Faster Hardware, Smarter Code
Optimizing X for Graphics Accelerators

Todd Newman

Digital Equipment Corporation
Workstation Systems Engineering
Palo Alto, CA 94301

ABSTRACT

Graphics accelerators are necessary to achieve acceptable performance for certain graphics operations like 3D hidden surface removal or image convolution. They can also boost simple 2D operations for key applications. But they add substantially to the cost of a workstation. This paper discusses techniques for obtaining the maximum performance from a graphics accelerator, some general principles to use in planning optimization, and some specific tricks for squeezing out the last drop of speed.

Introduction

Everyone wants their workstation to run infinitely fast and cost nothing. Since we live in a less than perfect world, systems designers must balance the performance of a system, which can be improved by adding special purpose hardware, against the price of the system, which can be improved by removing hardware. For applications that are not graphics intensive, like CASE, the optimum solution seems to be to just provide a block of memory which the display system will show as a rectangular array of pixels. For certain graphics operations like 3D hidden surface removal or image convolution, graphics accelerators are necessary to achieve acceptable performance. They also may boost performance on simple 2D operations for key applications. Offloading rendering work to a graphics accelerator leaves more CPU cycles for all applications. So graphics accelerators can be a big win. On the other hand, they can add substantially to the cost of a workstation. If you decide to add one, you want to be sure to get the full benefit from it. This paper discusses techniques for obtaining the maximum performance from a graphics accelerator. It discusses some general principles to use in planning optimization and some specific tricks for squeezing out the last drop of speed. Digital workstations are used as examples, but the guidelines suggested should work on any workstation.[1]

Types of Graphics Systems

The simplest graphics system is built by adding memory that gets scanned out to a display. The block of memory is called the frame buffer. The CPU sees it as just some I/O space memory. The video subsystem sees it as a rectangular array of pixels. For example, our CX board has 884,736 bytes of memory on it that the CPU sees at some memory location, call it FrameBufferBase, determined at run time. The video subsystem sees it as 1024x864 8 bit deep pixels. When the CPU writes a byte at the address FrameBufferBase + 1024 + 10, it changes the tenth pixel in the second row of the screen. Systems with frame buffers more than one bit deep are called Color Frame Buffers (CFBs).

Color frame buffers are great — up to a point. Because all the functionality is in software, they're flexible, easy to optimize, and the code is quite portable. But they're very CPU intensive. Furthermore, their rendering rates degrade noticeably on large area primitives. (See Table 1) This is simply because they access memory in roughly constant time, so performance degradation asymptotically approaches being

---

[1] And, of course, you don't need to apply them to Digital's workstations, because we've already done it for you.

linearly with area.[2] Performance is also a function of the number of bits in a pixel. Eight bit pixels packed four to a word are roughly four times faster than 24 bit pixels packed one to a word.

Performance degradation with increase in area	
rate per second	operation
180000	1x1 rectangle
81800	10x10 rectangle
2130	100x100 rectangle
89	500x500 rectangle

Table 1.

If we want a graphics system that runs any faster than that, we build some kind of hardware graphics accelerator. For the purposes of this paper, that's all I mean by a graphics accelerator -- that it makes something run faster than a CFB would. Most graphics accelerators are interposed between the CPU and the framebuffer. Usually they have some privileged access to the framebuffer that the CPU does not. For example, on Digital's PixelStamp graphics system each stamp chip accesses five pixels in parallel. And one could gang together up to four stamp chips in a system allowing access to 20 pixels at once. This parallelism accelerates many primitives. (See Table 2.) The price for this performance gain is that the CPU cannot directly access the framebuffer, but must make special requests through the PixelStamp. None of the MIT-provided CFB code could be used. We had to rewrite everything, but many primitives ran much faster than the CFB code ever could.

Performance acceleration with graphics engines		
CFB	PixelStamp	Operation
180000	139000	1x1 rectangle
81800	139000	10x10 rectangle
2130	13500	100x100 rectangle
89	585	500x500 rectangle
1020	13500	100x100 stippled rectangle
84	585	500x500 stippled rectangle
119000	190000	10-pixel line segment
156000	319000	10-pixel line
27600	100000	100-pixel line

Table 2. (measured with x11perf on DS5000/200 CX "CFB", PXGTurbo "PixelStamp")

All the graphics accelerators I've seen come in one of two variants. One sort has a set of registers that you load and then a final one that kicks off the operation. The other sort interprets a command packet which resides in memory somewhere. I believe that most of the points I have to make apply equally well to either sort of accelerator.

In the interest of space, I am ignoring geometry acceleration in the rest of this paper. 3D systems spend a lot of time in floating point math code and special purpose hardware can speed this up tremendously. Similarly, geometry acceleration helps for imaging primitives like rotation and convolution. Aside from the performance gain of having more floating point hardware to throw at a problem, many graphics accelerators tightly couple the geometry accelerator and the drawing accelerator (or rendering engine, as it's also called). Not having to move data over the system bus improves performance and frees the bus for moving other data. This leaves more cycles for the user's application to generate the next set of work for the graphics accelerator.

---

[2] Page mode access to RAM provides noticeable gain over strict random access times. Nevertheless, the degradation is noticeable with large areas.

Optimization Principles

Optimize What People Notice

No one cares if your server has low overhead per request or even how many polylines you can render in a second. They care if their application runs fast. What you're trying to optimize is the performance of their application. You only need to speed up the parts of the server that they use a lot and care about. Users care about background repaint, text, copy area, and zero width lines. I've found that users of CAD software care about stippled rectangles. And everyone will complain if the server can't tile the root window quickly. But don't bother to speed up your wide dashed stippled arcs. So set your priorities well and optimize what people notice. (However, from personal experience I can tell you that some one who receives a preliminary release of your server will always care deeply about the thing that you just hadn't gotten around to yet.)

Optimize the System As a Whole

Another useful principle is to optimize the performance of the whole system, both in the large scale and in the small. In the large scale, you want to be sure to leave CPU cycles for the client application to use. Even benchmark programs will suffer if they get no CPU cycles. Usually, an even mix of cycles between server and client will run faster than bursts of one and then the other. Designing your server to balance cycles between the server and the client is an example of large scale holistic optimization.[3] On the small scale, if the graphics hardware hogs too many of the system resources, especially the data buses, it may starve out other components leading to slower overall performance. Slowing down the graphics engine may speed up the system. So be careful not to fall into the trap of greedy local optimizations degrading global performance.

Know When to Stop Optimizing

Know the limitations of your hardware and remember that you can't optimize to run faster than your hardware can. You can waste a great deal of time worrying about it, but it just doesn't help. If you are working with new hardware, and you find yourself up against a performance wall, go back and make sure that you've accurately estimated the potential performance.

Remember also that the 90-10 rule applies here. You can get 90 percent of the performance in 10 percent of the time. If you want that last 10 percent, it will take nine times as long to get it. If you have a short time to market, you may have to trade off that last little bit against some other optimization.

Finally, remember that your work has a limited lifespan anyway. The time you spend optimizing may be eating out of that total time. If your product is only state of the art for 18 months, you don't want to slip your schedule by 6 months to get those last optimizations finely tuned. It would be much better to put out the product at 90 percent of its theoretical performance rate and move onto the next graphics accelerator. You need to strike a balance between the desire for perfection and the desire to satisfy customers.

Application of Optimization Principles

Before we decided on most the optimizations discussed below, we ran a benchmark suite (x11perf) and several representative applications. These gave us a good idea of where we had performance problems. We also considered the theoretical maximum performance to see where we fell short and by how much. This lead us to target areas of optimization. Some of these were fairly generic goals like speeding up the transport mechanism; some were quite specific like tiling the root window faster. Having actual, measured, performance data gave us both target areas to optimize and a metric to measure our progress.

---

[3] This may be a new nadir in California high tech babble.

Optimize Data References

Perhaps the most important idea in this paper is that *eliminating or simplifying memory references is the key to better performance.* Computers have always been slower at referencing memory than at referencing registers and over time this factor has become more significant. On a VAX it took referencing memory took roughly seven times longer than referencing a register. On the DECstation, that factor has gone up to 13. The difference is that while the CPU has gotten much faster, RAM chips have gotten only a little faster. Thus, accessing uncached memory is relatively slower. And every time you have to random access memory instead of hitting the cache you pay that cost again. In the following sections, I'll give some examples of how to minimize this factor.

Optimize Data References by Minimizing Copies in the Transport Mechanism

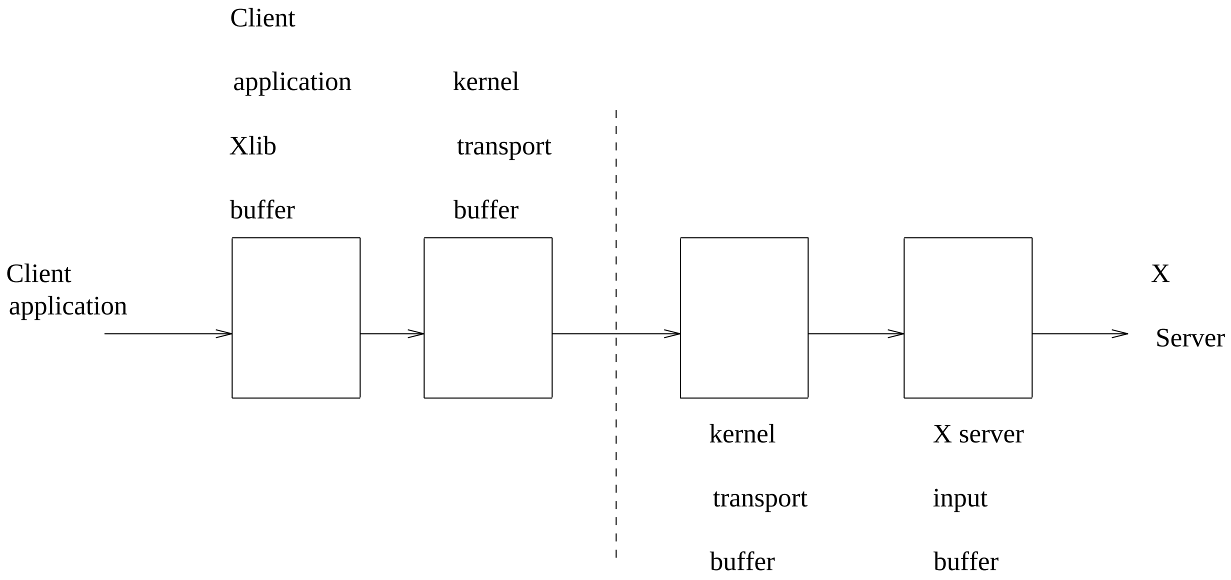

Figure 1. TCP/IP Transport Mechanism

When we decided to minimize memory references, the first thing we went to work on was the X transport mechanism. Let's consider the path from the client to the server using TCP/IP. (See Figure 1.) First the client puts the data into an Xlib buffer. Then that buffer is copied into a kernel buffer on the client machine. That buffer is pushed over the wire into a kernel buffer on the server machine. The buffer is then copied into the server's input buffer. The data is then massaged a bit and fed to the rendering hardware. The protocol is similarly complex for other socket protocols. This is tremendously inefficient. If the client and the server are running on the same machine, we could eliminate two copies and the most of the socket-based transport mechanism if we could find a way to make the Xlib buffer and the server's input buffer be the same physical memory. (See Figure 2.) Fortunately, Ultrix and System V provide a shared memory mechanism that makes this easy. The basic idea is to have an X transport buffer that is a shared memory segment. Both the client application and the X server can see the same memory as part of their address space. The client writes into that memory and the server reads from it.

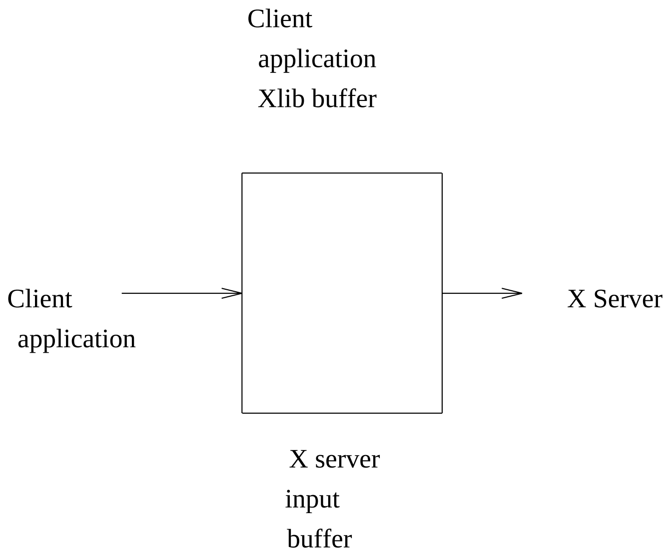

Figure 2. Shared Memory Transport Mechanism

We've seen impressive speedups from this. (See Table 3.) The gain is highest when there's lots of data per packet, as in imaging. In fact, small packets may slow down, because the overhead of looking for both shared memory transport packets and regular packets is higher. So this technique should not be used indiscriminately. But for image intensive 2D applications, this can be a big win. (It's an even bigger win for the PEX 3D extension. PEX applications tend to have much more data per packet, so they win more often.)

Performance Gain with Shared Memory Transport			
percentage gained	socket domain	SMT	operation
165.2	32	85	PutImage 100x100 square
92.7	50400	97100	10x10 4x4 tiled rectangle
65.8	234000	388000	Dot
50.5	105000	158000	1x1 stippled rectangle

Table 3. (measured through x11perf with sockets and shared memory transport)

There's a good bit more complexity here than I've indicated. The client needs some way to tell the server when a packet is ready to be read. And you'd like your server to be still able to talk to clients using TCP and local sockets. There's a complex interaction between the shared memory transport, TCP/IP, and local sockets. Still, it turns out to be easy to hide all the details inside Xlib and not expose any of the complexity to the client application.

Optimize Data References by Improving Locality of Reference

Locality of reference is critically important on a RISC system. Typical systems pay 1 cycle for a reference to data in a register, 2 cycles for cached memory, and 10 for random memory references. In a system like that, it's much better to move data into registers, process it, and never revisit it. Since typical graphics data sets are much larger than typical RISC system data caches, two passes over the data will guarantee that that all memory references are to uncached memory. If the data must be manipulated before being passed to the graphics subsystem, say for format conversion, it's much better to do it on-the-fly than to make two passes over the data.

One of the worst problems for this is clip-checking lines. Many graphics systems do not support the 16 bit address space that X uses, but only enough address space for their on-screen and off-screen memory, 10 to 12 bits. This means that every polyline, line segment, and polygon vertex must be tested to see if it's within the render engine's address space before the packet can be passed on. If not, the primitive must be rendered using the MI code provided by MIT. That will perform all the clipping and all the rendering in software — at an additional tremendous performance hit. However you choose to do it, you must make a pass over all the data. We can't afford to make a preliminary pass over the data, so we must check as we hand the data to the rendering engine. Even so, we measured a 15 percent size increase in the line code when we added clip checking. Since most vertices aren't outside the rendering engine's address space, this is an unfortunate price to pay. (The performance hit for lines that must be rendered with MI code can be a factor of several hundred.) If you have any say in the matter, try very hard to make your render engine's address space at least 16 bits in each dimension.

You also don't want to hop around too much in memory, as that also makes cache hits unlikely. The best example I know of this was when we designed the packet format for the VS3520 triangle strip primitive. Each triangle in the strip is defined by the last two vertices of the preceding triangle and one new vertex. We were toying with putting all the vertices first, followed by the vertex colors, followed by the vertex normals. As the vertex normals were always optional, and vertex colors were optional unless vertex normals were supplied, this seemed an ideal general purpose format. We could provide a word with flags saying which of the optional data were available, then always use the same code to send the data. But someone noticed that if all the optional data were supplied, the renderer would have to keep hopping back and forth to process each triangle. The cache would be useless, and our performance was shot. We switched to listing all the per triangle information in one place.

The primary key to optimizing X server performance is to make as few memory references as you possibly can. Those references that you must make should be as well localized as possible. The fastest thing is not to touch memory. The second fastest thing is to touch cached memory. The slowest thing is to touch uncached memory more than once.

Keep the Rendering Engine Busy

A secondary factor in improving the performance of graphics accelerators is the utilization rate of the rendering engine. The fastest rendering engine in the world can't help you if it's idle. Our first pass at the 2D accelerator for the DS5000/200 ran at only one third the rated speed. After some analysis, we realized that we only had data packets for it one third of the time. It executed them so fast that it was sitting idle two thirds of the time. There's two parts to this, one is to keep data flowing into the rendering engine, and the other is to minimize the overhead in the processing of the data. Once we reorganized things to keep the rendering engine constantly busy, our performance went up to 85 percent of the rated speed. (The rest of the performance we had to obtain through other optimizations.)

Keep the Rendering Engine Busy by Properly Choosing Between Polling and Interrupts

As stated earlier, we changed the interface to our rendering engine and trebled our performance. The basic idea was to make sure that we minimized the time between the rendering engine being ready and our feeding it data. To make things clearer, I'll have to start with a little background information. The PixelStamp system has a register on it that sets a bit when the engine is ready to receive more data. Systems programmers can decide whether or not that bit should generate an interrupt when it changes.

Our initial performance estimate was that we couldn't use the interrupt method, because it would be too slow to have the PixelStamp generate an interrupt, have the operating system pass that interrupt on to the server process and have the server feed new data to the PixelStamp. It just took too many cycles to context switch between the operating system and the server. We figured we could only sustain 1/2 of the PixelStamp's rated speed on the PolyLine primitive. So instead, we would have the server go into a tight polling loop on that register and feed it the next available packet as soon as it was ready. We'd eliminate the entire trip through the operating system. Unfortunately, as I mentioned, when we measured that, we were getting only one third of our rated performance.

The problem was that we hadn't factored in how long it took before the data was ready to be handed to the PixelStamp. There was a long path from the transport layer, through the clip-checking and packet format conversion. Since the server had been in a tight loop polling on the status register, all the data from the client was backlogged waiting for us to process it. We had to wait until the previous packets had completely executed before we could move on to the next set. Furthermore, this was ignoring that while all this was going on the client was getting no CPU cycles to execute in at all.

Our solution was to return to the interrupt scheme, but to eliminate the context switch that had been eating up all the time. Instead, the X server and the operating system shared knowledge about a packet buffer. The server knew how to add packets to the buffer, and not to add packets when the buffer was full. The operating system knew how to feed packets from the buffer to the PixelStamp, and how to tell the server when the space from the packet could be reclaimed. Shortening the software path we had to traverse to get data for the rendering engine was sufficient to recover the lost performance.

Ring buffers turn out to be the ideal data structure for this. First, they can buffer differences between the bursty rate at which the server produces packets, and the rate at which the PixelStamp can consume packets. Secondly, they can do this without using as much space as a variable length queue. By tweaking the queue length, you can optimize so that the server rarely sees the ring buffer full, but there isn't a lot of wasted space in the ring buffer. Thirdly, the server doesn't have to find out as each packet is finished and manages the memory for it. All it has to worry about is whether there's enough spaces for the next packet it wants to add. Decoupling the packet producer and the packet consumer lets them both run at their optimum speed, and doesn't waste much time synchronizing them.

Keep the Rendering Engine Busy by "Glomming" Small Packets Together

Packet-based systems tend to pay some number of cycles per packet in overhead. They make up for this by allowing you to put more than one primitive (of the same type) in the packet. The problem is that X requests don't always come bundled up. Xlib tries to do this for some primitives like PolyLines and PolySegments. It doesn't try for polygons. And it doesn't (and can't) bundle lines when the GC's foreground color changes in between. If your hardware allows you to specify different colors for each primitive in a packet, as the PixelStamp does, it may be worth trying to "glom" these separate primitives into one larger packet. In our server we did this by decided that if the current packet was of the same type as the previous, and the previous hadn't been processed yet, and a few other conditions were met, then it could merge the packets by changing the packet length and appending the primitive. (This violated the locality of reference rule mentioned earlier, but was a performance gain anyway. Rules were made to be broken.)

Tricks like this can make sure that you keep your rendering engine busy and that it's not wasting its time in overhead. Glomming PolySegments raised us from 8000 color-per-primitive PolySegments per second to 11,000. This is one of the more visible and satisfying ways to speed up your server.

Watch the Off-Screen Memory

Many graphics accelerators can only render in special video memory chips (called VRAMs). In these systems, there's some VRAMs that always map to the displayed screen, and some that are not displayed. We call that non-displayed memory off-screen memory. The configuration of the off-screen memory varies across systems. On the VS3520, the displayed screen was 1280x1024, but the VRAM was 2048x2048. So there was a 768x1024 strip and a 2048x1024 strip of non-displayed memory. On the PixelStamp boards, the screen memory is 1280x1024 and resides in one address range. There are one to three more buffers of off-screen memory all of them 1280x1024, each with its own address range. But in both these systems, and in most others that I've heard of, off-screen memory is a finite resource. And, like all finite resources, you eventually run out of it. X makes no restrictions on the number of pixmaps you can create. Sooner or later, some client or set of clients create more pixmaps than can fit in off-screen memory. At that time you have to start moving pixmaps out of off-screen memory into main memory (and probably onto disk eventually) in order to make room for pixmaps that are being accessed now. Moving pixmaps always takes time, and the optimization trick is to minimize the movement.

An obvious solution to the off-screen memory problem is not to have any, by allowing the renderer to draw into virtual memory. This solution makes life easier for the programmer, but is very expensive. First, it is expensive because it means that the video board must understand the virtual memory architecture and cooperate with the MMU in managing it. Second, it is expensive because it means that either there must be the same high speed path from the video chip to main memory that runs to the VRAMs, or that the chip must be able to talk to both VRAM and regular RAM chips. Some high-end workstations have adopted this technique, but it probably won't happen in the low- to mid- price range workstations for a long time, if ever.

In the meanwhile, we must manage off-screen memory ourselves. Writing an off-screen memory manager for X is tedious, but not too difficult. If your off-screen memory manager has to support multi-buffering or PEX the complexity rises astronomically, as does the likelihood of error. Once again, the problem here is to make a finite amount of VRAM serve the unlimited demand for pixmap memory from X and other extensions. We want to minimize shuffling data to accommodate this. There are two kinds of data shuffling that we must consider: one is moving pixmaps between off-screen memory and main memory; the other is moving pixmaps around within off-screen memory. Rarely are these equally efficient. In systems I've encountered, movement within off-screen memory is from 1.5 to 10 times faster than movement from off-screen memory to main memory. But repacking off-screen memory only occasionally can free up enough memory, while copying out to main memory can always work.

Hardware designers can choose to treat off-screen memory either as a linear array of memory or as a 2D array of memory. (The VAXstation 3520 could treat it either way, which was extremely convenient.) If you have any input in the matter, make sure the system can treat off-screen memory as a linear array. The computer industry has had more than 40 years working on algorithms for managing linear memory. There are cookbook solutions to most of the problems you'd encounter.

Efficient use of 2D memory turns into a bin-packing problem. These are known to be NP-complex. If you have any further constraints placed on your memory allocation (for example, this back buffer must be placed at this location in this buffer to align with a window in some other buffer), the problem is harder still. All you can do is use heuristics. This is a mixed blessing. Since there is no perfect answer, feel safe to give it your best shot.

In order to explain what we did on the PixelStamp system, I have to explain the problem a little more. We were implementing X with the PEX 3D extension and a double-buffering extension.[4] Windows are always in buffer 0. Double buffers have to be in a different buffer but aligned, that is, they have to start at the same x,y location as the window. PEX has a Z buffer which is used for hidden line and surface removal. Z buffers have to be in buffer 2 and aligned with the window. (On the low-end system buffers 0 and 1 are 8 bits deep and buffer 2, if present, is always 24 bits deep. Z buffers need that 24 bits, so they must be in buffer 2.) Pixmaps can be in any buffer, but unlike windows, they may not be clipped by any other drawable.

At first we considered placing any non-window drawable where ever it could go. Since back and Z buffers were tightly constrained, pixmaps could shoved in any spare space. But consider what happens when a Z buffered window moves. The system would have to look at every pixmap in the system to see if it would be overlapped and would have to move. And pixmaps can have Z buffers and they'd have to move with the pixmap. We were unable to come up with any sort of algorithm for visiting all the pixmaps that could possibly be affected in any sort of reasonable time.

So we borrowed an idea from operating systems and divided the off-screen memory into a manageable number of equal-sized pieces. (See Figure 3.) Since the pieces were two dimensional we called them "aliquots" and not "pages". Our 1280x1024 buffers could be divided into 80 128x128 aliquots. This size seemed reasonable for a list we expected to have to traverse often in a linear search. The server kept a list of what drawables, if any, were in each aliquot. Then when a window was moved, all we had to do was see which aliquots it would cover that it didn't before. If there were any pixmaps in the aliquot, they had to be moved. If there wasn't space for them in the buffer, they would have to be swapped out or moved to another buffer. In Figure 2, for example, suppose that Back Buffer A had to be moved from location (50, 50) to location (300, 300) in order remain aligned with its window. When it moved, it would overlap Pixmap C and Pixmap D. Pixmap C could be slid to the right by one aliquot. Pixmap D, however, would no longer fit in the buffer. Unless there was space for it in another buffer, it would have to be swapped out to main memory.

When a client wanted to allocate a pixmap, all the server had to do was see if there were enough empty adjacent aliquots. If there weren't it could wander over the aliquot table figuring out which drawables to swap out of off-screen memory.

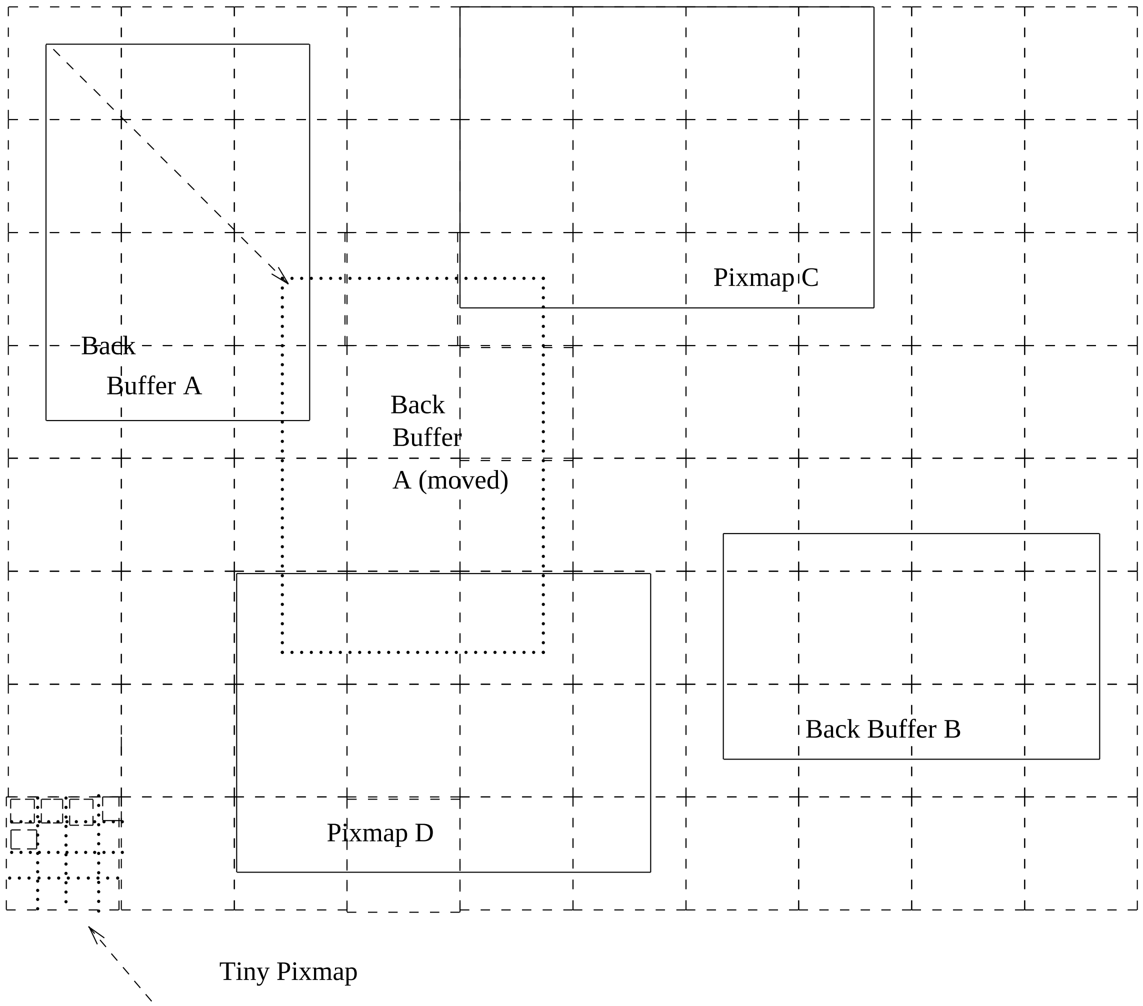

Figure 2. Two Dimensional Off-Screen Memory Allocation

Originally, we decided not to allow pixmaps to share an aliquot, because if they did, the server would have to do intersection calculations to see if there was danger of them overlapping. But we realized that a running X server has many small pixmaps active at the same time. There's a pixmap for every icon, after all. So we decided that tiny pixmaps could share one aliquot. The aliquot was divided into 16 32x32 pieces and pixmaps were just sequentially allocated them from there. When the tiny pixmap aliquot filled up, the server allocated a new one.

This scheme seems to work pretty well. Obviously, there are many details I've skipped over. And getting all those details fixed consumed one engineer full-time for several months. But I don't think any other scheme would have been easier. We could get much better performance from the allocator if we had more time to tune the heuristics for packing and replacement. One of the nice things about this scheme is that it's quite flexible, so it shouldn't be too hard to continue with our tuning.

Use the Hardware Creatively

In preparing this paper, I almost forgot to mention the obvious performance factor of making clever use of the rendering engine. It's hard for me to say much about this, because details are specific to a

---

[4] Actually, it's the proposed MIT multi-buffering extension. But we chose to limit the number of supported buffers to two, because that's all the hardware can do efficiently if Z buffers are also needed.

graphics system. But I will give two examples to highlight the dramatic effects you can obtain. Someone noticed that many clients create tiles that are small and only use two colors. The default X basket weave root tile also has this property. The PixelStamp has special stippling support. By turning appropriate bitonal tiles into opaque stipples, we sped the "tiling" (now stippling) of the root window by a factor of two to six, depending on the area.

Sometimes you can productively use hardware in ways utterly different than were intended. We once determined that the shapes extension could be implemented using Z buffering hardware in lieu of window enable planes. Set the Z value for the pixels behind the window to 0.0, and for the rest of the screen to 1.0. Set the hardware to render only if the new Z value is greater than the one in the Z buffer. Then render all your 2D with a Z value of 0.5. Pixels will be rendered only in the (arbitrarily-shaped) window just as if you had window-clipping hardware.[5] It pays to be creative in your use of the hardware.

The "Wormhole" Extension

Extending the functionality of X may also be a reasonable way to optimize it. Since basic X11 functionality is inadequate for 3D graphics, we have chosen to implement the PEX extension on our machines. Extensions can be added for other new functionality to support imaging, overlay planes, or video.

You could also consider new extensions to provide better support for drawing primitives already in X. Many applications draw huge numbers of triangles or quadrilaterals in a batch. Another large set of applications change the line color between every primitive. The X protocol provides no single primitive for rendering multiple polygons. Similarly, there is no way to bundle multiple lines with different colors into one packet. Either one creates a vast number of GCs and renders each line with a separate GC, or one executes a ChangeGC request between each polyline request. It would be really nice to have an official X extension to support a MultiPolygon request and a ColorPerPrimitiveLine request. We experimented with such an extension, which we called the "wormhole" extension. We saw a seven-fold improvement in polygon rates. This is another way that graphics engines can be kept busier by feeding them larger packets.

Conclusion

Extracting the full benefit of even a good graphics accelerator requires skillful optimization. Such optimization must focus on graphics as actually used by real applications. Optimization must also be made in the context of the whole system — again, as it is actually used by real users. These criteria help determine what features to optimize and how to optimize them.

The critical performance factors for graphics workstations are memory references and hardware utilization. You can minimize memory references by improving the data transport system and improving locality of reference. And you can keep the hardware busy by reducing idle time and by batching data into larger packets. The developer’s task is to optimize these critical factors for the focal areas determined by analysis.

Graphics accelerators provide the possibility of making applications run much faster than they ever can on a simple frame buffer. To turn that possibility into an actuality, to extract the maximum utility from the graphics hardware requires a lot of creativity and a lot of hard work.

---

[5] We didn’t do this in our server because we had already decided not to support the shapes extension in R3.