AT&T
Bell Laboratories

Subject: Bitmap Graphics SIGGRAPH'84 Course Notes
Work Program- 311403-0101 -- File- 39199-11

date: *May 21, 1984*

from: Rob Pike
Leo Guibas*
Dan Ingalls*

TM: *11271-840521-05TMS*

TECHNICAL MEMORANDUM

PREFACE

When the bitmap display on the Alto, Xerox's personal computer, was first being used, its programmers wrote a number of subroutines for special purpose tasks such as character drawing, highlighting, and copying rectangles. These subroutines all contained similar code to deal with problems such as bitfield insertion and rectangles with edges within words. Dealing with all these problems in a single, general operator looked forbidding, but the time spent reinventing the inner loops was becoming frustrating, so in 1975 Dan Ingalls and Diana Merry at Xerox PARC encapsulated the operation of copying a bit string from one location to another in a primitive they called *bitblt* for *bit*-boundary *blo*ck *t*ransfer. The first bitblt operated on a single scan line, but the outer loop was later added, making bitblt a rectangle operator. As bitblt was experimented with, it proved to be so useful that it is now the central graphics primitive on a number of bitmap displays.

These notes explain why bitblt is so successful. They are an overview of our understanding of bitmap graphics based on bitblt. They address the basic properties of bitmap displays, bitblt itself, and associated operators such as line-drawing primitives. Because the pixels on bitmap displays are usually represented by a single bit, Boolean algebra applies to the pixels, and the rectangular operators form a simple algebra. Using this algebra, the primitives may be composed to build algorithms for rotation, magnification, area filling and other traditional graphics applications.

Bitmap displays demand large amounts of memory and processing power. It is important that bitblt be implemented efficiently, since inefficiencies can result in (literally) visible degradations of performance, even for a single invocation. Unfortunately, though, implementing bitblt efficiently is a difficult problem. Later, we will present a complete, correct, very slow implementation of bitblt, but one simple and small enough to be understood easily. The following sections discuss techniques for improving its performance, and illustrate these by showing how they have been used, and how well they worked, in actual systems.

Next, we discuss how bitblt can be used for programming interactive graphics applications. Structured picture elements such as text, menus and windows are easily implemented using bitblt, but using them well requires some understanding of how bitblt itself behaves. Applications such as bitmap paint programs also depend critically on the semantics of bitblt.

The most important consideration through all these discussions is the integrated viewpoint that bitblt provides. The details of hardware and software implementation focus on a single operator that provides a rational, powerful model for raster graphics. Bitmap devices are popular because their style of graphics is convenient and flexible, but it is bitblt that makes that style manageable. By simultaneously addressing the issues of efficiency, representation and access, bitblt makes it possible to ignore the low-level detail inherent in bitmap displays, and attend to the more important and useful task of building an interactive graphics environment. Representative displays from some of the systems that have been built using bitblt are shown in Figure 1.

The reason we have assembled these notes is that, despite its importance, little has been written about bitblt in the literature, to the point that hardware manufacturers who are not 'in the know' make serious mistakes in the implementation of their systems. Until now, too much information about bitblt has been available only as folklore. By discussing the algorithmic, implementation and systems-level basics and implications of bitblt, we hope to enlarge the community of bitblt-knowledgeable people, and prevent the development of bitblt-antagonistic hardware.

---

*Xerox PARC, Palo Alto, California

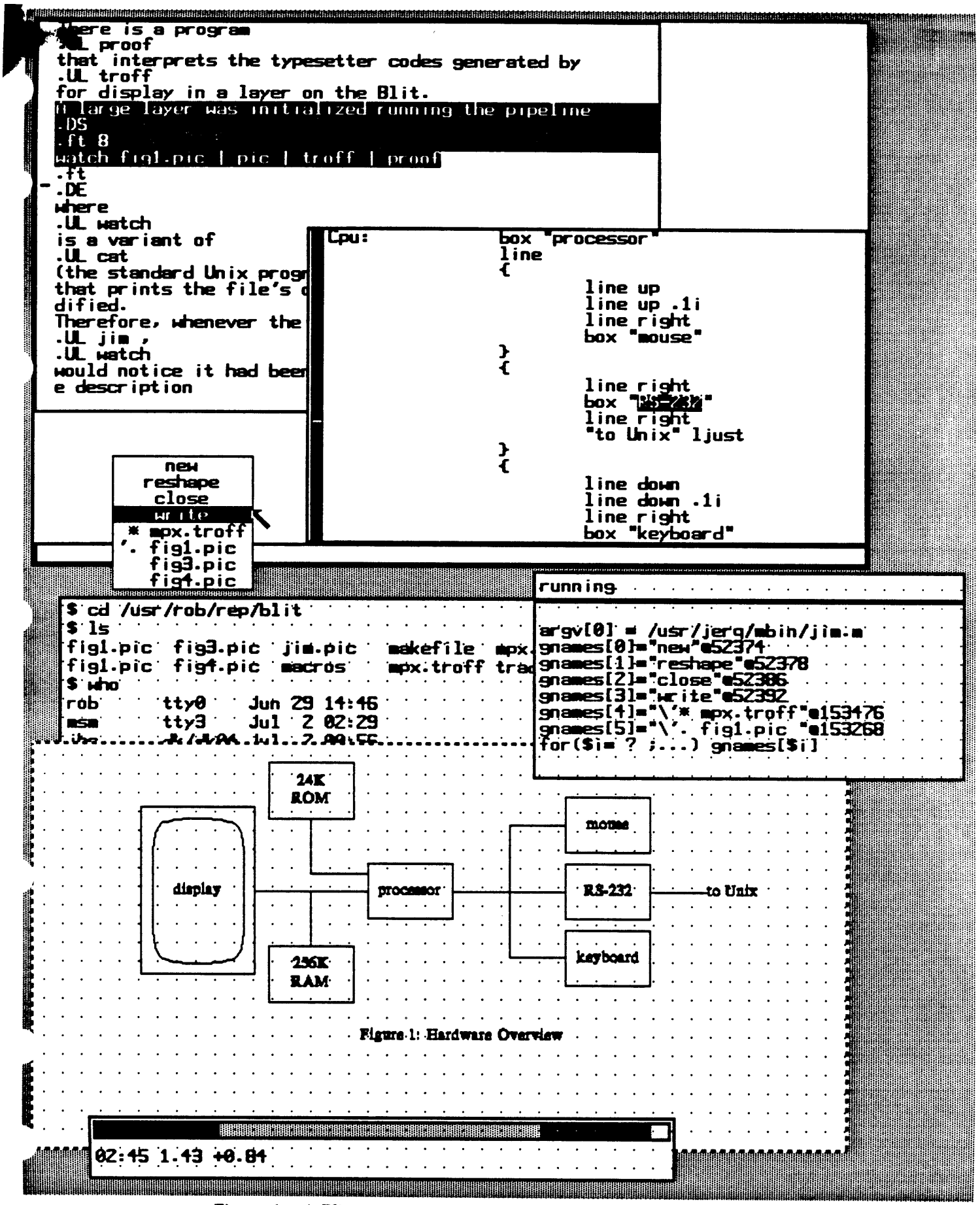

Figure 1: Hardware Overview

Figure 1a. A Blit screen

RASTER GRAPHICS †

The earliest computer displays, developed from the oscilloscope industry, plotted individual points, and drew all higher-level objects as sequences of dots. Later generations of displays were controlled by a *display list*: a list of instructions to the display, typically encoding the (x, y) coordinates of each point and its intensity. Subroutines in the display list allowed more complicated pictures — even beyond the capability of the display to draw quickly — to be displayed and dynamically changed. An obvious improvement was to interpret descriptions of lines and other simple curves directly by the hardware, so the display list was more compact and less expensive to compute. From this background has grown the *vector graphics* or *display segment* model of computer graphics, described in Newman and Sproull[19] and currently represented by the GKS standard.[1]

The *frame buffer* model, also called *raster graphics*, is rooted in the television industry, and instead represents an image as a two-dimensional array of intensities mapped onto a television tube to create the image. Frame buffers have one great advantage: the complexity of the displayed image does not affect the amount of memory consumed to display it. They have disadvantages, too, of course. The most obvious is economic: to store a reasonably complex picture might take 8 bits per pixel on a 512×512 display, which requires a quarter megabyte of relatively expensive high-speed memory. Also, the processing required for dynamic graphics on a frame buffer demands a dedicated CPU for reasonable performance, again because of the large amount of memory that must be updated. In time, though, the cost of TV tubes and memories has dropped, so a frame buffer is now less expensive to make than a vector display, and personal computers with frame buffers are becoming common.

During the early 1970's, researchers at Xerox PARC built a small personal computer called the Alto, and gave it a frame buffer with only a single bit per pixel — a *bitmap* display. Although binary pixels are clearly incapable of high-quality graphics, at least at typical frame buffer resolutions, the Alto's frame buffer was intended to simulate paper, for which only two values are required (print and background, or black and white). Unfortunately, the simple segmented graphics model that works well on vector displays is clumsy on bitmap displays. Perhaps surprisingly, the graphics model that has arisen to take the place of segments on bitmap displays capitalizes upon the lowest level of representation of single-bit-per-pixel images, rather than hiding it.

Traditionally, bitmaps have been viewed as imperfect approximations to the ideal images one would like to display. Such ideal images are assumed to be described by continuous variation in color, intensity and so forth. The imperfection is brought about by the discrete nature of the imaging devices we possess, as well as the digital nature of the computers themselves, which forces us to quantize these continuous variables into a discrete (but possibly very large) set of values. In particular, bitmap coordinates are integral, as are the pixel values within them. From this 'imperfect' viewpoint, bitmaps are essentially an implementation artifact, and the graphics programmer should not be exposed to them. Instead, the programmer should be given access to the ideal shapes of plane geometry, plus continuous functions for describing color and intensity modulation. The operators available in such a package always correspond to operations on these ideal objects. It is the responsibility of the package's implementor to discretize all these continuous functions internally and transform the ideal continuous operators into their discrete counterparts.

For a variety of reasons, though, no such package can hide fully the discrete representation involved beneath the surface. Perhaps the most fundamental reason is that certain laws that hold in the continuous domain cannot be made to hold in the discrete domain, no matter how careful one is about quantization. (We will have more to say about this shortly.) Also, good quantized approximations to continuous tone images naturally involve very large bitmaps. Such

† This section includes material from Guibas and Stolfi,[9] © ACM by permission of the Association for Computing Machinery.

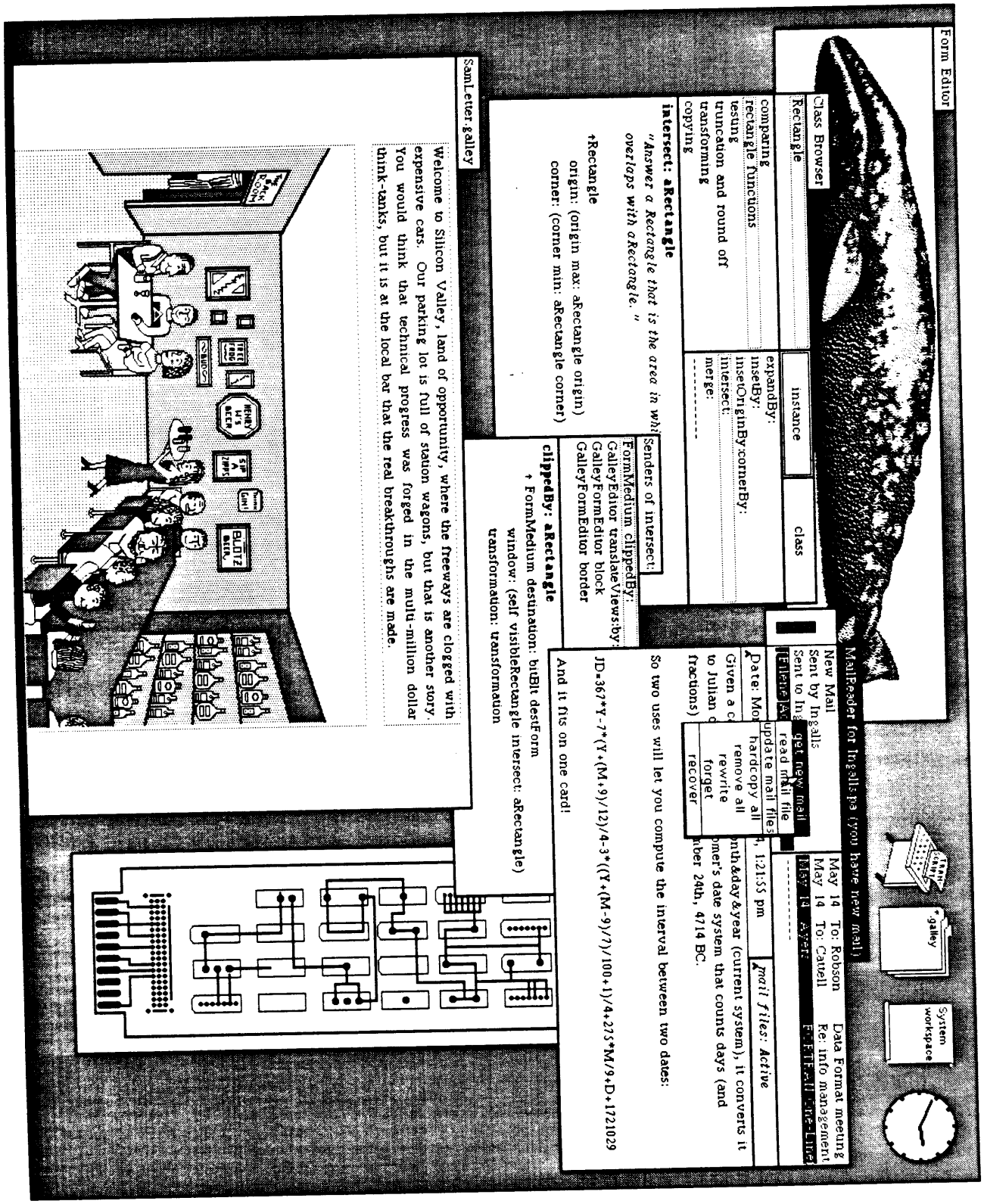

Figure 1b. A Smalltalk screen. (Cartoon courtesy Frank Zdybel.)

large bitmaps are expensive to store and manipulate. Thus efficiency considerations often force the graphics programmer to be aware of the implementation underneath.

As anyone who has taken a course in numerical methods knows, floating-point numbers do not satisfy many of the identities that hold true for real numbers (in the mathematical sense of 'real'). The science of numerical computing is largely devoted to compensating for these fundamental imperfections in floating-point arithmetic. Similarly, when images are discretized, certain errors are unavoidable. For example, consider the picket fence example shown in Figure 2. Suppose each picket is 1 pixel wide, but the pickets are spaced 2.5 pixels apart. One reasonable quantization method would rasterize the picket spacing to an alternating thickness of 2 and 3 pixels respectively, so as to make the overall length of the fence as close to its real value as possible, thereby maintaining the criterion of *global faithfulness*. Another method would simply choose either 2 or 3 pixels and make all spacings that thickness, maintaining *local faithfulness*. It is impossible to satisfy both criteria simultaneously.

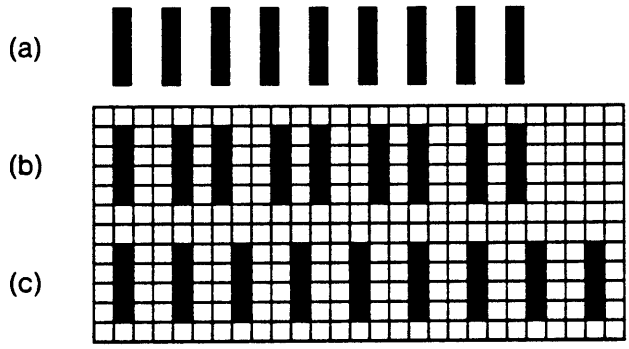

Figure 2. A picket fence image (a) and two discretized versions of it: (b) globally faithful, and (c) locally faithful.

Another reason bitmaps seep through to higher levels of system design is efficiency. Dynamic raster graphics require very high-speed manipulations of the raster memory, where the image being displayed is stored. In every case, the contents of the raster memory can be computed by scan conversion algorithms from higher level shape, illumination and color descriptions. Such algorithms are rarely, however, fast enough to cope with incremental updating of the display, as is often required in interactive applications. Although, as we remarked earlier, ideal image manipulations do not always have exact counterparts in discrete form, they sometimes do. For instance, the operation of copying or translating a subimage on the screen has an exact counterpart in the discretized form. There are immense speed advantages to be gained by doing these manipulations in the raster representations, rather than in the ideal ones and then repeating the scan conversion.

Any attempt to ban bitmaps from anything other than a temporary low-level representation for computer images is bound to encounter difficulties.

To facilitate raster manipulations, several novel computer systems have used specialized instructions dealing with the raster memory. Typically these systems have been personal computers, such as the Xerox Alto[25] or the M.I.T. Lisp Machine,[26] where a premium is placed on interactive graphics facilities. Their raster instructions are powerful primitives, often implemented in a combination of hardware and micro-code. Such an instruction is known as *bitblt* (bit boundary block transfer), or *RasterOp* in Newman and Sproull's terminology.[19]

The most common form of this instruction is a bitwise Boolean operation between two conformable (i.e., having the same dimensions), possibly overlapping, rectangles of pixels. If we call one rectangle S for *source* and the other D for *destination*, then bitblt performs the operation $d \leftarrow d * s$ for corresponding bits d and s in D and S. Here $*$ denotes some two-argument Boolean operation, which is a parameter to bitblt. Such an instruction can obviously be used to move rectangles of bits around the screen, by setting both D and S to the display bitmap.

There are many subtle issues regarding bitblt. How are the two rectangles to be specified? In whose coordinate system? What if they are not conformable? Useful stipple and grey-scale patterns can be obtained by replicating the image in a small rectangle across a large one. There are also several interesting issues about the implementation of the bitblt instruction itself.

Bitblt has been found to have an amazing number of uses, far beyond simple rectangle copying. (This fact has been part of the raster graphics folklore for some time.) For example, it is possible to rotate or transpose an $n \times n$ bitmap using a small constant times n bitblts, each of which touches $O(n)$ bits. Bitblt can also be used to fill in areas, count connected components, and do other interesting and useful bitmap computations in ingenious ways. Part of the reason for this power is that many interesting bitmap computations can be done entirely through local operations, that is, by uniformly replacing each pixel with a function of itself and its neighboring pixels. Such algorithms have been used to a certain extent in the theory of iterative arrays[18] (though not especially in a graphics context) and correspond naturally to invocations of bitblt. We will see some examples later.

Although bitblt was born of necessity, it has become the center of a powerful and convenient graphics model. The most important attribute of this model is conceptual economy. Bitmaps are a general representation of an image, and bitblt is a general primitive for manipulating them. Graphical algorithms based on bitblt work identically on bitmaps containing characters, synthetic shapes, lines, scanned images or any combination thereof.

Because bitblt operates at the lowest level of the representation, the hardware for a bitmap graphics system can be very simple. This simplicity can result in high performance; indeed, some of the most dynamic, interactive graphics systems have been developed for bitmap displays. And although bitblt uses the image representation directly, it hides that representation from the programmer and is general enough to be the *only* access to the display on well-designed systems.

There are other advantages to doing all raster manipulations through a single primitive. There is an economy of implementation: the details of accessing the bitmap need be expressed in only one place, and any performance improvements in bitblt benefit every application. Also, applications that use bitblt are portable from one display to another, and can take advantage, without change, of better hardware as it becomes available. From a software engineering viewpoint, if all the applications are based on a common data structure and operation, different applications coexist and compose more effectively.

DEFINITIONS AND DATA TYPES

This section presents a mathematical model for raster graphics. The model formally defines the notions of pixels, raster images, and the primitive operations applicable to them, providing a framework with clean and unambiguous operational semantics in which to present and discuss various raster algorithms. Part of this framework is a notation for succinctly describing such algorithms.

A raster image, or more simply *image*, is an infinite two-dimensional array, each of whose elements is a bit, or *pixel*. When indexing pixels it will prove convenient to assume that their centers occupy a square lattice in the plane and are located at half-integer coordinates. The axes themselves we will label with the letters x and y, where x grows to the right and y downwards. A *point* (x, y) is a pair of integers and denotes the Euclidean point located x units to the right and y units down from the origin (0, 0). This is unfortunately a left-handed coordinate system, but is consistent with all existing implementations we know (the orientation of the y axis was inherited from the way in which text is addressed on paper and hence on terminals). If we think of the pixels as unit squares centered at the half-integer lattice points, then a point is a common corner of four adjacent pixels. All these concepts are illustrated in Figure 3.

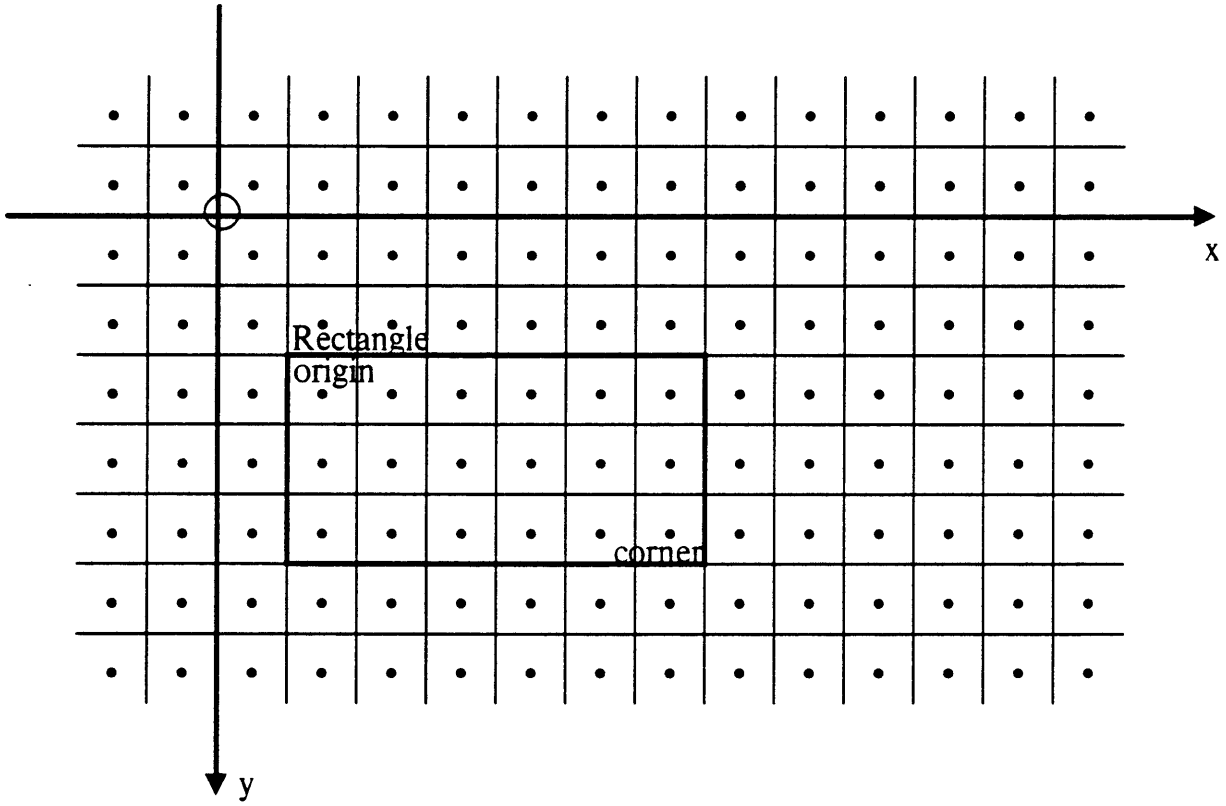

Figure 3. Pixels, points and rectangles. x increases to the right, and y downwards. The squares are pixels, and the dots their centers. The point (x, y) labels the pixel centered at $(x+½, y+½)$. A rectangle is specified by two points, its origin and corner, and includes those pixels whose centers it contains.

If p is the point (k, l), then $X[p]$ and $Y[p]$ denote respectively the x and y components of p, that is, k and l. For brevity, we will sometimes write p_x and p_y for $X[p]$ and $Y[p]$. A *rectangle* r is defined by a pair of points $r=(o, c)$, the origin o and corner c, as shown in Figure 3. We assume that $o_x \le c_x$ and $o_y \le c_y$. We also write $o=\text{origin}[r]$ and $c=\text{corner}[r]$. A non-empty rectangle is said to contain those pixels whose centers it contains. Since rectangles have sides positioned at integer coordinates and pixels are centered at half-integer coordinates, this notion is unambiguous. Another useful attribute of a rectangle is its *extent*, defined by

extent[r]=(corner$_x$[r]−origin$_x$[r], corner$_y$[r]−origin$_y$[r]), which is a point corresponding to what the corner of the rectangle would be if its origin was translated to (0, 0).

We will overload the normal arithmetic operations and allow the operations of addition and multiplication between scalars, points and rectangles when it is obvious what they mean. Thus, for example, if p and q are points, then $p+q$ denotes the point (p_x+q_x, p_y+q_y). If p is a point and r is a rectangle, then $p+r$ means the rectangle (p+origin[r], p+corner[r]), etc.

Since we intend to display raster images, they must have finite descriptions. This is achieved by assuming that the values of all pixels of an image m are fully specified by the values of all pixels lying inside a certain rectangle r, the bounding rectangle of m, denoted by bounds[m]. There are two kinds of images. A *bitmap* is an image where the value of each pixel falling outside the bounding rectangle is defined to be 0. A *texture* is an image where the value of pixels outside the bounding rectangle is defined by replicating the bounding rectangle so it tiles the plane. Note that the bounding rectangle for a texture need not correspond to the minimum tile whose repetition produces the given image.

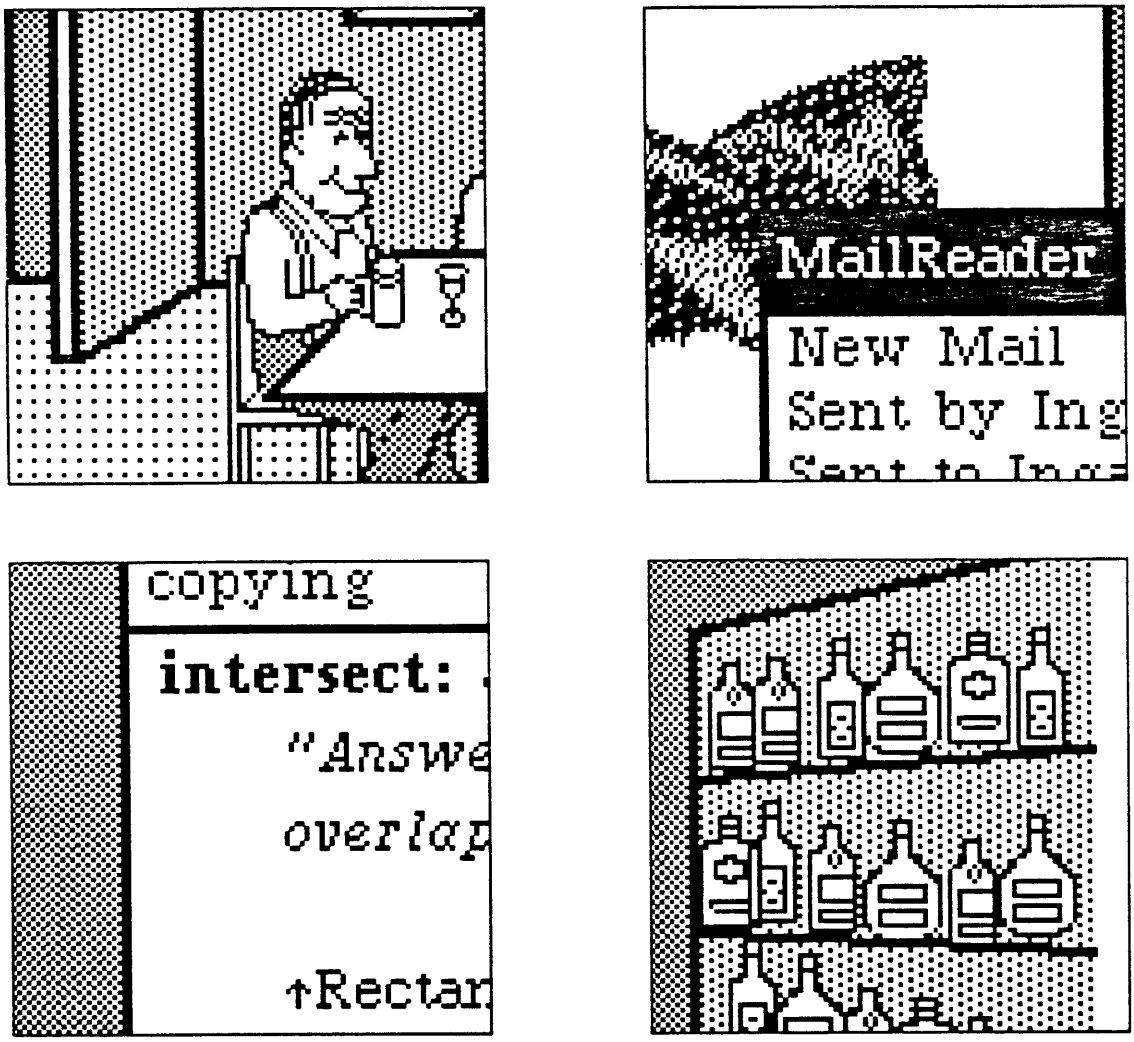

Figure 4. Magnified pieces of the Smalltalk screen in Figure 1b, showing portions of bitmaps (e.g. text and bottles) and textures (the regular patterns of dots).

For brevity, we will write, given an image m, origin[m] to denote origin[bounds[m]], extent[m] for extent[bounds[m]], etc.

The Bitblt Instruction.

Our manipulations on images will be done with a single primitive, the *bitblt* command. This command has the form

D {cut r} {clip c} {op} ⇐ {not} S {with T {shift sh}} {from p}

Here, D, S, and T are images, that is, bitmaps or textures, r and c are rectangles, sh and p are points, op is a member of the enumerated type {and, or, xor}, and each syntactic element within {}'s may be omitted. The operands have the following names:

D	destination
r	destination rectangle
c	clipping rectangle
op	operation
S	source
T	texture
sh	shift
p	source point

The bitblt command modifies the image D as follows. The type of D (bitmap or texture) is unaffected and the result has the same bounding box. Only pixels that lie within

bounds[D]∩r∩c

can change. Define a pair of pixels d in D and s in S to be *corresponding* if

1. d lies within the above intersection; and
2. d has the same position relative to origin[r] as s has with respect to p.

See Figure 5 for an illustration. For every pair of corresponding pixels d and s in the above sense, the bitblt command does the simultaneous assignment

$d \leftarrow d$ {op} {not}(s {and t})

in which the bracketed strings are omitted if the corresponding strings in the bitblt command are omitted, and t denotes the pixels of T with the same coordinates as s in S. If a shift point sh is specified, the texture in T is translated by sh before being applied to the source bitmap. If r or c are omitted then they default to bounds[D]; if p is omitted then it defaults to origin[S].

Usually the bitblt command is used with D and S being bitmaps, and T, if present, a texture. Two images are called *conformable* if they have bounding boxes with the same extent. A frequent use of bitblt is to store an image into another conformable one. With our notation this can simply be denoted as

$D \Leftarrow S$.

A number of textures are commonly used and therefore have special names. The textures *ALL0* and *ALL1* have bounding box ((0,0), (1,1)) and yield the value 0 or 1 everywhere, respectively. The textures *XBIT*[k] and *YBIT*[k] have bounding boxes ((0,0), (2^k,1)) and ((0,0), (1,2^k)) and yield at the pixel ($x+\frac{1}{2}$, $y+\frac{1}{2}$) values corresponding to the k-th bit of x and y respectively.

Note that if T_1 and T_2 are two textures, then $T_1 * T_2$, for any Boolean operation $*$, is also a texture. To see this, we argue as follows. If T is a texture, then there is an equivalent texture T', in the sense of having pixel values equal to those of T everywhere, such that bounds[T']=((0,0), extent[T]). Now if T_1' and T_2' are equivalent to T_1 and T_2 respectively in exactly this sense, then $T_1 * T_2$ is equivalent to $T_1' * T_2'$, which is clearly a texture since r=((0,0), extent[T_1]·extent[T_2]) can serve as a bounding rectangle. Of course the minimum rectangle whose repetition yields $T_1' * T_2'$ may actually be a subrectangle of r.

The bitblt command never modifies the bounding rectangle of the destination. Thus, for example, the assignment

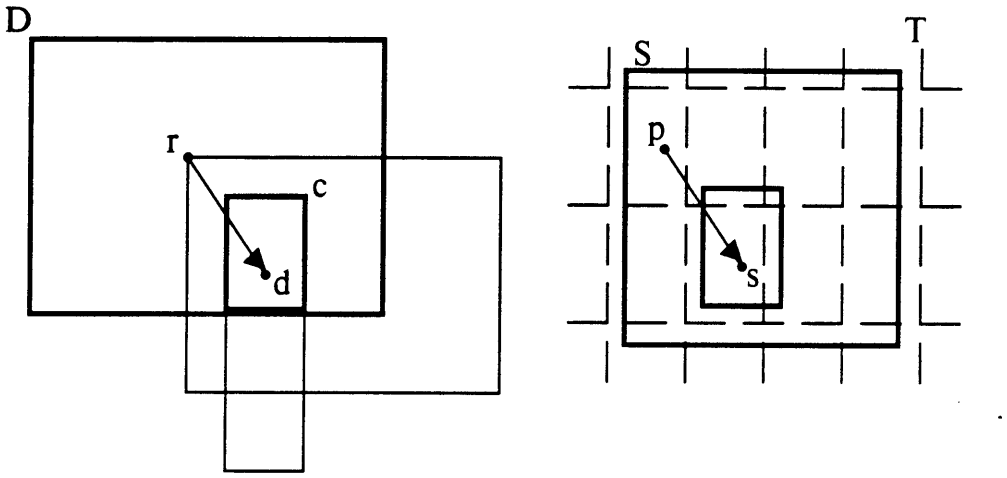

Figure 5. Bitblt parameters. Pixel correspondence between S and D is established by adjusting the coordinate system inside S so p and origin[r] are coincident; this maps the point s onto the point d. Those pixels in S inside the intersection of r, c, bounds[D] and the (translated) bounds[S] are then masked with the replicated, translated texture T and copied to D according to *op* (not shown).

D xor $\Leftarrow$ S

for textures D and S might not set D to the texture corresponding to the expression D xor S, whose smallest possible bounding rectangle might be larger than bounds[D]: if $D = XBIT[1]$ and $S = YBIT[1]$, then D xor $\Leftarrow$ S is simply no operation.

It is worth describing how line segments (or rather approximations to line segments) fit into the formalism. The primitive *line* is defined as follows:

line *pt1 pt2* {in *B*}

or's into the bitmap B the pixels approximating the line segment from *pt1* to *pt2*. How this approximation is made is described in a later section. If B is omitted, a bitmap of the minimum. size required to hold the line is created and filled with 0's before the line is drawn. In either case, the line primitive acts as a function and returns the bitmap as its result, so it can be used in a bitblt invocation.

For the creation of new bitmaps or textures we will use the command

$B \leftarrow NewIm[r]$

which yields a new bitmap or texture, according to the type of B, with bounding rectangle r and all pixels equal to zero.

Figures 6 and 7 show a couple of representative applications of bitblt: drawing a character and drawing menu.

Some familiarity with the laws of Boolean algebra is essential to understand bitmap algorithms. The operators and and or, as well as the major identities they satisfy, are probably familiar to most readers from set theory where they correspond to the operations of intersection and union. The operation xor is less familiar and it will be useful for us to review some of its properties. The first important one states that

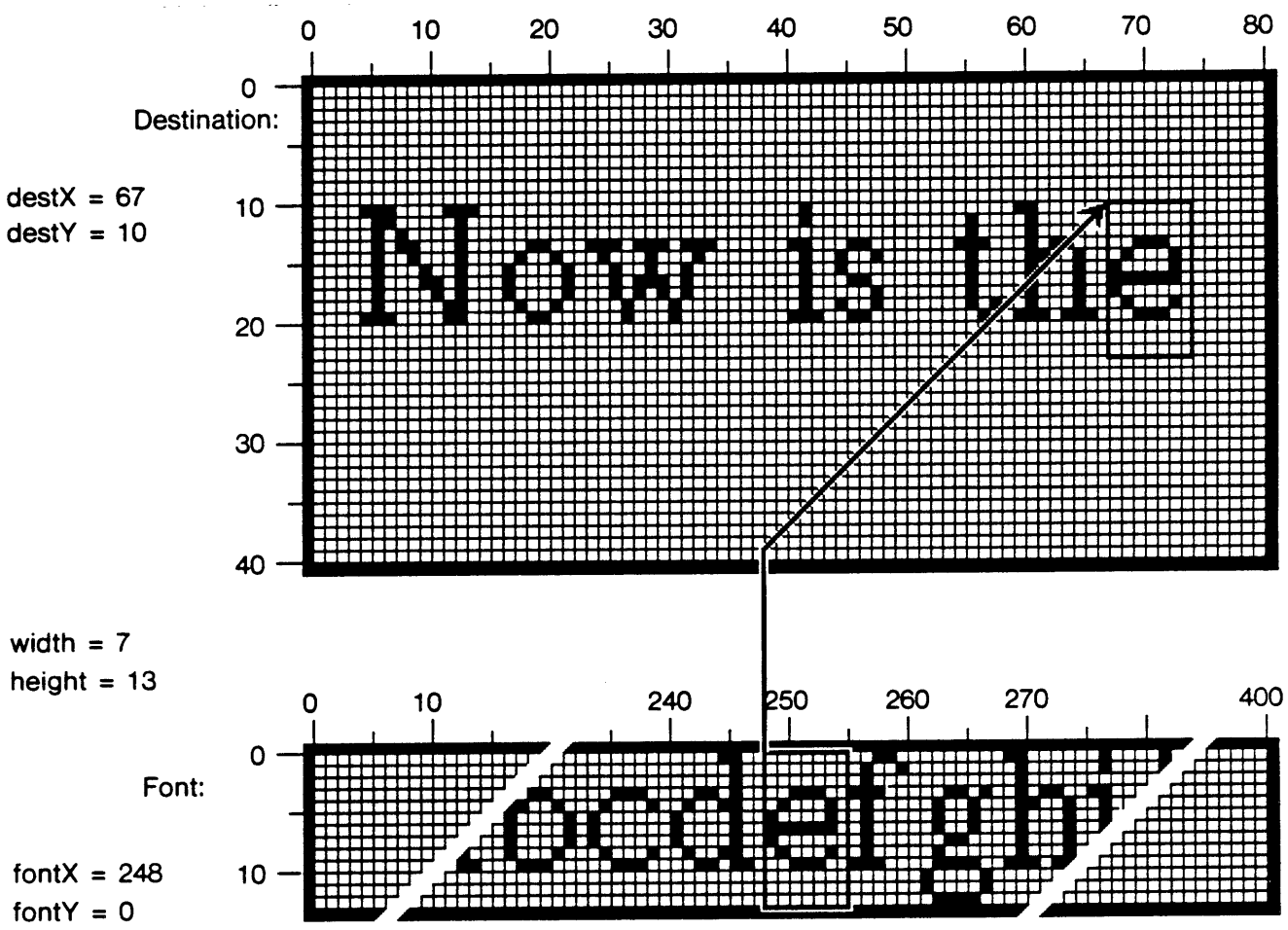

Figure 6. Drawing a character. The rectangle surrounding the image in the font bitmap is copied to the destination bitmap, typically the display. (Figure Copyright © Xerox Corp. Reprinted by permission.)

$(x \text{ xor } y) \text{ xor } x = y;$

in other words, that xor is commutative and its own inverse. If any two of x, y and x xor y are given, the third can be found. A second useful identity states that

$(x \text{ xor } y) \text{ and } z = (x \text{ and } z) \text{ xor } (y \text{ and } z);$

in other words, and distributes over xor.

As an example of the use of these laws, suppose that we are given two bitmaps A and B, a rectangle r in A and a point p in B, and we wish to exchange the contents of r in A and $(p, p+\text{extent}[r])$ in B, as shown in Figures 7 and 8. The following program accomplishes this:

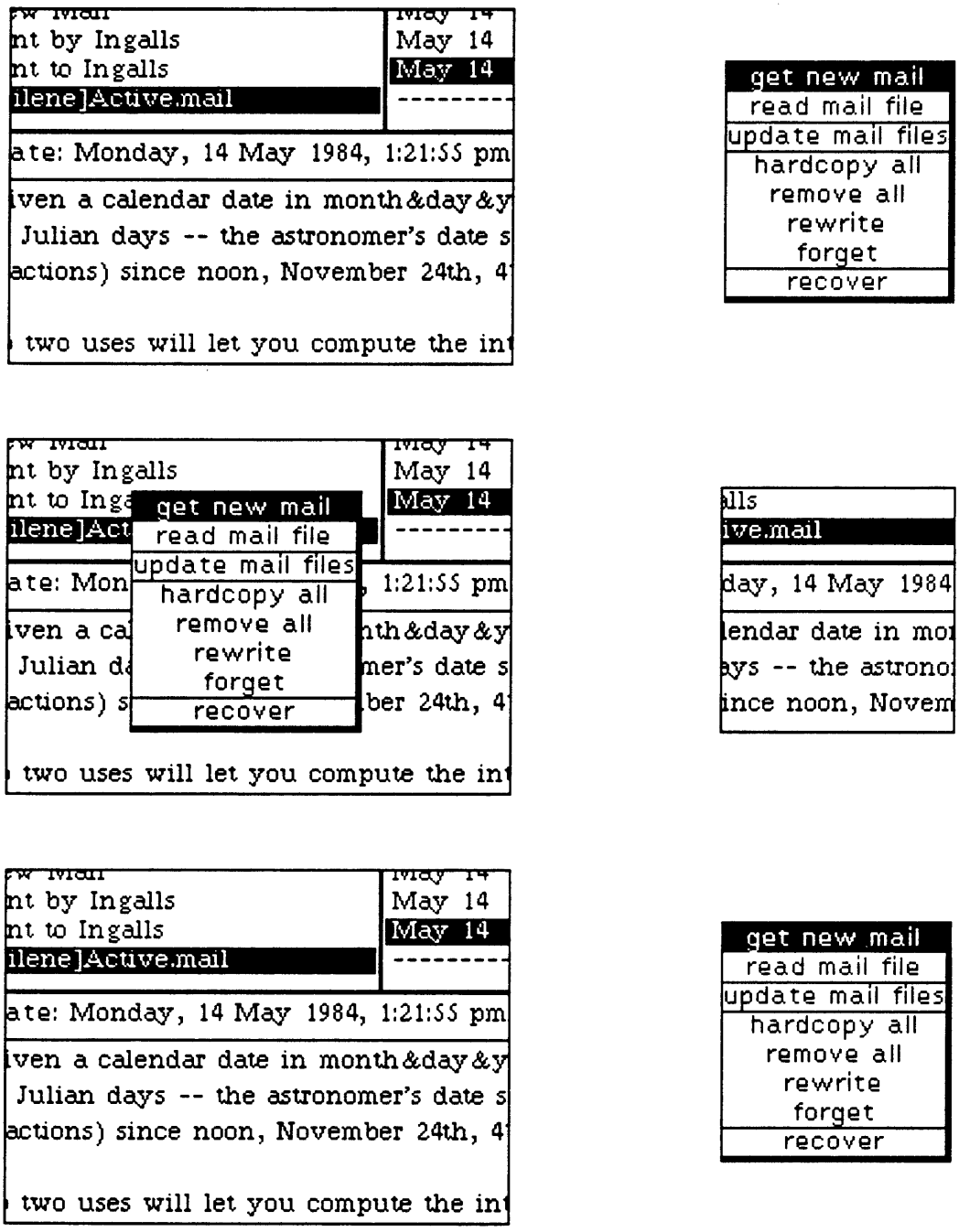

Figure 7. Drawing a menu. An off-screen bitmap is allocated and the menu drawn there. The off-screen bitmap and the screen storage at the position of the menu are then exchanged. When the selection is made, they are exchanged back and the menu bitmap deallocated.

```
Exchange[A, B, r, p]
        modifies Bitmap A, B
        Rectangle r
        Point p
{
        A cut r xor ⇐ B from p
```

B cut (*p*, *p*+extent[*r*]) xor ⇐ *A* from origin[*r*]
A cut *r* xor ⇐ *B* from *p*
}

An interesting exercise is to write the procedure that performs the same exchange, but only where allowed by a mask *M*. Note that this algorithm fails if the source and destination share pixels, because *x* xor *x* is zero.

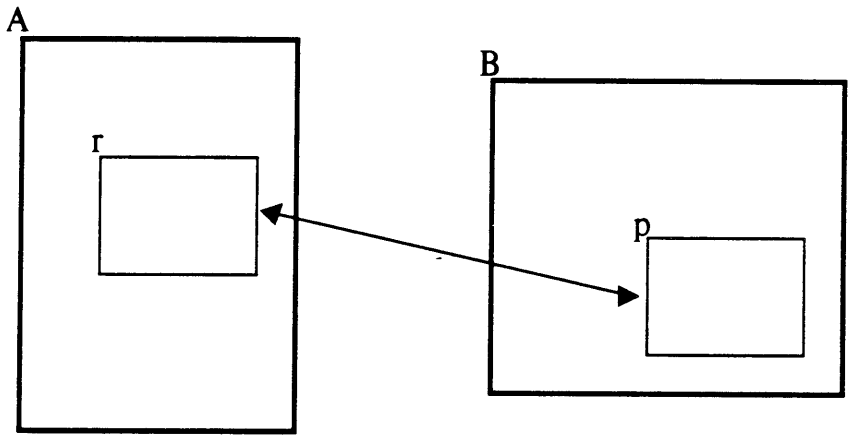

Figure 8. Exchanging images between two bitmaps.

BITBLT ALGORITHMS

Using the bitblt formalism, a number of image processing and graphics algorithms are easily expressible. Some of these, such as Levialdi's transform, are well-known image processing algorithms that are elegantly expressed using bitblt; others, such as the area fill algorithms, show how bitblt can be used to implement efficiently a number of traditional graphics operators that might normally be supplied as separate primitives in a library. This section is a catalogue of interesting algorithms, and the authors would appreciate hearing of other algorithms that readers know or discover.

Levialdi's transform

The Levialdi transform,[14] when repeatedly applied to a bitmap, slowly degrades the image. In particular, king-wise connected components of black pixels (1's) map to connected components of black pixels, and distinct components remain distinct. (Two pixels are king-wise connected if they share a corner.) Each component loses exactly one pixel after an application of the transform. Thus by counting the number of isolated black pixels that disappear as the transform is repeatedly applied, we can count the number of connected components of our bitmap. If p denotes a pixel and r, d, and x denote respectively the right, down and diagonally right-down neighbors of p, then the Levialdi transform sets

$$p' \leftarrow (p \wedge (r \vee d \vee x)) \vee (r \wedge d)$$

uniformly throughout the bitmap.

The following program executes the Levialdi transform on the bitmap *Bd*.

```
LevialdiTransform[Bd]
        modifies Bitmap Bd
{
        "Executes Levialdi transform on Bd"
        Bitmap T1, T2

        T1 ← NewIm[bounds[Bd]]
        T2 ← NewIm[bounds[Bd]]
        T1 ⇐ Bd from (0,1)
        T1 or ⇐ Bd from (1,0)
        T1 or ⇐ Bd from (1,1)
        T2 ⇐ Bd from (0,1)
        T2 and ⇐ Bd from (1,0)
        Bd and ⇐ T1
        Bd or ⇐ T2
}
```

Notice that although the algorithm is expressed in terms of pixel operations, the bitblt-based implementation applies the same operations to the entire image.

Figure 9 shows the result of applying this transformation repeatedly to our standard image. It is interesting to watch the image decay by this process: it gradually becomes unintelligible and collapses into the upper left corner. (Actually, the image needs a one-pixel-wide white border for the Levialdi transform to count isolated components near the edge.)

Figure 9. Levialdi's Transform

Picture clean-up

Inverse scan conversion is the process of taking scanned-in shapes and finding higher-level continuous representations for them, such as polygons or characters. Algorithms for doing so work better if the bitmaps they are given satisfy certain regularity conditions that avoid 'stray' pixels. Define the *closure* of a bitmap P to be the smallest bitmap containing P (in the sense of having a superset of the black pixels) and such that every white pixel is part of a 3×3 square of white pixels. Similarly, define the *interior* of a bitmap as the largest bitmap contained in P so that every black pixel is part of a 3×3 square of black pixels. The algorithm below implements a picture 'clean up' operation by replacing P by the interior of its closure.

```
clean[Bd]
        modifies Bitmap Bd
{
        "Cleans Bd to lines of thickness at least 3"
        Bitmap T, S

        S ← NewIm[bounds[Bd]]
        T ← NewIm[bounds[Bd]]
        A[Bd, S, T]
        B[Bd, S, T]
        B[Bd, S, T]
        A[Bd, S, T]
        Bd ⇐ S
}
A[Bd, S, T]
        Bitmap Bd
        modifies Bitmap S, T
{
        T ⇐ Bd
        T or ⇐ Bd from (0,-1)
        T or ⇐ Bd from (0,1)
        S ⇐ T
        S or ⇐ T from (-1,0)
        S or ⇐ T from (1,0)
}
B[Bd, S, T]
        modifies Bitmap Bd, T
        Bitmap S
{
        T ⇐ S
        T and ⇐ S from (0,-1)
        T and ⇐ S from (0,1)
        Bd ⇐ T
        Bd and ⇐ T from (-1,0)
        Bd and ⇐ T from (1,0)
}
```

The ABBA pattern of the algorithm reflects the antisymmetry of the white and black operations. AB generates the white regions by or'ing and then and'ing, whereupon BA thickens the black regions by and'ing and finally or'ing. Figure 10 shows the results of successive applications to an image, this time a (roughly) rook-wise connected outline derived (by hand) from our original bitmap. (Two pixels are rook-wise connected when they share an edge.)

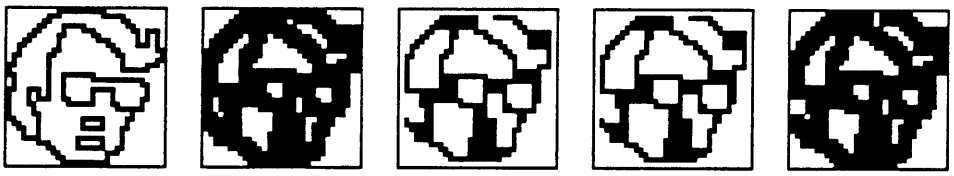

Figure 10. Picture clean-up

Area-filling by xor'ing scan lines

In this algorithm we are given a simple (non self-intersecting) rook-wise connected closed pixel path in the plane. The path is specified as a bitmap with 1's on the path and 0's elsewhere. The algorithm fills in (makes 1's) all pixels interior to the path, essentially by replacing each scan line with the xor of all scan lines up to and including that one. To avoid some boundary problems, the rightmost 1 of each run of 1's in a scan line is removed before this operation is applied. At the end, the path is or'ed back in to restore any rightmost pixels that might have been missed from a run.

```
fillByXorScanlines[Bd]
        modifies Bitmap Bd
{
        "Fills with 1's the connected contours in Bd"
        integer o, c
        Rectangle r
        Bitmap TB1, TB2

        o ← X[origin[Bd]]
        c ← X[corner[Bd]]
        r ← ((0,0), (c-o,1))
        TB1 ← NewIm[r]
        TB2 ← NewIm[r]
        for i ← Y[origin[Bd]] to Y[corner[Bd]]−1 {
                TB2 ⇐ Bd from (o,i)
                TB2 and ⇐ TB2 from (1,0)
                TB1 xor ⇐ TB2
                Bd cut ((o,i), (c,i+1)) or ⇐ TB1
        }
}
```

Figure 11 shows the intermediate results at each execution of the loop. The image used is not perfectly rook-wise connected — the lower part of the face is not a closed curve — and the result is therefore incorrect. The error is easy to characterize, however: because of its basis on the xor function, this algorithm essentially colors each region of the picture with its parity, white for even and black for odd. The 'beard' hanging from our subject therefore indicates an odd number of pixels in those vertical slices through the image. For comparison, the picture in the lower right shows the result of applying the algorithm horizontally instead of vertically: a different style of beard.

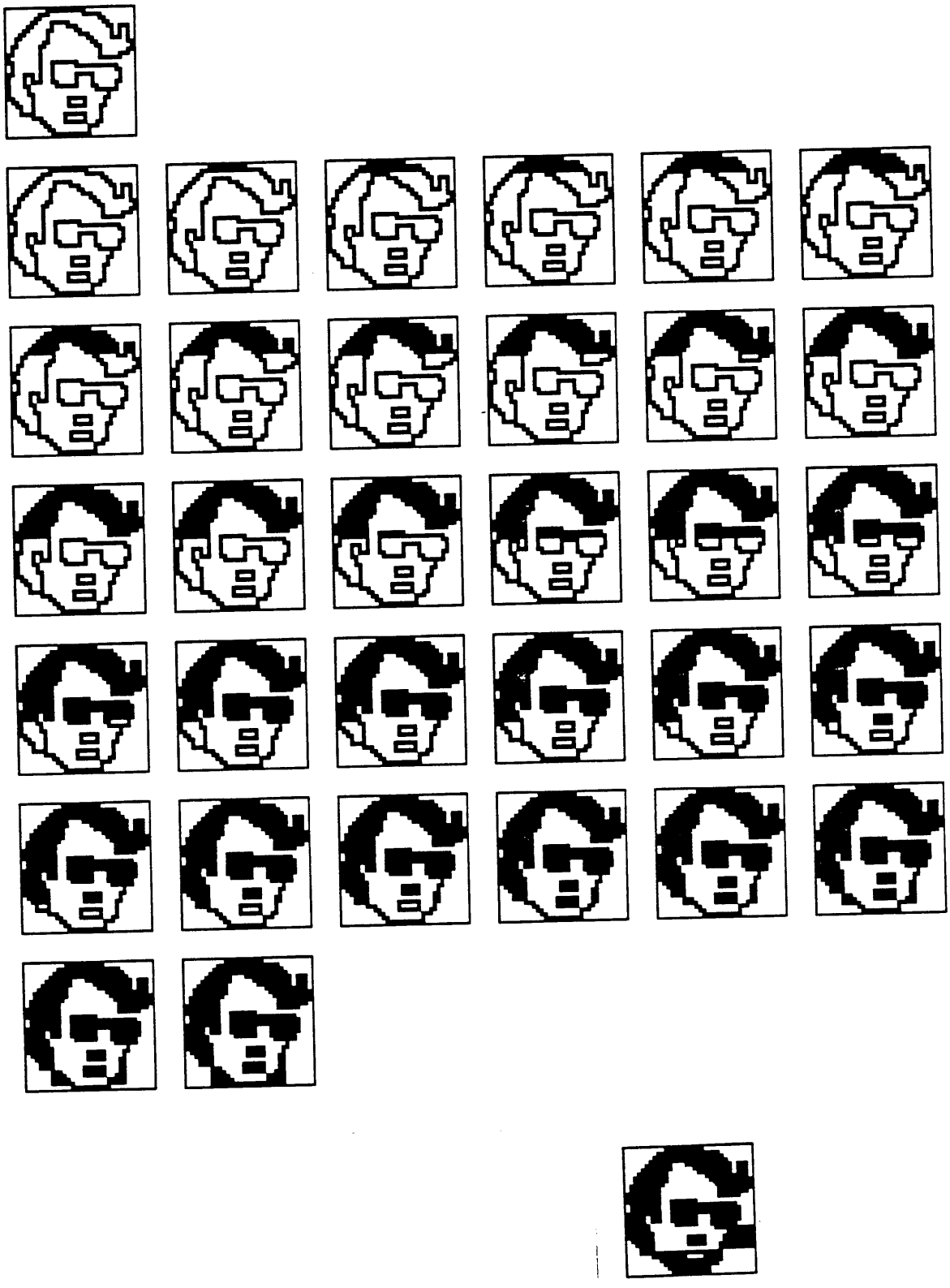

Figure 11. Area filling by xor'ing scan lines.

Area filling by growing seed

Area filling can also be accomplished by putting a single black pixel inside the path and then growing the black around the pixel, but stopping the growth from escaping outside the pixel path. This works for simple closed pixel paths that are only king-wise connected, unlike the xor'ing scan line method that requires rook-wise connectivity. A drawback, however, is that a particularly sinuous path in a bitmap of size $m \times n$ may require almost $mn/2$ iterations for the seed to reach from one end of the path to the other. This is a pathological case, though, and a more realistic termination criterion is to stop when the image stops changing. For simplicity, this version of the algorithm picks an incorrect limit, but it is a worthwhile exercise to write a safe version.

```
fillBySeed[Bd, seed]
        modifies Bitmap Bd
        Point seed
{
        "Fills with 1's the pixel path surrounding seed in Bd"
        Bitmap T, F

        T ← NewIm[bounds[Bd]]
        F ← NewIm[bounds[Bd]]
        F[seed] ← 1
        for k ← 1 to max[X[extent[Bd]], Y[extent[Bd]]] {          "see text"
                T ⇐ F
                T or ⇐ F from (0,1)
                T or ⇐ F from (0,−1)
                T or ⇐ F from (−1,0)
                T or ⇐ F from (1,0)
                F ⇐ T
                F and ⇐ not Bd
        }
        Bd or ⇐ F
}
```

Figure 12 shows the intermediate values of F for each iteration of the loop, starting from a seed pixel inside the hair contour.

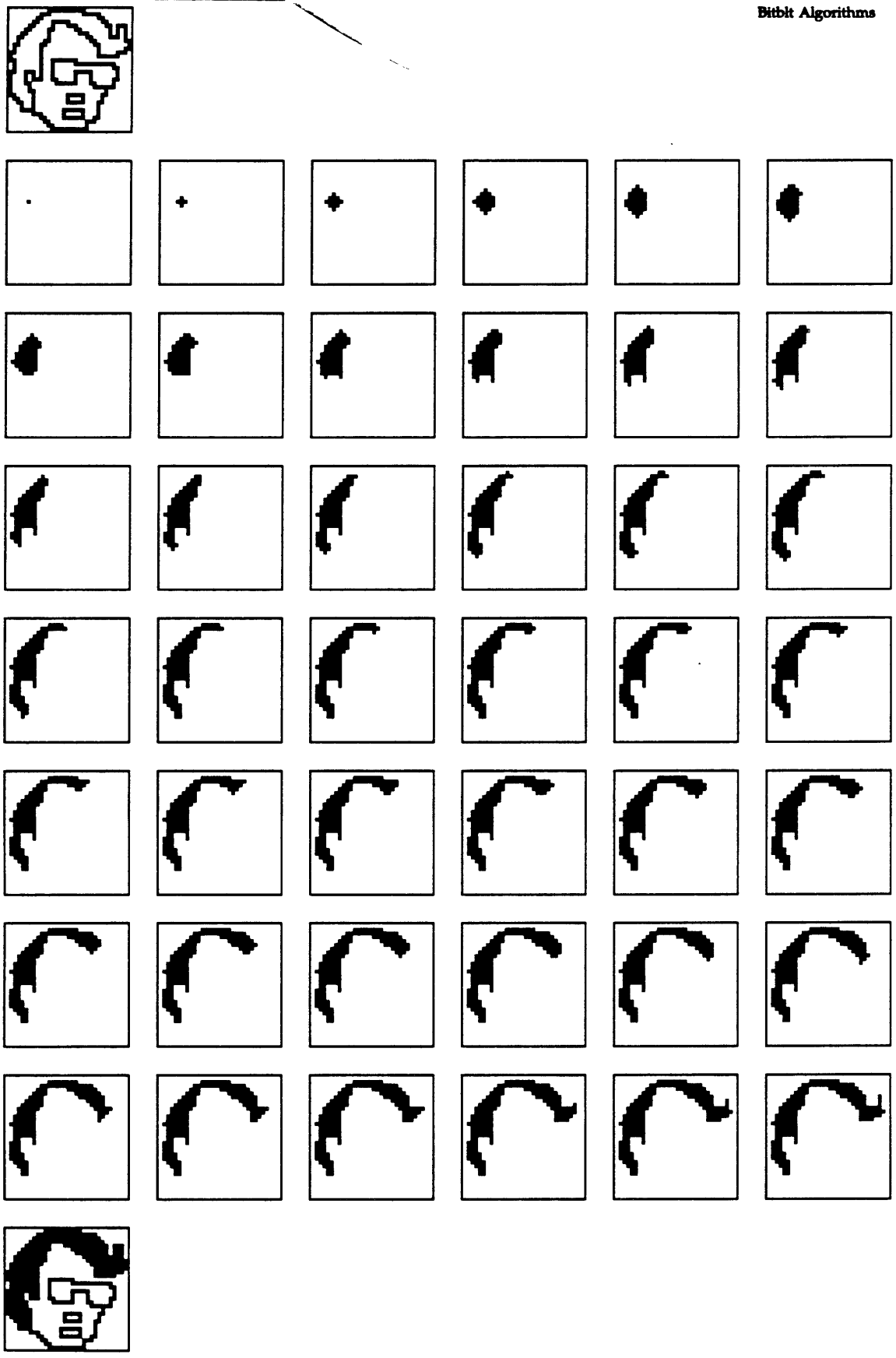

Figure 12. Area filling by growing seed.

Rotation of a bitmap by shearing

This method rotates an $n \times n$ bitmap clockwise by 90 degrees in $4n+5$ bitblts, each of $O(n)$ bits. It is essentially a discrete implementation of the transformation identity

$$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} -1 & 1 \\ 0 & -1 \end{bmatrix}$$

that expresses the rotation $\{y' \leftarrow x;\ x' \leftarrow -y\}$ as the composition of three shearing maps. Similar techniques can be used for bitmap transposition. The program below makes the simplifying assumption that *Bd* is a square bitmap with bounding box ((0, 0), (*n*, *n*)).

```
rotateByShear[Bd]
        modifies Bitmap Bd
{
        "Rotates Bd 90 degrees clockwise by multiple shearing"
        Bitmap M, A
        integer n

        n ← Y[corner[Bd]]
        assert n = X[corner[Bd]]
        M ← NewIm[((0,0), (2*n,2*n))]
        M cut ((0,0), (n,n)) ⇐ Bd
        A ← NewIm[((0,0), (2*n,2*n))]
        for i ← 0 to n−1
                A cut ((i,0), (i+1,2*n)) ⇐ M from (i,−i)
        M ⇐ A
        for j ← 0 to 2*n−1
                A cut ((0,j), (2*n,j+1)) ⇐ M from (j+1−2*n,j)
        M ⇐ A
        for i ← n to 2*n−1
                A cut ((i,0) (i+1,2*n)) ⇐ M from (i,n−i−1)
        M ⇐ A
        Bd ⇐ M from (n,n)
}
```

Figure 13 shows the intermediate values of *M* after executing each loop. An interesting exercise is to reconstruct the image after each shear, so it is always contained in the upper left quadrant of *M*, albeit folded over on itself. This method requires only $3n+6$ bitblts.

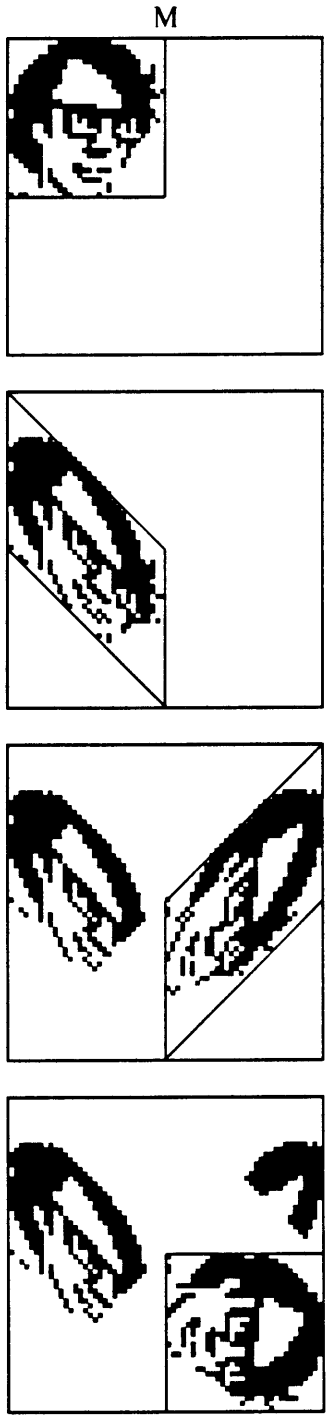

Figure 13. Rotation of a bitmap by shearing.

Rotation of a bitmap by parallel recursive subdivision

The following rotation algorithm uses a 'binary mask' technique based on the work of Floyd.[7] The bitmap size is assumed to be $m \times m = 2^n \times 2^n$, and the rotation is accomplished in n steps. After k steps, the current picture is the original picture except that it has been divided into square subarrays of size $2^k \times 2^k$, and each subarray has been rotated in place. (Of course, the zeroth iteration is trivial: a rotated single pixel is identical to an unrotated one.) The next step will assemble each four of these subarrays into a single rotated subarray of size $2^{k+1} \times 2^{k+1}$. All groups are operated upon simultaneously; the subarrays to be moved in each group are selected by a mask containing squares of ones of size $2^k \times 2^k$ in the appropriate places. The advantage of this method is that it requires only $7 \log_2 m$ bitblts to accomplish its task. This is less than $4m+5$ when $m>4$ (which is true even for characters). However, if m is not a power of two, the implementation of this rotation technique becomes considerably more complicated.

```
rotateByMasks[Bd]
        modifies Bitmap Bd
{
        "Rotates Bd 90 degrees clockwise by recursive subdivision"
        integer j, m, n
        Bitmap T
        Texture M

        m ← corner[Bd].y
        assert m = X[corner[Bd]]
        assert origin[Bd] = (0,0)
        n ← floor[log2[m]]
        assert m = 2^n
        T ← NewIm[bounds[Bd]]
        M ← NewIm[bounds[Bd]]
        for k ← 0 to n−1 {
                j ← 2^k
                M ⇐ XBIT[k]
                M and ⇐ YBIT[k]
                T ⇐ Bd with (M shift (−j,−j)) from (−j,0)
                T or ⇐ Bd with (M shift (−j,0)) from (0,j)
                T or ⇐ Bd with M from (j,0)
                T or ⇐ Bd with (M shift (0,−j)) from (0,−j)
                Bd ⇐ T
        }
}
```

Figure 14 shows intermediate values of T and M (the mask) in the loop.

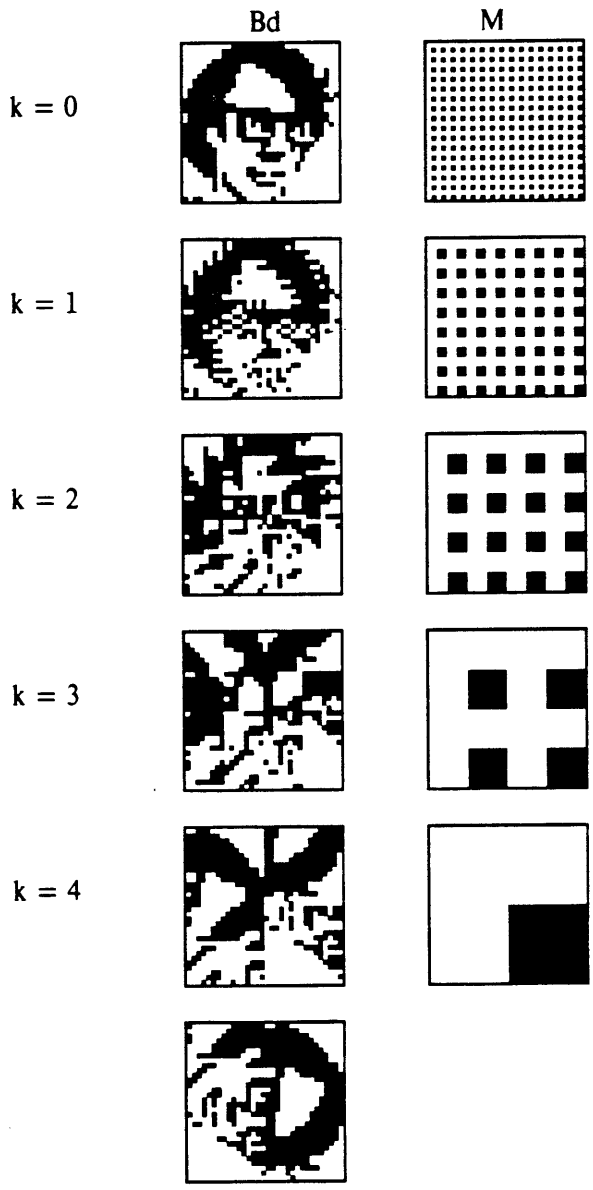

Figure 14. Rotation of a bitmap by parallel recursive subdivision

Magnification

Visually, one of the most striking applications of bitblt is the speedy 'magnification' of an image: the process of replacing each pixel by a larger rectangle of pixels of the same value as the source. In the procedure below we specify a rectangle *r* containing the image we want to magnify and the scale of the magnification (this is a point, to allow independent magnification factors in the *x* and *y* directions). The algorithm involves two invocations of the procedure *spread*, the first to accomplish horizontal and the second vertical magnification. Each invocation of *spread* first copies the appropriate entities (rows or columns) into the result bitmap, spacing them apart as it does so, and then smears them by or'ing the bitmap with itself.

```
Bitmap magnify[Bd, r, scale]
        Bitmap Bd
        Rectangle r
        Point scale
{
        "Returns a copy of the portion of Bd contained in r, magnified by p (an (x,y) pair)."
        Bitmap Wide, Big

        Wide ← NewIm[((0,0), (extent[r] * (X[scale],1)))]
        Big ← NewIm:[((0,0), (extent[r] * scale))]
        spread[Bd, Wide, r, X[scale], (1,0)]
        spread[Wide, Big, bounds[Wide], Y[scale], (0,1)]
        return Big
}
smear[T, scale, d]
        modifies Bitmap T
        Point scale, d
{
        "Smears pixels in T to size d.[xy] in specified direction"

        for i ← 1 to (scale . d)−1          "Smear out the slices"
                T or ⇐ T from −d
}
spread[F, T, r, scale, d]
        Bitmap F
        modifies Bitmap T
        Rectangle r
        integer scale
        Point d
{
        "Spread out in specified direction the image contained by r in F
                by factor d.[xy], placing result in T."
        Rectangle slice
        Point spt

        slice ← ((0,0), (corner[T] * (Y[d], X[d]))+d)
        spt ← origin[r]
        for i ← 1 to (extent[r] . d) {      "slice up original area"
                T cut (slice + origin[T]) ⇐ F from spt
                spt ← spt + d
                slice ← slice + d*scale
        }
}
```

Figure 15 shows the values of *Wide* and *Big*, after calling *spread* and *smear*, with *scale* (4, 3).

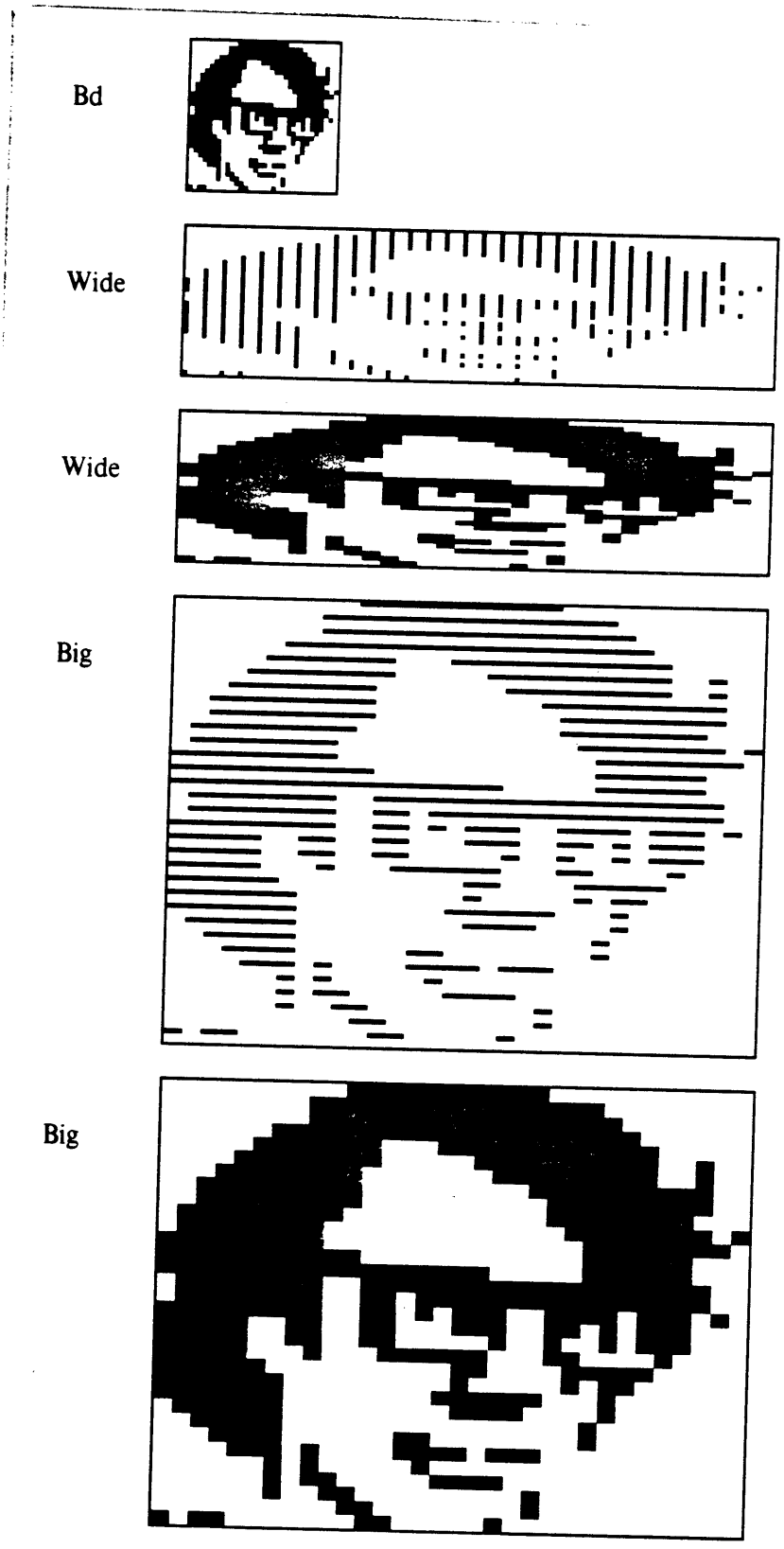

Figure 15. Magnification of a bitmap

IMAGING HARDWARE

Before discussing the software and hardware issues of bitblt implementation, it is worth describing how a CRT creates the final, dynamic display.

A raster display scans an electron beam across the front face of its tube. The inside of the face is coated with a *phosphor* — a polycrystalline substance that emits light when excited by the electron beam. The refresh rate is defined as the inverse of the time it takes for the beam to touch all pixels once, and is typically between 30Hz and 100Hz. In order to reduce the memory bandwidth exacted by high refresh rates, some displays *interlace* the refresh: they scan the tube twice vertically during a refresh cycle, touching even- or odd-numbered horizontal scan lines on alternate half-cycles.

Although conventional wisdom holds that phosphors have two forms of emission — fluorescent (short duration) and phosphorescent (long duration) — they are actually identical physical processes. For 'fast' phosphors (those designed for refresh from about 60Hz up) most of the light is emitted while the electron beam is touching the crystal, while slow phosphors emit light long after the beam has moved on.

A glowing phosphor is in a state of dynamic equilibrium. The electron beam excites electrons in the crystals, which decay to lower energy states through emission of light, and are excited again. If a pixel's value is constant, the light output integrated over the refresh rate will be constant. But when the value changes, the phosphor may respond slowly to the change. If the phosphor is slow, it may take several 'hits' from the electron beam to raise the phosphor to equilibrium brightness. Similarly, if the decay rate is higher than the refresh rate, the intensity will drop appreciably between refreshings, which can lead to visible flicker if the refresh rate is lower than the eye's flicker threshold.

The speed of phosphor is defined as the time for the light to decay to 10% of its original value after the excitation is removed. The properties of the standard Joint Electron Device Engineering Council (JEDEC) phosphors are listed in Table 1.

JEDEC/ Clinton Phosphor Number	Description			CIE Coord (Nominal)		Refresh Rate	Comments
	Steady State	Decay Color	Decay Time	X	Y		
P-1	YG	YG	15ms	.218	.712	60	Oscillography, radar
P-4	W	W	24μs	.270	.300	60	TV
P-6	W	W	.8ms	.338	.347	60	TV
P-11	B	B	30μs	.149	.144	60	Photography
P-22R	R	R	.7ms	.653	.343	50	60
P-22G	YG	YG	60	.346	.604	60	Projection
P-22B	B	B	60	.146	.061	60	Projection
P-31	YG	YG	70	.261	.549	60	Oscillography
P-39	YG	YG	520ms	.218	.720	30	Radar, display
P-40	W	YG	400ms	.270	.316	50	Display
P-45	W	W	1.5ms	.255	.292	60	Single comp. white
PC-164	W	YG		.265	.290	60I	Long white
PC-171	W	YG		.365	.400	60I	Soft white

Table 1. Properties of selected JEDEC (those labeled 'P') and Clinton ('PC') phosphors. Abbreviations: R: red, G: green, B: blue, Y: yellow, O: orange, V: violet, W: white, P: purple. I: interlaced.

P4 is the standard 'white' phosphor for screens refreshed at or above about 50Hz. It flickers badly at 50Hz to 60Hz, however, as discussed below. P39 is a slow, ugly green, usually refreshed around 30Hz. P40 is sometimes called 'slow P4.' It's a mixture of a blue and a yellow phosphor; the blue component is a bright phosphor that decays quickly, while the yellow component is slow. The color of a static P40 display is almost white, but when pixels change from white to black the decay is through strange purples and greens. P40 is usually refreshed

around 30Hz to 40Hz, but, again, flickers noticeably at 30Hz.

Flicker is a multi-faceted problem. Most screens are refreshed at either about 60Hz or interlaced 30Hz. Interlacing reduces the required memory bandwidth, but if an insufficiently slow phosphor is used, or if the alternate *fields* are very different (that is, if there is large contrast between adjacent scan lines), the eye is confused by the image and the flicker is highly visible. (This is an important consideration when designing textures.) For fast phosphors such as P4, even if the refresh rate is high, a person can subliminally detect the blackness between the brief fluorescent peaks. Sensitivity to flicker of any form is increased when the display is viewed peripherally, when the eye is moving (such as when reading text on the display), if ambient flickering light such as fluorescent illumination is present, if the display background is dim, if the image is bright, or if the ambient light is low. Usually when the ambient light level is low, the display brightness is turned down, which more than compensates. Also, a light source that is mostly off, with intermittent light (such as a fast-refresh low-persistence display), flickers more than a light source that is mostly on, with intermittent blackness (such as cinema, which is at 48Hz). Finally, of course, people's individual sensitivity to flicker is highly variable, although the eye can be fatigued by flickering light even when the flicker is invisible. The main physiological symptom of exposure to flickering light is a loss of adaptation, the ability to adjust to varying levels of illumination. The eye's response to flicker is the same as to glare.

There are too many factors to allow a bald assessment such as "above 60Hz, flicker is invisible." P4 is so fast that it must be refreshed at 90Hz or greater to render flicker invisible under all conditions (but even then, the eye can be stressed by the non-uniformity of the light). Even P39, the slowest phosphor commonly used, flickers at 30Hz if the interlace fields are wildly different.

The hardware designer would like to use a low refresh rate because it keeps the hardware cost down, so P39 might be considered a good choice for a phosphor. P39 has another advantage: the green color is near the center of the optical spectrum, where chromatic aberration is minimal, brightness sensitivity is good and the eye focuses most easily. But the green color is ugly, and some people react badly to it — they see a pink afterimage. Also, the phosphor is so slow that the display takes about a second to change in response to a change in the refresh pattern. This makes moving images such as animation and scrolling text hard to look at. One simple improvement is to reverse the sense of the image, so the background is light and the image (such as characters) dark. Because contrast is expressed as

$$\frac{L - L_b}{L_b}$$

where L is the image brightness and L_b the background brightness, the exponential tail is less noticeable on a bright background: erasing an image with light rather than dark is much more effective. Nonetheless, the image still changes slowly, and the color *is* ugly. Unfortunately, phosphor designer's best attempt at a slow white phosphor, P40, is not slow enough and is only white when at equilibrium. (Some proprietary phosphors, such as Clinton PC-171, may be better but are not registered by JEDEC and are not widely available.) Creating a truly white phosphor is hard, of course: it requires three colored phosphors with identical persistences.

A bright background helps animation but increases the flicker. The Ferry-Porter law states that (in the brightness range of interest) the highest frequency at which flicker is seen increases 10Hz for every factor of 10 increase in brightness. Therefore, if a bright background is used (at least on a fast display), the brightness knob should be kept turned down.

Overall, however, a bright background is superior. Experiments with data entry clerks have shown that people can work significantly more effectively with a bright background, even people that claim they prefer dark backgrounds. Because the image is bright, glare (ambient light reflecting off the screen) is less of a problem. The higher average light reaching the eye allows the eye to perform better, especially regarding visual acuity, distortion, and depth of field, which is particularly important if the eye is looking alternately at the display and something else, such as paper. The screen brightness can be matched, in fact, to that of the paper,

provided the room lights are adjusted properly.

Summing up, here are some guidelines for design and use of raster displays:

- Refresh the display beyond the flicker frequency of the phosphor.
- Keep contrast high on display (avoid glare, perhaps by using anti-reflection coatings rather than diffuse surfaces on the glass). When the contrast gets too low, reading ability is impaired by the loss of parafoveal vision.
- Keep room lights low, and the brightness knob turned down. The ideal lighting should have brightnesses about

  task:surround:background :: 9:3:1

  The room lighting should be somewhat lower than that typical for reading paper.
- Use a bright background on the display.
- Use incandescent room light, not fluorescent.

A couple of other display technologies deserve mention: plasma panels and liquid crystal displays (LCDs).

Plasma panels are quite old. They emit the same light as the glow in the back of a TV tube: the orange emission around the cathode of a 100V or so drop in a near vacuum. Plasma panels have two perpendicular arrays of wires, forming a grid, with about a 1mm separation between the planes of horizontal and vertical wires. The basic idea is to apply about ± half the required voltage to the x and y wires of the pixel to be lit; together, these provide a voltage difference large enough to induce the glow. In practice, an AC voltage on the wires is always present, and the peak voltage is increased or decreased when the value of a pixel is to be changed; the AC and a small metal plate cause each pixel to otherwise maintain its value. Plasma panels can be quite large and, because they do not require deflection magnets like a CRT, are flat rather than funnel-shaped. But they are complicated to drive and expensive because of the large voltages required, so it seems unlikely that a bitblt-based plasma panel will be built.

Liquid crystals are composed of large, cigar-shaped molecules that tend to align along static electric fields. A liquid crystal display has capacitive plates forming the image pattern under a very thin layer of liquid crystal. Small static voltages applied to the plates align the molecules and affect how light is reflected immediately above the plates. The crystals react fairly slowly to the voltage (time constants of order a second), so they produce a steady image. For reasons beyond the scope of these notes, black pixels must be refreshed by a 10ms or so voltage pulse before they decay, which limits the size of a single liquid crystal display to about (refresh time/10ms). The technology is improving, however, and some recently developed LCDs are arrays of pixels capable of displaying 24 lines by 80 columns of bitmap characters. Because liquid crystals reflect light, rather than emit it, and because the voltages are low and static, LCDs consume very little power. Also, like plasma panels, they are flat displays. Before too long, LCDs may make portable high-resolution bitmap displays possible.

Most of the material in this section has been accumulated from two references: Grandjean and Vigliani[8] and the JEDEC phosphor catalogue.[11]

IMPLEMENTATION

To implement a bitblt graphics library, several things must be assembled: a set of data types, a way to specify the Boolean operations provided by the graphics primitives, and the primitives themselves. In choosing the data representations and implementing the primitives, some simplifications of our formalism will be necessary or at least desirable; the full generality of the formal definition of bitblt is difficult to provide in an efficient implementation, and efficiency is a vital issue. For concreteness, this discussion will assume that the destination machine is a Motorola MC68000 microprocessor,[17] although the next section will describe briefly the implementations on other machines.

Our implementation language is C, which is widely known, high enough level to be convenient, but with low-level operators that allow a complete, portable bitblt to be written entirely in C. Nonetheless, we will rely on a couple of relatively recent changes to the language (they are in all the UNIX C compilers from the UNIX Seventh Edition onwards): enumerated types and structure assignment. Their names and their use in the example programs should be an adequate explanation.

The MC68000 is a 16-bit machine with 32-bit addresses, 8 32-bit wide data registers and 8 32-bit address registers. Although there are some 32-bit instructions, the microprocessor fetches data 16 bits at a time, so the types used for `bitblt` graphics will be built around 16-bit integers. More specifically, we will assume the C compiler interprets the built-in integer type `int` as 16-bit numbers. The first data type is a word of memory:

```
typedef unsigned int Word;
```

A `Word` is a 16-bit integer without sign. The bitmap memory, including the display, is a single one-dimensional array of `Words`.

Pixels are addressed by the data type `Point`:

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

`x` increases to the right, `y` increases down the display. Note that coordinates can be negative, and are integers, not reals. The `bitblt` graphics primitives are strongly oriented towards the hardware. `Points` label pixels in the plane; as in our formal definition, the `Point` (x, y) labels the pixel centered at $(x+½, y+½)$.

A `Rectangle` defines a region in a bitmap, and is specified by two `Points`:

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

Because pixels lie between lattice points, 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.

The `Bitmap` data type describes the storage for a rectangular image:

```
typedef struct {
        Word     *base; /* pointer to word that includes
                                the Point rect.origin */
        Rectangle rect; /* coordinates and size */
} Bitmap;
```

`Bitmap.base` points to the storage itself; we will return later to the details of how the image is held in memory. `Bitmap.rect`, related to the bounds operator in the formalism, serves two purposes. First, and more obvious, it defines the size of the image by marking its two-dimensional 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. Since all the graphics primitives take an argument `Bitmap` to describe where to draw the image

— the display is not the default — they clip to the region defined by `Bitmap.rect`. Note that because a `Bitmap` may have arbitrary width, the clipping edges might not lie on `Word` boundaries; see Figure 16.

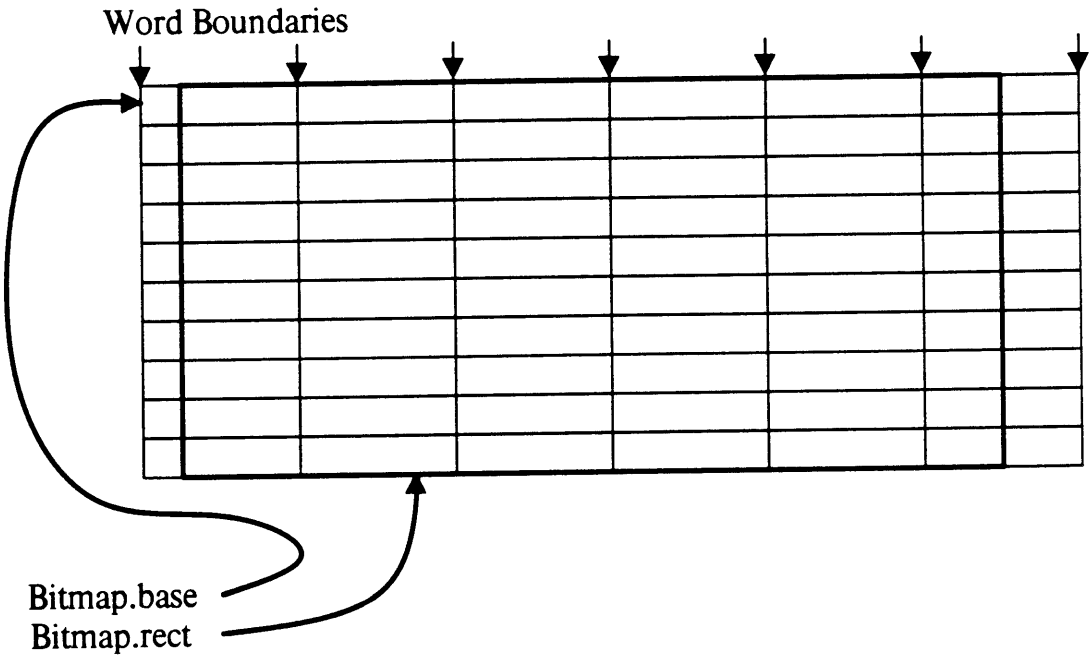

Figure 16. The `Bitmap` data structure associates a coordinate system, defined by `Bitmap.rect`, with the contiguous storage beginning at the `Word` pointed to by `Bitmap.base`. The vertical edges of the `Bitmap` need not lie on `Word` boundaries.

The final data type is a `Texture`:

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

Although our formal definition allows arbitrary sizes for textures, most implementations, including this one, define a texture to be a square of size the width of a word. Since the word size on the MC68000 is 16 bits, `Textures` are an array of 16 16-bit words, or a 16×16 array of bits. 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.

Most of the graphics primitives are, at least implicitly, two-argument operators. For example, a line-drawing primitive can be regarded as having two `Bitmap` operands: the `Bitmap` in which the line is to be drawn, and an imaginary `Bitmap` set to 0's except for those pixels approximating the line segment being drawn. To draw the line in the image, we must specify the Boolean combination of the source and (original) destination that will result. In our formalism, this is expressed as

image op ⇐ line *point1 point2*

Given two operands, the truth table of the Boolean operator is two by two, so there are 2^{2^2} or 16 possible Boolean operators, which can be numbered by the 4 bits of the truth table. Some of these functions are degenerate, of course: 0 sets the destination to all zeroes, and 15 to all ones, independent of the source (in our example, the bitmap with the line segment). Although all 16 functions are meaningful, only a few are commonly used in practice, and the degenerate ones can be subsumed using textures such as *ALL0* and *ALL1*. The common ones are given

mnemonic names, and are easily expressed (at the `int` level) in most conventional computers' instruction sets and in C:

```
typedef enum {STORE, OR, AND, CLR, XOR} Code;

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

The C syntax `a`*op*`=b` is equivalent to `a=a`*op*`b` but evaluates `a` only once. `STORE` mode assigns the source to the destination. `OR`, `AND` and `XOR` perform the simple Boolean operation on the source and destination and place the result in the destination. `CLR` is the same as `AND` with the source bitwise complemented.

Although `bitblt` is the most important bitmap operator, it is worth introducing a few other low-level operators for drawing in bitmaps. The first is the simplest:

```
point(b, p, c)
        Bitmap b;
        Point p;
        Code c;
```

changes, according to the Boolean code `c`, the pixel at point `p` in the `Bitmap b`. For example,

```
point(display, pt, XOR);
```

would flip the state of the pixel at x=`pt.x`, y=`pt.y` on the display. Not all the Boolean codes are meaningful for this operator; `STORE`, for example, yields an undefined result. Although it is possible to decree some meaning, it is not important enough to define here.

This example introduces a few conventions. First, the globally defined `Bitmap display` is the descriptor for that portion of memory visible on the screen. Second, all structures are passed by value, not by reference. Although the data structures `Bitmaps` and `Textures` are large and seldom change value, for consistency and simplicity we will pass all arguments by value in the examples (this does not mean passing the `Bitmap`'s pixels, of course, just its descriptor). In practice, `Bitmaps` and `Textures` might be passed by reference, but geometric data structures such as `Point` and `Rectangle` would always be passed by value to the primitives, because they are much more easily used that way. The programs below illustrate this.

A more interesting primitive draws approximate line segments:

```
line(b, p1, p2, c)
        Bitmap b;
        Point p1, p2;
        Code c;
```

draws the line segment from `p1` to `p2` according to Boolean code `c`. The line includes `p1` but excludes `p2`, to simplify the drawing of curves built from contiguous line segments. Again, we will not define what `STORE` mode means for line drawing, although it should be the same as for `point`. The implementation of `line` is discussed in a later section.

Finally, we come to the most important operator: `bitblt`. Our definition for it is:

```
bitblt(src, pt, txt, dest, rect, c)
        Bitmap src, dest;
        Point pt;
        Texture txt;
        Rectangle rect;
        Code c;
```

copies the rectangle in `src` conformable to `rect`, with origin `pt`, to the rectangle `rect` in `dest` according to the Boolean code `c`. The texture `txt` is aligned with the coordinate system of the source and and'ed with the source data before being applied to the destination. In our formal notation, this may be written

```
dest cut rect c ⇐ src with txt from pt
```

`dest` and `src` are `Bitmaps`; `txt` is a `Texture`. This is a reasonable, practical subset. The problems of default operands will not be discussed here, as C provides no syntax for defaulting.

Here are the factors that make `bitblt` difficult and expensive to implement: First, and most important, it must move an enormous amount of data. To scroll a window half a screen high on a typical display, almost a million pixels must be moved. Even if the inner loop moves a word at a time, it must move tens of thousands of words to scroll the screen once, yet it might be called to do so several times a second. Very few computer programs have loops that typically execute as many times.

Second, `bitblt` must check its operands, and clip the source and destination rectangles if required. We take it as a tenet that `bitblt` clips: if it does, no other software need think about clipping. Although clipping the source is easy, clipping to the destination `Bitmap` is slightly harder because the rectangle is translated to a different coordinate system; the clipping must be done in a coordinate system different from that in which the rectangle is defined. (This is just an offset, but it adds to the complexity).

The source and destination `Bitmaps` may contain the same storage, or even be the same `Bitmap`. `bitblt` must therefore copy the data in any of four directions to avoid overwriting the source information before it is copied.

The multiplicity of Boolean codes can complicate the program, although it depends greatly on the CPU and display hardware. For example, `OR` and the other ordinary Boolean operators are usually available as single instructions in the CPU, but `STORE` must be handled carefully if the operand is not a full word long; more on this below.

Finally, the texturing capability introduces another level of complexity. Each of these complications causes a factor of two or more in the number of distinct cases `bitblt` must handle, all of which must be executed with utmost speed. But before worrying about speed, let's examine a simple, very slow, but correct implementation of `bitblt` that illustrates the complexity of the operator even without concerns of performance.

The data declarations for this `bitblt` are slightly extended from our simple ones above:

```
#define WS      16      /* Word size in bits */
#define HIBIT   ((unsigned)1<<(WS-1))

typedef unsigned int Word;

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

typedef struct Rectangle {
        Point origin;
        Point corner;
} Rectangle;

typedef struct Bitmap {
        Word *base;
        int width;      /* bits across Bitmap, word edge to word edge */
        Rectangle rect;
} Bitmap;

typedef struct Texture {
        Word bits[WS];
} Texture;

typedef enum {STORE, OR, AND, CLR, XOR} Code;

typedef struct Bitptr {
        Word *word;
        int bit;
} Bitptr;

#define ltptr(p, q) (p.word<q.word || (p.word==q.word && p.bit<q.bit))

#define rot(w, n) (((w)<<(n)) | ((w)>>(WS-(n))))

int tmp;
#define mod(m, n) ((tmp=(m)%(n))>=0? tmp: (n)-tmp) /* in case % is not mod */
```

The main addition is the `width` field to the `Bitmap` structure: the number of bits in a scan line of the bitmap, rounded up to `Word` boundaries. This simplifies line-to-line addressing, and has a couple of other advantages discussed later. The `Bitptr` data type describes a bit address: a `word` address and a `bit` number within the word (on the 68000, bit 0 is the most significant (leftmost) bit). `Bitptrs` are used within `bitblt`, but are unknown to its clients. The last few lines of the declarations define a couple of macros that are basically simple operators that are best done in-line. `ltptr` compares two `Bitptrs` and returns true if the first `Bitptr` is a lower address than the second. (In C, `||` is Boolean `OR` and `&&` is Boolean `AND`; the bitwise operators are `|` and `&`.) `rot` rotates its first argument left by the number of bits specified by its second argument, which must be positive. It uses the shift operators `>>` and `<<`; C has no built-in bit rotate operator. The `mod` operator is defined because C's `%` operator may yield negative results when given negative operands.

The procedure `incptr` (`decptr`) advances (retards) the `Bitptr` addressed by its first argument by the number of bits specified by its second argument, which must be a `long` integer because there may be more than 2^{15} bits in a `Bitmap`.

```
incptr(p, n)
        Bitptr *p;
        long n;
{
        p->word += n/WS;
        if((p->bit+=n%WS) >= WS)
                p->bit -= WS, p->word += 1;
}
decptr(p, n)
        Bitptr *p;
        long n;
{
        p->word -= n/WS;
        p->bit -= n%WS;
        if(p->bit < 0)
                p->bit += WS, p->word -= 1;
}
```

The declaration

```
Bitptr *p;
```

declares `p` as a pointer to a `Bitptr`, which is dereferenced as `*p`. The notation `p->word` is equivalent to `(*p).word`.

Here is `bitblt`; `clip`, `rowbltneg` and `rowbltpos` are described below :

```
bitblt(map1, point1, text, map2, rect2, code)
        Bitmap map1, map2;
        Point point1;
        Texture text;
        Rectangle rect2;
        Code code;
{
        Bitptr p1, p2;
        int width, height;
        clip(map1.rect.origin.x, map1.rect.corner.x,
                &point1.x,
                 map2.rect.origin.x, map2.rect.corner.x,
                 &rect2.origin.x, &rect2.corner.x);
        clip(map1.rect.origin.y, map1.rect.corner.y,
                &point1.y,
                 map2.rect.origin.y, map2.rect.corner.y,
                 &rect2.origin.y, &rect2.corner.y);
        width = rect2.corner.x - rect2.origin.x;
        height = rect2.corner.y - rect2.origin.y;.
        if(width<=0 || height<=0)
                return;
        p1.word = map1.base, p1.bit = mod(map1.rect.origin.x, WS);
        incptr(&p1, point1.x - map1.rect.origin.x +
                map1.width*(long)(point1.y - map1.rect.origin.y));
        p2.word = map2.base;
        p2.bit = mod(map2.rect.origin.x, WS);
        incptr(&p2, rect2.origin.x - map2.rect.origin.x +
                map2.width*(long)(rect2.origin.y - map2.rect.origin.y));
        if(ltptr(p1, p2)) {
                incptr(&p1, height*(long)map1.width + width);
                incptr(&p2, height*(long)map2.width + width);
                if(ltptr(p2, p1)) return;  /* overlap, unequal widths */
                point1.x += width;
                point1.y += height;
                while(--height >= 0) {
                        decptr(&p1, (long)map1.width);
                        decptr(&p2, (long)map2.width);
                        point1.y -= 1;
                        rowbltneg(p1, p2, width,
                                rot(text.bits[mod(point1.y,WS)], mod(point1.x,WS)), code);
                }
        } else {
                while(--height >= 0) {
                        rowbltpos(p1, p2, width,
                                rot(text.bits[mod(point1.y,WS)], mod(point1.x,WS)), code);
                        incptr(&p1, (long)map1.width);
                        incptr(&p2, (long)map2.width);
                        point1.y += 1;
                }
        }
}
```

The calls to `clip` clip `x` and then `y` so the source and destination rectangles are both contained in their respective `Bitmaps`. `clip` deals with one coordinate at a time, and takes into account the implicit change in coordinate system between the source and destination:

```
clip(map1o, map1c, ppoint1, map2o, map2c, prect2o, prect2c)
        int *ppoint1, *prect2o, *prect2c;
{
        int t;
        t = *ppoint1 - map1o;
        if(t<0)
                *ppoint1 -= t, *prect2o -= t;
        t = *prect2o - map2o;
        if(t<0)
                *ppoint1 -= t, *prect2o -= t;
        t = *prect2c - map2c;
        if(t>0)
                *prect2c -= t;
        t = *ppoint1 + (*prect2c - *prect2o) - map1c;
        if(t>0)
                *prect2c -= t;
}
```

After clipping, `bitblt` determines the size of the rectangle to be copied; if the rectangle is degenerate, `bitblt` returns immediately. The `Bitptrs` `p1` and `p2` are initialized to point to the upper left (`origin`) bit in each `Bitmap`. The large `if` statement checks the direction of copy; if `p1<p2` (the destination address is greater than the source), it may be necessary to copy the rectangles starting from the lower right. Notice that only one test is needed for the two dimensions, not one for each dimension, since memory is really a one-dimensional bit string. The simplest way to characterize the direction of copy is that the inner loops must traverse the bitmaps in the direction defined from the origin of the destination rectangle to the origin of the source rectangle.

The `while` loops, one for each direction of copy, then call `rowbltpos` (or `rowbltneg`) to `bitblt` a single scan line, adjusting the `Bitptrs` one scan line each time through the loop.

The loop in `rowbltpos` runs across the scan line:

```
rowbltpos(p1, p2, n, tword, code)
        Bitptr p1, p2;
        Word tword;
        Code code;
{
        while(--n >= 0) {
                if(tword & HIBIT)
                        movbit(p1, p2, code);
                incptr(&p1, (long)1);
                incptr(&p2, (long)1);
                tword = rot(tword, 1);
        }
}
```

If the high bit of the `Texture` word, `tword`, is set, `movbit` is called to execute the appropriate bit copy, otherwise, the destination bit is unaffected.

Here is `movbit`:

```
movbit(p1, p2, code)
        Bitptr p1, p2;
        Code code;
{
        Word t = ((*p1.word<<p1.bit)&HIBIT)>>p2.bit;
        switch(code) {
        case STORE:
                *p2.word &= ~(HIBIT>>p2.bit);
                *p2.word |= t;
                break;
        case OR:
                *p2.word |= t;
                break;
        case XOR:
                *p2.word ^= t;
                break;
        case AND:
                *p2.word &= t;
                break;
        case CLR:
                *p2.word &= ~t;
        }
}
```

The mysterious assignment

```
t = ((*p1.word<<p1.bit)&HIBIT)>>p2.bit
```

does two things. `(*p1.word<<p1.bit)&HIBIT` selects the source bit by shifting it into the 0'th bit of the word and masking out the bottom 15 bits. `>>p2.bit` then shifts the bit into the correct position in the destination word. Therefore, `t` is either all zeroes or a single bit at the destination bit position. The `switch` statement is then straightforward: it simply executes the correct Boolean operation between `t` and the destination bit.

This version of `bitblt` is as slow as it looks (maybe slower!): scrolling an 800×1024 pixel `Bitmap` — a typical display — horizontally takes about 8 minutes using an 8MHz MC68000. The obvious first optimization is versions of `rowbltpos` and `rowbltneg` that work internally, without calling `movbit`, and operate a word at a time:

```
#define mask(n) (~(unsigned)0 << (WS-(n)))
rowbltpos(p1, p2, n, tword, code)
Bitptr p1, p2;
Word tword;
Code code;
{
        Word t;
        int w;
        while(n > 0) {
                t = ((p1.word[0]<<p1.bit) | (p1.word[1]>>(WS-p1.bit))) & tword;
                w = p2.bit + n;
                if(w < WS) {            /* right end */
                        t = (t&mask(n)) >> p2.bit;
                        if(code == STORE)
                                *p2.word &= mask(p2.bit) | ~mask(w) |
                                        ~rot(tword, p1.bit);
                } else if(p2.bit > 0) { /* left end */
                        t >>= p2.bit;
                        if(code == STORE)
                                *p2.word &= mask(p2.bit) | ~rot(tword, p1.bit);
                } else if(code == STORE)
                        *p2.word &= ~tword;
                switch(code) {
                case STORE:
                case OR:
                        *p2.word |= t;
                        break;
                case XOR:
                        *p2.word ^= t;
                        break;
                case AND:
                        *p2.word &= t;
                        break;
                case CLR:
                        *p2.word &= ~t;
                }
                w = WS - p2.bit;
                incptr(&p1, (long)w);
                incptr(&p2, (long)w);
                tword = rot(tword, w);
                n -= w;
        }
}
```

The operation of the inner loop is to construct the source word aligned with the destination (in general, this will be made from two source words), and then the `switch` statement is like that in `movbit`, but operates on a complete word.

This code has a couple of problems, typical of the sorts of architectural dependencies `bitblt` excites. First, it assumes that shifting by `WS` bits works, but on some machines (not the MC68000), shifts are interpreted modulo the word size. Also, the program may access the first word *after* the destination bitmap, which may not exist.

Performance is better; scrolling the big `Bitmap` takes about 30 seconds, 16 times less than the pixel-at-a-time version. There is still much to be gained, though: the Blit `bitblt` implementation described in the next section, running on the same hardware, scrolls the screen horizontally in 0.36 seconds. The next section exposes the remaining factor of 100.

PERFORMANCE ISSUES

In this section, we will discuss how to make bitblt as fast as possible. Offhand, this might be addressed as two main subtopics: improvements to the software and improvements to the hardware. Although there is certainly speed to be gained by improving both hardware and software, the main lesson is that ultimate performance comes through cooperating hardware and software. Software that doesn't take advantage of the hardware is clearly weak, but hardware designed without the eventual software in mind can be just as ineffectual.

Before getting involved, we must ask the question: what must be fast? The heart of bitblt-style bitmapped graphics is bitblt, so obviously it must be fast. It may be possible, however, to optimize special cases appropriate to the intended use of the display. Tradeoffs will arise, and they should be resolved to optimize dynamic (actual) rather than static (benchmark) performance. If the display is to be used largely for textual work, bitblt might be designed to optimize character drawing and related processes such as scrolling. If instead the display is to be part of a CAD system, line-drawing and area-filling performance might be critical. For some systems, small-area bitblts might be enormously more common, while another window-based terminal might do many large copies.

These tradeoffs do not compromise the idea of bitblt itself. Although particular cases of bitblt might perform better than others, the conceptual simplicity is the same. Bitblt gives us a focus for implementation issues, and therefore is the place to concentrate design and tradeoff decisions. In what follows, we will examine various techniques that can be applied to the optimization of bitmap graphics, then see how the tradeoffs interact by describing a few commercial systems and their implementations of bitblt.

As the pedagogical implementation of bitblt above showed, it is advantageous to process as many bits as possible in the inner loop. Since the video memory is usually organized into horizontal scan lines, this means operating a word at a time, and making the word as wide as possible. For example, although the MC68000 is a 16-bit machine, it is possible to address 32 bits of memory in a single instruction, and the simple bitblt could gain a factor of two in some cases were it to exploit this feature (provided that the cost of determining that the feature can be used does not outweigh the gains achieved by exploiting it).

Hardware features can have great influence on performance. Memory speed is clearly important; to scroll the screen of a typical bitmap display requires reading and writing about one quarter megabyte, which takes a significant fraction of a second on medium-sized computers, so faster memory will result in visibly faster display update. For bitblts in which the source and destination are not bit-aligned, considerable time may be spent rotating the source words into alignment. The MC68000 takes time proportional to the shift amount to align the bits; other machines have a barrel shifter and execute all shifts in the same small time. Still others are bit-addressable in some way, so that the issue of alignment has no bearing on the instructions in the inner loop.

Another obvious improvement is simply speeding up the CPU. This is usually outside the direct control of even the hardware designer, but system-level decisions can influence the effective speed: bus bandwidth, memory latency, caching† and the like will all influence the speed of bitblt by increasing the 'picture bandwidth' through the processor.

One technique of speeding up the CPU is to implement special instructions such as bitblt or a one scan line version of it in micro-code. The machines developed at Xerox make good use of their micro-code for bitmap graphics and other resource-intensive applications.

Many calls to bitblt move relatively large amounts of memory, and therefore execute the inner loop thousands of times, in fact, many times compared to almost all loops in computer programs. If the inner loop is executed 100,000 times on a microsecond-per-instruction machine, each instruction in the inner loop will consume 0.1 seconds of real time. It is therefore worthwhile to execute the tightest loop possible, even at the expense of increased setup

† Bitblt wreaks havoc on a cache. The short shrift given the issue here is disproportionate to its importance.

time. One rarely-used but highly successful technique is to compile high-quality code for the loop on the fly when bitblt is called. Bitblt on the Blit actually generates *optimal* code, and is discussed in detail below. This technique can also work very well for lines and other curve-drawing primitives.

At the other extreme, small bitblt calls, such as to draw characters, are dominated by setup rather than by moving the data: drawing a typical character on a 16-bit machine probably requires changing fewer than 10 words of memory. This is one of bitblt's problems, in fact: the great unification in programming is at the expense of an operator that must perform well over an astonishingly large domain. The complexity of the operator itself may be a controlling issue in its design and implementation. The nature of the operator requires considerable complexity and difficulty of a good implementation, and it is therefore advantageous to design the hardware and software around bitblt so as to minimize or even reduce the complexity.

There are sometimes special cases that can be applied to simplify bitblt and perhaps improve performance without seriously reducing its generality. For example, most implementations, like ours, assume a fixed size for textures — the word size of the machine. Although a dynamic size might be more useful, the functionality is not in practice reduced significantly by decreeing a convenient size.

A more interesting example is a special case exploited, without loss of generality, in the Blit implementation. Many bitmaps are dynamically allocated to hold a piece of the display memory temporarily, while something like a window or menu covers it up. By observing that the rectangle the bitmap is copied from is often the rectangle it will copied *to* later, the bitmap can be allocated so the word boundaries are aligned with those in the source bitmap (and therefore the later destination) (see Figure 17). This allows bitblt to copy the temporary bitmap without shifts or rotates, which can save considerable time on a machine without a barrel shifter, such as the MC68000. This trick saves a factor of two or more on a Blit, and also simplifies the software somewhat. By demanding that *all* bitmaps be aligned so pixels at $x=0 \bmod 16$ are on word boundaries, bitblt can exploit that fact to convert bit addresses into word/bit pairs easily, and it is straightforward to arrange that textures (which are 16 bits wide) are also aligned with the coordinate system in the bitmap.

Hardware can have an overwhelming influence on the complexity of bitblt. Perhaps the single greatest factor is the uniformity of the address space and memory. Even if all memory is directly addressable by the CPU, and all behaves the same way, bitblt must deal with thousands of distinct cases for the two-dimensional copy loop, and the problem in implementing it is keeping the case analysis manageable without degrading performance. If some piece of memory is different — some machines have special properties such as bit addressability— the case analysis increases combinatorially with the number of variations. Also, as will be explained below, the well-meaning designers of hardware sometimes add features intended for performance that actually cause more problems than they resolve. One of the most common is that of making memory two-dimensional (after all, it's two-dimensional on the screen) which makes memory allocation a 2-D bin-packing problem, and makes it impossible to allocate a bitmap that is wider than the display (see Figure 18).

The Blit: A Case Study

Our first analysis is of the Blit experimental bitmap terminal, which is discussed in considerable detail because it was designed primarily as a graphics system to execute bitblt, and because it is well understood (one of the authors (Pike) was half of the design team). Also, it is a very simple, low cost frame buffer with good performance, and it is interesting to see why. Finally, it is an excellent illustration of the main point of this section: if the hardware is cleanly designed, a software implementation of bitblt may be good enough that special purpose hardware is unjustified and unnecessary. Much of this material is excerpted from Pike, Locanthi and Reiser.[21]

The Blit has an 8MHz MC68000 and 256 Kbytes of contiguously addressable dual-ported

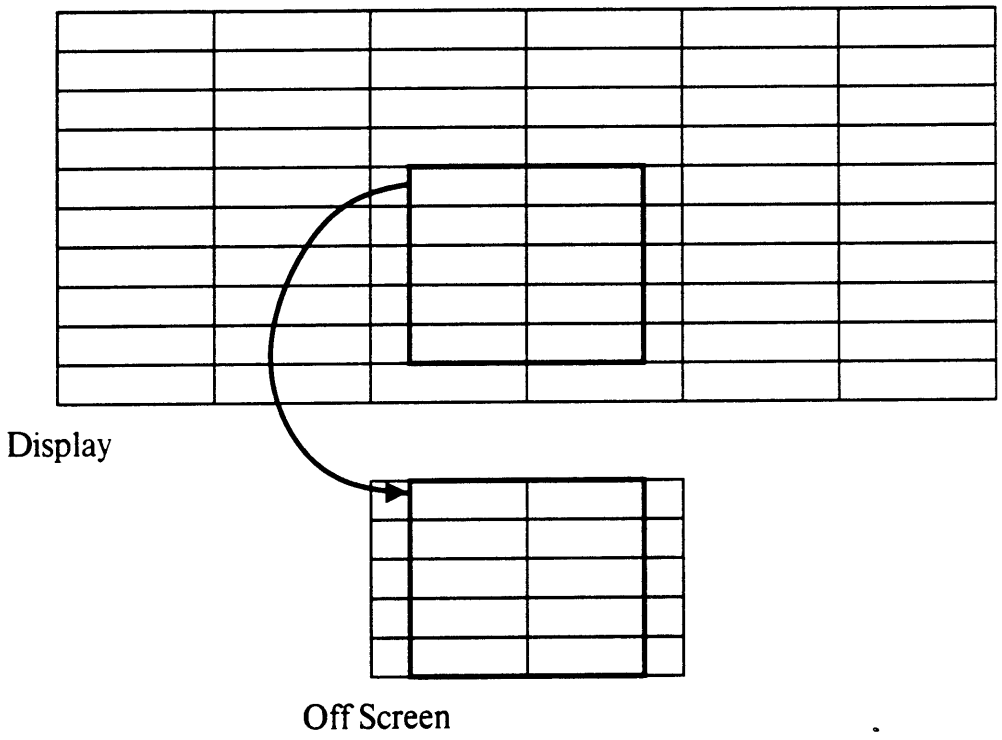

Figure 17. Alignment of allocated bitmaps. Bitmaps created to back up pieces of the display bitmap can be allocated so the word boundaries are aligned, so saving and restoring the display pixels does not involve bit rotates in bitblt.

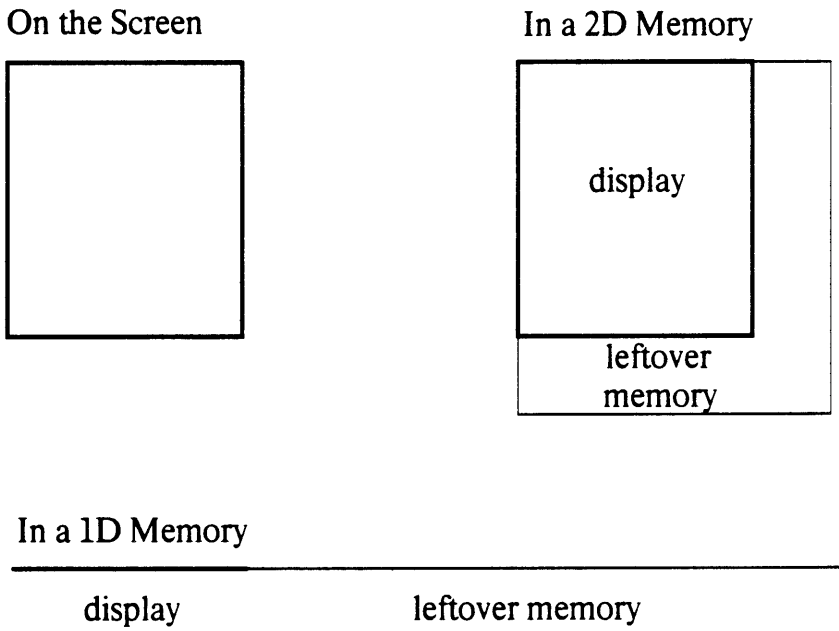

Figure 18. Two-dimensional memory introduces severe problems when trying to use those portions of memory not used by the display — memory allocation requires 2-D bin packing.

RAM, a connected 100 Kbytes of which is scanned to the display to form the 800×1024 pixel image. 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 region of RAM, shared by the processor and the display.

The center of the Blit is the dual-ported memory: 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, so the display is a `Bitmap` as discussed above: the video controller copies an arbitrary connected 100 Kbytes to the 800 pixel wide by 1024 high display. The remaining 156 Kbytes are used for program and data, including auxiliary bitmaps. 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 */
        {0, 0, 800, 1024}           /* 800×1024 screen */
};
```

The single address in `display.base` and the coordinates in `display.rect` are all that is needed for the software to address the bitmap. (One other field, `width`, the number of words across a scan line, is provided for the window system so bitmaps may share storage.) Memory is one dimensional, and any two-dimensionality of interpretation is enforced by software, not hardware.

The Blit is a terminal used largely for character-oriented applications, so it has 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 small — drawing the letter `'a'` in `XOR` mode 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, whose characters are 9 pixels wide and 14 high. 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[17]

```
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, which can be made fast by the alignment trick discussed above. 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.

Bitblt runs two, nested, loops: an outer loop over scan lines and an inner loop across words in a scan line. 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 20. In fact, Figure 20 does not present the most complicated case, because the source and destination can span a different number of words. (The Blit's bitblt does not have a texture operand; texturning is provided by a separate primitive. This is for historical reasons, and could easily be changed.)

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 '2×' (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 small 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.

The number of cases depends on the rules for counting. Table 2 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.

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 2. Cases in generated code for the inner loop for Blit `bitblt`.

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.

Figure 19 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 2×, 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 19. Generated code for scrolling the whole display.

Figure 20 shows the relative alignment of source and destination words for a complicated case. Figure 21 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

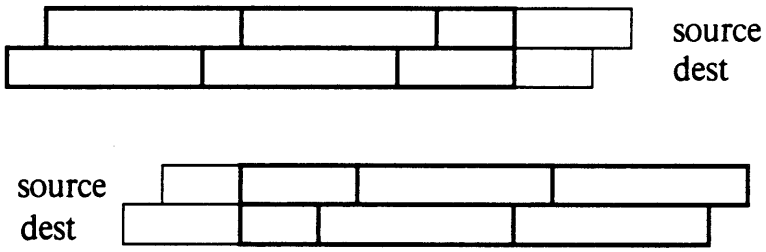

Figure 20. Right edge and left edge for a complicated case.

```
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 21. Code for a complicated XOR.

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 20 and 21), saving 5.5 instructions per scan line. Destination edges coinciding

with word boundaries omit the and. 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. (A comparison of bitblt implementations and systems appears in Table 3.) 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. Despite the lack of direct hardware support, the implementation of `bitblt` on the Blit is entirely satisfactory, and in fact outperforms some systems with hardware assist. A related point is that the Blit's performance is predictable: there are no anomalously fast or slow cases of `bitblt`.

Apollo

The Apollo DN400[5] is an MC68000-based personal computer with an integral 800×1024 pixel bitmap display, refreshed at 60Hz. 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. Bitblt on the Apollo therefore runs at very different speeds depending on its arguments; XOR mode, for example, must be executed by the CPU and is therefore considerably slower than STORE mode. The speed of the bit mover is so high that the invisible portion of the 1024×1024 memory is used to store fonts, so that characters can be drawn very quickly.

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. Unfortunately, we were not able to run our benchmarks on these newer systems.

Dorado

The Xerox D machines are a set of three heavily micro-programmed personal computers with a range of performances. Although they are different in detail, they are similar in concept, varying mainly in performance. For our purposes, therefore, we will consider them a single class of computer. The Dorado[13] and the Dolphin are benchmarked below; the intermediate performance machine, the Dandelion, was not measured.

The processors are based on the idea of micro-tasks: at the start of each micro-instruction, control switches to the highest priority process, in micro-code, that is ready to run. These context switches are essentially instantaneous, and are used to allow each I/O device on the computer to have the full programmability of the CPU at its disposal. For example, one of the higher priority micro-tasks fetches the memory to update the display, which means the memory need not be dual-ported. The lowest priority micro-task executes the user instruction stream, as on a traditional processor.

Bitblt is written in about 300-400 instructions (depending on the machine) in micro-code, and is interpreted as a user instruction by the lowest-priority micro-task. The micro-instruction

words are 32 to 48 bits wide, and can (in a single instruction on the Dorado, two or three on the Dolphin) encode a move, shift offset (through a barrel shifter), test and branch address. This allows the inner loop of most cases of bitblt to be one or two micro-instructions. To all the D machines, the display is simply a part of memory; although the hardware supports more complicated display addressing, all current Xerox software sets the display control blocks so the display is a single connected block of memory.

The Dorado is the highest performance D machine, and in fact the highest performance bitblt processor, largely because of a complex, very high bandwidth memory architecture. Some Dorados have a 512×512 pixel 24-bit deep color frame buffer, refreshed out of main memory by a CPU micro-task that degrades overall performance only about 30%!

Note that although the D machines were designed to execute bitblt efficiently, this was done largely by optimizing CPU and memory performance, rather than changing the processor or memory specifically for bitblt itself. Nonetheless some features, such as a pre-loadable shift and mask unit on the Dolphin and Dorado, were installed specifically for bitblt.

Ridge

The Ridge 32[6] is a high-performance pipelined RISC processor with virtual memory and an interesting bitmap display architecture. The computer can support up to four bitmap displays simultaneously because of a shadow memory scheme that off-loads the memory bandwidth required for refreshing the displays (about 10MHz per display).

Each display is a DMA device on the I/O bus, with 128K bytes of local memory, from which the 1024×800 pixel display is refreshed. A program running in the Ridge does not access the display directly, but instead changes the contents of a 128K bitmap (using bitblt) in main memory. Because the Ridge has virtual memory, the "page modified" bit can be used by the hardware to detect when a part of memory has been touched. For ordinary data, this may require copying the page to disk in order to swap the process out. For the display pages, however, the backup device is the shadow memory in the display controller. The page tables for the user's main memory copy of the display bitmap are examined at 30Hz, and any modified 4Kbyte pages are then asynchronously copied to the display controller by a DMA request, which moves a 32-bit word every 750ns. Although this requires memory bandwidth, main memory is used to update the display only when the image changes, and only to retrieve those pages that have been touched. In fact, if the display bitmap is not accessed for a while, it may be paged out!

Bitblt on the Ridge is written in assembly language, and executes quickly because of the uniform address space and high performance of the CPU, which can execute up to 8 million instructions per second. Because of the simple design of the instruction set, however, the inner loops of bitblt are somewhat longer than on machines with 'horizontal' micro-code, such as the Dorado, especially for those cases of bitblt with non-aligned source and destination.

Sun

There are two implementations of a bitmap display for the SUN, both with a separate display memory with hardware assistance for shifting, Boolean operations and masking. The first implementation[2] 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^{16} 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 (22MHz 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.

Symbolics

The Symbolics 3600[10] is a micro-coded personal computer designed specifically to support interactive programming in Lisp, and includes an integrated 1150×900 pixel bitmap display refreshed at 60Hz. The frame buffer architecture is fairly simple. A separate memory, slightly larger than the display, is dual-ported between the processor and the video refresh to offload the refresh bandwidth from the main memory. The display memory is, however, regular (but slower) memory in the address space of the CPU.

Bitblt is written in a mixture of micro-code and Lisp. The inner, horizontal loop is written in micro-code, and the outer loop and setup is in Lisp. Symbolics bitblt does not clip or automatically determine the direction of copy. There is somewhat more micro-code for bitblt on the 3600 than on comparable machines such as the Dorado, primarily to take advantage (when profitable) of a special hardware memory access method called 'streaming mode.' This method uses RAM chip page mode access and an eight word cache to improve performance for sequential addressing.

Benchmarks

Four simple tests of were run on these various machines. The tests were:

- Scroll 800wide×1024high bitmap 1 pixel horizontally
- Scroll 800wide×1024high bitmap 1 pixel vertically
- Draw 8wide×7high character in XOR mode at random bitmap positions
- Texturing in XOR mode a 40×40 square at random bitmap position

Although many systems have special primitives for operations such as character drawing, the tests were run using bitblt directly for each test. They therefore indicate the approximate performance of bitblt itself. Most tests were run with source and destination both in regular memory; where results are significantly different in memory vs. the frame buffer, this is indicated in the table.

A note on costs: a Blit costs about $3,500 and the other machines range from $20,000 to $100,000.

Machine	Vert. Scroll	Horiz. Scroll	8×7 Character	40×40 Texture	Code size, notes
Apollo	25	27	1.0[1]	2.6	hardware (disp. to disp.)
Apollo	642	656			software (disp. to disp.)
Blit	130	370	0.41	1.20	1542 bytes instr's[2]
Dolphin	85.2	128.1	2.02	2.40	~300 micro-instr's
Dorado	23.3	23.8	0.051	0.156	~300 micro-instr's
Ridge	39	112[3]	0.244	0.830	5439 bytes instr's
Sun 1	187	194	1.1	1.89	All separate routines
Sun 2	109	110	0.34	0.82	3344 bytes instr's(disp. to disp.)
Sun 2	82.2	311	0.74	1.78	3068 bytes (mem. to mem.)
Symbolics	30.5	36.8[4]	0.52	1.64	~500 micro-instr's[5]

All times are in milliseconds.

[1] 0.06ms in STORE mode (done in hardware). Scrolling done by hardware in display; much slower if source or destination is not display.

[2] Texturing is a separate operator

[3] 67ms scrolling left. The scrolling tests were 1024wide×800high.

[4] This are memory-memory times. The display memory scrolls about 2.5 times slower.

[5] No clipping, programmer must specify direction of copy

Table 3. Bitblt benchmarks. Thanks to Luca Cardelli, Ken Church, L. Peter Deutsch, Tom Duff, Emden R. Gansner, Peter Langston, George Trow and Dave Ungar for testing the various machines.

LINES

The problem of drawing an approximate line segment on a raster device is old but interesting and important. Drawing a line segment is most easily done by the simplest example of a class of curve-drawing algorithms called Digital Differential Analyzers, or DDAs. Other DDAs can draw circles, arcs, ellipses, splines and so on.

The basic problem is, for a line segment between two points (x_0, y_0) and (x_1, y_1), to illuminate those pixels that form the best approximation of the true line (henceforth we shall use the adjective 'true' to distinguish the line from its approximation). Certain compromises must be made, of course. The most obvious is that the line must have thickness at least that of a pixel. Also, at least for the one-bit pixels we are considering, the true line must be approximated by a sequence of horizontal or vertical rectangles with visible jumps, or 'stair steps', where adjacent rectangles abut.

The DDA is a simple algorithm that maintains the minimum distance between the real line and the pixels, measured perpendicular to the axes. The algorithm moves from the start point of the line to the end point one pixel at a time, tracking the distance between the pixel being plotted and the true line. When the error becomes greater than half a pixel width, the subsequent pixel is moved one unit in the direction to return the error term to near zero. The behavior of the error is illustrated by Figure 22.

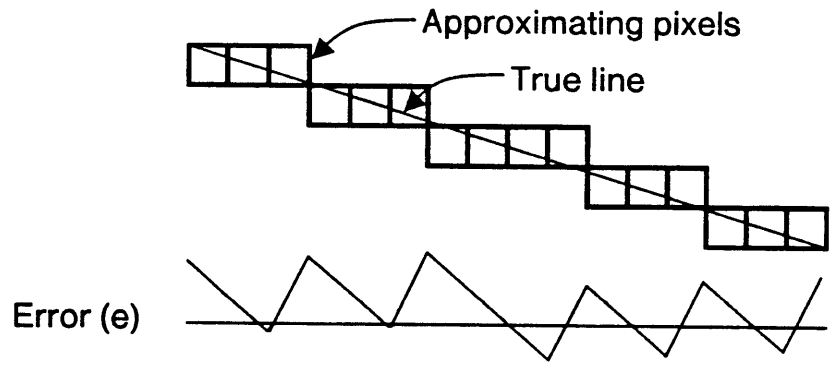

Figure 22. The error term for approximating a line minimizes the distance between the true line and the approximation.

Formally, assume a true line of positive slope $0 \le m < 1$, with $(x_0, y_0)=(0, 0)$ and $x_1=\Delta x$ and $y_1=\Delta y$ integers. The pixel labeled by the point (x, y) is at $(x+½, y+½)$,† so the error at (x, y) is

$$y+\frac{1}{2}-m(x+\frac{1}{2})$$

with

$$m=\frac{\Delta y}{\Delta x}.$$

At the origin, the initial error is

$$\epsilon_0=\frac{1}{2}-\frac{\Delta y}{2\Delta x}$$

so at (dx, dy) the error is

† Most derivations of this algorithm assume the pixel is centered on (x, y). We have chosen a different convention because it makes the notion of a rectangle less ambiguous for `bitblt` graphics. Its only influence on the algorithm is the initial value of e derived later; for pixel centers at (x, y), $e_0=\Delta x-2\Delta y$.

$$\epsilon' = (y + dy + \frac{1}{2}) - m(x + dx + \frac{1}{2}).$$

Setting $dx = 1$, the change in the error between the first pixel, at the origin, and the second pixel is

$$\epsilon' - \epsilon = dy - mdx = dy - \frac{\Delta y}{\Delta x}$$

Rearranging,

$$\Delta x \epsilon' = \Delta x \epsilon - \Delta y + dy \Delta x.$$

Here, dy is either 0 or 1, the choice per pixel determined by the constraint

$$-\frac{1}{2} \leq \epsilon' < \frac{1}{2}.$$

Next, we scale and offset ϵ to form a computationally more convenient error term, e:

$$e = 2\Delta x \epsilon + \Delta x$$

that satisfies

$$0 \leq e < 2\Delta x.$$

Initially,

$$e_0 = 2\Delta x \epsilon_0 + \Delta x = 2\Delta x - \Delta y$$

The loop over pixels in `line` therefore plots a pixel, increments x, subtracts

$$2m\Delta x = 2\Delta y$$

from e, and tests its sign. If $e < 0$, y is incremented and e increased by $2\Delta x$. Simplifying somewhat, e can be tested before it is updated by subtracting $2\Delta y$ from the initial value. This yields the following program to draw a line of shallow positive slope (it is easier to write separate versions for the other orientations than to fold them into one loop):

```
line(b, p1, p2, c)
        Bitmap b;
        Point p1, p2;
        Code c;
{
        int e, Δx, Δy;
        Point p;
        Δx=p2.x-p1.x;
        Δy=p2.y-p1.y;
        p=p1;
        e=2Δx-3Δy;
        for(i=1; i<Δx; i+=1){
                point(b, p, c);
                p.x+=1;
                if(e<0){
                        p.y+=1;
                        e+=2Δx-2Δy;
                }else
                        e-=2Δy;
        }
}
```

This algorithm was first derived by Bresenham.[4] The basic idea of a DDA — approximating a curve by minimizing the error while walking across the display — can be applied to circles, ellipses, splines and any other curve. These are discussed in Newman and Sproull.[19] There is an extensive literature regarding the algorithms for drawing curves on raster displays, and we

probably needn't dwell further on it here. On the other hand, the considerations of efficiency in executing these algorithms are not as widely known, at least in print.

The first observation regarding the efficiency of `line` is that it is unnecessarily expensive to call the subroutine `point` for each pixel to be drawn. But besides the obvious inefficiency of calling a subroutine, if the line is nearly horizontal this method can waste time by missing the opportunity to write several pixels in the same word; the word address might not change between pixels. To address this shortcoming, we would therefore like the DDA to generate multiple pixels per trip through the loop, so that horizontally adjacent pixels can be written in a single memory reference.

A simple idea to speed up line drawing is to draw the line as a set of precomputed line segments of some short, convenient length such as the width of a word. If the segments are chosen from a short 'catalogue,' lines may be drawn very fast, but will not have minimum error at each pixel. In fact, some lines will not even be monotonic. To draw the correct line, the catalogue must be indexed by the (rational) slope of the line and the error term at the first point of the short segment. To precompute this for all lines is prohibitively expensive. Nonetheless, for some applications the lines need not be perfect, and a short catalogue may suffice.

The imperfections must be reproducible, however; one important property that the lines must always maintain on a bitmap display is that the same sequence of pixels is generated for every call to `line` with the same end points, because a line might later by erased by a call to `line` with `Code XOR`. The implementation of `line` described by Pike[20] goes even farther, and allows an arbitrary subsegment of the line to be correctly regenerated or erased by passing the initial value of the DDA to the line-drawing primitive.

Considerable work has been done in trying to write a segment-at-a-time line-drawing algorithm that generates the correct sequence of pixels, but so far no practical method has been found. For more information on these and other algorithmic improvements, see the papers by Sproull[23] and McIlroy.[15]

Optimizations relating more to implementation than algorithms may also be applied to line drawing. One trivial optimization is to write special-purpose code for horizontal and vertical lines. For many applications, these are the most common lines to be drawn, and especially for horizontal lines, special code can execute much quicker than a DDA. Even for general lines, careful code design can avoid the unnecessary memory accesses required to write multiple pixels in the same word. It is possible to compile code on the fly to avoid tests in the inner loop (indeed, McIlroy's work on Farey series was motivated by a design for a compiler for line drawing).

Unfortunately, the speedup to be gained by improving the obvious software for line drawing is not nearly as great as that for improving the obvious bitblt code. The fundamental optimization in bitblt — word at a time processing — does not achieve as much for DDA-driven algorithms, for which most words accessed contain only one pixel of the image (except special cases such as nearly horizontal lines). Unlike for bitblt, special purpose graphics hardware can make a tremendous difference in line-drawing performance, however. DDAs are ideal to run in hardware, and changes to the memory architecture, such as adding cache or making memory two-dimensionally addressable, can reduce greatly the bandwidth necessary to draw a line. These ideas focus on the property of line drawing that successive pixels are not at adjacent addresses, which is not true of bitblt.

BUILDING AN INTERACTIVE GRAPHICS SYSTEM USING BITBLT

Graphics is an essential element of an interactive programming system and any interactive application. People think with images, so good use of images is an aid to the user. Pictures make interaction more effective: the normal rate for reading text is less than 100 characters per second, but the human visual system can process two-dimensional information equivalent to millions of characters per second.

On a graphics display, many styles of object can be drawn, including charts, graphs, diagrams, shaded images, and so on. But these are only output on the display; to interact with a computer, there must be a means for two-dimensional input from the user back to the computer — some form of pointing device to let the user manipulate what the computer draws. With a device such as a pen or mouse, the process of selecting from graphical objects such as text displayed on the screen is natural and rapid. By tracking the pointer with a program that simulates a pen or paintbrush, users can input line drawings and freehand sketches.

A general principle for the design of interactive systems is that

Any object accessible to the user should present itself suitably for observation and manipulation.

The Smalltalk-80 system is a representative display-based interactive computing environment. Figure 1b shows a typical Smalltalk-80 screen, and illustrates the wide range of graphical styles possible on a bitmap display. Rectangular areas of arbitrary size are filled with white, black and various halftone patterns. Text, in various typefaces, is placed on the screen from stored images of the individual characters. Halftone shades are "brushed" by the user to create freehand paintings. Moreover, images on the display may be moved or sequenced to provide animation.

Rectangles

Although it is almost trivial, the simplest bitblt operation — coloring a rectangle black or white — is frequently used in user interfaces for tasks such as erasing a paragraph of text or drawing a command button on the display. Another operation frequently associated with rectangular areas is drawing a border. An elegant way to draw a bordered rectangle using bitblt is first to color a large rectangle black, and then to color an inset rectangle white. It is worth pointing out, however, that this algorithm may be distracting in some applications. If the area being cleared is large, the momentary "flash" of dark then light may be obvious.

Rectangular operations in xor mode can indicate selections. A software switch may be shown to be "active" by reversing the black/white sense of its label on the display. As a more interesting example, a block of text may be reversed to indicate that it has been selected for editing. Dynamic effects can be useful for directing the user's attention. For instance, an invalid entry in a table can be flashed several times while a diagnostic message is displayed to indicate the location of the problem. Such techniques must be used sparingly, though, to avoid creating a system that behaves like a video game.

Convenient Rectangle Functions

There are several rectangle functions useful in bitmap graphics because they simplify the expression of geometric operations. The function *inset* is useful for drawing borders, and for placing images inside an area but inset from the edge. *Inset* yields a circumscribing rectangle if given a negative width argument.

```
Rectangle inset[rect, width]
        Rectangle rect
        integer width
{
        "Return rect inset by width"
        return ((origin[rect]+(width, width)), (corner[rect]−(width, width))
}
```

The *intersect* function returns the rectangular intersection of its two arguments. It is particularly useful for clipping.

```
Rectangle intersect[r1, r2]
        Rectangle r1, r2
{
        "Return the rectangular intersection of r1 and r2"
        return ((max[X[origin[r1]], X[origin[r2]]], max[Y[origin[r1]], Y[origin[r2]]]),
                ((min[X[corner[r1]], X[corner[r2]]], min[Y[corner[r1]], Y[corner[r2]]])
}
```

The function *includes* is a similar computation; it returns true if the point lies within the rectangle. It is useful for determining whether the mouse cursor is pointing at a given object.

```
boolean includes[rect, pt]
        Rectangle rect
        Point pt
{
        "Returns whether pixel at pt is inside rect"
        return X[pt]≥X[origin[rect]] and Y[pt]≥Y[origin[rect]] and
                X[pt]<X[corner[rect]] and Y[pt]<Y[corner[rect]]
}
```

The *merge* function returns smallest rectangle that includes its two argument rectangles.

```
boolean merge[r1, r2]
        Rectangle r1, r2
{
        "Return the rectangular union of r1 and r2"
        return ((min[X[origin[r1]], X[origin[r2]]], min[Y[origin[r1]], Y[origin[r2]]]),
                ((max[X[corner[r1]], X[corner[r2]]], max[Y[corner[r1]], Y[corner[r2]]])
}
```

The *include* function returns the smallest rectangle that includes the argument rectangle and point. This function is useful for computing the bounding box of complex objects.

```
Rectangle include[rect, pt]
        Rectangle rect
        Point pt
{
        "Returns the smallest rectangle that includes rect and pt"
        return merge[rect, (pt, pt)]
}
```

Minus is useful for drawing borders and for decomposing the problem of displaying an image occluded by other rectangular areas. It computes a list of rectangles that cover the area inside its first argument rectangle and outside its second argument rectangle.

```
RectangleList minus[r1, r2]
        Rectangle r1, r2
{
        "Returns the list of rectangles inside r1 and outside r2"
        RectangleList l
        l ← nil
        if not (includes[r1, r2.origin] or includes[r1, r2.corner])
                return RectangleList[r1]
        if X[origin[r1]] < X[origin[r2]] {
                l ← cons[l, (origin[r1], (X[origin[r2]],Y[corner[r1]]))]
                X[origin[r1]] ← X[origin[r2]]
        }
        if Y[origin[r1]] < Y[origin[r2]] {
                l ← cons[l, (origin[r1], (X[corner[r1]],Y[origin[r2]]))]
                Y[origin[r1]] ← Y[origin[r2]]
        }
        if X[corner[r1]] > X[corner[r2]] {
                l ← cons[l, ((X[corner[r2]],Y[origin[r1]]), corner[r1])]
                X[corner[r1]] ← X[corner[r2]]
        }
        if Y[corner[r1]] > Y[corner[r2]]
                l ← cons[l, ((X[origin[r1]],Y[corner[r2]]), corner[r1])]
        return l
}
```

Note that these functions all depend on geometric objects being data structures that are passed to and returned from procedures; consider how awkward they would be if a rectangle was specified by four integers.

Horizontal and Vertical Lines

Horizontal and vertical lines are worthy of special mention because they are very common and easy to draw with bitblt. The principal use of such lines is probably as rectangle borders, but they are also used to underline text, draw axes, and make connections in diagrams. As an example, the following code uses some of the rectangle functions above to draw a border surrounding a rectangle R in a bitmap B:

```
for r in minus[inset[r, −2], R]
        B cut r ⇐ ALL1
```

Lines and Curves

Lines and curves can be drawn by the application of a "brush" along a path. (See Figure 23.) The simplest such path is the sequence of points generated by a line-drawing primitive. The simplest implementation simply draws the brush shape at every point along the path, but this fails in xor mode if the brush is larger than a single pixel because the overlapping regions of adjacent instantiations of the brush will interfere with each other. To handle this problem, differential brush shapes can be precomputed for each possible offset from one brush location to the next. These differential shapes are the non-overlapping area of two adjacent brushes. The generalized line-drawing code chooses the appropriate brush shape for each point, depending on its location relative to the previously drawn location.

Besides lines, DDA's can also draw circles and other curves such as quadratic and cubic

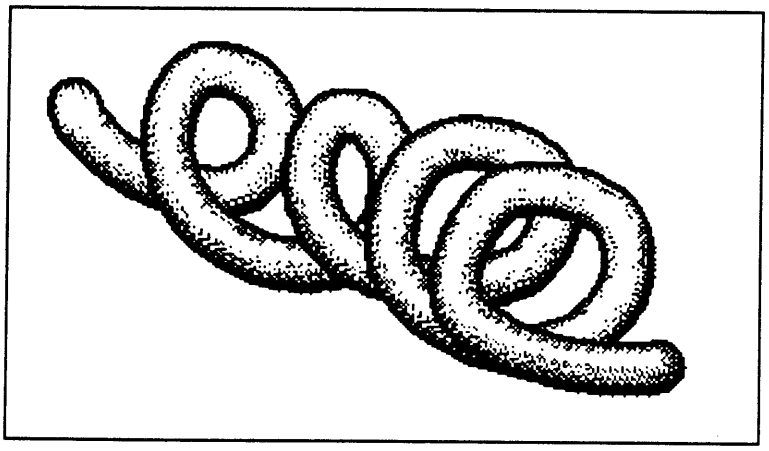

Figure 23. A curve drawn by dragging a differential brush along a path.

splines. The main issue in choosing which type of spline is control. Some splines are specified by the points they touch, others by the values of the tangents at their endpoints. Which is most useful depends on the application. Metafont[12] draws curves by dragging brushes along curved paths, even brushes that change shape along the path. This has proven to be an effective simulation of drawing graceful characters with pen and ink, independent of the size of the character. An attractive property of this method is that a family of fonts can be generated by changing the brush shape between fonts.

Characters and Text

The presentation of text has been a dominant mode of computer communication ever since the first teletype was connected to a computer. With the advent of electronic displays, the bandwidth rose from ten characters per second to thousands of characters per second, and it became possible to make changes to the display incrementally instead of retyping the changed lines. Because of limitations of the hardware, character sets on early video terminals were very ugly. This has changed.

Bitblt simplifies computer display of text. Because it handles the bit-alignment and clipping problems in one place, it makes it as easy to support multiple fonts and bit-aligned justification as it is to display ugly fixed-width characters.

The basic operation in displaying text with bitblt is to copy a rectangle containing a character in a font bitmap to a displayed bitmap. This process is illustrated in Figure 6. A font contains a bitmap with all the character glyphs in it, together with a structure that gives the coordinates for each character, indexed by their character code. Such a storage format is quite compact, as the characters can be packed next to one another regardless of their variation in width. Even more compactness can be achieved by separating characters of different heights, but the added complexity of decoding is seldom worth the saving.

Text Alignment

Text display is an intricate subject, but there are instructive simple examples. For instance, consider the problem of drawing text in the four different alignments: flush along the left margin, flush along the right margin, centered between margins, and padded to be flush along both left and right margins. Figure 24 illustrates these processes.

The first step is to scan the text, measuring the widths of each character to locate the first word that will not fit within the margins. Given this position in the text, the line can be ended at a natural word break, and the amount of extra space, or "padding," is known. For normal

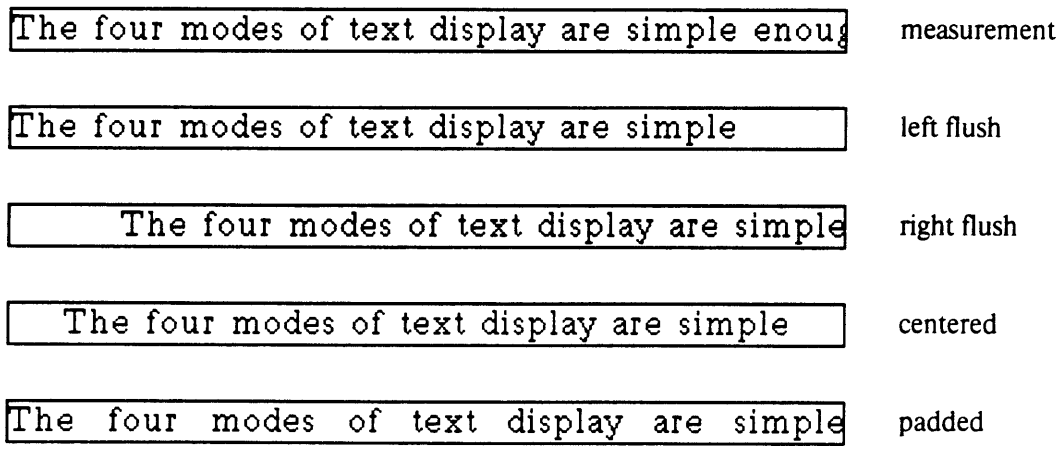

Figure 24. Text alignment.

left-flush text, it is sufficient to display the text up to the final word break; all padding appears after the last word. For right-flush display (often used in columns of figures and for labeling axes), it is sufficient to space over by the padding distance prior to displaying the text. For text centered between the margins, half the padding is placed at each end of the line. Finally, to achieve flush margins at both the left and right side, the padding must be distributed evenly over the internal spaces in the line, as shown in the figure.

There are many other details that crop up when supporting full text formatting. These include tab stops, handling of double spaces at sentence ends, and especially the ability to use the mouse to select characters. Pointing at characters, since it is a fairly low-bandwidth operation, can most simply be done by executing the text display code and checking for the cursor coordinates at each step.

Text Emphasis

There are some simple tricks using bitblt to derive approximate bold, italic and other fancy fonts from a simple base font. Figure 25 shows text in five different emphases, all synthetically derived from the same font. The bold text is made by or'ing the image over itself shifted one bit horizontally. Italics can be produced by sliding horizontal strips of the text over different amounts. Underlining is trivial, of course. A line above the baseline can indicate text that has been struck out but is still a part of a document. Several more decorated versions of a font can be produced by more use of bitblt — the outlined text shown is produced by or'ing up, down, left, right, and then and'ing the complement of the original text to leave white bits in the middle of the smear.

Synthetic emphasis of text is simple to compute and economical of memory but does not produce high quality fonts. The problem of creating good fonts is aggravated by the combined filtering of the display tube (the pixels are not perfect squares) and of the eye, so that the best representation of a character is sometimes a surprising pattern of dots that is unintelligible when magnified. Finally, bold and italic fonts tend to have character widths different from their base font. The last sample in Figure 25 shows a true bold font for comparison with its synthetic counterparts.

In the discussion of brushed line-drawing, we noted that it was sometimes necessary to compute differential brush shapes to produce the desired effect. Emphasized text sometimes requires the same sort of processing to avoid interaction with any background behind the characters.

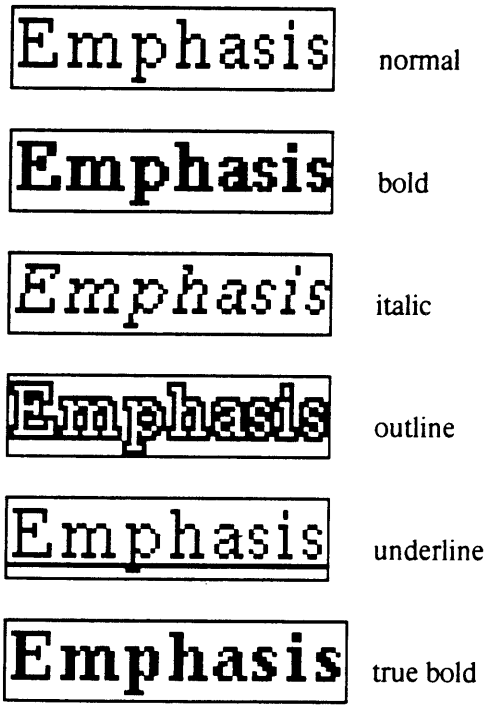

Figure 25. Text emphasis.

What You See is What You Get

For accurate layout of text, it is necessary to use character widths that correspond to the fonts used in printing. Thus the width of a character might be stored both as an integer that is the width of the display font glyph, and as a real or rational number that is the scaling of the true printer width to the display resolution. Although information is lost in rounding off the character positions for display, the loss does not accumulate. The display of small fonts in this "hardcopy mode" looks uneven because of round-off, but it guarantees that character positions correspond closely to printer positions. The central panel in Figure 26 shows this compromise strategy; the crowding of characters compromises legibility as well as ease of pointing.

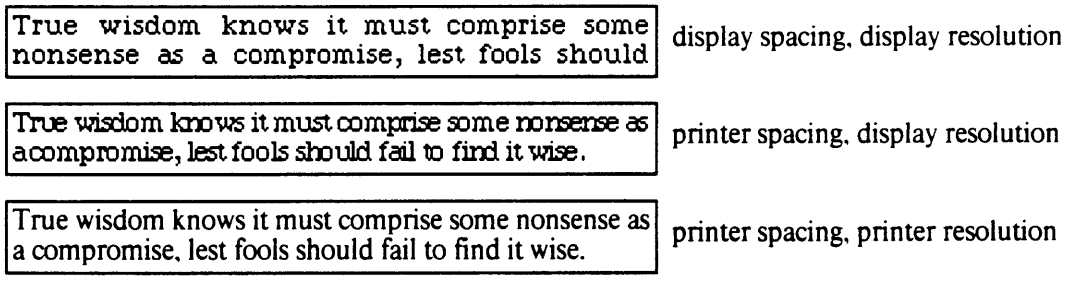

Figure 26. Simulating a high-resolution printer on a bitmap display. The simulation must show the characters at printer spacing but display resolution.

Polygons, Paragraphs and other Compositions

We have seen how dots can be repeated to form lines, and how characters shapes can be repeated to form a line of text. Lines in turn may be repeated to form polygons, and lines of text may be repeated to form paragraphs. The property that must be added to these aggregate

structures is connectivity. With polygons, there is the problem of filling in the interior, recognizing when the cursor is pointing at constituent points and lines, and adjusting adjacent parts properly when a given point or line is altered. With paragraphs, the connectivity issue involves locating characters pointed to by the cursor (find which line is being indicated, then which character within the line), and especially in recomputing the composition of the paragraph when a change has been made by editing. Such structures call for caching of auxiliary state to assist in the incremental changes in display or in the processing needed to detect cursor position in the image. In the case of text, it is common to save an index of character positions at the beginning of every line.

Other images can be replicated to achieve higher-order graphical units. For instance, a circuit diagram is built from a vocabulary that includes the symbols for resistors, capacitors, transistors, and so on. These appear in different locations and orientations, with wire lines, connection nodes and conventional text. Integrated circuit designs are a natural application for the basic bitblt operations, since the world being modeled is heavily based on rectangular areas and spatial replication. Very high speed image generation can be achieved in such applications by storing the bitmap image of a cell, and using that bitmap to draw later instantiations of the cell. Applied recursively, this technique can draw a hierarchically-defined circuit of n rectangles in O(log n) bitblts.

Display Structures

A display structure is a data structure plus support for producing an image and for relating a pointing event to its components. Design of effective display structures is difficult. Just as in choosing effective data structures, the basis set must be chosen to cover the most needs with the fewest, most convenient elements possible. Then the functionality must be arranged to isolate implementation issues from the software above. For example, it is often useful to be able to change the display device (bitmap display, printer, file, etc.) without having to change any of the code that translates from data structure to image. This requires a procedural specification for the generation of images. Bitblt is such a specification at a very low level. We will see later how different interpretations of this specification provide the flexibility to support display in occluded windows. The higher level specifications that form the basis of complete graphics languages are beyond the scope of these notes. Instead, we will concentrate on a representative sample of graphical idioms, and with the mechanics of creating and manipulating such images.

Common Graphical Idioms

If a graphical representation of an object is successful, the images take on features and properties of the objects they simulate.

Figure 1b shows a Smalltalk display from a Dorado. It is full of information encoded and presented in different ways. To begin with, the screen is divided into rectangular windows, each corresponding to a given application program. Several of these not in active use have been collapsed into more compact stylized representations, or *icons*. The basic image components include text in lists and paragraphs, rectangles, borders, lines, hand-drawn images and halftone shading. Within each application, these effects work together to provide a natural model for the user, represented by an idiomatic picture. For example, when a person glances at a screen wanting to know what time it is, a clock face is instantly recognizable and intelligible.

Mouse input is also idiomatic: one quick motion with a pointing device can make a window active, or activate a command, or scroll some text, or whatever is natural to the user.

A window on a display may be simple, divided into parts, or collapsed into a representative icon. Although there may be many types of windows in a system, all windows are accessed (at least in part) by a common graphical syntax with components such as changing the size and shape of a window or browsing through its contents.

An enormous amount of communication is possible using text. In the mail reader in Figure 1b, pointing to elements of the top left list allows the user to choose categories of messages. Within a given category, pointing among the message names in the top right list causes a specific message to be displayed in the bottom of the window. These lists of choices and their layout are the elements that make up the mail reading idiom. The window to the left is similar, but it is used instead for retrieving and modifying programs. The list components are the same, but they have a different meaning — one chooses functional categories, another chooses procedures within the category, and the bottom of the window displays the source code for the procedure selected. The graphical syntax is the same in each case — lists of items to choose from by pointing — but the semantics are different, depending on the contents of the window.

Management of Space

The designer of a user interface must manage the space on the screen. For example, windows may overlap or simply tile the screen. Overlapping windows use the screen area more effectively but can be confusing. Occluded portions of a window may be saved as off-screen bitmaps or regenerated from its display structure when necessary. One approach saves time, the other space. The layers notion[20] is a uniform system for managing occluded windows that takes advantage of the procedural interface provided by bitblt; essentially, it implements windows as generalizations of bitmaps.

Another decision regards elements that are not always visible. Consider the grey rectangle in a white area to the left of the top left list in the mail reader. This is a *scroll bar;* it represents the visible portion of the list of message categories relative to the complete list. This scroll bar is shown only when the cursor is in the window or sub-window in question. When the cursor moves elsewhere, the scroll bar disappears and the background behind it is restored. This allows a cleaner display to be presented; imagine the appearance if scroll bars were shown for all six scrollable parts of this window at once.

Temporary menus are lists of choices that are visible only when invoked by a special button on the pointing device, or when a button is pressed in a reserved spot on the screen. One such menu is shown over the mail reader in Figure 1b. The currently highlighted command will cause new mail to be retrieved if the pointer is released. As with temporary scroll bars, temporary menus offer a cleaner display. In the case of menus that "pop up" at the touch of a button, the cost is a little user effort to access the intended functions. The lists of messages or procedure names in Figure 1b are a sort of static menu — they remain visible, and they are thus easier to use, but they consume screen area all the time. The decision about how to manage such components must be based on an intuitive sense of what model most users will have of the overall set of functions provided.

Incremental Change and Redisplay

Drawing complex images brings up the problem of incremental change: how to avoid recreating an entire image when only one component of it changes. As mentioned above, *ad hoc* techniques are often used for text. For more general images a simple approach is to allow any node of the display structure to cache the entire image resulting from the structure below it. For instance, a fairly complex structure may be needed to represent a fuel gauge or a thermometer complete with numerical graduations. To regenerate the entire image for each change in value measured is expensive. Instead, the image can be assembled as an overlay of background image (meter face) and value-dependent image (meter needle), with the background saved in a bitmap. To update the display for a new measured value, a single copy operation will restore the background, after which the value dependent image can be displayed from its new value.

Image caching is frequently motivated by an attempt to reduce computation. However, dynamic appearance may also be a factor, even when performance is not a problem. For

instance, there may be sufficient computing power to redraw the needle ten times a second, but the dynamic appearance of frequent update may be distracting. In such cases, further buffering can be applied to improve the appearance. The new needle can be drawn in an off-screen buffer, then copied to the display.

There are occasions when even simple copy operations lead to objectionable flashing. This often shows up with small images, because the image update may beat with display refresh, leading to a wavering of the image. In such a case, it is necessary to create the entire new image off-screen. This image can then be copied to the display in a single operation, usually without causing flicker. A common application that requires this attention is moving small images across a display while restoring the background underneath. Sometimes, though, it is even necessary to synchronize the update with the display refresh to eliminate flicker. This is typically done by triggering the final buffer copy by an interrupt supplied by the hardware that tracks the vertical display scan. Most display hardware provides such an interrupt.

Pointing, Input Events, Modes and Cursors

Some pointing devices have several buttons, each of which can have a different interpretation. If used consistently, multiple buttons can greatly simplify some common interactions. It is also possible to interpret multiple clicks of a button. A common example is for a double click within text to request selection of a word or line. Some systems require the two clicks to be within a certain time interval, others interpret a double click regardless of the interval between the clicks.

Another element of a user interface is to indicate modes of interaction such as inking style in a painting program. In some cases it is adequate to highlight a static image representing the mode, but more often the tracking cursor image is used for feedback. The user's attention is always focused on the cursor — it is the one place where feedback is certain to be recognized.

Cursor feedback can be dynamic. One example of this is in editors where the cursor is kept small to minimize interference with the material being edited. It may be useful to flash or otherwise vary the cursor so that it can be seen easily. Another use for dynamic cursor images is to indicate ongoing and preemptive processing. Such situations are always to be avoided, but if some process such as file backup has to complete before other actions can be resumed, then it may be helpful to explain the situation using an animated cursor. Not only does this alert the user to the preemptive activity, but it also provides reassurance that the system is still active, even if it may be momentarily unresponsive.

Opaque Whites

It is complicated to draw non-rectangular images containing black and white pixels so that some of the white pixels appear "opaque" rather than transparent. Some of the 0's in an image are white, and some are simply outside the outline of the image, but there is no way to encode the difference in a one bit deep image. This is the one-bit case of the image compositing discussed by Porter and Duff.[22] To solve this problem, a parallel image, or mask, is constructed with 0's at all interesting pixels and 1's outside the image. The display operation is first to and the mask into the background image, and then to or the original image. Such pictures can be useful for cursors which must appear clearly over white and black while retaining a non-rectangular outline. The cursor that appears over the menu in Figure 1b makes use of this effect.

Example: The Game of Life

Conway's game of Life is an example of image processing that requires more complex operations than those specified for bitblt. It is a fairly simple rule for successive populations of a bitmap. The rule involves the neighbor count for each cell — how many of the eight adjacent cells are occupied. Each cell will be occupied in the next generation if it has exactly three neighbors, or if it is now occupied and has exactly two neighbors. Since bitblt supports only single-bit pixels, separate bitmaps must be used to store separate bits of the neighbor count. Then, bitblt's xor (partial sum) and and (carry) modes can be used for the addition. With some ingenuity and a fair amount of extra storage, the next generation of any size of bitmap can be computed using a constant number of bitblts.

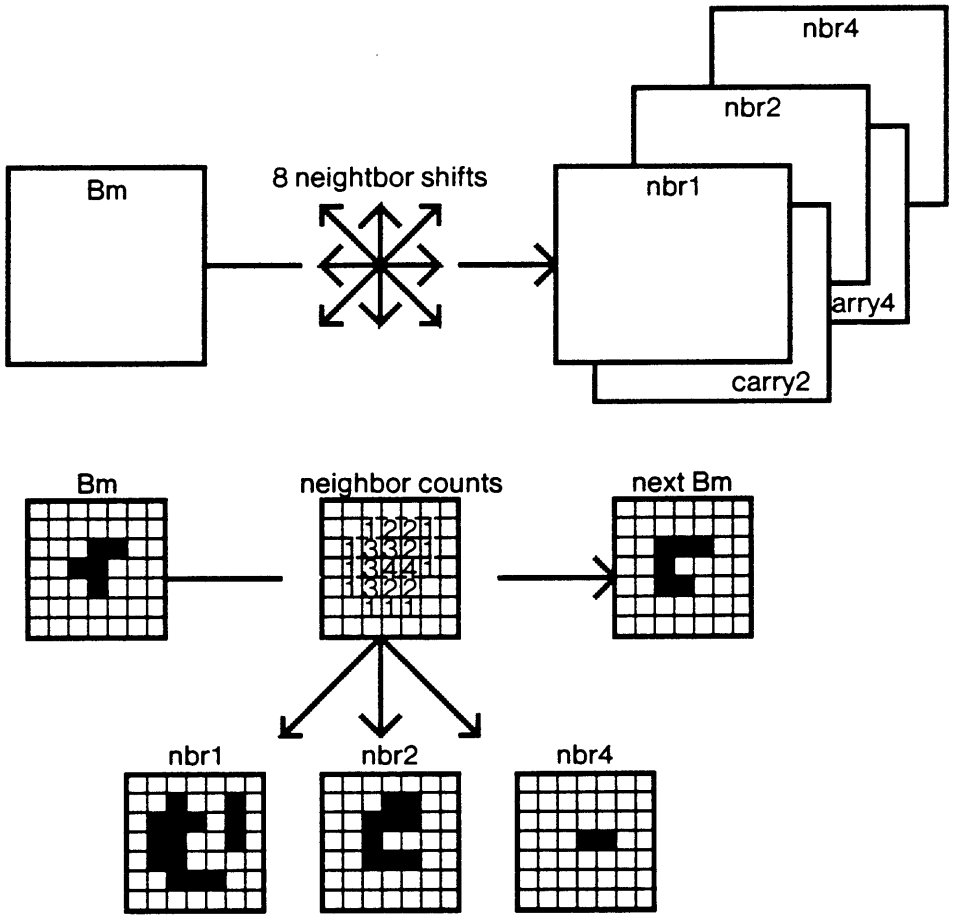

Figure 27. Implementing the game of Life using bitblt. (Figure Copyright © Xerox Corp. Reprinted by permission.)

As shown in the figure, the number of neighbors is represented using three image planes for the 1's bit, 2's bit and 4's bit of the count in binary. (See Figure 27.) The 8's bit can be ignored since the result for 8 neighbors is the same as for 0.

ALTERNATE REPRESENTATIONS AND ARCHITECTURES

The form of bitblt we have presented tries to achieve a balance between generality and ease of implementation. Other forms are certainly possible. One design decision we have not made particularly explicit is the rectangular form of bitblt, which is obviously imposed by the inherent rectilinearity of the raster. Since the sides of the rectangle must be plumb, however, there are serious limitations on the images bitblt can manipulate. For example, rotating a bitmap other than by multiples of 90 degrees (however that might be implemented!) results in something that is no longer a bitmap in our sense of the term. One fruitful variation is a triangular bitblt. Although it is not as intuitively appealing, rotations of triangles are still triangles, and arbitrary polygons are easily tiled by triangles, with the tiling transforming as the polygon. Polygons may also be tiled by trapezoids, but the allowable transformations must then either exclude rotations or allow retiling. In practice, trapezoidal bitblt is probably easier to implement than triangular, but less general. Both these variations have specification problems due to the discrete nature of the raster. Because the edges of an arbitrary triangle are not square to the raster, it is necessary to define carefully how a triangle is represented. Unfortunately, a compromise must be made in the definition, because two desirable features of the definition — connectedness of individual triangles, and having each pixel in a tiling be a member of only one triangle — are not simultaneously realizable.

There are interesting variations possible for rectangular bitblt, too. The SUN workstation[2] implements a three-operand Boolean function $D \leftarrow f(D, S, T)$. This makes it possible at times to save on the number of bitblts required compared to our notation. Our decision to use (S and T) rather than an arbitrary Boolean function was driven by extensive positive experience with this form, but exactly how the texture is integrated into bitblt is a topic worth exploring further. On the other hand, in the implementation bitmaps and textures need to be handled differently, so a less general but still adequate form of bitblt has two variants: $D \leftarrow f(D, S)$, with D and S bitmaps, and $D \leftarrow g(D, T)$, with D a bitmap and T a texture.

It is also worth noting that in practice bitblt is usually used either for very small bitmaps, such as characters, or for very large bitmaps, such as windows. In the former, set-up time dominates, while in the latter the main loop execution dominates. So, from a practical point of view, it makes sense to have two different implementations, each optimized for one of these two situations. However, given the loss in conceptual economy, it is debatable whether this results in an overall gain. The designers of the Blit spent hours debating this topic, but eventually decided the simplicity was worth the 30% or so performance loss for characters, and threw away their special-purpose character bitblt.

The operation of setting down a paint-brush, that is, an arbitrary shape filled with a texture, is sufficiently common that perhaps it should become a single primitive. This will significantly improve the inner loop for painting wide lines or curves. Another possibility is to allow a class of operators that modify both the source and destination, such as exchange. This would simplify the operation of saving and restoring off-screen bitmaps.

One of the complications in the implementation of bitblt is deciding in what direction the loops should run, to avoid destroying source bits before they are moved. A case can be made that it should be possible in the specification of bitblt to set these directions at will (although the default would certainly be as we have defined it). For example, the algorithm that does area-fill by xor-ing scan lines takes only a constant number of bitblts. The trick is to apply the same algorithm, but on the whole bitmap at once rather than one scan line at a time, and specify that the outer loop run the 'wrong' way. This has the effect of doing the cumulative xor needed.

We would be remiss not to mention how color displays and bitblt might coexist. The following brief summary is basically an admission that we don't understand the problem very well.

There are two basic representations of color images: integer values, in which the intensity of each pixel is stored directly as an integer; and bit planes, in which each bit in a pixel corresponds to a color, and the displayed color is the sum of the colors corresponding to the bits set in the pixel. Intensities are used for image synthesis, while bit planes are common in CAD/CAM applications.

Bit planes are a simple generalization of single-bit bitmaps. Because the planes are independent, the simplest way to generalize bitblt for such displays is to add a couple more operands to control bit selection and, perhaps, rearrangement. Textures still make sense on such displays, but might be implemented as one bit deep or many, depending on the application.

Intensities are not as easily handled. Usually, a pixel has (at least conceptually) three different integers associated with it, one for each of the red, green and blue components of the displayed color. The pictures on RGB displays have a qualitatively different type of processing applied to them, such as anti-aliasing to remove the discretization artifacts described earlier. Bitblt as it stands is inappropriate for such displays. One interesting possibility, though, is a 'byteblt' operator with built-in linear interpolation (lerp). The analog of bitblt, at least for straight assignment, is to merge a picture into a frame buffer, but this must be done with care at the edges to avoid aliases. The idea is something like using a mask for writing opaque whites on one-bit bitmaps, but with a fractional mask rather than a Boolean one. Porter and Duff[22] describe the algebra for such an operator.

One possibly confusing element in color displays is the color map: a correspondence table that converts the pixel value to displayed color (even on regular RGB displays, a color map is useful for contrast correction and the like). Color maps don't change the formalism, though: they are transparent to bit copying in bit planes, and for intensity displays the result of merging two images with different color maps is undefined. So color maps are probably not an issue with respect to variations on bitblt.

What does the future hold for bitmap hardware? One of the most interesting possibilities is two-dimensional memory. Although large bitblts sweep sequentially through memory, smaller images such as characters and non-rectangular images such as lines and curves are as likely to access words at adjacent y values as at adjacent x values. Some existing displays support such 'vertical' address generation, but a more interesting possibility is to have two-dimensional words of memory, rather than one dimensional. This would allow small characters to be written by touching a single word of memory (depending on word boundaries), and would guarantee that the number of pixels of a line that fit in a word is independent of the orientation of the line. The 8 by 8 display is an experimental bitmap display built to try the idea using words 8 bits square. The paper by Sproull, Sutherland, Thompson and Minter[24] describes the display in detail, so here we will just summarize the results. As expected, lines and characters can be drawn quickly — lines, for example, are about three times faster to draw that on a conventional frame buffer. Also as expected, the two-dimensional offers no performance advantage large bitblts. Unfortunately, the hardware is complex and expensive, and the special nature of the display memory requires more case analysis in bitblt, as discussed earlier. Still, the idea is appealing, and deserves more development as hardware becomes less expensive.

As was mentioned, one of the reasons for bitblt's success is that many image processing algorithms can be expressed using only local operations. Bitblt has an inherent parallelism — it is a form of parallel assignment operator — so it is likely that graphics hardware with parallel processing capabilities could easily provide very high performance implementations of bitblt. The extreme of this trend is a system with a processor attached to every pixel, capable of communication with some subset of its neighbors and perhaps some global controller.

And, of course, no discussion of raster graphics would be complete without mentioning VLSI.[3]

MH-11271-RP/LG/DI-unix

R. Pike TM

L. Guibas

D. Ingalls

Atts.
References (1-26)

REFERENCES

1. ANSI, "Graphical Kernel System Proposal X3H3/83-25r3," *Computer Graphics* (February 1984).
2. A. Bechtolsheim and F. Baskett, "High-Performance Raster Graphics for Microcomputer Systems," *Computer Graphics* 14(3), pp. 43-47 (July 1980).
3. Tom Blank, Mark Stefik, and Willem vanCleemput., "Parallel Bitmap Processor," *Proc. First Design Automation Conference* (June 1981).
4. J. F. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems Journal* 4(1), pp. 25-30 (1965).
5. Apollo Computer, *Apollo Manual*, 1981.
6. Ridge Computer, *Ridge-32 Manual*, 1983.
7. R. W. Floyd, *Complexity of Computer Computations*, Plenum, New York (1972).
8. Grandjean and Vigliani, *Ergonomic aspects of video display terminals*, Taylor and Francis, London (1980).
9. Leo J. Guibas and Jorge Stolfi, "A Language for Bitmap Manipulation," *ACM Transactions on Graphics* 1(3), pp. 191-214 (1982).
10. Symbolics Inc., *3600 Technical Summary*, 1983.
11. JEDEC, "Optical characteristics of cathode ray tube screens," JEDEC Publication 16-B (1971 (a new edition was published in 1981)).
12. Donald E. Knuth, *TEX and Metafont*, Digital Press, Bedford MA (1979).
13. B. Lampson and K. Pier, "A Processor for a High-Performance Personal Computer," *Proc. 7th Symp. Comp. Arch. SIGARCH/IEEE*, La Baule, p. 146 (May 1980).
14. S. Levialdi, "On Shrinking Binary Picture Patterns," *Communications ACM* 15(1), pp. 7-10 (January 1972).
15. M. D. McIlroy, "A Note on Digital Representations of Lines," *Bell Labs Tech. Journal* (to appear).
16. Sun Microsystems, *System Interface Manual for the SUN Workstation*, 1983.
17. Motorola, *MC68000 16-Bit Microprocessor User's Manual, Third Edition*, Prentice-Hall, Englewood Cliffs, NJ (1982).
18. J. Von Neumann, *Theory of Self-Replicating Automata*, Univ. of Illinois (Urbana) Press (1966).
19. W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics*, Second Edition, McGraw-Hill, New York (1979).
20. Rob Pike, "Graphics in Overlapping Bitmap Layers," *ACM Transactions on Graphics* 2(2), pp. 135-160 (1983).
21. Rob Pike, Bart Locanthi, and John Reiser, "Hardware/Software Tradeoffs for Bitmap Graphics on the Blit," *Software — Practice & Experience* (to appear).
22. Tom Porter and Tom Duff, "Compositing Digital Images," *SIGGRAPH'84 Proceedings* (1984).
23. Robert F. Sproull, "Using Program Transformations to Derive Line-Drawing Algorithms," *ACM Transactions on Graphics* 1(4), pp. 257-288 (October 1982).
24. Robert F. Sproull, Ivan E. Sutherland, Alistair Thompson, and Charles Minter, "The 8 by 8 Display," *ACM Transactions on Graphics* 2(1), pp. 1-31 (January 1983).
25. C. P. Thacker, B. L. Lampson, and E. McCreight, "Alto — A Personal Computer, in," *Computer Structures: Readings and Examples* (1979).
26. D. Weinreb and D. Moon, in *Lisp Machine Manual* (1981).