AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Computing Science Technical Report No. 999

Graphics in Overlapping Bitmap Layers

Rob Pike

April 1, 1983

Graphics in Overlapping Bitmap Layers

Rob Pike

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

One of the common uses of bitmap terminals is to store multiple programming contexts in multiple, possibly overlapping, areas of the screen called windows. Windows traditionally store the visible state of a programming environment, such as an editor or debugger, while the user works with some other program. This model of interaction is attractive for one-process systems, but to make full use of a multiprogramming environment, windows must be asynchronously updated, even when partially or wholly obscured by other windows. For example, a long compilation may run in one window, displaying messages as appropriate, while the user edits a file in another window.

This document describes a set of low-level graphics primitives to manipulate overlapping asynchronous windows, called *layers*, on a bitmap display terminal. Unlike previous window software such as [mey81] and [tes81], these primitives extend the domain of the general bitmap operator *bitblt* [gui82] to include bitmaps which are partially or wholly obscured.

April 1, 1983

Graphics in Overlapping Bitmap Layers

Rob Pike

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.4 [Computer Graphics]: Graphics Utilities.

General Terms: Algorithms

Additional Key Words and Phrases: Raster image, bitmaps, bitblt instruction, windows.

Introduction

Bitmap displays are in vogue. Despite their drawbacks, they are sufficiently inexpensive to be personal equipment rather than shared computer center resources. Unfortunately, although their use has become widespread, most bitmap displays are controlled by primitive software.

Bitmap screens are commonly divided into a set of rectangular areas called *windows* each of which maintains the input and output context for a particular program or graphics application. A bitmap display can become several screens at once — different windows can be emulating standard terminals, graphics terminals, drafting tables or even arcade games. The screen areas need not be disjoint — the display areas of two programs may overlap, with one area fully visible and the other partially or wholly obscured. This paper describes software to manipulate these overlapping windows, or *layers*, on a single display. A program drawing in a layer is fully isolated from other programs drawing on the same screen, even if one layer overlaps, and thereby obscures part of, the other. By extending the basic bitmap operators to work with layers, which may be regarded as possibly obscured, subdivided bitmaps, each program manipulates its layer just as it would manipulate the full screen or any other bitmap in a more conventional environment. Each program is removed from considerations regarding, for example, which layers obscure which, and two programs may (almost) simultaneously draw in their respective layers, even though some of the drawn objects are currently obscured from view.

Comparison with previous work

A number of window systems have been implemented by others [lan79, tes81, mey81, wei81], but the software presented here has some distinguishing characteristics, dictated largely by the environment under which it is run. The environment is a bitmap display connected to a multiprogrammed timesharing system, with a limited but useful amount of off-screen memory (about a screenful) for storing bitmaps. Because the system is multiprogrammed, any program may want to update its window at any time, regardless of whether it is attached to the keyboard and mouse or even whether it is fully visible. It is also important that the user program be removed from concerns about window update: changes in the window configuration should be invisible to the program. This precludes the simple ''repaint'' mechanism, wherein a program is instructed to redraw its window whenever its visibility changes, because repainting forces the user program to deal with issues better left to lower-level software. Like the file system, which makes a user file look like a contiguously addressable stream of bytes although it might be scattered across the disk, the window system should provide a simple programming interface so a program can always draw in its window as though it is a contiguous bitmap, regardless of its position or visibilty on the screen.

There are two basic approaches to providing this transparency. The first, exemplified by the BRUWIN system [mey81], is to maintain a display list that is re-evaluated when the window configuration changes. Although this method typically uses a relatively small amount of memory to hold the display list, and makes it easy to rearrange the layout of the windows on the display, it has a few difficulties. One problem is that the time to redraw a freshly uncovered portion of a window is substantial if the window's display list is long. More important, it is only possible to draw what the display list can encode. BRUWIN, for example, only supports characters, contradicting the generality of the bitmap data structure. Even adding bitmap primitives to the display list is not entirely satisfactory, however. Consider using a bitmap paint program to create a realistic scene: the bitmap containing the scene is probably smaller than the display list to describe it, and certainly takes much less time to draw on the screen. Display lists introduce an unnecessary level of indirection.

The second method, used on the Lisp Machine [wei81], is to keep the window image in a contiguous off-screen bitmap and copy data from there to the visible portions of the on-screen window. This simple method has the advantage of supporting any bitmap operation, but takes a constant large amount of auxiliary memory and can be inefficient — it may be necessary to execute each bitmap operation twice: once off-screen and once on-screen.

The layers software is in some sense an optimization of this ''shadow bitmap'' method. By dividing the windows into visible and obscured parts, and only keeping obscured parts off-screen, it uses the minimum memory possible (assuming no display lists) by letting the screen memory itself hold the visible parts. The basic idea is to extend the domain of the general bitmap operator *bitblt* [gui82] to include windows that are wholly or partially obscured. Layer drawing operations therefore touch the same number of pixels regardless of visibility, although the number of bitblt operations may be larger if there are on-screen and off-screen components. The data structures and software are designed to make drawing in a layer as fast as possible, at the expense of slower creation and deletion operators (although creation and deletion are still fast). The most common layer shuffling operator, making a layer fully visible, is also efficient because it touches only those pixels that need to be updated. There are disadvantages, of course: the data structures become unwieldy with more than about a dozen overlapping layers, and translating a layer across the screen is clumsy, but the software has been implemented and works comfortably in our time-sharing environment, where the number of layers is fairly small and their average lifetime long.

Definitions

The programs described herein, written in a pseudo-C dialect, use several simple defined types and primitive bitmap operations. This section describes the basic concepts for bitmap operations, but does not attempt to be complete. The next section defines bitmaps precisely.

A Point is an ordered pair

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

that defines a location in a bitmap such as the screen. A Rectangle is defined by a pair of Points spanning a diagonal:

```
typedef struct{
    Point origin;    /* min x,y */
    Point corner;    /* max x,y */
}Rectangle;
```

By definition, `corner.x` ≥ `origin.x` and `corner.y` ≥ `origin.y`. Rectangles are half-open: a Rectangle contains the horizontal and vertical lines through `origin`, and abuts, but does not contain, the lines through `corner`. Two abutting rectangles r_0 and r_1, with r_1`.origin=(`r_0`.corner.x, `r_0`.origin.y)`, therefore have no point in common. The same applies to lines in any direction: a line segment drawn from (x_0,y_0) to (x_1,y_1) does not contain (x_1,y_1). These curious definitions simplify drawing objects in pieces, which is essential to the layer primitives.

The routine `rectf(b, r, f)` performs a function specified by an integer code `f`, in a rectangle `r`, in a bitmap `b`. The function code is one of:

`CLR:`	clear rectangle to zeros
`OR:`	set rectangle to ones
`XOR:`	invert bits in rectangle

The routine `bitblt(sb, r, db, p, f)` (bit-block transfer) copies a source Rectangle `r` in a bitmap `sb` to a corresponding Rectangle with `origin p` in a destination bitmap `db`. `bitblt()` is therefore a form of Rectangle assignment operator, and the function code `f` specifies the nature of the assignment:

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

For example, `OR` specifies that the destination Rectangle is formed from the bit-wise OR of the source and destination Rectangles before the `bitblt()`. Here, `|`, `&`, `^` and `~` are C's bit-wise OR, AND, exclusive-OR and NOT operators, and the notation `x *= y` is equivalent to `x = x * y`, but is notationally and semantically closer to the machine.

Often, `bitblt()` is defined to copy through a two-dimensional mask, to permit stippling and related operations. Our original `bitblt()` had implementation problems (which have since been resolved) which forced us to not incorporate a mask into the primitive; for compatibility, our current `bitblt()` also does not have a mask. However, whether a mask is present has no bearing on the ideas presented here.

`bitblt()` is a fundamental bitmap operation. It is used to draw characters, save screen rectangles and present menus. Defined with a bit mask, it includes `rectf().` For a thorough description, see [ing81] or [gui82]. `bitblt()` can be a very expensive operation, however. In the general case, the data from the source Rectangle must be shifted or rotated and masked before being written to the destination Rectangle, and the amount of data processed in a single `bitblt()` can be surprisingly large.

Bitmaps

A bitmap is a dot-matrix representation of a rectangular image. The details of the representation depend on the display hardware, or, more specifically, on the arrangement of memory in the display. For the idea of a bitmap to mesh well with the software in the display, the screen must appear to the program as a bitmap with no special properties other than its visibility. Because images (bitmaps) are stored off-screen, off-screen memory should have the same format as the screen itself, so that copying images to and from the screen is not a special case in the software. The simplest way to achieve this generality is to make the screen a contiguous array of memory, with the last word in a scan line followed immediately by the first word of the next scan line. Under this scheme, bitmaps become something familiar to the programmer: a two-dimensional array.

Given a two-dimensional array in which to store the actual image, some auxiliary information is required for its interpretation. Figure 1 illustrates how a bitmap is interpreted. The hatched region is the location of the image. When a bitmap is allocated, the allocation routine, `balloc()`, assumes its data will correspond to a screen rectangle, for example, a part of one layer obscured by another. `balloc()` creates the left and right margins of the bitmap to word-align the bitmap with the screen, so word boundaries in the bitmap are at the same relative positions as in the screen. This technique can make a significant performance difference for the common operation of saving and restoring screen rectangles.

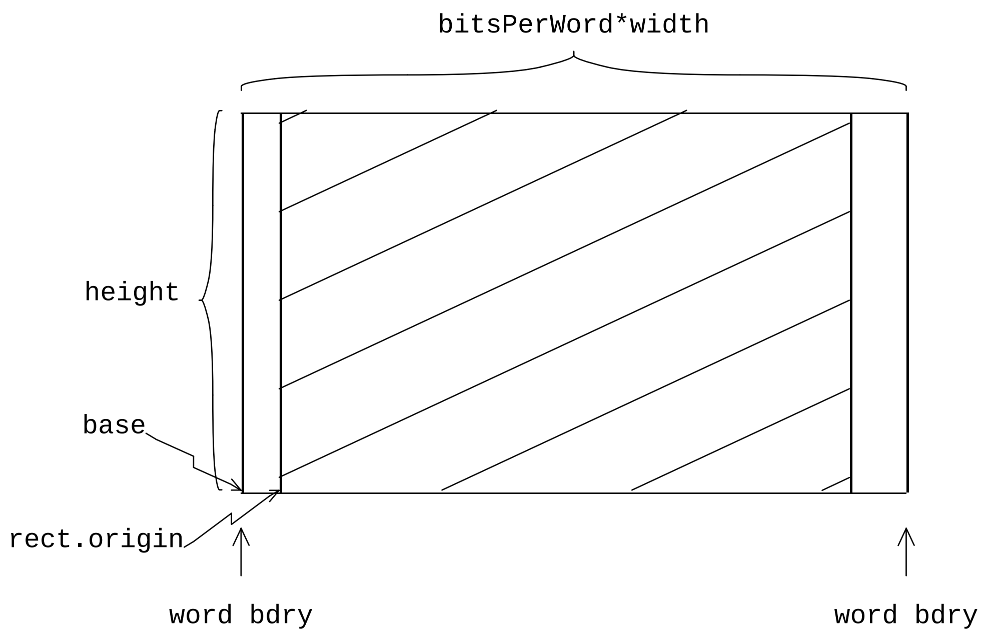

Figure 1. Layout of a bitmap. The storage begins and ends on word boundaries, with the last word of one scan line followed immediately by the first word of the next scan line. The hatched area, `Bitmap.rect`, stores the image.

`balloc()` takes one argument, the on-screen rectangle which corresponds to the bitmap image, and returns a pointer to a data structure of type Bitmap. Bitmap is defined thus:

```
typedef struct{
    Word *base;      /* start of data */
    unsigned width;  /* width in words */
    Rectangle rect;  /* image rectangle  */
}Bitmap;
```

The declaration `Word *base` states that `base` points to an object of type `Word`, or rather that `*base` is a `Word` pointed to by `base`. (Pointers in C can point to objects not allocated dynamically, and the layers software depends on this.) The elements of the structure are illustrated in Figure 1. `width` is in Words, which are some convenient unit of storage. `rect` is the argument to `balloc()`, and defines the coordinate system inside the Bitmap (the next section, about the Layer data structure, explains why this is done). The storage in the Bitmap outside `rect` — the unhatched portion in Figure 1 — is unused, as described above.

Typically, `width` is the number of Words across the Bitmap, between the arrows in Figure 1. A Bitmap may be contained in another Bitmap, however, if `width` is the `width` of the outer Bitmap, and `base` points as usual to the first Word in the Bitmap — the Word containing the pixel at the `origin` of the Bitmap. Although such Bitmaps are not created by `balloc()`, they have utility — such as representing the portion of the screen occupied by a layer.

`balloc()` and its obvious counterpart `bfree()` hide all issues of storage management for bitmaps. Because the sizes of bitmaps are widely variable, `balloc()` does garbage compaction, moving all allocated bitmaps to one end of the arena when space runs low, thereby ensuring that memory does not fragment.

The Bitmap structure is used throughout the layer software. Graphics primitives operate on points, lines and rectangles within Bitmaps, not necessarily the screen. The screen itself is simply a globally accessible Bitmap structure, called `display`, and is unknown within the graphics primitives.

Layers

A layer is a rectangular portion of the screen and its associated image. It may be thought of as a virtual display screen. Layers may overlap (although they need not), but the image in the obscured portion of a layer is always kept current. Typically, an asynchronous process, such as a terminal program or circuit design system, draws pictures and text in a layer, just as it might draw in any bitmap (such as the full screen if it were the only process on the display). Because processes are asynchronous, drawing actions can take place at any time in an obscured layer, and a graphical object such as a line may be partially visible on the screen and partially in the obscured portion of the layer. The layer software isolates a program drawing in a layer on the screen from other such programs in other layers, and guarantees that the image on- and off-screen is always correct, regardless of the configuration of the layers on the screen.

Layers are different from the common notion of windows†. Windows are used to *save* a programming or working environment, such as a text editing session, to process ''interrupts'' such as looking at a file or sending mail, or to keep several *static* contexts, such as file contents, on the screen. Layers are intended to *maintain* an environment, even though it may change because the associated programs are still running. For example, a programmer could be editing the program source in one layer while watching for diagnostic messages from a long compilation taking place in another layer, and waiting for intruders into his corner of the labyrinth in a real-time game in a third layer. There is, of course, nothing deep in the distinction between layers and windows; the term ''layer'' was coined to avoid the more cumbersome phrase ''asynchronous windows.'' Nonetheless, the difference between layers and windows is significant. The concept of multiple active contexts is natural to use and powerful to exploit.

Truly asynchronous graphics operations are difficult to support, because the state of a layer may change while a graphics operation is underway. The obvious simple solution is to perform graphical operations atomically by inhibiting process switches during calls to the layer software.

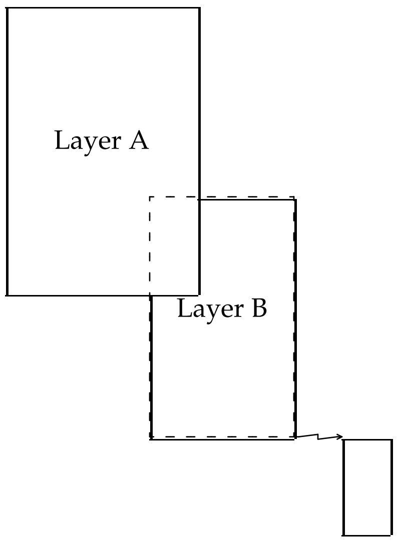

Figure 2. Obscured portions of a layer are stored off-screen and linked to the obscured layer.

The data structures for layers are illustrated in Figures 2 and 3. A partially obscured layer has an *obscured list*: a list of disjoint rectangles *in that layer* obscured by another layer or layers. In Figure 2, layer A obscures layer B. Layer B's obscured list has a single entry, which is marked ''obscured by A.'' If more than one layer obscures a rectangle, the rectangle is marked as

†The term ''window'' in common usage is a misnomer (see e.g. [new79]). The meaning is actually closer to that of the older term ''viewport.''

obscured by the *frontmost* (unobscured) layer intersecting the rectangle. This is illustrated by rectangle 4 in Figure 3. Rectangle 4 is an obscured part of both layers B and C, so these layers store their obscured pieces off-screen, and mark them blocked by layer A.

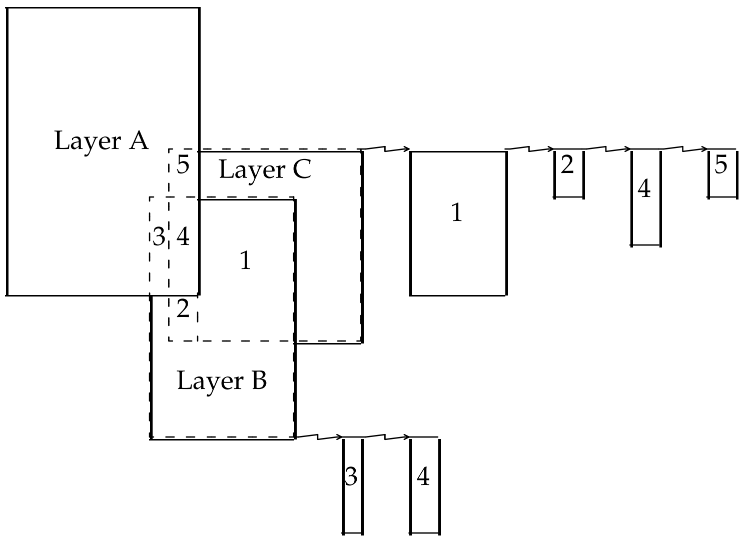

Figure 3. Rectangle 4 is obscured in layers B and C, and is maintained off-screen by both. The addition of layer C divides the obscured part of layer B into two rectangles, 3 and 4.

Rectangles 3 and 4 in layer B (Figure 3) may be stored as a single rectangle, as they were in Figure 2. They are stored as two because if layer C is later moved to the front of the screen (i.e. the top of the pile of layers), it will obscure portions of both layers A and B. Rectangle 4 in layer B would be obscured by C, but rectangle 3 would still be obscured by A. To simplify the algorithms for rearranging layers, the layer creation routine does all necessary subdivision when the layer is first made, so when layer C is created*, the obscured rectangle in B is split in two along the edge of the new layer.

The layer structure is:

```
typedef struct{
    Word *base;      /* start of data */
    unsigned width;  /* width in words */
    Rectangle rect;  /* image rectangle  */
    Obscured *obs;   /* linked list of obscured rectangles */
    Layer *front;    /* adjacent layer in front */
    Layer *back;     /* adjacent layer behind */
} Layer;
typedef struct{
    Layer *lobs;     /* frontmost obscuring Layer */
    Bitmap *bmap;    /* where the obscured data resides */
    Obscured *next;  /* chaining */
    Obscured *prev;
} Obscured;
```

The first part of the Layer structure is identical to that of a Bitmap. Actually, the Bitmap structure has an extra item to it: a `NULL` `obs` pointer, so a Bitmap may be passed to a graphics routine expecting a Layer as argument. The next section explains, among other things, why a

* Despite Figure 3, layer C is actually created at the *front*, but the argument is nonetheless true.

Bitmap with a `NULL` `obs` pointer looks like a Layer (and vice versa).

The individual Layers are chained together as a doubly-linked list, in order from ''front'' to ''back'' on the screen (when they do not overlap, the order is irrelevant). Besides the link pointers, a Layer structure contains a pointer to the list of obscured rectangles and the bounding rectangle on the screen. The obscured lists are also doubly linked, but in no particular order. Each element in the obscured list contains a Bitmap for storing the off-screen image, and a pointer to the frontmost Layer that obscures it. As we shall see later, an Obscured element need only record which (unobscured) Layer is on the screen ''in front'' of it, not any other obscured Layers which also share that portion of the screen. The screen coordinates of the obscured Rectangle are held in the `rect` field of the `Bitmap` structure pointed to by the `Obscured` list entry, `Obscured.bmap->rect`. All coordinates in the layer manipulations are screen coordinates.

Note that a Bitmap holding an obscured rectangle of a Layer has in its `rect` field a definition of the screen space it would occupy were it visible. This is an appealing model: as a Layer's contents are subdivided into visible and obscured portions, so is its coordinate system. More generally, since a Layer structure must contain a rectangle defining the portion of the screen it occupies, so must a Bitmap, because Layers are simply a generalization of Bitmaps.

Layerop

`layerop()` is the main interface between layers and the bitmap graphics primitives. Given a Layer, a Rectangle within the Layer, and a bitmap operator, it recursively subdivides the Rectangle into Rectangles contained in single Bitmaps, and invokes the operator on the Rectangle/Bitmap pairs. To simplify the operators, `layerop()` also passes along, unaltered, a pointer to a set of parameters to the bitmap operator. For example, to clear a rectangle in a layer, `layerop()` is called with the target Layer, the rectangle within the layer in screen coordinates, and a procedure (the bitmap operator) to invoke `rectf()`. `layerop()` divides the rectangle into its components in obscured and visible portions of the layer, and calls the procedure to clear the component rectangles. A complete example is worked through at the end of this section. `layerop()` itself does no graphical operation, it merely *controls* graphical operations done by the bitmap operator handed to it. It is a functional that turns a bitmap operator into a layer operator.

`layerop()` first clips the target Rectangle to the Layer, then calls the recursive routine `Rlayerop()` to do the subdivision:

```
/*
 * Clip to outer rectangle of layer, then call Rlayerop()
 */
layerop(lp, fn, r, otherargs)
    Layer *lp;
    procedure (*fn)();   /* Pointer to bitmap operator */
    Rectangle r;
    misc otherargs;  /* Other arguments used by (*fn)() */
{
    r = intersection of r and lp->rect;
    if(r not null)
        Rlayerop(lp, fn, r, otherargs, lp->obs);
}
```

The argument `(*fn)()` is the pointer to the bitmap-level procedure. `Rlayerop()` recursively chains along the obscured list of the Layer, performing the operation on the intersection of the argument Rectangle and the obscured Bitmap, and passing non-intersecting portions on to be intersected with other Bitmaps on the obscured list. When the obscured list is empty, the rectangle must be drawn on the screen.

The code to test if two rectangles overlap is simple, but not well known:

```
rectXrect(r, s)  /* Do r and s intersect? */
    Rectangle r, s;
{
    return(r.origin.x < s.corner.x &&
           s.origin.x < r.corner.x &&
           r.origin.y < s.corner.y &&
           s.origin.y < r.corner.y   );
}
```

The C && operator is a logical AND that does not evaluate the right operand if the left operand is false. Similarly, the == operator used below is the equality operator.

Here is Rlayerop:

```
/*
 * Rlayerop -- recursively subdivide and intersect
 *      rectangles with obscured bitmaps in layer
 */
Rlayerop(lp, fn, r, otherargs, op)
    Layer *lp;
    procedure (*fn)();
    Rectangle r;
    misc otherargs;
    Obscured *op;    /* Element of obscured list with which
                        to intersect r */
{
    if(op == NULL)       /* This rectangle not obscured */
        (*fn)(lp, r, display, otherargs, op); /* Draw on screen */
    else if(rectXrect(r, op->bmap->rect) == FALSE)    /* They miss */
        Rlayerop(lp, fn, r, otherargs, op->next); /* Chain */
    else{       /* They must intersect */
        if(r.origin.x < op->bmap->rect.origin.x){
            Rectangle temp = piece of r left of op->bmap->rect;
            Rlayerop(lp, fn, temp, otherargs, op->next);
            r->origin.x = op->bmap->rect.origin.x;
        }
        /* etc. for other three sides of rectangle */
        /* What's left goes in this obscured bitmap */
        (*fn)(lp, r, op->bmap, otherargs, op);
    }
}
```

The Layer pointer and Obscured pointer are passed to the bitmap operator (`(*fn)()`) because, although they are clearly not needed for graphical operations, `layerop()`'s subdivision is useful enough to be exploited by some of the software to maintain the layers themselves; we will see an example of such usage later on. Note that if `layerop()` is handed a Layer with a `NULL obs` pointer, or a Bitmap, its effect is simply to clip the rectangle and call the bitmap operator.

So far, `otherargs` has been referred to in a deliberately vague manner. In practice, `layerop()` works something like the standard C formatted output routine `printf()`: after the arguments required by `layerop()` (the Layer, bitmap operator and Rectangle), the calling function passes the further arguments needed by the Bitmap operator. `layerop()` passes the address of the first of these arguments through to the operator, which therefore sees a pointer to a structure containing the necessary arguments.

The following example illustrates the action of `layerop()`. Although we will see several examples later, this is the simplest and most common use. `lblt()` uses `layerop()` and `bitblt()` to copy an off-screen Bitmap to a Rectangle within a Layer. The Bitmap may contain, for example, a character. The general case of copying a Rectangle from one Layer to another will be discussed later.

```
Lblt(l, r, db, fp, o)
    Layer *l;
    Rectangle r;
    Bitmap *db;      /* Destination Bitmap */
    struct{
        Bitmap *sb;  /* Source Bitmap */
        int f;       /* Function code */
    } *fp;
    Obscured *o;
{
    bitblt(fp->sb, r, db, r.origin, fp->f);
}

lblt(l, sb, r, f)
    Layer *l;
    Bitmap *sb;
    Rectangle r;
{
    layerop(l, Lblt, r, sb, f);
}
```

Notice that the bulk of the code is declarations. This code assumes the source Bitmap is in screen coordinates; if it were not, extra arguments to `lblt()` would be passed to the operation routine `Lblt()` to describe the necessary coordinate transformation. The ''extra'' arguments to `layerop`, `sb` and `f`, are seen by `Lblt` as elements of a structure pointed to by `fp`.

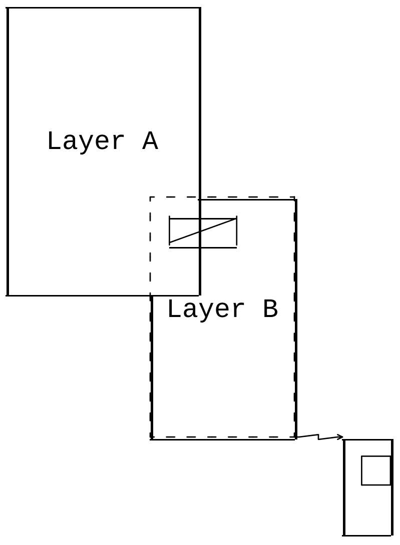

Figure 4. Drawing a rectangle (for example a character) in a layer. Some of the rectangle is drawn on the screen, some in the obscured parts of the layer. The portion of the rectangle overlapping layer A is drawn in the obscured bitmap, not on the screen.

As an example, consider drawing a character (the hashed rectangle) in Layer B in Figure 4. The first call to `Rlayerop()` has `op` set to the obscured, off-screen, element of layer B. `op` is not `NULL`, nor is the overlap of `r` and `rp->rect`, so we fall into the last block of the large **if-else**: part of the rectangle must be drawn in this obscured bitmap. The test

```
    if(r.origin.x < op->bmap->rect.origin.x)
```

succeeds, so the right portion of the rectangle (to the right of Layer A) is cut off and passed on to a second invocation of `Rlayerop()`. Here, the first `if` statement succeeds, the visible portion of the rectangle is drawn on the screen, and `Rlayerop` returns to its first invocation. No further clipping is required, so the left portion of the rectangle is drawn in the obscured bitmap and the job is complete.

Although it is a complicated operation, `layerop()` is fast compared to the expensive graphics operations it invokes. When `layerop()` is called, its overhead is usually unimportant. This is because the data structures to manipulate layers were explicitly designed to make *drawing* as rapid as possible, because that is the most common operation. The less common inter-layer manipulations such as creating and deleting layers are slower, but their price is still reasonable, considering how relatively infrequently they are invoked.

Upfront

There are three basic transformations that can in principle be applied to layers: changing the dimensions of a layer (scaling); changing the position of a layer on the screen (translation); and changing the front-to-back positions of overlapping layers (stacking).

The issues of scaling — changing a layer's size, as distinct from changing the scale of the coordinate systems inside the layer — and translation are not handled at all by the layer software, primarily because the subdivided nature of the data structures makes them difficult to implement. This is not to say that scaling and translation are not important transformations. Translation services in particular must be provided, perhaps by creating a new layer at the destination position, using `lbitblt()` (the generalized `bitblt()` operator, discussed later) to copy the data to the new position, and deleting the old layer. Note that the *data* in the layer can be dragged dynamically across the screen, using an auxiliary bitmap to hold the data obscured by the dragged rectangle or even just dragging a rectangular outline, without dynamically and incrementally modifying the layer data structure. Given the relative complexity of the subdivided layer structure, this approach is more reasonable, and certainly adequate (it is the way our higher-level software does translation). The notion of copying the data rather than signalling the application program to recreate it is consistent with the layer model: the application program is freed from the responsibility of maintaining its layer during rearrangement of the layers on the screen.

Scaling, however, does not fit as well into the software. Because there is no display list, the data in the different-sized layer cannot be recreated perfectly by the lower-level software — at best, an intersection of the source and destination layers can be maintained. We have no solution to this problem, but don't regard it as critical; the importance of the extended bitmap idea is powerful enough that we prefer to give up low-level support for scaling rather than revert to display lists. Also, it is unclear whether a layer should be regarded as a window or a viewport. Does a different-sized layer show the same thing at a different scale or a different-sized portion of the same thing? For example, if a large layer contains a picture composed of incompressible bitmaps, such as a page of characters in a text editor, what should happen to the image if the layer is shrunk to half its size? In particular, should the same configuration of smaller characters be drawn, or fewer characters the same size? Either could be correct under some conditions, and rather than make a decision we chose to ignore the issue.

Any stacking transformation can be defined as a sequential set of one-layer rearrangement operations, moving a single layer to another position, such as the front or back of the stack of layers. For example, the stack can be inverted by an action similar to counting through a deck of cards. `upfront()` is an operator that moves a layer to the front of the stack, making it completely visible. It is the only stacking operator in the layer software, because in the couple of places where a different operation is required, the desired effect can be achieved, with acceptable efficiency, by calls to `upfront()`. The action of pulling a layer to the front was chosen because it is the most natural. When something interesting happens in a partially obscured layer, the instinctive reaction is to pull the layer to the front where it can be studied. `upfront()` also turns out to be a useful operation during the creation and deletion of layers.

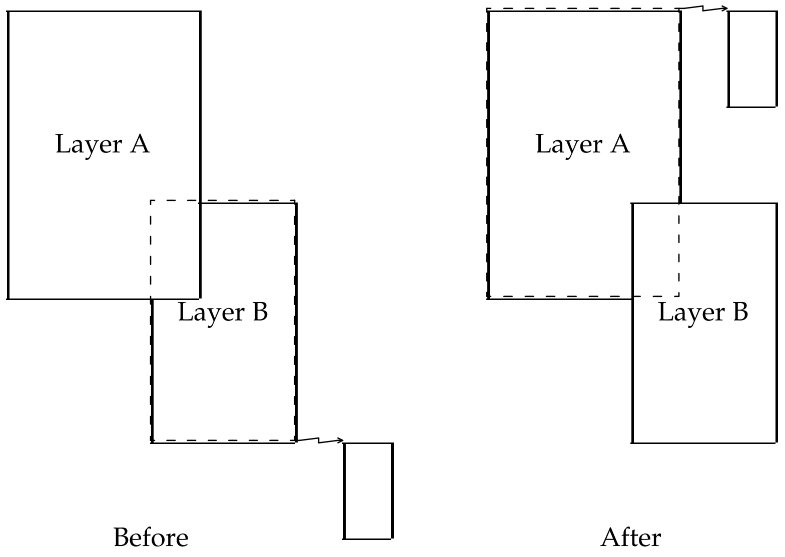

Figure 5. Moving a layer to the front requires interchanging its off-screen bitmaps with screen rectangles.

`upfront()` has a simple structure. Most of the code is concerned with maintaining the linked lists. The basic algorithm is to exchange the obscured rectangles in the layer with those of the layer obscuring them, swapping the contents of the obscured bitmap with the screen (Figure 5). Since the obscured rectangle has the same dimensions before and after the swap, the exchange can be done in place, and it is not necessary to allocate a new bitmap, we need merely link it into the new obscured layer. Obscured rectangles are marked with the *frontmost* obscuring layer for `upfront()`'s benefit: the frontmost layer is the layer that occupies the portion of the screen the rectangle would occupy were it at the front.

```
/*
 * upfront -- pull layer to the front of the screen
 */
upfront(lp)
    Layer *lp;
{
    Layer *fr;   /* a layer in front of lp */
    Layer *beh;  /* a layer behind lp */
    Obscured *op;

    for(fr = each layer in front of lp){
        for(op = each obscured portion of lp){
            if(op->lobs == fr){  /* fr obscures op */
                screenswap(op->bitmap, op->rect);
                unlink op from lp;
                link op into fr;
            }
        }
    }
    move lp to front of layer list;
    for(beh = all other Layers from back to front)
        for(op = each obscured portion of beh)
            if(lp->rect overlaps op->bmap->rect)
                op->lobs = lp;   /* mark op obscured by lp */
}
```

`screenswap()` interchanges the data in the bitmap with the contents of the rectangle on the screen, in place. It is easily implemented, without auxiliary storage, using three calls to `bitblt()` with function code `XOR`. Note that because of the fragmentation of the obscured portions done when a new Layer is created, if `lp->rect` and `op->bmap->rect` intersect, the Layer must *completely* obscure it. Note also that it is `upfront()` which enforces the rule that the *frontmost* Layer obscuring a portion of a second Layer is the Layer marked as obscuring it. Only if these two Layers are interchanged is the screen updated.

The last loop is required; it is not sufficient to mark obscured rectangles only in the main loop, because rectangles initially behind `lp` may also need to be marked.

Dellayer

It is simpler to delete a Layer than to create one, so we will discuss deletion first. The algorithm is easy:

1) Pull the layer to the front. It now has no obscured pieces, and is a contiguous rectangle on the screen.
2) Color the screen rectangle the background color.
3) Push the layer to the back. All storage needed for the obscured portions of the layer is now bound to the layer, since it obscures no other layer.
4) Free all storage associated with the layer.
5) Unlink the layer from the layer list.

A special routine, the opposite of `upfront()`, could be written to push the layer to the back, but `upfront()` can be used for the task:

```
/*
 * dellayer -- delete a layer
 */
dellayer(lp)
    Layer *lp;
{
    Obscured *op;

    upfront(lp);
    background(lp->rect);
    /* Push to back using upfront */
    while(lp not rearmost layer)
        upfront(rearmost layer);
    /* Free the storage */
    for(op = each obscured part of lp){
        bfree(op->bmap);
        free(op);
    }
    unlink lp from Layer list;
}
```

Using successive calls to `upfront()` to push a layer to the *back* gives the screen a curious appearance during the deletion. `dellayer()` is so simple, though, that it seems unnecessary to give it extra support by providing a more efficient "`downback()`" routine. `upfront()` does *not* join disconnected obscured bitmaps which could be joined because of the deletion, although it could. The difficulty of recognizing which bitmaps can be joined is too high to justify the usually small storage compaction and execution efficiency that would be gained thereby. Joining would be facilitated if the obscured lists of the layers contained information about which layer caused which edge to be cut during creation (see, for example, rectangle 2 in Figure 3). In retrospect, it would probably have been better to use some form of tree structure for the obscured lists, with an indication at each node of the layer ''responsible'' for creating the obscured element or dividing the underlying structure into two or more elements.

Newlayer

Making a new layer is the most difficult layer operation, for it may require modifying obscured lists of other layers. If the new layer creates any new overlaps, the obscured list of the overlapped layer must be restructured so that `upfront()` need not subdivide any rectangles to pull the obscured layer to the front. The creation routine, `newlayer()`, pays the price for the simplicity of `upfront()` and `layerop()`.

The basic structure of `newlayer()` is to build the layer at the back, constructing the obscured list by intersecting the layer's rectangle with the obscured rectangles and visible portions of the current layers. After allocating storage for the obscured bitmaps, the layer is pulled to the front, making it contiguous on the screen and forcing the rectangles obscured by the new layer to contain the new storage required by the addition of the new layer. Finally, the screen rectangle occupied by the new layer is cleared to complete the operation.

Several ancillary routines are used by `newlayer()`. `addrect()` adds rectangles to the obscured list, `obs`, of the new layer. Since the new layer is built at the ''back'' of the screen, any obscured rectangle of the new layer will be obscured by a layer already on the screen. `addrect()` builds the list of unique obscured rectangles, marked by which layer is currently occupying the screen in each rectangle. To be sure that a rectangle is unique, it is sufficient to check just the origin point of the rectangle. The rectangles passed to `addrect()` are ordered so that the first layer associated with a particular rectangle occupies the screen in that rectangle.

```
/*
 * addrect -- add (unique) rectangle to
 *      obscured list of new layer
 */
Obscured *obs;   /* Pointer to obscured list for new layer */
addrect(r, lp)
    Rectangle r;
    Layer *lp;   /* Layer currently occupying r on screen */
{
    Obscured *op, *newop;

    for(op = each element of obs)
        if(op->rect.origin == r.origin)
            return; /* Not unique */
    newop = new Obscured;
    newop->rect = r;
    newop->lobs = lp;
    link newop into obs list;
}
```

Because it is called once for each rectangle, `addrect()` takes time proportional to the square of the number of obscured rectangles in the new layer. But since `newlayer()` will ultimately perform a series of expensive `bitblt()`'s the time taken by the list operations is insignificant.

`addobs()` does recursive subdivision of the obscured rectangles that intersect the new layer, calling `addrect()` when an overlap is established. It is similar to `layerop()` except that it does not chain along the obscured list, and no special action (i.e. storage allocation) is required if the rectangles match exactly. As subdivided pieces are added to the obscured list of a current layer, the original rectangle must remain in the list until all the subdivided pieces are also in the list, whereupon it is deleted. New pieces must therefore be added *after* the original piece. When the topmost call to `addobs()` returns, the subdivision (if any) is complete, and the return value is whether the argument rectangle was subdivided. `newlayer()` then removes the original rectangle from the list if `addobs()` returns `TRUE`.

```
/*
 * addobs -- add obscured rectangle to list, subdividing obscured
 *      portions of layers as necessary
 */
boolean
addobs(op, argr, newr, lp)
    Obscured *op;
    Rectangle argr;  /* Obscured rectangle */
    Rectangle newr;  /* Complete rectangle of new layer */
    Layer *lp;       /* Layer op belongs to */
{
    Obscured *newop;
    Rectangle r;
    Bitmap *bp;

    r = argr;    /* argr will be unchanged through addobs() */
    if(rectXrect(r, newr)){
        /* This is much like layerop() */
        if(r.origin.x < newr.origin.x){
            Rectangle temp = piece of r left of newr;
            addobs(op, temp, newr, lp);
            r.origin.x = newr.origin.x;
        }
        /* etc. for other three sides */
        /* r is now contained in rectangle of new layer */
        if(r == argr){   /* no clip, just bookkeeping */
            addrect(r, lp);
            return FALSE;    /* No subdivision */
        }
        addrect(r, lp);
    }
    bp = balloc(r);
    newop = new Obscured;
    /* Copy the subdivided portion of the image */
    bitblt(op->bmap, r, bp, bp->rect.origin, STORE);
    newop->bmap = bp;
    newop->rect = r;
    newop->lobs = lp;    /* Layer lp obscures this part of the new layer */
    link op into lp->obs;
    return TRUE;   /* Subdivision */
}
```

`newlayer()` is long but straightforward.

```
Obscured obs;    /* obscured list of new layer when at back */
/*
 * newlayer -- make a new layer in rectangle r of bitmap *bp
 */
Layer *
newlayer(bp, r)
    Bitmap *bp;
    Rectangle r;
{
    Layer *lp, *newlp;
    Obscured *op;

    /* First build, in obs, a list of all obscured rectangles that
     * will be obscured by the new layer, doing subdivision with
     * addobs()
     */
    obs = NULL;
    for(lp = each layer from front to back){
        for(op = each obscured portion of lp){
            if(rectXrect(r, op->rect) &&
               addobs(op, op->rect, r, lp)){
                unlink op from lp->obs;
                bfree(op->bmap);
                free(op);
            }
        }
    }
    /* Now add the rectangles not currently obscured, but that will
     * be obscured by new layer, by building layer & calling layerop()
     */
    newlp = new Layer;
    Bitmap part of newlp = *bp;
    newlp->obs = obs;    /* currently obscured ... */
    for(lp = each layer from front to back)
        layerop(lp, addpiece, lp->rect);
    newlp->obs = obs;    /* ... and soon to be */
    for(op = each element of obs)
        op->bmap = balloc(op->rect);
    link newlp into back of layer list;
    upfront(newlp);
    rectf(newlp->rect, CLR); /* Clear the screen rectangle */
    return newlp;
}
```

`newlayer()` takes an argument `Bitmap`, which is typically the screen `Bitmap display`, but may be any other. It is a simple (untried) generalization from Layers within Bitmaps to Layers within Layers, and a true hierarchy.

`addpiece()` is a trivial routine to add to the obscured list the rectangles that are currently unobscured (i.e. have only one layer) but that will be obscured by the new layer.

```
addpiece(lp, r, bp, otherargs, op)
    Layer *lp;
    Rectangle r;
    Bitmap *bp;
    char *otherargs; /* Unused */
    Obscured *op;
{
    if(op == NULL)   /* This piece occupied by one layer only */
        addrect(r, lp);
    /* Otherwise it's already in obs list */
}
```

Scrolling and the general bitblt operator

So far, we have only discussed *support* for graphics, not graphical actions themselves. This and the next section, on scrolling and line drawing, illustrate how difficult graphics can be on a bitmap display, and how `layerop()` helps.

For simplicity of exposition, this section uses as an example the case of scrolling a layer: although the subdivision of rectangles can get complicated in sophisticated examples, the issues are all addressed by a simple case. Since the general case is not significantly harder to implement, this section presents a general `lbitblt()` operator that extends the bitmap-level `bitblt()` functionality to layers.

Scrolling a layer — moving a layer full of text up one line — is a special case of a bitblt operation. Scrolling a screen rectangle `r` is a single bitblt:

```
#define NLSZ 16 /* height of a text line */
bitblt(display, Rect(r.origin, Pt(r.corner.x, r.corner.y-NLSZ)),
    display, Pt(r.origin.x, r.origin.y+NLSZ), STORE);
```

that copies all but the first line of text up one line. `Rect()` and `Pt()` create Rectangles and Points from their arguments. Scrolling a layer requires a series of bitblts, which must also be done in a certain order. Figure 6 illustrates a simple example of scrolling a partially obscured layer. The scroll is broken into three bitblts, so that the source and destination rectangles each lie within a single bitmap.

The routine `lbitblt()` has semantics identical to `bitblt()`, but accepts Layer arguments as well as Bitmaps, and does the necessary subdivision of obscured rectangles and executes the atomic `bitblt()` calls in the correct order. There are two basic methods to organize `lbitblt()`. The first method calls `layerop()` to create a list of source rectangles by dividing the original rectangle ((B+C+D) in Figure 6, with (A+B) off-screen and (C+D) visible) into a set of rectangles contained within single bitmaps. `layerop` is then called for each source rectangle to subdivide it so the destination rectangles are also contained in single bitmaps (separating C and D in Figure 6 because c is in a different bitmap from d). This has the disadvantage that it calls `layerop()` once for each source rectangle, although simpler, faster code can accomplish the task.

The second method, and the one we use, calls `layerop()` twice: once to subdivide the source rectangles and once to subdivide the destination rectangles. The loop to intersect the source and destination rectangles is then trivial.

`lbitblt()` uses the defined type ListElement to couple the bitmaps to the rectangles:

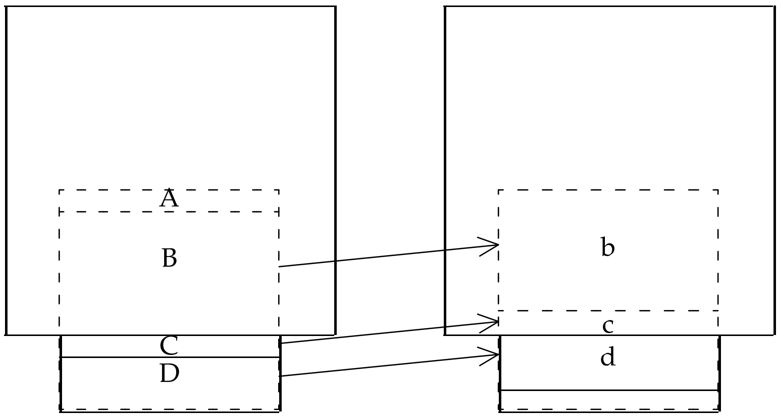

Figure 6. To scroll the rear layer, the source rectangles B through D are copied to the destination rectangles b through d.

```
typedef struct{
    Rectangle rect;
    Bitmap *bp;
} ListElement;
```

The type List chains together ListElements. Two calls to `layerop()`, one in the coordinate system of the source Layer and one in the coordinates of the destination, generate the lists of rectangles in sorted order. A simple nested loop then executes the component `bitblt()` calls:

```
Point delta;     /* difference between source and dest. origin */
List SrcList;    /* list of source rect/bitmap pairs */
List DestList;   /* list of destination rect/bitmap pairs */

lbitblt(sl, rect, dl, pt, f)
    Layer *sl, *dl;
    Rectangle rect;
    Point pt;
    Code f;
{
    ListElement *s, *d;
    Rectangle r;

    delta = pt - rect.origin;
    layerop(sl, Pass, rect, SrcList);
    layerop(dl, Pass, raddp(rect, delta), DestList);
    for(s = each element of SrcList)
        for(d = each element of DestList) {
            r = intersection of d->rect
                and raddp(s->rect, delta);
            if(r not null)
                bitblt(s->bp, rsubp(r, delta),
                    d->bp, r.origin, f);
        }
}
```

`raddp()` adds its second argument, a `Point`, to the `origin` and `corner` of its first argument, a `Rectangle`; `rsubp()` is the corresponding subtraction operator. `Pass()` is a trivial routine to create the lists. The rectangles in the lists are sorted according to an ordering operator `lessthan()`.

```
Pass(lp, r, sb, ap, op)
    /* the usual declarations */
    struct {
        List l;
    } *fp;
{
    ListElement *e;

    for(e = each element of fp->l)
        if(not lessthan(r, e->rect)) {
            insert the ListElement {sb, r} into fp->l
                before e;
            return;
        }
    append {sb, r} to fp->l
}

boolean
lessthan(a, b)
    Rectangle a, b;
{
    if((a->origin.y < b->corner.y) && (b->origin.y < a->corner.y))
        return ((a->origin.x - b->origin.x)*delta.x) >= 0;
    else
        return ((a->origin.y - b->origin.y)*delta.y) >= 0;
}
```

The first test sees if the two rectangles overlap in *y*, and if so checks the *x* ordering; otherwise the *y* ordering is sufficient. The ordering of the lists simplifies doing the bitblts in the correct sequence. For the scrolling example, `delta.y` is positive, and `lessthan()` orders the rectangles so they are copied in decreasing *y* order (see Figure 6).

There are a few efficiency issues. If the source and destination bitmaps are distinct in the uppermost call, such as in drawing a character in a layer, the rectangles need not be sorted, but `lbitblt()` sorts them anyway. Also, if one or both of the layers has an empty obscured list, `lbitblt()` could recognize these degenerate cases and call `bitblt()` directly if neither layer has obscured components, or `lblt()` if only one does.

More seriously, the implementation of `lbitblt()` presented here contains some code with quadratic complexity — the inner loops of `Pass` and `lbitblt()`. (The first method described above only executes *n* times, but calls `layerop()` to (recursively) run down the destination lists, so the complexity is the same.) In our application the obscured lists are typically short, so this complexity is dwarfed by the expense of the underlying `bitblt()` calls. If this were not true, it would be straightforward to rewrite both `Pass` and `lbitblt()` to use $n \log n$ search algorithms. The structure of the code would be the same, however, and the use of `layerop()` to organize the clipping would be unaffected.

Lines

Drawing lines on a dot-matrix display is an old problem (see e.g. [new79]). Drawing lines in layers is more difficult, because the sequence of dots approximating a line must be independent of the structure of the layer. To draw a line efficiently in a partially obscured layer, the line-drawing algorithm must draw the line as a set of line segments in bitmaps, but the endpoints of the segments (which must be in integer coordinates) might not be points on the actual line being approximated. A line from (0,0) to (300,100) is not approximated by the same points as two lines, one from (0,0) to (100,33) and one from (100,33) to (300,100). Of course, if efficiency is not an issue, lines may be drawn by generating the set of dots for the approximation and, for each dot, deciding in which bitmap it will be drawn. Here we will assume that a line segment may be drawn efficiently in a bitmap (which is true on any reasonable display) and that the problem is one of dividing up the original line.

Most line-drawing algorithms simulate a digital differential analyzer, or DDA, to reduce the strength of the rational arithmetic in

$$y = \frac{\Delta y}{\Delta x} x + y_0$$

to increments and decrements. Different DDA's approximate a given line by different sets of points. Bresenham's algorithm [bre65] has no multiplications or divisions, and uses only integer arithmetic:

```
int e, Δx, Δy;
e=2Δy-Δx;
for(i=1; i<Δx; i++){
    drawpoint(x, y);
    if(e>0){
        y++;
        e+=2Δy-2Δx;
    }else
        e+=2Δy;
    x++;
}
```

(As mentioned earlier, this algorithm is usually implemented efficiently; `drawpoint()` is not a subroutine call, and the constants come out of the loop). Here, `e` is an (offset and scaled) error term that represents the vertical distance between the actual line and the approximating points. Each time the algorithm draws a point, `e` is adjusted to track the local deviation from the actual line. When `e` is positive, the actual line is too far above the dots, and the dots are therefore moved up one unit in y.

The line algorithm can be restarted as it passes from obscured bitmap to bitmap in the layer by carrying the value of `e` across the boundary. But since `layerop` calls the bitmap-level operators in a non-deterministic order, a simpler solution is to evaluate `e` at the (x,y) location where the line intersects the bitmap, and reset the line algorithm in each component bitmap. We are assuming $0 < \Delta y/\Delta x \le 1$, a shallow line of positive slope. If we further assume the real line passes through (0,0) we can analytically specify points generated by Bresenham's algorithm. The equation of the actual line is

$$y\Delta x - x\Delta y = 0,$$

so at a generated point

$$y\Delta x - x\Delta y = r$$

where r is the residual — the deviation from the actual line, measured parallel to the y axis. For computational efficiency, `e` is offset and scaled from r:

$$\mathtt{e} = -2r + 2\Delta y - \Delta x \tag{1}$$

To minimize r, we round to calculate y:

$$y = \left\lfloor \frac{x 2\Delta y + \Delta x}{2\Delta x} \right\rfloor \tag{2}$$

implying, as above,

$$r = y\Delta x - x\Delta y.$$

Substituting,

$$\mathtt{e} = (2x + 2)\,\Delta y - (2y + 1)\,\Delta x \qquad (3)$$

Note that if $(x,y) = (0,0)$, $\mathtt{e} = 2\Delta y - \Delta x$, as in the original algorithm. Equation 3 can also be derived directly from the algorithm (and vice versa). Given equations 2 and 3, we can break a line into a set of segments in bitmaps independently of the layer's configuration by resetting the DDA for each segment. This method generates a unique approximation because the state of the DDA is a strict function of x.

One problem remains. The clipping routine (see [new79]) that divides the line into segments individually within bitmaps may clip to a horizontal or a vertical edge of a bitmap. If the edge is horizontal, several possible x values may be chosen (see Figure 7). Because lines are half-open, the desired value is the least x such that (x,y) is a point on the approximated line, as defined by (1). It can be shown that the desired value is

$$x = \left\lceil \frac{2y\Delta x - \Delta x}{2\Delta y} \right\rceil . \qquad (4)$$

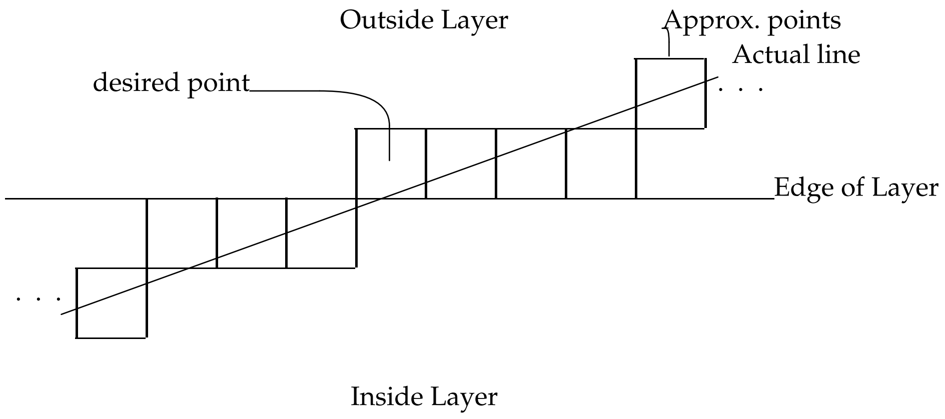

Figure 7. Lines must be clipped to the first point outside a layer or bitmap.

Turning these equations into working code involves a few obvious transformations to cover all possible cases. The transformations must be done carefully, to preserve the half-open property of lines. Given a routine `clipline()` which clips a line segment to a bitmap and draws it, using a line-drawer which loads the DDA based on the initial x value, `layerop()` can control drawing a line in a layer:

```
Lline(lp, r, db, fp, op)
    Layer *lp;
    Rectangle r;
    Bitmap *db;
    struct{
        Point p0, p1;    /* Endpoints of complete line */
        int mode;        /* Function code */
    }*fp;
    Obscured *op;
{
    clipline(db, r, fp->p0, fp->p1, fp->mode);
}
lline(lp, p, q, f)
    Layer *lp;
    Point p, q;
    int f;
{
    setline(p, q);   /* Initialize global parameters for line() */
    layerop(lp, Lline, lp->rect, p, q, f);
}
```

`setline()` initializes the global variables for the line, in particular Δx and Δy, and canonicalizes the line by coordinate transformation so that $0 < \Delta y / \Delta x \leq 1$. This line algorithm has advantages over an unadorned DDA, even for non-overlapping bitmaps: the sequence of dots generated is independent of the direction in which the line is drawn, and portions of a line may be ''undrawn''. Both these advantages stem from the DDA being a strict function of x, instead of the distance from the starting point of the line.

The point of this section is not that lines can be drawn in layers, but how `layerop` helps draw them. Given an object to be drawn — line, circle, ellipse, polygon, spline, etc. — and an operator that can draw the object in a bitmap, `layerop` will invoke the operator with the clipped rectangle/bitmap pairs that draw the complete object in a layer. The implementation of the bitmap operators can be tuned for performance; in our application, we build lines out of line segments with a restartable DDA for maximum speed, but draw arcs and circles as sequences of points because we don't draw them nearly as often. If circles were important to an application, the same sort of DDA restarting could be applied, using `layerop` to define the clipping rectangles for each component of the circle.

Conclusions

The software described here is in day-to-day use on hundreds of ''Blit'' terminals throughout Bell Labs, providing an asynchronous window interface to the multi-programmed Unix system. Unlike previous window systems, the layers software does not keep display lists [mey81], force ''repainting'' during window operations [tes81] or restrict graphical operations in any way. Instead, it supports general bitblt-style raster graphics by extending the notion of a bitmap to encompass rectangular regions which are subdivided into visible and invisible portions, and provides an elegant interface between the simple, traditional bitmap operations and the more general ones. The role of the traditional ''window manager'' is reduced to providing a user interface. In [mey81] appears the sentence, ''Research needs to be done to develop a way in which to conveniently store and manipulate graphics data in the context of a window manager.'' `layerop()` and its associated primitives meet this challenge.

Acknowledgements

Many people helped develop the algorithms discussed herein. The data structures grew out of discussions with Ken Thompson; Doug McIlroy guided me through the mazes of integer geometry; Peter Weinberger got my line-drawing theorems straight; Andrew Koenig simplified the Bitmap data structure; Andrew Hume spent a difficult week getting the details of `lbitblt()` right; John Reiser wrote an astonishing implementation of `bitblt()`, on which it all depends; and Brian Kernighan, Dave MacQueen and Chris van Wyk made helpful constructive criticism on an early manuscript, which led me to rethink and rewrite large portions of the software as well as the paper. I thank them all. I am most deeply indebted, however, to Bart Locanthi, who wrote the first `bitblt()`, greatly improved the interface to `layerop()`, and, most important, designed and built a bitmap terminal and placed it on my desk, thereby challenging me to tame the bitmap world.

References

[bre65] Bresenham, J. E., ''Algorithm for computer control of a digital plotter,'' *IBM Syst. J.,* **4** (1):25-30, 1965

[gui82] Guibas, L. J. and J. Stolfi, ''A Language for Bitmap Manipulation,'' *Transactions on Graphics,* **1** (3): 191, July 1982.

[ing81] Ingalls, D. H. H., ''The Smalltalk Graphics Kernel,'' *Byte,* **6** (8): 168, August 1981.

[lan79] Lantz, K. and R. Rashid, ''Virtual Terminal Management in a Multiple Process Environment,'' *Proceedings of the 7th Symposium on Operating Systems Principles,* p. 86, December 1979.

[mey81] Meyrowitz, N. and M. Mosher, ''BRUWIN: An Adaptable Design Strategy for Window Manager/Virtual Terminal Systems,'' *Proceedings of the 8th Symposium on Operating Systems Principles,* **15** (5): 180, December 1981.

[new79] Newman, W. M., and R. F. Sproull, *Principles of Interactive Computer Graphics, 2nd Ed.,* McGraw Hill, New York, 1979

[tes81] Tesler, Larry, ''The Smalltalk Environment,'' *Byte,* **6** (8): 90, August 1981.

[wei81] Weinreb, D. and D. Moon, ''Introduction to Using the Window System,'' Symbolics, Inc. 1981.