Hardware/Software Tradeoffs
for
Bitmap Graphics on the Blit

*Rob Pike**

*Bart Locanthi**

John Reiser

ABSTRACT

The Blit is an experimental bitmap graphics terminal built for research into interactive computer graphics on the UNIX time-sharing system. The hardware is inexpensive and the graphics functions are implemented entirely in software. Nevertheless, the graphics performance of the Blit is comparable or superior to some displays with special purpose graphics hardware. This paper explains the paradox by referring to some principles of design: the hardware and software should be designed together to complement each other; carefully designed software can outperform infelicitous hardware; and simplicity of design leads to efficiency of execution. These principles are illustrated by examples from the Blit hardware and software and comparisons with other systems.

March 30, 2000

* Current address: Bell Laboratories Murray Hill, New Jersey 07974

AT&T Bell Laboratories Holmdel, NJ 07733. Current address: Valid Logic Systems, Mountain View CA 94043

Hardware/Software Tradeoffs
for
Bitmap Graphics on the Blit

*Rob Pike**
*Bart Locanthi**
John Reiser

Overview of the Blit graphics system

The Blit[15] is an experimental bitmap graphics terminal for UNIX systems, providing approximately the graphics power of the Xerox Alto,[19] but using a 16-bit microprocessor, fused logic arrays and 64K RAM chips to keep the hardware cost comparable to that of an ''intelligent'' terminal. (The Teletype DMD 5620 terminal is a commercial product based on the Blit, but this paper is about the Blit itself.) Unlike most systems, the Blit's hardware and software were designed together, by the same people (Locanthi and Pike). Although the exact use of the terminal was unknown while the hardware was designed, it was decided that the graphics functions in the software would be based on the bitblt[8] primitive that had proved so successful on the Alto. Proposed hardware designs were therefore evaluated by comparing portions of the assembly code to execute bitblt for the various architectures.

Although most trial designs had explicit hardware support for bitblt, the final design of the Blit's graphics hardware is characterized more by what it does *not* have than by what it does have. The Blit has no special graphics memory, no display list processor, no hardware for moving the mouse cursor, and no pixel-moving hardware. Instead, it has 256 Kbytes of contiguously addressable dual-ported RAM, a connected 100 Kbytes of which is scanned to the display at 30Hz (60Hz interlaced), forming an image 800 one-bit pixels wide and 1024 high. The remaining 156 Kbytes are used for program and data, but the division of the address space is imposed entirely by software: there is only one address space, shared by the processor and the display. The CPU executes instructions from the display memory, because that is the only RAM in the terminal. There are two registers in the display hardware. One sets the base address of the display, so the program can display any part of memory, and the other determines how bits in the memory are displayed. The software sets this register so 0 bits are bright, 1's dark. In the processor's address space are also 24 Kbytes of ROM.

To change the displayed image, the processor, a Motorola MC68000 running at 8 MHz, changes the contents of the portion of memory scanned by the display. Four software primitives — bitblt, texture, segment and point — form the complete set of graphics operators; all changes to a picture go through one of these primitives. We have found no need to augment the primitives by special cases designed for efficiency. As will be explained, the uniformity of the Blit's memory structure allows the general primitives to perform as well as special-purpose code; the Blit outperforms costlier commercial displays that have special purpose graphics hardware.

The following sections introduce the data types and graphics primitives, describe the memory structure and how it influences software and performance, provide a detailed discussion of the Blit implementation of bitblt, and compare the Blit design to some commercial displays. The next two sections discuss the texture and segment primitives, and the last section talks about the implementation of graphics input on the Blit, because it is a tradeoff microcosm all its own.

---

* Current address: Bell Laboratories Murray Hill, New Jersey 07974
Bell Laboratories Holmdel, NJ 07733. Current address: Valid Logic Systems, Mountain View CA 94043

Software I: Data Objects

One feature of the Blit software is the use of data structures, which are passed by value to the graphics primitives. This is in contrast to passing four integers to describe a rectangle, or six for a bitmap. For example, bitblt has only 5 arguments, although they constitute 19 integers and addresses. Of course to specify, say, an arbitrary rectangle, four integers must be provided, but in practice the rectangle needed is already available or can easily be assembled from existing rectangles. Because the geometric structures are passed by value, they can be assembled as in-line expressions, which simplifies clipping and other transformations. Code such as

```
draw_point(transform(pt))
```

is easy to read and understand, and is efficient (although it took some work on the code generator of our compiler to make it so). Also, the data structures are built somewhat hierarchically, so, for example, a rectangle consists of two points rather than (say) one point and a diagonal. All the components of the objects are therefore of the same type and in the same space, which helps the data structures and graphics primitives fit together smoothly.

The Point data structure is made of two 16-bit integers that specify the coordinates of a pixel in a bitmap, and is declared in C (our implementation language) as:

```
typedef struct {
        int     x, y;
} Point;
```

All bitmaps share a single coordinate space, with (0, 0) in the upper left. (The orientation of the y axis is one thing we might change were we to do it again.) Coordinates are integers, not reals, and can be translated but not scaled; the primitives all work in ‘screen’ coordinates, one unit per pixel, because that is the coordinate system supported by the hardware.

A Rectangle is built from two Points that specify the pixels at the minimum and maximum (x, y) positions in the rectangle:

```
typedef struct {
        Point   origin; /* minimum x, y; upper left  */
        Point   corner; /* maximum x, y; lower right */
} Rectangle;
```

Rectangles are half-open: they exclude the horizontal and vertical lines that pass through Rectangle.corner. This implies that two horizontally adjacent rectangles r_0 and r_1 share no pixels when r_0`.corner.x` = r_1`.origin.x`, which simplifies subdivision of rectangular regions. It is convenient enough to abut geometrical objects by sharing coordinates without duplicating pixels that the half-open property is extended to line segments, to simplify construction of longer lines and curves.

The fundamental data structure in the Blit software is a Bitmap:

```
typedef struct {
        Word                    *base;  /* pointer to
                                                word that includes the Point rect.origin */
        unsigned int    width;  /* number of full words across scan line */
        Rectangle               rect;   /* coordinates and size */
} Bitmap;
```

A Word is the hardware’s quantum of memory: a 16-bit integer. width and base are related to storage allocation and will be discussed later. rect serves two purposes. First, and more obvious, it defines the size of the image by marking its extent in rect.origin and rect.corner. It also provides a coordinate system inside the Bitmap by specifying a clipping region inside the storage containing the image. Because a Bitmap need not be an integral number of Words wide, the clipping edges might not lie on Word boundaries.

By convention, the only access to the display is through a global Bitmap called display. The display

C programmers will recognize an analogy with 0-indexed arrays.

bitmap has no special properties in the software. Its rect is {(0, 0), (800, 1024)}, so to draw an object on the screen, a program calls a bitmap primitive operator with argument display Bitmaps are passed by reference, because they are relatively static objects) and the operator will clip the object to the visible screen. But the operator has no notion of ''display bitmap;'' it instead decodes the Bitmap structure it is passed and uses the information rect, width and base, for all clipping and pixel operations. Because any Bitmap has an internal coordinate system, exactly the same procedure will draw the object in any Bitmap, visible or not, and clip the result appropriately. To the graphics primitives, there is no distinction between the display bitmap and invisible or ''off-screen'' bitmaps; they are all equivalent.

Although a one-bit-per-pixel display cannot show colors, it can simulate colors by textures regular patterns of pixels. Since the word size on the MC68000 is 16 bits, textures are defined as 16×16 bit arrays:

```
typedef struct {
        Word    bits[16];
} Texture;
```

A Texture is replicated across a bitmap by aligning its upper left corner with the points (0, 0) mod 16, regardless of clipping. This guarantees that an area can be textured with results independent of any partitioning.

Due to difficulty of implementation in the early versions, our bitblt does not have a Texture argument. Instead, a separate operator, texture, executes the special case of replicating (with various boolean combinations) a texture across a rectangle in a bitmap. The current bitblt implementation could easily be extended to include texturing, but we are content with a separate texture operator. The separation of function is not important to the tradeoffs and design details discussed here.

Software II: Primitives

All the graphics primitives take a Bitmap pointer argument to indicate in which image the object is to be drawn. There is no default Bitmap; even the display bitmap is always named explicitly. The bitmap operators execute a boolean function between a source and destination. The destination is a Bitmap, but the source can be a Bitmap, Texture or an implicit infinite lattice of 1's. A set of function Codes specify the function:

```
STORE:          dest =  source
OR:                     dest |= source
CLR:                    dest &= ~source
XOR:                    dest ^= source
```

Of the 16 dyadic boolean functions, these four have proven sufficient for most applications. They are the four most easily implemented in C, and also the ones used on the Alto, which is how they were chosen. The list could be extended easily because of the architecture of our current bitblt implementation. The use of these Codes will be illustrated in the next few paragraphs.

The simplest primitive changes the value of a single pixel in a bitmap. The subroutine

```
point(bp, p, f)
        Bitmap *bp;
        Point p;
        Code f;
```

sets the pixel p in Bitmap *bp according to f. (This is the prolog of the C declaration for the point subroutine; it declares the subroutine and its arguments). The meaning of f is interpreted by imagining the source to be a 1 bit at p in an otherwise empty (all 0's) bitmap conformal to the destination. Then, Code OR or STORE will turn on the pixel in the destination bitmap, CLR will turn it off, and XOR will flip its state. The discussions of texture and bitblt, below, will manifest the difference between OR and STORE. The Code XOR is particularly convenient because it is its own inverse: the drawing of a pixel with XOR may be undone by another XOR call to point.

Another obvious bitmap operation is drawing a line segment. On the Blit, the primitive

```
segment(bp, p, q, f)
        Bitmap *bp;
        Point p, q;
        Code f;
```

draws an approximation to the line from p to q in the Bitmap *bp, effectively by calling point for each pixel on the line, although the implementation is much more efficient. Lines are half-open, so the pixel at q is not touched; the line is an integer grid approximation to [p, q).

The texturing operator

```
texture(bp, r, tp, f)
        Bitmap *bp;
        Rectangle r;
        Texture *tp;
        Code f;
```

replicates the Texture pointed to by tp over the rectangle specified by r in the Bitmap *bp. The meaning of the codes is much as in the discussion of point, except that since the texture can contain 0 bits, if f is STORE the operator may store 0 and 1 bits in the destination.

One obviously common case of texture is with a texture that is all 1's. The operator will then clear a rectangle CLR), make a rectangle black OR or STORE), or invert white and black in the rectangle XOR). For convenience, and due to the history of the software, an operator

```
rectf(bp, r, f)
        Bitmap *bp;
        Rectangle r;
        Code f;
```

is identical to texture with an implicit texture of 1's. In fact, rectf is implemented by a call to texture, and is in practice by far the most common use of texture. An example is to clear a bitmap, say b, but leaving a MARGIN-wide black border:

```
rectf(&b, b.rect, OR)
rectf(&b, inset(b.rect, MARGIN), CLR)
```

Here, inset is a Rectangle-valued function that returns its argument Rectangle with origin and corner inset by its second argument, an integer. Of course, it would touch fewer pixels, although require somewhat more complicated code, to draw the border as four thin rectangles. Our goal here, however, is primitive bitmap operators that are fast enough that we evaluate complexity by the number of calls to the primitives, not the amount of memory accessed.[7]

The most important bitmap operator is bitblt,[8] called RasterOp in Reference [13]:

```
bitblt(sb, sr, db, dp, f)
        Bitmap *sb;
        Rectangle sr;
        Bitmap *db;
        Point dp;
        Code f.
```

This operation copies, with Code f, the Rectangle sr from the Bitmap pointed to by sb to the congruent Rectangle with origin dp in the destination Bitmap pointed to by dp. On the Blit, the image is clipped to both the source and destination by bitblt. Other systems usually clip at a higher level, but Blit programs call bitblt directly, so the clipping is done there. Enough has been written about the importance of bitblt,[7,8] so we here just show a few examples and defer the discussion on implementation to a later section. The first example is scrolling the display DY pixels:

```
bitblt(&display, Rectangle{0, DY, 800, 1024}, &display, Point{0, 0}, STORE)
```

Notice that the source and destination are the same bitmap. Another frequently executed case is copying an off-screen image such as a menu to the screen from a separate bitmap:

```
bitblt(menu, menu->rect, &display, p, STORE)
```

where p is the point at the upper left corner of the image on the display. Of course, the image in menu would be assembled by calling the bitblt and texture operators with destination Bitmap menu. Blit menus actually work differently. The menu is drawn directly on the screen after saving the screen contents in a dynamically allocated bitmap. Another trick is that three similar calls with Code XOR will exchange the *data* on the screen and in the off-screen bitmap using no auxiliary storage.

The most common bitblt call is to draw a character on the display, but the specifics of character instantiation depend on uninteresting details of the font data structure. The basic idea, however, is simple: store all the characters of a font in a single bitmap, and store a (compressed) Rectangle for each character in the font. Schematically, to draw the character on the screen at p:

```
bitblt(font->bp, font->rect['a'], &display, p, STORE)
```

Although this example uses STORE, most characters on the Blit are drawn in XOR mode, because it is about 10% faster the inner loop of bitblt is simpler and only the minimum enclosing rectangle need be drawn, not the entire vertical span of the font and because it avoids interaction with the mouse cursor; see the section about the graphics mouse, below.

A traditional graphical model, at least on vector displays, is to build a picture from a set of graphics each of which contains some piece of the image, such as a background grid, a menu, or a line drawing of a mechanical part. At any time, the displayed image is formed from some subset of the segments, and segments may be created, destroyed or modified as the program runs. Bitmaps are often used in a similar manner: as well as the obvious global bitmap for the screen, bitmaps are used to hold most of the structured objects drawn on the display character sets, menus, elements of CAD drawings, and so on. But bitmaps are also allocated dynamically, for example to store pieces of the display in a backup bitmap so an object such as a window or pop-up menu can be drawn temporarily in its place. When the object is deleted, the contents of the backup bitmap can then be copied to the display and the bitmap deleted. This style of use treats bitmaps like variables, whereas segments resemble graphical procedures.

The primitive for allocating a bitmap is called balloc:

```
Bitmap *balloc(r)
        Rectangle r;
```

balloc allocates off-screen storage large enough to hold an array of pixels of the size specified by r, and returns a pointer to a Bitmap structure describing the allocated memory. The internal Rectangle in the bitmap is set to r, so for the typical example of saving a piece of the display, the resulting bitmap contains the coordinates of the rectangle where it is to be restored, as in the menu example above.

Hardware I: Memory

The center of the Blit is the dual-ported RAM. In some sense, it is all the graphics hardware in the system: software must do all the work; hardware just displays the result.

The Blit memory is 256 Kbytes of 32-bit wide RAM built from thirty-two 64K by one bit dynamic RAMs. A video processor fetches 32-bit words from the memory and copies them to the display, refreshing the display at 30Hz (60Hz interlaced), and has the side effect of refreshing the dynamic memory chips. Video refresh takes approximately a third of the RAMs' bandwidth. The other memory port is connected to the MC68000 data bus, which is 16 bits wide. An asynchronous arbiter controls access to the memory: if the memory is busy, the requesting entity is suspended until it is freed; otherwise the request is serviced immediately. In practice, the MC68000 usually does no waiting, but when it does it must wait about 500ns. The fraction of memory cycles lost to the video depends greatly on the application, varying from about 10% for CPU-intensive tasks to perhaps 50% for memory-intensive tasks, because the 16-bit data bus must execute two fetches to retrieve a 32-bit quantity.

The memory is one-dimensional and contiguously addressable; to the MC68000, it has no special properties. The video controller copies an arbitrary connected 100 Kbytes to the display. Two adjacent scan lines on the display reside in adjacent 100-byte pieces of memory; the last word of one scan line is followed immediately in memory by the first word of the next scan line. There is no wasted space off the top or side of the display. The remaining 156 Kbytes are used for program and data, including bitmaps.

The most critical property of the memory is that it is a single address space. (Of course, the processor must have sufficient address space to directly access the entire memory at any time.) There is no pure display memory, there is just memory, some of which is displayed. Programs execute from the same memory that is refreshing the display. The processor would obviously be faster if program execution did not rely on the dual-ported memory, but there are performance and functionality advantages that stem from the single memory structure and how it interacts with the Bitmap data type; these will be discussed in the next few sections.

The Bitmap structure is, as discussed above:

```
typedef struct {
        Word                    *base;
        unsigned int    width;
        Rectangle               rect;
} Bitmap;
```

The data structure does not contain the storage for the image, it only describes it. The storage itself begins at the address in base, which is the address of the (16-bit) Word containing the pixel with coordinates rect.origin. width is the number of Words across a scan line; usually, it is the width of the bitmap itself, but in a layer data structure[14] it is the width of the containing bitmap. Since the graphics library puts the display at the last 100 Kbytes of memory, the display structure is initialized as:

```
Bitmap display = {
        156*1024,                       /* size of memory less 100K for display */
        50,                             /* 800 pixels/line divided by 16 bits/Word */
        {0, 0, 800, 1024}
};
```

The single address in display.base and the coordinates in display.rect are all that is needed for the software to address the bitmap. For simplicity and generality of programming and ease of storage management, memory is one dimensional, and any two-dimensionality of interpretation is enforced by software, not hardware. balloc calls a conventional garbage-compacting allocator for storage, rounding the length of a scan line up to a word boundary when necessary. If memory were two-dimensional, balloc would require a 2-D bin packing algorithm, and bitmaps wider than the display could not be created.

This simple memory structure is in contrast to graphics systems that make display memory two-dimensional (typically 1024 bits wide), or place it in a different address space or on the other side of hardware registers. These systems must have special code for drawing on the display bitblt, for example, must execute different code for the cases of memory-to-memory copy, memory-to-display, display-to-memory, and display-to-display. The designers of these systems probably perceive that the special code is the code that does use the display, and thereby force the programmer to avoid off-screen memory for performance reasons. On the Blit, bitblt never knows that it has been passed the display bitmap. That information is irrelevant (although some higher-level code checks base to avoid interference with the mouse cursor).

A couple of examples should illustrate the power of off-screen bitmaps. The most important is layers,[14] an implementation of windows[18] that allows them to overlap and operate independently. A partially obscured layer is a viewport and a collection of off-screen bitmaps whose coordinate systems map to the obscured portions of the layer. The viewport is specified by a bitmap data structure that defines the visible portions of the layer to share storage with the display bitmap, with width set to that of the display; this trick of sharing the width creates a software version of two-dimensional memory. Bitmap operations are filtered through the off-screen bitmaps with any remainder going to the visible part of the viewport. When a layer is made fully visible, its off-screen bitmaps are exchanged with the previously obscuring areas. This scheme demands several properties of bitmaps. Bitmaps must be allocated dynamically, because a layer may have ten or twenty off-screen components on a busy screen. They must have arbitrary dimensions, because the sizes of the off-screen bitmaps depend on the screen layout, and change when new layers are created. Finally, all bitmaps must look the same to the graphics primitives. Because the vast majority of Blit graphics is done under the layers software, the true display bitmap is rarely used; for most programs, running under mpxterm,[15] the variable called display is actually each program's name for its layer, not the whole screen.

The other example is more an efficient algorithm than a new function. An integrated circuit plotting or editing program can take advantage of the hierarchy in many designs by drawing each cell once off-screen and copying it to the screen for each instantiation. This can be done recursively, requiring $O(\log n)$ bitblt calls for a perfectly hierarchical n-rectangle design, but even in practice a modern design has considerable regularity that can be exploited for efficient drawing. A program on a Blit, communicating with the host at 9600 baud but allocating bitmaps in the terminal, can draw a typical circuit faster than a dedicated IC design station without bitblt or dynamic bitmaps.

Integration: Building a terminal using bitblt

For applications in terminals, there are three cases of bitblt that dominate: drawing characters, scrolling windows, and window-window operations such as exchanging off-screen data with the display. These cases also cover the most common graphics operations on personal computers.

Drawing a character requires decoding a font structure to find the location of the character in the font bitmap and calling bitblt to draw the character on the display. For a general font format and typical character sizes, over half the total time to draw a character on the Blit goes into overhead: at least one subroutine call and setup, opening the font, building the argument list for bitblt, calling bitblt, and having bitblt in turn decode and clip its arguments and decide how to draw the image. Because the characters are so small drawing the letter touches 7 words of memory actually changing the pixels in the destination bitmap is relatively unimportant. Our overhead is not unreasonable; the Blit draws about 2500 characters per second in the standard font, which is 9 pixels (not 8) by 14. An experimental version with eight-bit wide characters drawn only on byte boundaries, that avoided the overhead of calling bitblt and used a special font format that was easy to decode (the current format is somewhat compressed for economy of memory), was only a factor of two faster. This is insufficient speedup for so great a loss of generality.

The second common case of bitblt is scrolling a rectangular region of a bitmap, usually the display. Since the word boundaries in the scan lines of a bitmap are at the same place in each line, the speed of scrolling depends primarily on the speed of the MC68000 instruction[12]

```
mov.l   %a0@+, %a1@+
```

or, in C,

```
register long *p, *q;
*p++ = *q++;
```

For typical rectangles, the edges, which must be handled with more complicated code, do not dominate the performance. There is nothing hardware can do to accelerate this loop except provide faster memory access. If the display were accessed through a narrower or clumsier interface, it would take longer to move the data.

The last common case is shuffling on- and off-screen rectangles. It can be made fast by a simple observation: the off-screen bitmaps are allocated by balloc, which is given as argument the rectangle on the display occupied by the data. This rectangle is assigned to rect in the resulting Bitmap. balloc can therefore allocate the bitmap so the word boundaries occur at the same places in the image as they do in the display, reducing to the scrolling case the bitblt call that copies the data. This is the last feature of the Bitmap data structure: Bitmap.rect defines not only the coordinate system but also the word fragmentation; the coordinate modulo 16 is 0 at the first bit of the word in every bitmap. This results in a factor of two to four speedup for window-shuffling bitblt operations and combines neatly with the way textures are generated without diminishing the generality of the graphics primitives. Of course, there is also the wide, non-aligned case of bitblt to be supported, but almost by construction it occurs rarely, and the memory and software are clean enough to make it acceptably fast when it is executed.

Software III: Blit bitblt

bitblt runs two nested loops: an outer loop over scan lines and an inner loop across words in a scanline. Either loop may need to run toward higher or lower addresses, depending on overlap of the source and destination. Although the inner loop has the greater effect on performance, the outer loop must set up the registers for the inner loop. This initialization is probably the most error-prone part of the coding, because it is rich in possibilities for off-by-one errors.

In the general case, separate code in the inner loop must deal with the partial word at the left edge, the partial word at the right edge and the words between, which can be moved 16 or 32 bits at a time. If the source and destination are not bit-aligned (for example, when scrolling horizontally one pixel), the code is complicated further, as in Figure 2. In fact, Figure 2 does not present the most complicated case, because the source and destination can span a different number of words.

There have been five implementations of bitblt for the Blit. Version 1, by Pike, did STORE operations only and treated characters (source width less than 17 bits) with a separate charblt primitive. Locanthi wrote Version 2, which implemented all the Codes, and was significantly faster, but still treated characters specially. This uncomfortable distinction was removed in a third version, also by Locanthi. All these implementations did case analysis in static code and were written in C, although the inner loops of the various cases were liberally sprinkled with in-line assembly language. Reiser next wrote Version 4, in assembler, that used coroutines to process the various portions of each scan line, but this version was slower in some cases such as scrolling.

The latest bitblt, Version 5, also written in assembler by Reiser, is the topic of this section. It compiles optimal code on-the-fly for each invocation, then jumps to the generated code. The code is optimal in the following sense: any faster correct sequence of instructions has its loops unrolled to more than '2X' (the loop has been duplicated in-line more than once), or depends on the values of the bits inside the rectangles involved. (For example, if the destination rectangle already contains the correct answer, a null sequence of instructions would be faster than the sequence generated by bitblt.)

Finding the optimal sequence of instructions takes time, of course. And actually, characters are hopeless. The strategy is to detect when width is less than 17 and jump immediately to static code that always does 32-bit operations although 16-bit operations might suffice in some cases. Characters are so short that there is no time to plan.

In the worst case, `bitblt` executes 400 instructions to generate the optimal sequence. Cases requiring no shift, or which are all edges (inner loop is empty; width as much as 64 pixels), take significantly fewer instructions.

factor	*reason*
3	Code: `STORE`, `CLR`, other
2	adjust source pointer at end of line or not
2	adjust destination pointer at end of line or not
2	scan: left-to-right, right-to-left
3	rotation: left, right, no shift
3	left edge: full, partial word, partial long
3	right edge: full, partial word, partial long
3	inner loop: small, big and odd, big and even
1944	*lower bound on cases*

Table 1. Cases in generated code for the inner loop.

The number of cases depends on the rules for counting. Table 1 gives a lower bound on the essentially distinct cases. `OR` and `XOR` are certainly different operations and result in different instructions, but they behave the same way as far as compiling code is concerned. CLR is different because it requires two instructions (a not and an and) instead of one; STORE is different because a zero bit is not an identity for partial-word operations. The magnitude of a rotate ranges from 1 to 8 positions and is an immediate constant in the rotate instructions, but the value is otherwise immaterial. Whether the rotation is to the right or to the left is important, however. The table does not count some tricky cases where the inner loop is small

(0, 1, or 2 words) and interacts intimately with the edge cases.

The size of the generated code ranges from 16 to 72 bytes. If all the cases were written out (including all the shifts and Codes) then the total size would approach one megabyte. Thus it is impractical to expand all the cases ahead of time and merely jump to the right one; the generation process is useful because it saves space. The generated code resides in a local array on the stack.

It is reasonable to ask if some small subset of the generated cases might dominate execution, and if table lookup might therefore suffice. Character drawing is the most common case, but since it is handled by static code, it doesn't enter into this calculation. Of the cases generated on the fly, aligned STOREs and XORs executed by the window and menu software are the most common, but they are compiled quickly because they are aligned, so the compilation overhead is minor. The remaining cases arise infrequently in the standard software, but may be the performance bottleneck of an application program there is no way to predict what arbitrary graphics programs will do. The care taken to generate optimal code is therefore worthwhile. Finally, Version 5 is actually the smallest implementation, because it does not contain explicit loops for dozens of cases.

Figure 1 gives the generated code for scrolling the whole display. The display is 800/16 = 50 words wide. This case requires no shifting for alignment; each instruction can process 2 words at a time. The left and right edges coincide with word boundaries, so partial-word operations are not necessary. Autoincrement processes each scan line left-to-right, and the side effects leave the pointers ready for the next scan line without further adjustment. The inner loop is unrolled to 2X, and control enters at the middle since the operation count of 50/2 = 25 is odd. Except for the odd inner loop, this code is the shortest and fastest of all the generated cases.

```
        br.b L20              initial entry for odd inner loop
L10:    mov.l %a2@+,%a0@+     32 bits moved here
L20:    mov.l %a2@+,%a0@+     and 32 more here
        dbr %d6,L10           until scan line finished
        mov.w %a4,%d6         reload inner counter with 12
        dbr %d7,L20           until no more scan lines
        jmp %a5@              return to fixed control
```

Figure 1. Generated code for scrolling the whole display.

Figure 2 shows the relative alignment of source and destination words for a complicated case. Figure 3 gives the code generated for XOR with this alignment and an overlap requiring right-to-left scan. The code processes each scan line right-to-left using autodecrement, compensates for a three-bit difference between the source and destination in the location within a word of the edges, handles a partial word at the right edge and a partial long word at the left edge with appropriate masks, uses additive correction constants to move from one scan line to the next, and falls through into an even inner loop.

Version 5 of bitblt is written in assembler for efficiency. The overhead of starting bitblt dominates for small areas. By writing in assembler, the on-the-fly compile time can be minimized, primarily because all the variables required for the compilation can be held in registers, avoiding memory fetches for data.

The generator itself consists of three sections. The first section analyzes the rectangles, sets up masks and the rotate instruction, and determines which special cases apply. The last section lays down instructions to process one word or long word. The middle section supervises generation. It calls the last section four times, using flags and parameters to distinguish the first word, two inner words, and last word.

Version 5 performs slightly better than the coroutine-based Version 4; everything, including cases requiring no shift, is some 20% faster than Version 3, the best of the C implementations. The structure of the generator makes it fairly easy to add new function Codes or to extend the definition by masking the data with a texture; the versions in C would need more complicated analysis and more inner loops written in-line.

bitblt recovers the time spent compiling by amortizing the cost of generation over the left and right edges of the rectangles. On the average, one edge can use 32-bit operations (the left edge in Figures 2 and 3), saving 5.5 instructions per scan line. Destination edges coinciding with word boundaries omit the and.

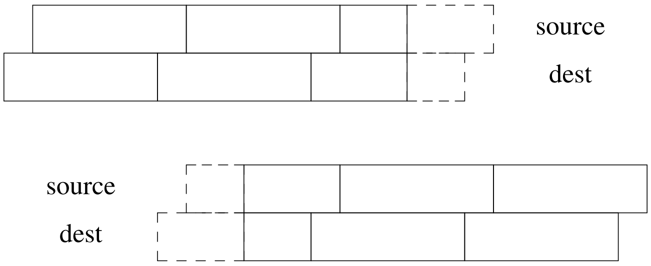

Figure 2. Right edge and left edge for a complicated case.

```
L40:                          top of outer loop; partial word at right edge
      mov.l %a2@-,%d0               first fetch always 2 words
      mov.l %d0,%d1                 prime the pump for inner loop
      ror.l &3,%d0                  rotate right 3 bits
      and.w %d4,%d0                 mask might be 0xffc0
      eor.w %d0,%a0@-               exclusive-or with destination
L50:                          top of inner loop
      mov.w %a2@-,%d1               fetch next source word
      swap.w %d1                    proper order with previous word
      mov.l %d1,%d0                 prepare for other side of loop
      ror.l &3,%d1                  align
      eor.w %d1,%a0@-               exclusive-or with destination
                              second half of inner loop
      mov.w %a2@-,%d0               fetch next source word
      swap.w %d0                    proper order with previous word
      mov.l %d0,%d1                 prepare for other side of loop
      ror.l &3,%d0                  align
      eor.w %d0,%a0@-               exclusive-or with destination
      dbr %d6,L50                   until scan line finished
                              partial long word at left edge
      mov.w %a2@-,%d1               fetch next source word
      swap.w %d1                    proper order with previous word
      ror.l &3,%d1                  align
      and.l %d5,%d1                 mask might be 0x007fffff
      eor.l %d1,%a0@-               exclusive-or with destination
                              move to next scan line
      add.w %a3,%a2                 adjust source
      add.w %a1,%a0                 adjust destination
      mov.w %a4,%d6                 initialize inner count
      dbr %d7,L40                   until no more scan lines
      jmp %a5@                      return to fixed control
```

Figure 3. Code for a complicated XOR.

The generator similarly elides add instructions when possible. These savings completely recover the cost of generation after 70 scan lines, and after that the generated code starts winning. Narrow rectangles and ones requiring no shift (by construction, the most common case) start winning sooner because generation is shorter.

The code STORE requires special attention because the MC68000 has no partial-word bit-field operations for handling fragmentation at the left and right edges of a bitmap. Standard computation uses AND and OR in the formula `d=(s&m)|(d&~m)` for combining source and destination under mask. The formula `d^=((s^d)&m)` uses XOR and AND to compute the same answer in less time and fewer registers (depending on the instruction set).

Simplicity of design leads to efficiency of execution. In this case, 'design' is in hardware and 'execution' is in software. The complete bitblt subroutine is 1524 bytes of code, and can scroll the screen one pixel vertically in 0.13 seconds, and horizontally in 0.38 seconds. These are appreciable times, but the vertical case is near the bandwidth limit of the memory, the horizontal case is essentially never executed, and most bitblt calls are at most the size of a window, not the entire display, and so are proportionately faster. The implementation of bitblt on the Blit is entirely satisfactory from a user's viewpoint, most calls to bitblt are instantaneous.

Hardware II: Other systems

The Blit is the simplest bitmap display in existence, and it is interesting to compare its structure and performance with other graphics systems.

There were a few early bitmap displays, such as the Yale Gem system[11] and the Datadisk at Stanford, but the first modern bitmap display is that on the Xerox Alto.[19] Like the Blit, the Alto has a uniform memory structure, and the main processor executes bitblt, but in microcode. (The Alto's memory is single-ported; the screen refresh is also in microcode!) Because it is interpreted at a lower level, Alto bitblt is comparable in speed to that on the Blit, although more traditional computation is slower. The Alto has a separate charblt-like instruction called CONVRT[9] that draws characters only. CONVRT is about 30% faster than bitblt for the same parameters, which is approximately the difference in speed between bitblt and charblt (now defunct) on the Blit, but we prefer to have a single primitive.

One interesting property of the Alto is that the display memory is mapped by an array of registers each of which specifies the physical addresses for a piece of the screen of size between one scan line and the whole display. The minimum unit they address is one scan line. Although a couple of programs, such as the text editor, made use of these registers to save memory and speed up scrolling, once bitblt was invented (the first bitblt was on the Alto) most programs used its model of display management.

Several current personal computers, such as the Three Rivers PERQ[20] and the Xerox Dorado,[10] are structured like larger, faster Altos. These machines use higher memory bandwidth, memory 64 bits wide instead of 16, faster processors and some low-level architectural support such as hardware barrel shifters to speed up the bitblt microcode by a factor of ten or more. Despite these complications, the display architecture of these machines is still that of the Blit or the Alto without display registers.

The hardware in these large systems is expensive, so it is reasonable to ask if adding a relatively small amount of hardware graphics support can accelerate bitblt without increasing the cost unduly. The implementation of bitblt on the Blit provides some clues. For small pictures, such as characters, the overhead of starting bitblt dominates the performance. But, the complexity of bitblt is due more to the complexity of the operator itself than to difficulties associated with architectural peculiarities of the display or microprocessor. Most of the code is concerned with case analysis. Any hardware support must therefore complicate the case analysis as little as possible or bitblt, already unwieldy, may become unwritable. This suggests that the memory structure should be kept uniform, and as will be explained below, uniformity can also help performance.

One hardware feature that could help bitblt is bit-addressable memory, because it removes the distinction between the cases where the source and destination are or are not word-aligned. It still requires special adjustment at the end of each scan line, however, so bitblt is not made trivial. A simpler possibility is to provide a hardware barrel-shifter that allows the source data from anywhere in memory to be loaded in a register already word-aligned with the destination. This was tried in an early design of the Blit, but was rejected because, as noted above, un-aligned bitblts are rare except when drawing characters, and characters are dominated by overhead. We estimated the barrel shifter would speed up character drawing at most 30%, which was too small an effect to warrant the extra hardware complication and expense. Our feeling was that hardware support should assist all bitblt calls uniformly.

For large areas, bitblt on the Blit is within a factor of two of what the memory can support, the missing cycles being consumed by instruction fetches. Therefore, the forms of hardware assistance that can help the most involve raw speed: speeding up the memory or (to a lesser extent) the processor. The dual-ported structure of the memory costs some bandwidth, and this can be partially won back by adding a processor-only cache or prefetch buffer so processor memory requests are blocked less often by video refresh cycles. An elegant way to reduce the overhead of dual-ported memory is used on the Ridge computer,[16] which is a virtual-memory machine. Memory pages used for the display (which are ordinary memory from the processor's viewpoint) are backed up in a shadow memory local to the display. The 'page modified' bit in the virtual memory hardware is used to notify a clock-driven routine to copy data from main memory to the display. Basically, the shadow memory is used like a paging device for the display memory, so memory cycles are lost for video update only when the picture is modified. Also, because the pages are 2 Kbytes, while the display is 100 Kbytes, for typical character-based graphics only a small portion of the display memory must be copied to the display.

Some bitmap display designs change the format or access method of display memory in the interest of performance. The most intriguing of these designs is the 8-by-8 display,[17] which gives display memory a granular two-dimensionality. Only the display memory is two-dimensional, however, so the functionality provided by the microcode in the bitmap processor attached to the display must be duplicated on the main computer to make use of off-screen bitmaps; every non-uniformity in the address space results in more bitblt code. Also, although the 8-by-8 display can draw characters very fast, optimum speed is attained only if clipping is ignored. The 8-by-8 display has another problem, common among frame buffer designs: the display memory is globally two-dimensional, making allocation of invisible portions of the display memory difficult to manage. Access to the invisible memory would be fast because of the bitmap processor, but in practice only the display bitmap can be conveniently used, so off-screen bitmaps suffer a performance penalty. Overall, the 8-by-8 display is a fascinating idea, but the theoretical advantages of locally two-dimensional memory have not been realized in practice: the performance is not as high as the hardware investment should provide, and the non-uniformity of the address space complicates the software and influences it in meddlesome ways.

Another two-dimensional bitmap display is that on the SUN[2,3] personal computer. There have actually been two implementations, the SUN 1 and the SUN 2. Both have a separate display memory with hardware assistance for shifting, Boolean operations and masking. The SUN 1 uses a 1024×1024 two-dimensional memory on a Multibus card, and accesses the 1024×768 display through a 16-bit interface. Through the use of display registers and the high, unused address bits of the MC68000, a 16-bit word in the display memory can be picked up, shifted, masked with a 16-bit word and written to an arbitrary bit address in the display memory by executing a single word copy instruction:

```
mov.w a1@+, a2@+
```

There are several problems with this technique. The memory of the display bitmap cannot be accessed directly by the CPU, but must instead be reached through the 16-bit interface on the Multibus. This means that twice as many instructions must be executed as the MC68000 might optimally require, because the CPU can move 32 bits in an instruction. Also, the two-dimensional nature of the display memory makes it awkward to use the invisible portion. Most important, though, is that a general bitblt implementation must deal with the four separate cases of display to display, display to memory, memory to display and memory to memory, each of which is very different code. This makes bitblt unnecessarily complex.

The Sun 2 addresses these problems by providing the same functionality (this time in a custom LSI chip), but with the display memory on the same card as the CPU and directly in its address space. This allows the CPU to treat the display as regular memory when that is convenient, or to use the hardware assistance when advantageous. Bitblt is therefore reduced to two main cases display to display and all else which are very different but manageable. To reduce the memory bandwidth required for refreshing the display (114MHz for the 70Hz refreshed 1024×1280 display), the tube is refreshed from a shadow memory that watches write access from the CPU to the display, and copies the data from all writes to memory into the shadow memory.

Table 2 gives the time for some display operations on the two SUNs and on a Blit. All measurements are elapsed time on existing configurations of hardware and software; the SUN 2 timings are given for both display-to-display copy and memory-to-memory, since these are actually distinct versions bitblt. The SUN timings are from Tom Duff and Peter Langston.[6] The characters were drawn in a 9×14 pixel font, but the bounding box for the letter 'a' is only 8×7. Both systems used bitblt to draw the characters, rather than special purpose primitives, and executed clipping code.

The SUN display hardware is designed to make the general, non word-aligned case of bitblt as fast as

operation	SUN 1	SUN 2 (disp.)	SUN 2 (mem.)	Blit
scroll screen vertically	187	109	82.2	129
scroll screen horizontally	194	110	311	376
letter ‘a’ at random positions on screen	1.1	0.34	0.74	0.42
texturing a random 40x40 square	1.89	0.82	1.78	1.60

Table 2. SUN and Blit timings in milliseconds.

possible, but as discussed above, this general case is executed rarely in practice. The 16 bit window on the SUN 1 demands twice as many instructions to be executed to move the data as on the Blit, because the MC68000 can move 32 bits per instruction.

The worst problem with the SUN 1, however, is implementation, not performance. Because the display memory is peculiar, bitblt must execute 4 different inner loops for the various combinations of source and destination in main memory or the display. This extra case analysis makes bitblt intractable; we know of no complete bitblt implementation for the SUN 1. The version of Smalltalk on the SUN 1 uses the display much as does the Ridge, maintaining the screen image in main memory, so all bitmaps have the same structure, and copying data to the display using a special simple primitive when the image changes.[5] (To be fair, it should be pointed out that the SUN 1 is very good at line drawing, in fact, better than the SUN 2.)

The SUN 2 was designed so that for cases where bitblt hardware is not profitable such as for vertical scrolling the hardware can be disregarded and a purely software implementation can get good performance. The implementation is also manageable, because there are only two major cases: display-to-display and everything else, which can be implemented independently. The SUN 2 hardware is, however, considerably more complex and expensive than either the SUN 1 or the Blit.

Another display that stops short of providing complete bitblt in hardware is the Apollo DN400, Apollo’s first computer.[1] The display memory is separate, although accessible directly from the CPU, and has a hardware ‘bit mover’ that executes bitblt directly, and very quickly, for a subset of the possible cases. The main restrictions of the bit mover are that it is STORE mode only, and only works on the display memory. Since providing the different Codes is only a small part of bitblt, the software on the Apollo is not made simpler by the hardware support. Instead, because of peculiarities of the display memory, the software bitblt is significantly slower than on the Blit, while operations through the bit mover are significantly faster. There is roughly a factor of ten performance difference between otherwise identical STORE and XOR operations; although display-to-display STORE mode bitblts are faster than the Blit, every other case is slower.

The lack of generality of the bit mover on the DN400 has been corrected in more recent Apollo systems, which have bit movers that support all Boolean operations and are not restricted to the display memory.

In summary, the crucial observation is that *half-hearted hardware support for bitmap graphics does not speed up the cases that matter.* Only a fast processor and memory can make a significant difference in character-style graphics, which is the most common application of bitmaps. Clumsy hardware can speed up the general case of bitblt, which is rarely executed, while slowing down the common special cases by interfering with the relationship between the processor and memory. As well, bitblt can become enormously difficult to write. There are only two viable structures for hardware to support bitblt-style bitmap graphics: either provide complete bitblt in hardware or microcode (such as on the PERQ or Dorado), or provide a clean, simple memory structure and let software do the rest (such as on the Blit or Macintosh). (The SUN 2 is essentially the sum of these.)

Texture

The performance of the texture operator is limited by the speed of the MC68000 instruction

```
mov.l   %d0, %a0@+
```

derived from the C statement

```
*p++ = x;
```

where p and x are both registers. The inner loop of texture can be unrolled to this, or some trivial variant for the other Codes. If p points to 32-bit words, memory bandwidth is the crucial factor, and only increasing the bandwidth can improve the performance of texture. Although our texture is written in assembly language, it replaces a slightly slower and larger version written in about two pages of C. texture is not a difficult operator to program. Nonetheless, some hardware designers try to help it by providing some form of texture register in the display memory interface. Since each location of the rectangle must be touched by the software, and the texture can be stored in a processor register (with trivial adjustment between scan lines), a hardware texture register offers no advantages. If texturing is provided in bitblt, the additional complexity of doing the job entirely in software is insignificant. Again, if the hardware does not execute the complete bitblt instruction, it should do nothing.

Lines

Because we draw lines on the display relatively rarely most of our graphics involves manipulating characters the line-drawing software has not had as much attention devoted to its performance as have the bitmap operators. Nothing precludes this being done, of course; a special-purpose compiler like that for bitblt could make line-drawing significantly faster, but we don't feel it is worth the effort for our typical use of the display. Our current software draws sloping lines at about 50,000 pixels per second, and that is adequate.

Most lines drawn on Blits are horizontal or vertical. The line code recognizes these cases and runs special code much like that in texture to draw almost a million pixels per second horizontally (which is still suboptimal).

Mouse

Graphics input on the Blit is provided by a three-button mouse that is integrated into the hardware and software. The software that tracks the mouse asynchronously copies a 16×16 image to the pixel pointed to by the mouse as it moves.

User-level software to access the mouse uses polling loops, which is acceptable on the Blit but would be troublesome in more traditional computing environments. It would probably be cleaner and faster to rewrite the mouse software to be event driven, so software could wait for buttons to be depressed or released or for the mouse to enter some rectangular region on the display.

One of the commonest uses of the mouse in the software is making menu selections. A single subroutine, menuhit, takes as arguments an array of character strings to be displayed and an indication of which button is depressed (the routine is always called after the button has been pressed). menuhit displays the menu on the screen, accepts a selection from the user, restores the display contents and returns the user's selection. The menus 'pop up' where the mouse is pointing, and the user makes a selection by lifting the button with the cursor over the menu entry desired, or outside the menu if no selection is wanted. This is much like menus in Smalltalk.[18] The most interesting thing about menus on the Blit is that the standard software provides only one kind of menu; although some fancier menus have been developed for special purposes, the simple style of menu has proved convenient and almost always adequate.

The mouse itself[4] generates x and y quadrature information and the status of the three buttons on parallel unencoded wires. The mouse interface hardware handles the position and buttons separately. The connection is direct to the Blit circuit board, and not (for example) through a UART.

For each axis of motion, two square wave signals are produced, out of phase by 90 degrees, providing 15 transitions per millimeter. At maximum hand speed, the transitions occur about once per millisecond. The obvious interface to this hardware is to generate an interrupt for each 'click,' but the original terminal had a noisy tablet, and we were worried about continuous interrupts. When the terminal was redesigned for mice, we kept a tablet-like interface to the processor, which required hardware to count the clicks from the mouse. Were we to redesign now, we would probably change to the interrupt-driven interface, and let the software do the counting.

Each axis has a PAL (a fuse-programmable logic array) that decodes the quadrature information, taking care to not slip when the mouse changes direction, and computes the bottom 4 bits of the position. The high 8 bits are then computed by a counter, and the 12-bit result presented to the processor in a read-only register. The mouse coordinates are therefore asynchronously updated variables in the program's address space. The high bit of each register indicates whether current data is valid, because during updates the registers may contain garbage. The PAL counts 1 for each element of the quadrature, so the cursor on the display moves 15 pixels per millimeter of mouse motion. The entire display can be reached by moving the mouse about 50mm horizontally and 70mm vertically. This is significantly more sensitive than most input devices, but having become accustomed to the high-resolution mouse, we find the hand motion required by lower-resolution systems awkward.

The software reads the mouse position registers during the vertical retrace of the video monitor (the hardware generates a 60Hz interrupt at the start of vertical retrace), so the cursor can be moved while the beam is not sweeping through the memory occupied by the cursor. This prevents annoying flicker, and the 60Hz update of position is adequate for the terminal's applications. When the mouse is moving, this consumes about 3 percent of the processor, and the load is much less when the mouse is stationary, which is vastly more common.

60Hz is, however, not fast enough for the buttons. Quick clicks of a button could be lost if they fell between polls. Therefore, the buttons generate an interrupt when their state changes, so the program is instantly notified of any button activity.

Although the hardware provides absolute coordinates, the software interprets them relative to a moving origin, so, for example, the mouse does not wrap around when it reaches the edge of the display. Unlike clipping, this method also guarantees that the mouse responds instantly to motion in any free direction, rather than sticking to the edges of the display, and makes it easy to move the cursor under software control. Because of this algorithm, it would be possible to use smaller hardware registers (and therefore fewer chips) to hold the mouse coordinates. The registers only need to be about twice the size of the maximum increment possible in a sixtieth of a second, or about 5 bits. 8-bit registers would be adequate.

Some graphics systems provide circuitry to place the mouse cursor in the video signal instead of the memory, to avoid interference with graphics operations, while others provide no hardware at all. The Blit lies in the middle: the mouse is integrated into the hardware, but software must cope with the interference problem. Because there are only a few graphical operators bitblt, texture, segment and point it is not prohibitively expensive to check the arguments to the operators for potential interference with the mouse cursor. If the operation is not XOR, which would commute with the XOR bitblt that moves the mouse cursor on the screen, the cursor is removed from the display. The cursor is not redrawn immediately, though: it is instead marked ''uninhibited,'' so the next clock interrupt will redraw the cursor. Although this makes the cursor flicker during heavy non-`XOR` activity, it is inexpensive (the cursor is only drawn at 60Hz maximum), and not bothersome, because the majority of calls to bitblt and texture are in XOR mode, to draw characters and highlight portions of the display.

In summary, the mouse hardware is very simple (although it could be simpler), but software can cope easily with the problems of mouse tracking and interference with displayed graphics. Although hardware cursor tracking would certainly simplify the software, as it stands there are only a few dozen lines of code to deal with the mouse. Simplification seems unneccesary, especially if (as with the Blit) hardware cost must be kept down.

Conclusions

Perhaps the most important lesson from the Blit is that *hardware and software should be designed together*, by people that are going to use it. The Blit's designers understood the issues and were able to place functionality where it would be most effective. Hardware designers unaware of the software architecture of a bitmap graphics system might, with good intentions, add special hardware that would interfere with the software. Instead, the Blit hardware was designed to make the software efficient and easy to write.

For bitmap graphics, a simple, uniform memory and processor architecture can result in higher performance than a system with special purpose graphics hardware, even if the hardware is fairly simple. The bitblt style of graphics is easier to support in a more regular environment; non-uniformities complicate the implementation of the graphics primitives. Predictions about the intended use of the display can guide architectural decisions; it is easy to provide hardware assistance that accelerates general purpose bitblt at the expense of the commonly executed cases. *Carefully designed software can outperform infelicitous hardware*.

Sometimes, of course, a designer is faced with demands for utmost speed for a particular application. This may require direct hardware support, but such support should be provided in a way that enables efficient, general software to be written for the device. One way to do this is to make the hardware transparent, so it can be bypassed when the software wants direct access to the display. Forcing the software to use the special hardware may impede other applications.

Acknowledgements

Most of the basic ideas about bitmap graphics were developed at Xerox, and no discussion of the subject would be complete without crediting the contributions of the Alto architects B. Lampson, E. McCreight, and C. Thacker, and the inventor of bitblt, D. Ingalls. This paper was improved by thoughtful comments made by Jon Bentley, Tom Cargill, Gerard Holzmann, Brian Kernighan and Doug McIlroy.

References

1. Apollo Computer, Apollo Manual, Chelmsford, Massachusetts, 1981.
2. Bechtolsheim, A., and Baskett, F., 'High-Performance Raster Graphics for Microcomputer Systems', Computer Graphics **14**, 3 (July 1980), 43-47.
3. Bechtolsheim, A., et al., 'The SUN Workstation Architecture', Computer Systems Laboratory, Stanford University, Jan. 15, 1982.
4. Depraz Mouse P-4 description, Le Lieu, Switzerland, April 1982
5. Peter Deutsch, personal communication.
6. Tom Duff and Peter Langston, Lucasfilm Ltd., personal communication.
7. Guibas, L. J. and J. Stolfi, 'A Language for Bitmap Manipulation', Transactions on Graphics **1**, 191, July 1982.
8. Ingalls, D. H. H., 'The Smalltalk Graphics Kernel', Byte **6** 168, August 1981.
9. Dan Ingalls, personal communication.
10. Lampson, B., and Pier, K, 'A Processor for a High-Performance Personal Computer', Proc. 7th Symp. Comp. Arch., SigArch/IEEE, La Baule, 146, May 1980.
11. Levine, J., 'An Overview of the Yale Gem System', Software Practice and Experience **12**, 1133 (1982).
12. *MC68000 16-bit Microprocessor User's Manual*, third edition, Prentice-Hall, Englewood Cliffs, NJ, 1982.
13. Newman, W. M., and R. F. Sproull, *Principles of Interactive Computer Graphics, 2nd Ed.,* McGraw Hill, New York, 1979
14. Pike, R., 'Graphics in Overlapping Bitmap Layers', ACM Trans. on Graph. **2**, 135, April 1983.
15. Pike, R., 'The Blit: A Multiplexed Graphics Terminal', Bell Labs Tech. J., to appear, 1984.
16. Ridge Computer, Ridge manual, Sunnyvale, CA, 1983.
17. Sproull, R. F. et al., 'The 8 by 8 Display', ACM Trans. on Graph. **2**, 32, January 1983.
18. Tesler, L., 'The Smalltalk Environment', Byte **6**, 90, August 1981.
19. Thacker, C. P. et al., 'Alto: A Personal Computer', in D. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Readings and Examples*, second edition, McGraw Hill, New York, 1980.
20. Three Rivers Computer Corporation, PERQ Manual, Pittsburgh 1980.