SEPTEMBER 1989

WRL Technical Note TN-9

Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer

Joel McCormack

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a technical note. We use this form for rapid distribution of technical material. Usually this represents research in progress. Research reports are normally accounts of completed research and may include material from earlier technical notes.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
DARPA Internet:	`WRL-Techreports@decwrl.dec.com`
CSnet:	`WRL-Techreports@decwrl.dec.com`
UUCP:	`decwrl!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Smart Code, Stupid Memory:
A Fast X Server for a Dumb Color Frame Buffer

Joel McCormack

September, 1989

Abstract

Processor speeds are improving faster than memory access speeds. The current generation of RISC processors can perform many 2-dimensional graphics operations at memory bandwidth speeds, rendering specialized hardware unnecessary. This paper describes the DECStation 3100 color frame buffer hardware and several of the graphics algorithms used in the X server implementation. Measured performance numbers are presented and compared to memory bandwidth speed, and possible frame buffer improvements are discussed.

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

1. Introduction

Color workstations have typically had a screen size of 1024 columns by 768 to 1024 rows, 8 bits per pixel, and a 1 to 3 MIPS processor. Except at the very low end, color workstations are traditionally equipped with special-purpose graphics accelerators in order to achieve reasonable performance.

The DECStation 3100 supports a 1024x864x8 color display, but uses no special graphics hardware. The 14 MIPS R2000 processor[1] paints directly into a simple frame buffer. We originally hoped to obtain graphics performance equivalent to the 3 MIPS VAXStation 3100, which has two specialized graphics chips built using fairly old technology. We ultimately found that the frame buffer offers graphics performance roughly 2 to 4 times better than the chip set.

This paper discusses the video memory system and some useful characteristics of the R2000 instruction set. Graphics algorithms for painting lines and text and for copying data are shown to illustrate how the server exploits the capabilities of the R2000, and to show the memory bandwidths these algorithms are capable of using. Finally, comparing the relative performance impact of processor speed and memory bandwidth suggests that increasing bandwidth is more important than using special-purpose hardware to improve rendering computations. Complex 2-dimensional graphics accelerators can still outperform a frame buffer, but for most applications this is costly overkill: smart code talking to stupid memory is sufficient.

2. Frame Buffer Hardware

The DECStation 3100 color frame buffer is arranged as 1024 rows of 1024 8-bit pixels. The display hardware uses each pixel as an index into a 256-entry colormap, which delivers 24 bits of RGB data to the digital-to-analog video converter. The smallest address in a scanline (last ten bits are all 0) displays at the leftmost edge of the screen; the largest address in a scanline (last ten bits are all 1) displays at the rightmost edge of the screen.

Only the first 864 rows are actually displayed. In systems with hardware graphics accelerators, the extra 160 rows would be used as an off-screen cache for font information or pixmaps. We don't bother using this memory: it offers no performance advantage over main memory, and would only add to the complexity of memory management.

The processor can read or write any contiguous group of 1 to 4 pixels in a frame buffer word with one memory access. Read and write times are nominally 6 cycles each, where a cycle is 60 nanoseconds. Measured times are 400 nanoseconds for a write, 410 for a read, and 760 for a read followed by a write. Sustained write bandwidth is about 10 megabytes per second; sustained copy bandwidth is about 5.3 megabytes per second.

The frame buffer has an 8-bit planemask, which is replicated 4 times across the word. A 1 in the planemask allows the corresponding destination bit to be overwritten, a 0 in the planemask

[1]Throughout this paper MIPS refers to millions of instructions per second. R2000 refers specifically to the RISC chip designed by MIPS Computer Systems, and more generally to the associated support software. Statements about the R2000 usually apply to the R3000, R4000, and R6000 architectures as well.

leaves the destination bit unchanged. This allows the server to avoid a read/modify/write cycle when a graphics request specifies a planemask with one or more 0 bits.

3. CPU Hardware

The DECStation 3100 uses a MIPS Computer System R2000 processor [5] with a 60 nanosecond clock. This processor is generally rated at 12-14 times the speed of a VAX 11/780 on integer operations.

The processor has a 64 kilobyte direct-mapped instruction cache, and a 64 kilobyte write-through direct-mapped data cache. We assume that small amounts of code and small data tables are cached if they have been accessed recently. Information from the data cache is not available to the instruction immediately following a load, but the R2000 instruction scheduler usually fills this load-delay slot with useful work.

Writes to both cached and uncached memory go through a 4-word deep write buffer. The buffer is generally good for unrolling loops: the processor executes several store instructions, then performs loop overhead ‘‘for free’’ while the write buffer drains to memory. The write buffer also coalesces sequential writes to contiguous bytes, but the algorithms described below usually write contiguous bytes in one memory access anyway.

The R2000 chip can be configured for ‘‘big-endian’’ or ‘‘little-endian’’ byte order. For compatibility with the VAX architecture, the DECStation 3100 is little-endian, which means that byte 0 of a word is the least significant 8 bits, and that byte 3 of a word is the most significant 8 bits. All algorithms are described in little-endian terms; with minor modifications they would work equally well on a big-endian R2000.

4. Addressing the Frame Buffer

In theory, the R2000 memory addressing architecture offers four possibilities for accessing the frame buffer: addresses can be physical or virtual, and the referenced data can be cached or uncached.

In reality, physical addressing is available only in the protected kernel mode; the processor traps a user program attempting to use a physical address. The X server can use physical addresses only if the server is incorporated into the operating system kernel, but putting the server into the kernel makes it hard to debug and update the server, and reduces the reliability of the kernel. Instead, the server runs in user mode, and uses virtual addresses to access the frame buffer; this causes some performance degradation that is discussed later.

We don’t cache accesses to frame buffer memory for several reasons. First, the displayed portion of the screen is 864 kilobytes, but the data cache is only 64 kilobytes. Painting the contents of even a small window would saturate the cache with screen data, pushing aside large amounts of useful data already in the cache. Second, nearly all accesses to the color frame buffer are write instructions. Finally, we measured performance of cached and uncached access to the monochrome frame buffer, which uses 108 kilobytes; due to the configuration of video RAMs used, this memory is mapped onto only 32 kilobytes of the data cache. The monochrome paint-

ing code often performs several read/modify/write cycles on the same word, so we expected the cache to help some operations substantially. Benchmarks showed that caching improved painting at most 10%, and often degraded performance.

5. Useful R2000 Instructions

The R2000 supports several instructions, particularly loads and stores for data smaller than a 32-bit word, that are not found in all RISC architectures. This section describes the semantics of these instructions; later sections show how the server uses them to avoid various read/modify/write idioms, and thus increase performance.

This section describes only the store instructions; there are corresponding load instructions in each case. Also note that all load and store instructions contain a 16-bit signed byte offset, which makes unrolled loops quite efficient. The server uses a small constant offset for each unrolled source and destination reference, then increments both pointers by the total number of bytes processed at the end of the loop.

`Store Byte` (`sb`) stores the least significant byte of a register into the specified destination. `Store Halfword` (`sh`) stores the least significant two bytes of a register into the specified destination, which must be halfword-aligned (the lowest bit of the address must be 0). `Store Word` (`sw`) stores all four bytes into the specified destination, which must be word-aligned (the low two bits of the address must be 00).

`Store Word Right` (`swr`) writes the least significant bytes of the source register to the most significant bytes of the destination word. It writes one to four bytes, depending upon the alignment of the destination address. Conceptually, it starts at the lowest byte in the source register, copying byte by byte until it reaches the highest byte of the destination word. If the destination address is word-aligned, `swr` writes all four bytes, and acts just like a `Store Word` instruction. If the destination address points to the highest byte of a word (the low two bits of the address are 11), `swr` writes the lowest byte of the register into the highest byte of the word. Figure 1 shows an example of `Store Word Right` in action.

`Store Word Left` (`swl`) writes the most significant bytes of the source register to the least significant bytes of the destination word. It writes one to four bytes, conceptually starting at the highest byte in the source register, copying byte by byte until it reaches the lowest byte of the destination word. If the destination address points to the highest byte of a word, `swl` writes all four bytes of the register to the destination word. If the destination address points to the lowest byte of a word, `swr` writes the highest byte of the register to the lowest byte of the word. Figure 2 shows an example of `Store Word Left` in action.

6. X Imaging Modes

X imaging can be broken into two independent problems: computing the shape of the image to paint, and then filling the interior. Conceptually, the shape of an image is broken down into a list of ‘‘spans,’’ where each span is a contiguous sequence of pixels on a scanline. Then each span is painted according to the specified fill style. (For increased speed, many algorithms fill each span immediately, and may also precompute information common to all spans in an object.)

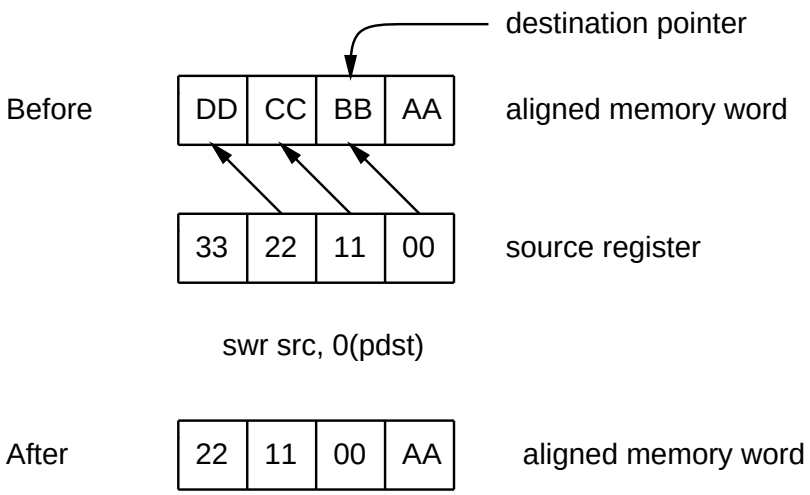


Figure 1: The Store Word Right instruction

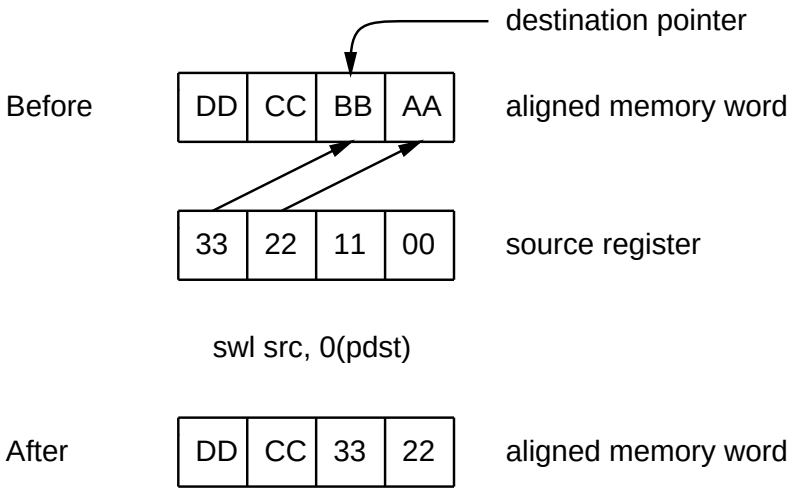


Figure 2: The Store Word Left instruction

There are four fill styles in X: solid, transparent stipple, opaque stipple, and pixel copy. The following sections describe these fill styles, and sketch ways to deal with the ragged left and right edges of a span. Most of the actual implementation techniques for painting these fill styles are described later.

All fill styles are described in terms of a single span, but in reality a shape consists of an arbitrary number of spans. And the word ‘‘overwrite’’ should really be interpreted as ‘‘the current graphics function is applied to the source pixel and the destination pixel.’’ Only the most common graphics function, `GXcopy`, actually overwrites the destination with the source pixel.

6.1. Solid fill

A solid fill style is the simplest possible. The server paints the foreground pixel into the span. Figure 3 shows the first three pixels of a solid fill.

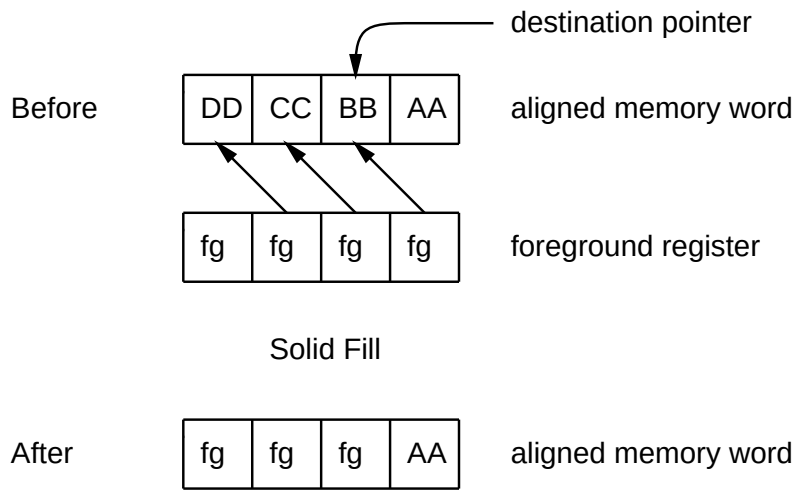


Figure 3: Solid fill style

Whenever possible, the server uses 32-bit writes to the frame buffer for maximum speed. For solid fill, the server replicates the foreground pixel across all four bytes of the source register, and fills the interior of the span using `Store Word`.

Spans are aligned only to pixel boundaries, and not word boundaries, so the server must treat the left and right edge of a span specially. These ‘‘ragged’’ edges may not fill an entire word: at the left edge of a span the server might have to paint one to three bytes into the highest bytes of a word; at the right edge it might have to paint one to three bytes into the lowest bytes of a word.

On most architectures, the server could paint the ragged left edge in one of two ways. The server could perform shifting and masking operations based on the bottom two bits of the destination address, or it could use these two bits to branch into four separate pieces of code. At the right edge, the server could use the same technique, but use the bottom two bits of the number of pixels still left to paint.

On the R2000, the `Store Word Left` and `Store Word Right` instructions, which were originally intended for writing unaligned 32-bit words, prove unexpectedly useful for painting ragged edges.

To paint the ragged left edge, the server writes the source data using `swr`, then increments the destination pointer and decrements the width count by the number of bytes written. Since `swr` writes one to four bytes, the server doesn’t bother special-casing a pointer that is already word-aligned, but unconditionally uses `swr` at the left edge of a span. (If the span is so short that it is completely contained within a single word, then the server just executes the proper store instructions.)

Painting the ragged bytes at the right edge of a span is slightly more complex. When the server exits the word painting loop, there are four or fewer bytes remaining to paint. The server then uses `swl` with the address of the last byte to be painted. (In non-solid fill styles, where the source data does not contain the same pixel in all four bytes, the server has to shift the source data up into the most significant bytes of the source register.) Again, since `swl` writes one to four bytes, the server unconditionally uses `swl` at the right edge of a span.

Figure 4 shows a prototype of the solid fill painting code. This fragment is used only if the span is wide enough to touch two words in the frame buffer.

```
Pixel8    *pdst;
int       width, raggedLeftBytes;
Pixel32   foreground;

/* Paint ragged left edge */
StoreWordRight(foreground, pdst[0]);
raggedLeftBytes = 4 - (pdst & 3);
width -= raggedLeftBytes;
pdst += raggedLeftBytes;

/* Paint full words until 1-4 pixels left */
while (width > 4) do {
    StoreWord(foreground, pdst[0]);
    width -= 4;
    pdst += 4;
}

/* Paint ragged right edge */
pdst += width;
StoreWordLeft(foreground, pdst[-1]);
```

Figure 4: Solid fill painting code

6.2. Transparent stipple

Transparent stipple maps a bitmap into pixels. For each 0 in the bitmap, the server leaves the corresponding destination pixel unmodified. For each 1 in the bitmap, the server overwrites the corresponding destination pixel with the foreground pixel. Figure 5 shows the first three pixels of a transparent stipple fill.

All techniques for painting transparent stipples are fairly big and ugly. Rather than repeat this ugliness several times—at least once for narrow spans, and three more times for the left edge, middle, and right edge of wider spans—the server exploits the ‘‘nop’’ behavior of 0 to handle ragged edges.

At the left edge of a span, the server always aligns the destination pointer to a word boundary. It does this by logically shifting the stipple bits left, using the bottom two bits of the destination pointer to determine the length of the shift. If the bottom bits are 00, the server shifts the stipple left 0 bits; if the bottom bits are 11, the server shifts the stipple left 3 bits. The server then compensates for the extra 0 bits shifted in by setting the bottom two bits of the pointer to 00, which backs up the destination pointer to the beginning of the word.

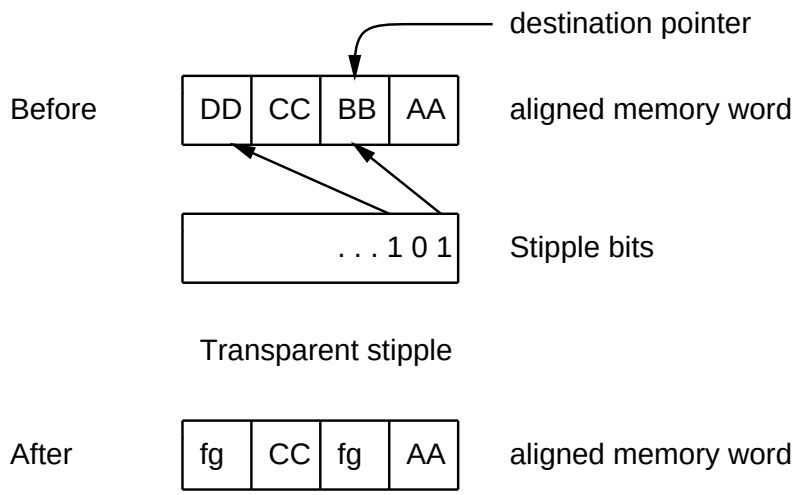


Figure 5: Transparent stipple fill style

At the right edge, the server just masks off any stipple bits past the end of the span. These adjustments to the stipple data effectively make all transparent stipple spans word-aligned at both the left and right edge.

The server exploits the nop property of 0 for speed, too: it stops painting the span when all remaining stipple bits are 0, rather than processing all stipple bits for the width of the span.

6.3. Opaque stipple

Opaque stipple also maps a bitmap into pixels. For each 0 in the bitmap it overwrites the corresponding destination pixel with the background pixel; for each 1 it writes the foreground pixel. Figure 6 shows the first three pixels of an opaque stipple fill.

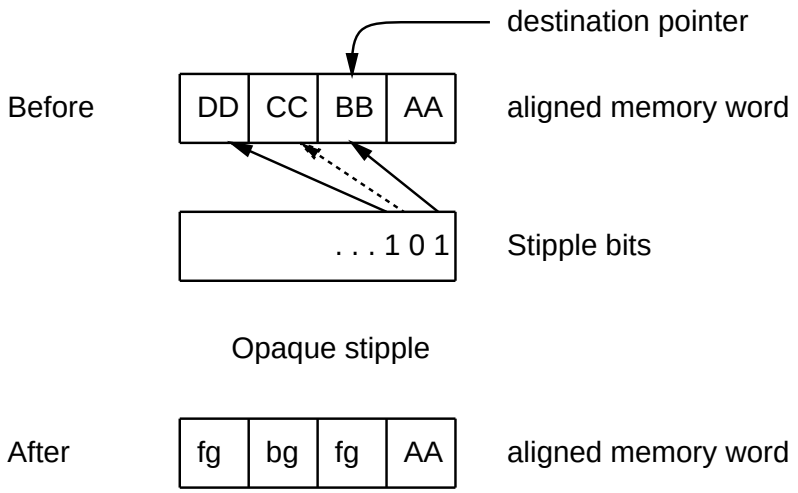


Figure 6: Opaque stipple fill style

The server uses `Store Word Right` at the left edge, and `Store Word Left` at the right edge. At the right edge it must shift data to the most significant bytes of the source register before writing with `swl`.

6.4. Pixel copy

The pixel copy fill style copies pixels from the specified source to the destination. Source and destination can be in any alignment with respect to one another. To maximize throughput, the server usually reads and writes aligned words, making alignment adjustments by shifting the data in intermediate registers. Figure 7 shows the first two pixels of a pixel copy fill.

The source data for the ragged right edge may span two words, so the server doesn't use `swl`, but copies byte-by-byte for the final one to three pixels. The source data for the ragged left edge may span two words as well, but setup for the word painting loop makes it easy to use `swr`.

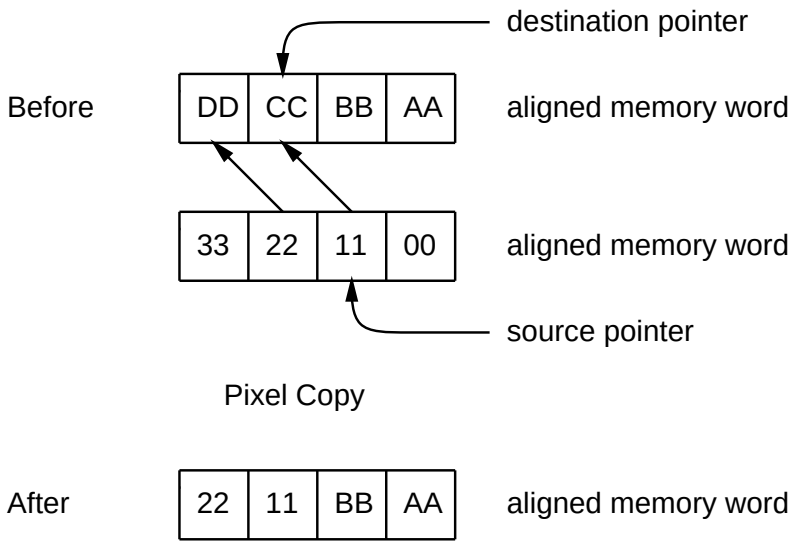


Figure 7: PixelCopy fill style

7. X Graphics Requests

The X protocol contains several types of drawing requests. This paper discusses only a handful of them, in order to illustrate how to efficiently implement the various fill styles in realistic settings. `PolyText` and `ImageText` show implementation techniques for transparent and opaque stipples, respectively. `CopyArea` shows techniques for pixel copies. Finally, the section on line drawing shows how published algorithms are often expressed in a form that is non-optimal, and how these algorithms can be rewritten into more efficient forms using simple mathematical transformations.

The X protocol permits drawing with any of 16 graphics functions, which correspond to the 16 possible Boolean functions of two arguments. The source pixels form the first argument, the pixels already residing at the destination address form the second argument. In all cases the algorithms described are designed to work best with a graphics function of `GXcopy`, which

overwrites the destination pixels with the computed source pixels. The other 15 graphics functions are used much less frequently—less than 1% of the graphics requests in most applications. There are a small class of applications that allocate several planes of the colormap and use graphics functions like `GXxor`, `GXand`, and `GXor` much more frequently. In some cases, different implementation techniques would result in slightly faster algorithms for these graphics functions.

All performance numbers shown are from the `x11perf` X server evaluation program, which measures elapsed time for the server to execute various graphics and window management requests. We used this program extensively in order to time and tune our code.

7.1. PolyText

Applications often use the `PolyText` request to paint a string of characters onto the screen. The server first looks up the font-specific glyph for each character, which is simply a bitmap picture. The server then paints one glyph at a time using transparent stipple semantics: it paints the foreground pixel where there is a corresponding 1 in the glyph, and leaves the destination unmodified where there is a 0 in the glyph. Figure 8 shows how the character code for ‘‘Z’’ is painted.

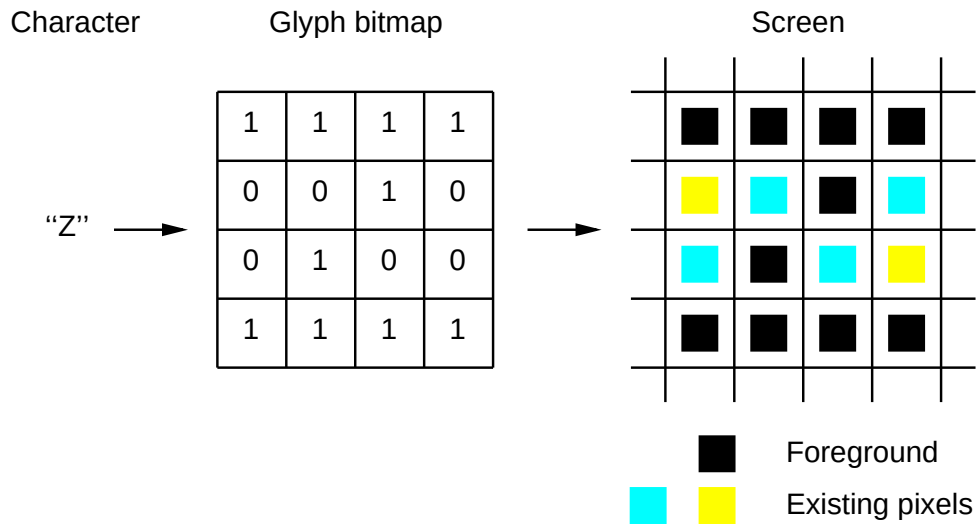
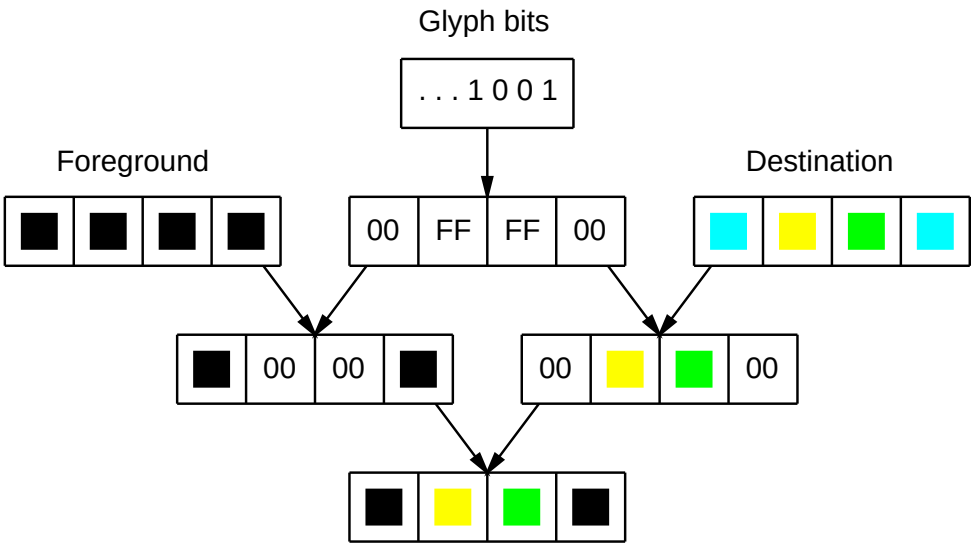


Figure 8: PolyText paints a ‘‘Z’’

`PolyText` is most often used by WYSIWYG editors; such editors paint a line of characters by painting a background rectangle as high as the tallest font in the line, then painting characters with `PolyText`. This avoids problems with mixing fonts of different heights on the same line.

The traditional way to implement transparent stipple is to read the destination word, mask out the pixels to be overwritten, merge in the new pixels, and finally write the modified word back out to memory. Figure 9 shows this process graphically, as well as the C code to implement it.

The main problem with this technique is its poor performance. The best possible code takes 16 cycles for each 4 bits of glyph, including loop overhead. Even if all computation could be hidden, the reading and writing would still cost nearly 13 cycles.




```
index = glyphbits & 0xf;  /* Get 4 bits of glyph     */
mask  = masktable[index]; /* Get mask                */
dstm  = *pdst & mask;     /* Mask destination        */
srcm  = fgword & ~mask;   /* Reverse mask foreground */
*pdst = dstm | srcm;      /* Merge the two           */
```

Figure 9: PolyText implemented with read/modify/write

Another problem arises when this technique is applied to graphics functions other than `GXcopy`. The server cannot simply zero out the destination pixels and merge in the foreground pixels, but must perform equivalent operations appropriate to the graphics function. For example, `GXxor` shouldn't mask anything from the destination, while `GXinvert` must invert the appropriate destination bits and ignore the foreground completely.

The server avoids both problems entirely. By using the store instructions judiciously, the server can write almost any group of one to four contiguous bytes of a word in one memory cycle. The only exception is the middle two bytes, which require two separate writes. Thus, out of the 16 possible pixel patterns represented by 4 glyph bits, 1 case needs 0 write instructions, 9 cases need 1 write instruction, and the remaining 6 cases need 2 write instructions, as shown in figure 10. (The fg register has the foreground pixel replicated across all four bytes.)

Figure 10 shows a total of 21 write instructions, so if all 16 cases are equally likely, each four bits of the glyph requires an average of 1 5/16 writes, or 8 3/4 cycles. In reality, character glyphs are composed of contiguous groups of bits more often than random chance, and the painting code stops as soon as all remining glyph bits are zero, so these numbers are somewhat conservative. Even the worst case of two writes for each four bits is better than the read/modify/write strategy.

How quickly can the R2000 examine four bits of the glyph and branch to the right sequence of writes? The R2000 C compiler is quite good overall, but the front end does not work well in this case, nor does the instruction scheduler. The C front end generates excessive code to dispatch a `switch` statement—evidently the compiler designers assumed that the code in each branch

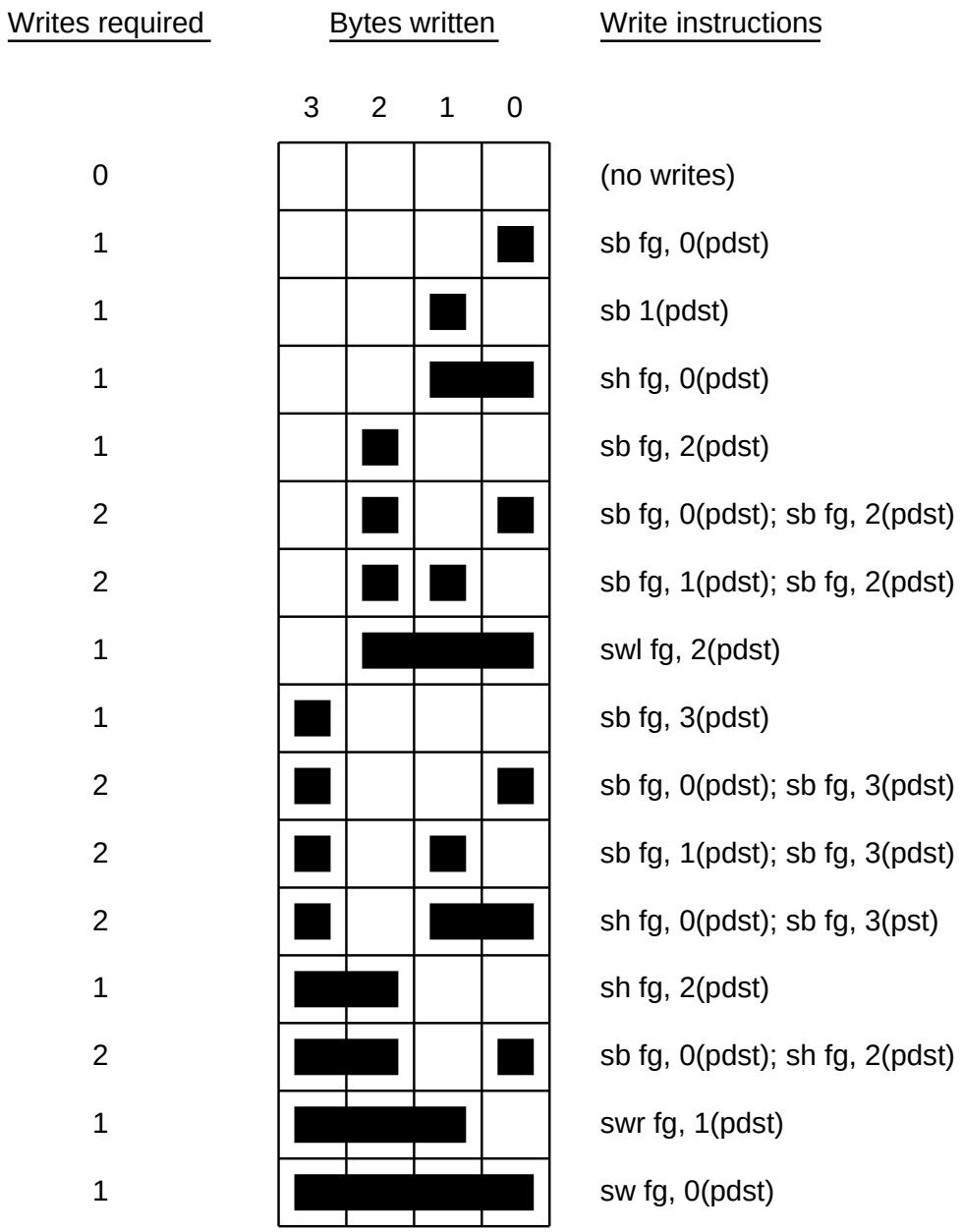


Figure 10: PolyText using only write instructions

would be relatively large. The instruction scheduler does not move code up through the case branches in order to fill load-delay and jump-delay slots, nor does it copy tiny (one or two instruction) basic blocks up to blocks that unconditionally jump to them. The resulting code has tests that needn't be executed, unfilled delay slots, and a high branching frequency.

The result? The compiler generates 16 instructions of branch and loop overhead. Assuming an average of 1 5/16 write instructions, this totals 17 5/16 instructions. The C code paints the variable-pitch font Times Roman 10 (75 dpi) at about 45,000 characters per second, and the fixed width and height font 6x13 at about 32,000 characters per second. These are respectable numbers, but don't approach what is possible.

The color server uses assembly code to implement the above algorithm. Coding in assembler allows optimizations that even a very good compiler and code scheduler would probably miss. In the assembly language version, all 16 case branches are exactly 8 instructions long. Rather than using the bottom 4 glyph bits to index into a case branch table, the assembly code computes an offset from a known instruction, and branches directly to the computed location. Each case contains loop termination code for all three nested loops in which the painting code is embedded—the inner ‘‘loop’’ executes only one to three times per glyph scanline for commonly used fonts, so minimizing all loop overhead is important. And in the assembly version, all delay slots are filled with useful work.

The assembly code paints four glyph bits in an average of 8 5/16 instructions, which is slightly below the (pessimistic) average memory write time of 8 3/4 cycles. The assembly code paints Times Roman 10 at 58,000 characters per second. With one more improvement described in the section below, the assembly code paints 6x13 at 65,000 characters per second, which works out to 5 megabytes per second (though only the foreground pixels are actually painted, of course).

7.2. ImageText

The `ImageText` request is similar to `PolyText`, but paints each glyph using opaque stipple semantics. Since different glyphs in the font may have different heights and widths—the character ‘‘:’’ might map to a 2x7 glyph, while the character ‘‘H’’ might map to a 5x9 glyph—each glyph is conceptually extended up and down with 0 bits to the overall font height. The space between glyphs is also conceptually filled with 0 bits.

`ImageText` is intended mainly for use in terminal emulators, which use fonts in which every glyph has the same width and height (for example, every glyph in 6x13 is 6 bits wide by 13 bits high). Some text windows that allow only one (possibly variable-pitch) font also use `ImageText`, since all glyphs in the font are extended to the same height.

The easiest way to implement `ImageText` is to clear the background rectangle, then use the `PolyText` code to paint in the foreground pixels. This simple technique works surprisingly well, and the color server paints all variable-pitch fonts this way. Trying to paint background and foreground simultaneously is hard: each glyph must be extended up and down, the space between glyphs must be filled in, and in some fonts information from two adjacent glyphs can overlap (as with an overstrike character). Rather than deal with these complications, we ultimately settled for an `ImageText` rate of 43,000 characters per second for the Times Roman 10 font.

The server painted the 6x13 font at only about 35,000 characters per second using this technique. Since fixed-metric fonts contain glyphs that are all the same size and which never overlap, it is easy to paint the background and foreground simultaneously. We thought it would be much faster to map each four bits of the glyph into 4 pixels, as shown in table 1, then write the derived word to memory.

The results were disappointing. Assembly language `ImageText` code painted 6x13 only 3% faster than the simple code that let `PolyText` do most of the work. The problem lies in the narrowness of the glyphs. The `PolyText` technique clears a large rectangle, and deals with alignment effects only at the left and right edges. It then dabs the foreground onto this area by

Glyph bits	fg/bg pixels
0000	bg bg bg bg
0001	bg bg bg fg
0010	bg bg fg bg
. . .	. . .
1111	fg fg fg fg

Table 1: Foreground/background mapping table for opaque stipple

painting a few pixels here and there. The `ImageText` code deals with alignment on every glyph; averaging over the four possible alignments of a 6-bit wide glyph gives us 1.5 writes per four bits of glyph. And overhead is quite high, as the inner ‘‘loop’’ is executed at most once.

Several people had suggested that all text painting might be done a scanline at a time, rather than glyph by glyph. That is, the server should paint the entire top line of a string, then the next line, etc. Sketch implementations suggested that a general scanline algorithm would shuffle overhead around without improving painting rates, as the write-through cache can’t be used to assemble data very efficiently.

However, this technique works well on a smaller scale, using registers to hold one row of information from a small number of glyphs. The assembly-language implementation of `ImageText` for fixed-width fonts of width 8 or less processes four glyphs at a time. All four glyph rows fit into a single 32-bit word, and assembling the word is easy. The 6x13 `ImageText` rate is now 60,000 characters per second, or about 4.7 megabytes per second.[2] At this speed, it is faster to regenerate text windows from scratch than it is copy them to and from backing storage.

The server should be extended to use the same technique to paint three glyphs at a time for fonts of width 9 or 10, and to paint two glyphs at a time for fonts of width 11 to 16. Though these fonts are used infrequently today, users will switch to larger fonts as screen densities increase.

7.3. CopyArea

The `CopyArea` request copies pixels from one rectangle to another rectangle of the same size. If the origins of the source and destination rectangles are assumed to be random, they are aligned (have the same two low-order bits) 1/4 of the time. In reality, many applications scroll data vertically, so the aligned case occurs much more frequently.

It would seem an easy matter to optimize the scrolling case, but experience proved otherwise. The server does a bit of work at the left and right edges, but copies all words in between directly from source to destination, with no data massaging necessary. The hardware designers claimed that turning around the memory controller chip from read to write mode took two cycles; in order to reduce this cost, the server read in four words from the source, then wrote four words to the destination. This code should have achieved just over 5 megabytes per second, but only hit 4.7 megabytes.

[2]Keith Packard at MIT has implemented this technique in C, and gets 46,000 characters per second.

Some time later, we measured performance of the X11 version 3 color frame buffer code from MIT. The MIT code scrolled large areas at 4.95 megabytes per second. Investigation showed that the R2000 instruction scheduler had pessimized rather than optimized our code: it had moved all loop overhead code before the stores, so the overhead code no longer executed in the write buffer's shadow.

Rescheduling the compiler-generated assembly sources by hand didn't improve performance. The faster MIT code copied four words at a time in an unwound loop, but always wrote each word as soon as it read it. An even simpler loop resulted in even better performance: the current loop copies a single word per iteration, and achieves a scrolling rate of 5.3 megabytes per second.

No one has adequately explained this oddity. The fastest memory writing loop takes about 400-410 nanoseconds per word. The fastest memory reading loop is about 10 nanoseconds slower. The fastest read/write loop is only about 760 nanoseconds per word. The moral of the story: tuning based on measured performance gets better results than tuning based on theoretical calculations.

For unaligned source and destination pointers, the server reads each source word once, and writes each destination word once, which requires the processor to rotate and merge words. On the R2000, this operation takes three instructions: shift one data word left, shift the other data word right, and merge the shifted words. The server uses logical shifts that bring in 0 bits in order to avoid masking operations. (Some RISC machines, such as the WRL Titan [7], offer a register extract instruction. This instruction gets a 32-bit word from any position in the concatenation of two registers, allowing the rotate and merge operation to execute in one cycle.)

Loop overhead plus merging words is slower than the available memory bandwidth, so the server unrolls the loop four times. Copying large areas when source and destination are evenly distributed among all four alignments executes at 4.9 megabytes per second.

7.4. Lines

The X protocol precisely specifies which pixels are affected for line widths of 1 or greater. The protocol attaches much looser constraints to a line width of 0; servers still paint a line approximately 1 pixel in width, but use much faster software algorithms or existing graphics hardware. This section deals only with painting 0-width lines.

Bresenham's line drawing algorithm [3] uses integer arithmetic to keep track of a line's distance from the nearest pixel, and paints one pixel in each iteration. For a line with a slope between 0 and 1, the algorithm always steps right one pixel. If the vertical distance to the line from this new position is less than 1/2 pixel, the algorithm stays on the same scanline; otherwise, it steps up one pixel as well.

In the literature, Bresenham code for lines is almost always non-optimal for a software implementation. For example, References [4], [6], [8] all show algorithms akin to the following:

```
*pdst = fg;
if (e > 0) {
    pdst += scanlineWidth;
    e += e2;
} else {
   e += e1;
}
pdst++;
```

The variable `e` represents the distance of the pixel from the line. The adjustment `e1` represents the incremental distance to add for a horizontal step, while `e2` is the incremental distance to add for a diagonal step.

The derivation of integer algorithms for lines and conic sections in Reference [8] have a clear advantage over the other works. The resulting algorithms show how the derivations were achieved, and also keep terms broken into pieces that are easy to rearrange into more efficient forms.

For example, the `else` part can be removed entirely by unconditionally adding `e1` to `e`, and compensating for this by using `e2′ = e2 - e1`. Though not an issue on RISC machines, the `if` usually generates a test instruction to set condition codes on CISC machines. This test is unnecessary if the new value of `e` is computed immediately before the test. We can move the computation if we offset `e′ = e - e1`. The improved algorithm is:

```
*pdst = fg;
e′ += e1;
if (e′ > 0) {
    e′ += e2′;
    pdst += scanlineWidth;
}
pdst++;
```

The color server unrolls the loop surrounding this code four times. Each point takes 5 3/4 instructions on average for lines with slope less than 1, and 6 3/4 instructions on average for lines with greater slopes.

Early benchmarks of line drawing speed on the DECStation 3100 showed performance at least five times slower than projected. A program that measures the time it takes to handle Translation Lookaside Buffer (TLB) traps showed that the kernel was taking about 38 microseconds every time it accessed a page of frame buffer memory not currently in the TLB. On the R2000, the TLB effectively contains 56 4-kilobyte pages. Each scanline of the color frame buffer is 1 kilobyte, so a page spans 4 scanlines. The entire screen is 216 pages, or almost four times the size of the TLB. Any line that covered much vertical distance spent most of its time faulting, not painting.

The R2000 kernel is capable of handling most TLB faults in 17 instructions, which is a tad longer than a microsecond. But the kernel had incorrectly set up the virtual memory interface to the frame buffer, which caused every frame buffer TLB fault to take a very slow path. Once this problem was remedied, line drawing sped up about 9 times, and many other operations sped up 2 to 3 times.

Even so, any line with a sloper greater than 1 hits a new page every four pixels. While it is theoretically possible to paint long lines (500 pixels) at about 2.5 megabytes per second, TLB filling reduces this rate to 2.0 megabytes per second. Lines that are 10 pixels long and evenly distributed among all orientations paint at 47,000 lines per second, or about 1/4 available bandwidth.

The trend toward larger page sizes will reduce TLB faulting on frame buffer accesses somewhat, but frame buffers are getting larger at the same time. The best solution is to include a special TLB entry dedicated to mapping a frame buffer into virtual memory; if this entry points to an 8 or 16 megabyte page, accesses to even a very large frame buffer never cause a TLB fault.

8. Building a Better Frame Buffer

The R2000 can saturate or nearly saturate memory bandwidth for a large number of graphics operations. The R3000 chip has been available for several months, and runs about 50% faster than the R2000. Faster processors are on the drawing board. As processors get faster, more and more operations will fall into the bandwidth-limited domain. Obviously, memory bandwidth needs to be increased.

Graphics processors from Sun Microsystems and Silicon Graphics have memory bandwidths in the range of 80 to 110 megabytes per second, which is about 10 times faster than the DECStation 3100. High-speed graphics processors get bandwidth in several ways: they use memory interleaving, they don't communicate with memory over a general-purpose bus, and they often use a 64-bit (or wider) path to memory. Note that this bandwidth is very asymmetric: though wide objects paint at 80 megabytes per second, vertical lines usually paint at less than 4 megabytes per second.

By definition, a frame buffer shares the bus with the processor, which leaves only memory interleaving for increasing bandwidth. But as memory densities increase, interleaving possibilities decrease. The DECStation 3100 uses 8 256x4 bit video RAMs to provide 1 megabyte of memory. At this density, the only way to interleave is to double or quadruple memory, and let the server use the extra memory as it sees fit. Note, however, that video RAMs cost about twice as much as ordinary dynamic RAMs.

Fortunately, video RAMs are now providing ‘‘page-mode’’ or ‘‘static-column’’ addressing. The first access to a memory location requires a complete memory cycle, but subsequent read or write accesses to the same internal page are much faster. If the bus is designed carefully, it should be possible to get data rates of about 32 to 40 megabytes per second without much additional expense. The internal page size of a 256x8 video RAM translates into two 1024-byte scanlines, so page-mode access improves vertical line drawing to 4 to 5 megabytes per second.

9. Frame Buffers vs. Graphics Processors

Frame buffers are simple and cheap to build. Better yet, you don't have to get everything right to ship a machine. Each release of the DECStation 3100 color server has incorporated significant performance enhancements. Even so, some graphics requests are still slow; as we work out improved algorithms for these requests, we can offer better performance with no hardware

changes. And page-mode access could increase performance substantially with only minimal code changes.

On the other hand, a frame buffer offers no parallelism between setting up paint requests, and performing the actual painting. The color frame buffer paints short lines at only 1/4 of currently available bandwidth, and this ratio will get worse with faster memory. A server might paint short lines two to ten times faster if some of this work were offloaded to a specialized graphics processor. This parallelism seems to be the greatest advantage of hardware accelerators. Ironically, preliminary performance numbers for the Silicon Graphics and Sun GX X implementations show that the dumb frame buffer code outperforms these implementations on small objects, and only falls behind on large objects. (These early implementations should improve substantially, as applications that write directly to the graphics controllers obtain much better performance.)

Even though specialized graphics hardware can offer better 2-dimensional performance than a frame buffer, this advantage will soon be inconsequential for most applications. Like nuclear arsenals, at some point enough is enough. Users don't care if they can paint text at 200,000 characters per second using a graphics accelerator, but only 150,000 characters per second on a frame buffer: both devices paint a page of text instantaneously for all practical purposes. The frame buffer, once a poor man's graphic device, is sufficient for all but the most demanding applications.

10. Acknowledgements

Susan Angebranndt, Raymond Drewry, Phil Karlton, and Todd Newman wrote most of the X sample server and documented how to port it [1, 2].

Todd Newman wrote code to fill spans, copy pixels, paint window backgrounds, and draw polygons. Vasudev Bhandarkar and Rich Hyde wrote the device driver and driver interface.

Susan Angebranndt, Chris Kent, and Phil Karlton wrote the original `x11perf` program, which I later rewrote and added to. If you are writing or tuning X servers, you should have this program, which is now available from MIT via anonymous FTP to expo.lcs.mit.edu:/contrib/x11perf.tar.Z.

Bob Alverson wrote the thrash program to measure TLB fault handling time.

Mary Jo Doherty made useful comments on an earlier version of this paper.

11. References

[1] Susan Angebranndt, Raymond Drewry, Phil Karlton, Todd Newman.
Definition of the Porting Layer for the X v11 Sample Server
X Version 11 Release 3 edition, Software Distribution Center, Massachusetts Institute of Technology, Cambridge, MA, 1988.

[2] Susan Angebranndt, Raymond Drewry, Phil Karlton, Todd Newman.
Strategies for Porting the X v11 Sample Server
X Version 11 Release 3 edition, Software Distribution Center, Massachusetts Institute of Technology, Cambridge, MA, 1988.

[3] J. E. Bresenham.
Algorithm for Computer Control of a Digital Plotter.
IBM Systems Journal 4(1):25-30, 1965.

[4] J.D. Foley, A. Van Dam.
Fundamentals of Interactive Computer Graphics.
Addison-Wesley , Reading, Massachusetts, 1982.

[5] Gerry Kane.
MIPS R2000 RISC Architecture.
Prentice Hall, Englewood Cliffs, NJ, 1987.

[6] William M. Newman, Robert F. Sproull.
Principles of Interactive Computer Graphics.
McGraw-Hill Book Company, New York, NY, 1979.

[7] Michael J. K. Nielsen.
Titan System Manual.
Research Report 86/1, Digital Equipment Corporation, Western Research Laboratory, September 19, 1986.

[8] Jerry Van Aken and Mark Novak.
Curve-Drawing Algorithms for Raster Displays.
ACM Transactions on Graphics 4(2):147-169, April, 1985.

WRL Research Reports

‘‘Titan System Manual.’’
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’
David W. Wall.
WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’
David W. Wall.
WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’
Paul J. Asente.
WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’
Brian K. Reid.
WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’
Jeremy Dion.
WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’
Silvio Turrini.
WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

‘‘Leaf: A Netlist to Layout Converter for ECL Gates.’’
Robert L. Alverson and Norman P. Jouppi.
WRL Research Report 89/12, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’
David W. Wall.
WRL Research Report 89/17, September 1989.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’
Joel McCormack.
WRL Technical Note TN-9, September 1989.

Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer

Table of Contents

1. Introduction **1**
2. Frame Buffer Hardware **1**
3. CPU Hardware **2**
4. Addressing the Frame Buffer **2**
5. Useful R2000 Instructions **3**
6. X Imaging Modes **3**
6.1. Solid fill **5**
6.2. Transparent stipple **6**
6.3. Opaque stipple **7**
6.4. Pixel copy **8**
7. X Graphics Requests **8**
7.1. PolyText **9**
7.2. ImageText **12**
7.3. CopyArea **13**
7.4. Lines **14**
8. Building a Better Frame Buffer **16**
9. Frame Buffers vs. Graphics Processors **16**
10. Acknowledgements **17**
11. References **18**

List of Figures

Figure 1: The Store Word Right instruction **4**
Figure 2: The Store Word Left instruction **4**
Figure 3: Solid fill style **5**
Figure 4: Solid fill painting code **6**
Figure 5: Transparent stipple fill style **7**
Figure 6: Opaque stipple fill style **7**
Figure 7: PixelCopy fill style **8**
Figure 8: PolyText paints a ‘‘Z’’ **9**
Figure 9: PolyText implemented with read/modify/write **10**
Figure 10: PolyText using only write instructions **11**

List of Tables

Table 1: Foreground/background mapping table for opaque stipple 13