes. Those occasional swaps involving a context not in the local RAM are handled by CPU copying to and from the main memory contexts, and take under 200 μsec..

Since all drawing commands atomically update the drawing engine state, swaps are legal between any commands. Drawing commands are never interrupted or aborted, since some (such as RASTER-OP [12]) may not be idempotent or may use temporary storage in mid-execution. Swaps are initiated by issuing SAVE and RESTORE commands between the drawing commands of one process and the next. The fault handler inserts these commands between the commands of the old and new owning processes.

Exception-based command flow control

Flow control can best be handled by an exception mechanism. Usually, graphics hardware handles requests faster than a CPU can generate them, but whenever this is untrue, the CPU must wait until the graphics hardware has caught up. In our implementation, a command FIFO with 512 entries buffers between the bursty request and execution traffic. It permits a requesting CPU and the drawing hardware to proceed asynchronously. When a FIFO is nearly full, all further requests trigger interrupts [1], which force the requesting process to wait until the FIFO signals it has reached half full. (An interrupt is issued upon **each** request to insure the nearly-full condition is communicated successfully, since processes may migrate between CPUs at any time.) This throttling of the requesting process is separate from the access mechanism, although it may be factored into the access policy decisions.

Since the CPU is increasingly the bottleneck in graphics-based applications, it is advantageous to slow it down only in the exceptional case of being limited by the graphics hardware. CPU busy-flag tests, "done" interrupts, and other hardware/software locks are eliminated by handling flow control transparently, streamlining and simplifying the co-processor interface.

Window-boundary clipping during drawing

Drawing into a window usually requires 2-D clipping to the window boundaries. This can be cumbersome, especially if the window is partially occluded [6,16]. The burden of screen clipping makes it more difficult for applications to fully participate in a shared screen, and slows down their rendering if they do. Since we added rectangle clip logic in hardware, drawing can simply ignore window boundaries, or only trivially reject primitives prior to drawing.

In our implementation, every potential pixel write is compared with up to seven rectangles using 28 parallel X or Y comparators. These 28 rectangle coordinates are part of the per-process state swapped when changing drawing contexts. Two clipping tactics are supported. Simple windows are handled by one "clip outside" rectangle occluded by up to six "clip inside" rectangles. More complex windows are drawn in multiple passes, with each pass clipped to the union of seven "draw inside" rectangles. In both modes, the clip decision is made in parallel with address synthesis, and imposes no speed penalty.

Pixel format switching per pixel

Even if multiple windows can be drawn rapidly, having only a single pixel format or single color lookup table for the whole screen thwarts their concurrent display. For example, a page-layout system may wish to mix full-color photographs with false-color text. Another application may wish to alter the color lookup table to highlight part of its image. And a third may want to double-buffer a false-color CAD image. If they are to effectively share the screen, the video display mode must change on-the-fly at each window boundary. Switching the display hardware mode at pixel rates satisfies the requirement of most window managers to arbitrary window position, size, and complexity [15].

By appending each bitmap pixel with a 4-bit tag, the video logic obtains control information synchronously with the raw bitmap pixel data. Using this tag as an index into a 16-entry *display attributes table* expands the tag into its current display mode [17]. See Figure 3. This table

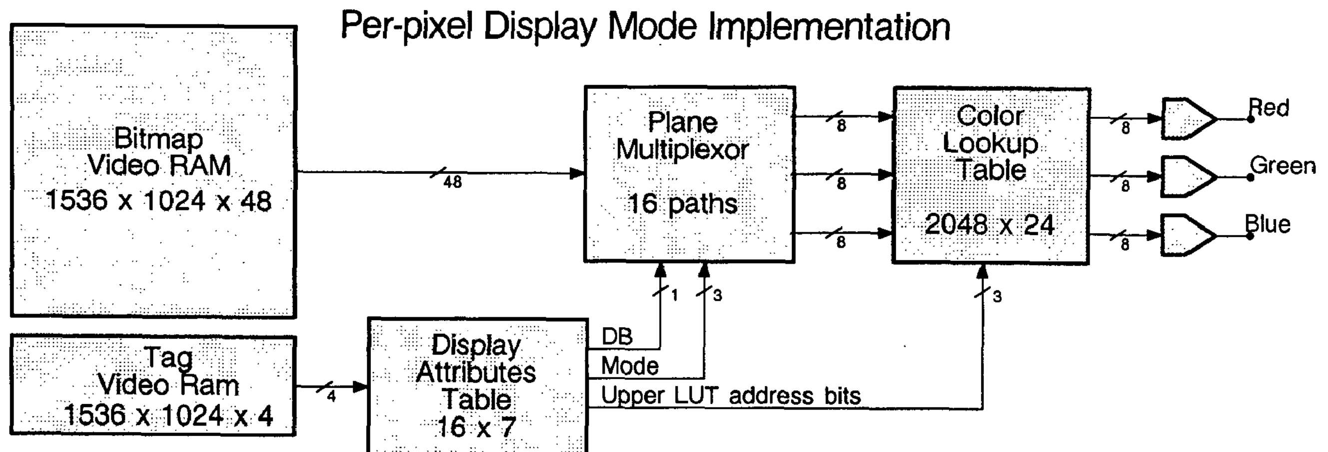

Figure 3

lookup is analogous to the color lookup table for pixel data. Per-pixel display attributes in our implementation include:

- Color Lookup Table (LUT) selection (1 of 8)
- Plane multiplexing mode:
  - 8-bit false color
  - 10-bit false color
  - 12-bit real color
  - 24-bit real color
  - Mixed false and real color
  - Substitute a cursor color
- Overlays on/off
- Double buffer select

Typically, all pixels in a particular window are displayed in the same mode. The actual number of windows is not limited by the 16 *display attributes table* size, since several windows will often share the same attributes (e.g. text windows). Further, the limitation of 8 x 256 LUT entries is ameliorated by allocating color blocks smaller than 256 on demand, and by sharing LUT blocks (e.g. a shared real color Gamma function).

BENEFITS

Together, these six mechanisms and the regular graphics hardware appear to each application as a fast dedicated rendering and display pipeline. Applications are unaware of sharing the screen area, drawing context timeslicing, screen display mode cooperation, or flow control. Consequently, graphics is presented to applications as a clean and useful abstraction. The abstraction allows instructions embedded within an application to request drawing directly. Neither active coordination with other applications nor intervening layers of graphics system software are required. Moreover, this memory-mapped interface is easily used by high level languages.

The most challenging part of virtual graphics is access management. We chose to base our implementation on virtual memory management, thus reducing it to an already-solved problem.

Performance is improved by the reduction in both software layering and context management overhead. Indeed, there is no state-management overhead when there is only one active drawing process, and minimal overhead when there are six or less active processes.

A more subtle performance gain is achieved through the asynchrony of the CPU and graphics executions allowed by the FIFO. Inserting swap requests in the drawing instruction sequence inherently synchronizes drawing context and operations. When a process gains access to the FIFO, it does not need to wait for its state to be swapped in. Drawing instructions can be issued immediately, even though they presume post-swap context, because they are preceeded in the FIFO by the swap request. Since the FIFO preserves the sequence, the swap will emerge from the FIFO and occur before these instructions emerge. This is possible because the graphics context is swapped synchronous with drawing and asynchronous from the CPU by a post-FIFO mechanism.

HARDER PROBLEMS

Our six mechanisms do not address several more difficult graphics resource management limitations. Any sharing abstraction breaks down when its underlying resources have insufficient capacity or are too slow to multiplex their context. For example, thrashing occurs in a virtual memory system if there is insufficient physical memory. Also, software locks are needed for single-threaded peripherals such as tape drives. Similarly, virtual graphics, while effective for common rendering, cannot hide finite resources and cannot transparently share what it cannot swap.

Screen space is the most visible finite resource. There is an obvious extension of virtual memory to virtual windows [8], which allows drawing to proceed oblivious to window occlusion, and allows a potentially larger bitmap. However, crossing discontiguous page boundaries is awkward, and gathering the scattered windows at video rates is either very expensive or requires a massive copy. Such copying cuts into drawing bandwidth, and may in turn cause jerky screen update. Moreover, a fixed-size video-RAM array offers much lower cost; and, because it can be tightly interconnected to a dedicated drawing engine, it can achieve higher potential performance.

Off-screen images, such as image preparation areas, down-loaded fonts, texture maps, or tile patterns, represent a huge context which defies rapid swapping. Many bitmaps have some off-screen area where the drawing hardware can work at full speed, but it is expensive and must be allocated very carefully. Moreover, high-level software locks are often needed to single-thread access to large blocks of the limited space. Virtual graphics can do little to manage these large images.

Similarly, color and display attribute lookup tables sizes are finite, and the number of entries needed for all the windows may ocassionally exceed what the hardware can support. In time, however, technology will push the limit beyond common user needs. Other features, such as per-window pan and zoom, remain particularly difficult to implement without copying pixels. High display rates and the horizontal shifting efficiency of video RAMs [13] limit flexibility in video back-end hardware. Fortunately, with improving redraw speeds, video panning and pixel-replicating zoom are becoming obsolete.

Finally, with increasingly asynchronous CPU and drawing execution and with more automatic state management, software has less control. These features optimize output-only rendering while penalizing operations which require the CPU to know the state of the bitmap or drawing operation. For example, if an application wishes to read the bitmap, or be signaled at the end of a drawing operation, or ascertain the current interpolated colors, then the long pipeline must be drained and synchronization reestablished. Since most of such "upstream" communication is to support human input, the frequency is quite low. Response time, however, must remain adequate, and any forced re-synchronization conflicts with concurrent update.

CONCLUSIONS

The abstraction of virtual graphics elevates graphics to a first-class system resource. Its implementation pushes the multiplexing of control and context from the device driver level down into the hardware. Hardware accelerates context swaps, FIFO full detection, and window clipping. By moving these mechanisms into the hardware, we eliminate a layer of software and achieve a significant reduction in multiplexing overhead. Faster multiplexing can quantitatively improve performance as well as qualitatively smooth the drawing of multiple windows.

The software sees a dedicated graphics co-processor. Because the ad-hoc jumble of low-level driver software has been replaced, applications can simply access the hardware by embedding graphics instructions directly in their code. Additionally, several applications each retain a clean view despite simultaneously running on multiple processors.

The user sees a screen supporting multiple images, each displayed properly and updated smoothly. Thus virtual graphics enhances the concurrency and effectiveness of modern workstations by addressing the meta-rendering problem of multiple image update.

ACKNOWLEDGEMENTS

We wish to thank Inwhan Choi, Robert Feldstein, Thomas Fincher, Jeffery Kurtze, Eliot Polk, and Robert Taylor for their insightful contributions and refinements during the implemention of this architecture. We also appreciate Dr. Terence Lindgren for his helpful observations in the presentation of these ideas.

REFERENCES

[1] Asal, M., Short, G., Preston, T., Simpson, R., Roskell, D., and Guttag, K., "The Texas Instruments 34010 Graphics System Processor", *IEEE Computer Graphics and Applications*, Vol. 6, No. 10, October 1986, pp. 24–39

[2] Barton, R. S., "A New Approach to Functional Design of a Digital Computer", *Proc. AFIPS Western Joint Comp. Conf.*, 1961, Vol. 19, pp. 393–396.

[3] Buzen, J. P. and Gagliardi, U. O., "The Evolution of Virtual Machine Architecture", *Proc. of AFIPS*, NCC 1973

[4] Cohen, E. S., Smith, E. T., and Iverson, L. A., "Constraint-based Tiled Windows", *IEEE Computer Graphics and Applications*, Vol. 6, No. 5, May 1986, pp. 35–45.

[5] England, N., "A Graphics System Architecture for Interactive Application-Specific Display Functions", *IEEE Computer Graphics and Applications*, Vol. 6, No. 1, January 1986, pp. 60–70.

[6] Foley, J. and van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass., 1982

[7] Guttag, K., Van Aken, J., and Asal, M., "Requirements for a VLSI Graphics Processor", *IEEE Computer Graphics and Applications*, Vol. 6, No. 1, January 1986, pp. 32–47.

[8] Ilgen, S. and Scherson, I. D., "Real Time Virtual Window Management for Bit Mapped Raster Graphics", *Proc. 5th International Conf. on Computer Graphics in Japan*, Springer-Verlag, Tokyo, 1987, pp. 145–158.

[9] Kilburn, T., Edwards, D. B. G., Lanigan, M. J., and Sumner, F. H., "One-level Storage System", *IRE Trans. on Electronic Computers*, Vol EC–11, No. 2, pp. 223–235.

[10] Lantz, K. A., Tanner, P. P., Binding, C., Huang, K., and Dwelly, A., "Reference Models, Window Systems, and Concurrency", *Computer Graphics*, Vol. 21, No. 2, April 1987, pp. 87–97.

[11] Meyer, R. A. and Seawright, L. H., "A Virtual Machine Timesharing System", *IBM Sys. Journal*, Vol. 9, No. 3, 1970

[12] Newman, W. M. and Sproull, R. F., *Principles of Interactive Computer Graphics*, 2nd ed., McGraw–Hill, New York, 1979, pp. 262–265.

[13] Pinkham, R., Novak, M., and Guttag, K., "Video RAM Excels at Fast Graphics", *Electronic Design*, Vol. 31, No. 17, Aug. 18, 1983, pp. 161–182

[14] Scheifler, R. W. and Gettys, J., "The X-Window System", *ACM Trans. on Graphics*, Vol. 5, No. 2, April 1986, pp. 79–109

[15] Shires, G., "A New VLSI Graphics Coprocessor — The Intel 82786", *IEEE Computer Graphics and Applications*, Vol. 6, No. 10, October 1986, pp. 49–55.

[16] Sproull, R. F. and Sutherland, I. E., "A Clipping Divider", *Fall Joint Computer Conf. 1968*, Thompson Books, Wash. D.C., pp. 765–775.