```
/* Expand the width of an image by p/q. */
void WeimanExpansion(
        const grayscalePixmap source,      /* Source image, size n × k */
        grayscalePixmap target,            /* Target image, width at least k * p/q */
        int n, int k,                      /* Size of source image */
        int p, int q)                      /* Scale factor is p/q */
{
    char roth[MAX];                        /* The array must hold at least p items. */
    int i, j, s;                           /* Loop indices */

    /* Source image is n × k, target image is to be n × ceil (k * p/q).[3] */
    int targetWidth = ceil (k * p / (double) q);

    Rothstein (roth, p, q);                /* Store the Rothstein code for p/q in array roth. */
    SetToBlank (target, n, targetWidth);   /* Clear the target array. */
    for (i = 0; i < p; i++) {              /* For several passes through the algorithm... */
        int sourceCol = 0;
        Permute (roth);                    /* Apply cyclic permutation to Rothstein code. */
        /* For each column of the target */
        for (j = 0; j < targetWidth; j++) {
            if (roth[j] == 1) {            /* If code says to copy source column */
                for (s = 0; s < n; s++) {  /* Copy all the pixels. */
                    target[s][j] += source[s][sourceCol];
                }
                sourceCol++;               /* Go to next column. */
            }
        }
    }

    /* Divide by q to compensate for adding each source column to target q times. */
    for (i = 0; i < n; i++)
        for (j = 0; j < targetWidth; j++)
            target[i][j] = rint (target[i][j] / (double) q);
} /* WeimanExpansion */
```

Fig. 17.5 The Weiman algorithm for expanding an image.

perform the identity transformation on an image, it blurs the values. In Section 17.5.3, we discuss this and other drawbacks of transformation algorithms.[4]

At this point, it is worthwhile to separate two aspects of image transformation. The first is computing which point in the source image is mapped to the center of the pixel in the

[3] The ceiling of a number is the smallest integer greater than or equal to the number; ceiling(1.6) = 2, ceiling(1.1) = 2, and ceiling(6.0) = 6.

[4] Feibush, Levoy and Cook note that any filter can be used, but describe the algorithm in terms of a filter of diameter 2. The algorithm generally performs better with this filter than it does with a unit-area box filter.

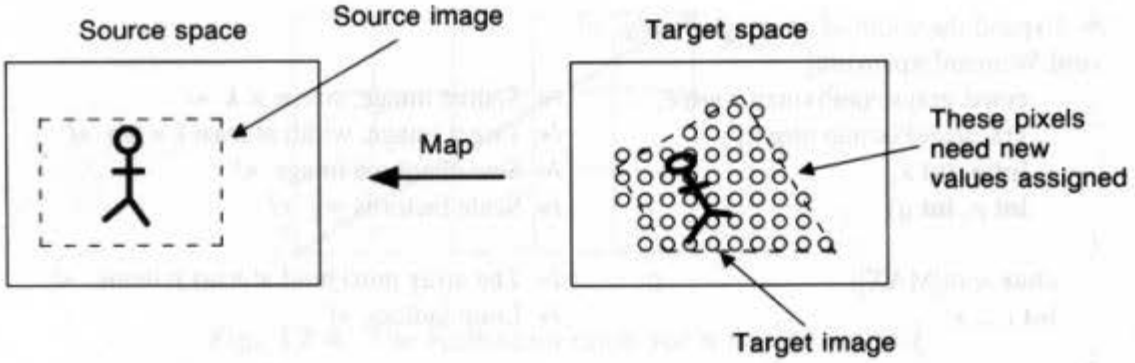


Fig. 17.6 The relationship of the source space, source image, target space, and target image in the Feibush–Levoy–Cook algorithm.

target image. The second is computing the value for the pixel in the target image. The first task is merely algebraic, in that it involves computing values (and inverse values) of a transformation. This may be done efficiently by various incremental methods. The second task also has numerous solutions, all of which involve choosing some filtering function to apply to the original image. The method described in the next few pages assumes a filtering function that is circularly symmetric and has a modest size (i.e., is nonzero only on a small part of the plane).

The algorithm starts with a source image (thought of as lying in one copy of the Euclidean plane, called the *source space*), a projective map[5] from another copy of the Euclidean plane (the *target space*) to the source space, and a polygonal region in the target space. The target image is the collection of pixels in the target space that are near the polygonal region, and it is these pixels whose values need to be assigned (see Fig. 17.6). Note that the projective map here goes from target to source, the reverse of the usual naming convention for mathematical functions.

To start, we choose a symmetric filter function that is nonzero only for (x, y) very close to $(0, 0)$ (perhaps within a 2-pixel distance). The *support* of this filter function is the set of points on which it is nonzero. We take a copy of the bounding rectangle for the support of the filter and translate it to each pixel in the target space. Whenever this rectangle intersects the target polygon, the pixel is considered to be in the target image. This translated rectangle is called the *bounding rectangle for the target pixel*, and the translated support of the filter function is called the pixel's *convolution mask* (see Fig. 17.7).

The vertices of the target polygon are transformed to source space just once, for repeated use. The resulting polygon is called the *source polygon*. The bounding rectangle of each target pixel is transformed to the source space, where it becomes a quadrilateral. A bounding rectangle for this quadrilateral is computed, then is clipped by the source polygon (because clipping a rectangle by the source polygon is much easier than clipping a general quadrilateral). The pixels in the source space that lie in this clipped quadrilateral are transformed to the target space; only those that fall within the target pixel's bounding rectangle are retained.

---

[5]A *projective map* is a map represented by a 3×3 matrix operating on the plane using homogeneous coordinates, in the manner described in Chapter 5.

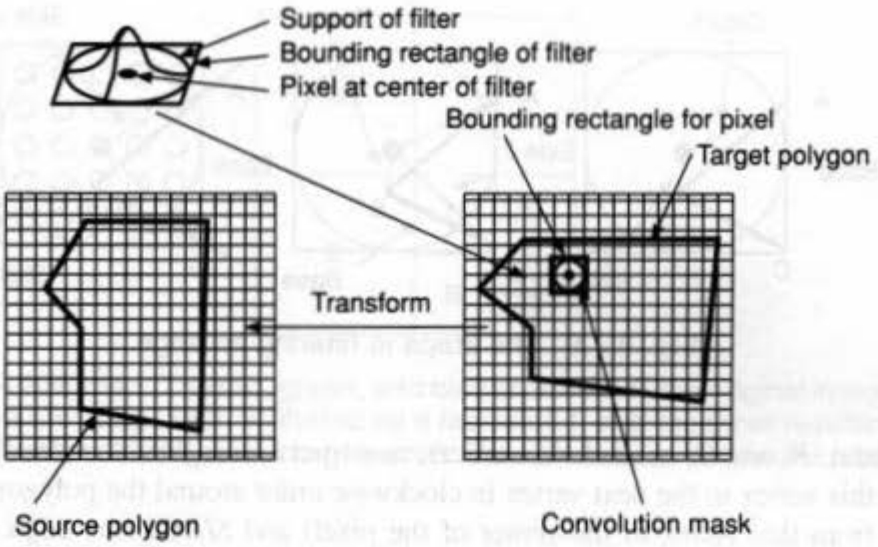


Fig. 17.7 Terms used in the Feibush–Levoy–Cook algorithm.

These transformed pixels are then averaged together by the weights given by the filter to yield a value for the target pixel. This target pixel value is correct only if the entire pixel is within the transformed image boundaries. If the image has been rotated, for example, then the transformed edges of the image may cut across pixels (more precisely, across their convolution masks).

Thus pixels are not entirely determined by the value just computed; that value only contributes to the pixel's value, in proportion to the coverage of the pixels. The contribution can be determined analytically. Figure 17.8 shows the transformed edge of the source image passing through a pixel's bounding rectangle, and within that rectangle passing through the pixel's convolution mask. To find the contribution of the computed value to the pixel's final value, we do the following:

1. Clip the image polygon against the bounding rectangle for the pixel (see Fig. 17.9). The points of intersection with the edges of the bounding rectangle were already computed in determining whether the pixel was in the target image.
2. For each vertex of the clipped polygon (in Fig. 17.9, a single triangle with vertices

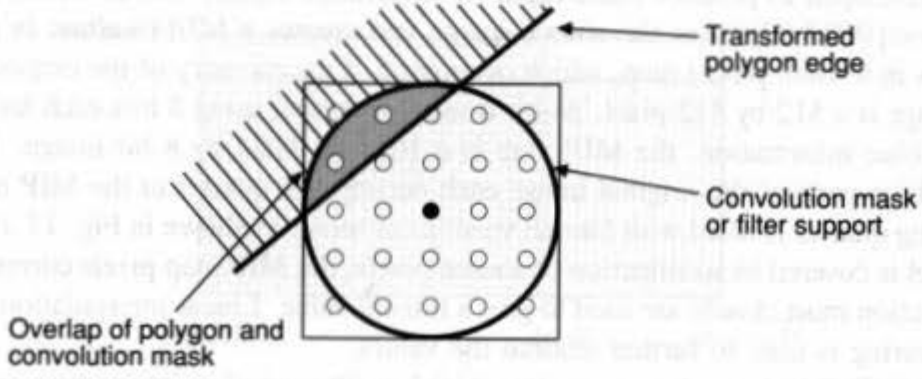


Fig. 17.8 Filtering for a pixel at the edge of the polygon.

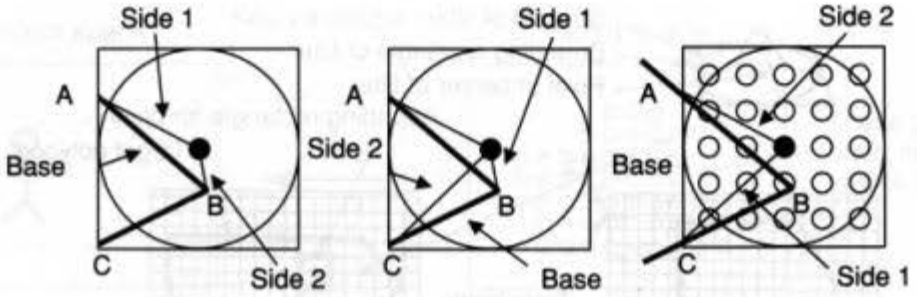


Fig. 17.9 The steps in filtering an edge.

labeled *A*, *B*, and *C*, in clockwise order), construct a triangle with sides *BASE* (the edge from this vertex to the next vertex in clockwise order around the polygon), *SIDE1* (the edge from this vertex to the center of the pixel) and *SIDE2* (the edge from the next vertex to the center of the pixel).

3. Consider the filter function as being plotted in a third dimension above the convolution mask. The weight a region contributes to the total for a pixel is proportional to the volume above that region and under the graph of the filter. So we now compute the volumes above each triangle. As Fig. 17.9 shows, some of these volumes must be added and some subtracted to create the correct total contribution for the region. The rule is that the volume is added if the cross-product of *SIDE1* and *SIDE2* points into the page; otherwise, it is subtracted (see Exercise 17.4).

Computing the volumes in step 3 is easier than it might appear, in that they can be precomputed and then extracted from a look-up table during the actual filtering process. Exercise 17.5 shows how to do this precomputation.

17.4.3 Other Pattern Mapping Techniques

The Feibush-Levoy-Cook algorithm provides excellent results for pattern mapping onto polygons, but requires computing a filtered value at each point, so that for each pixel in the image a filtering computation is performed. In a perspective picture of a plane (receding to a vanishing point) a single pixel in the final image may correspond to thousands of pixels in the source pattern, and thus require an immense filtering computation. Several techniques have been developed to produce more rapid (if sometimes slightly less accurate) filtering.

Williams [WILL83] takes the source image and creates a MIP (*multum in parvo*—many things in a small place) map, which occupies $\frac{4}{3}$ of the memory of the original. If the original image is a 512 by 512 pixel, 24-bit true color image, using 8 bits each for the red, green, and blue information, the MIP map is a 1024 by 1024 by 8 bit image. The red, green, and blue parts of the original image each occupy one quarter of the MIP map, and the remaining quarter is filled with filtered versions of these, as shown in Fig. 17.10. When a target pixel is covered by a collection of source pixels, the MIP map pixels corresponding to this collection most closely are used to give a filtered value. Linear interpolation between levels of filtering is used to further smooth the values.

Crow [CROW84] devised a scheme by which box filtering of an image over any aligned rectangle can be done rapidly. For quick pattern mapping, this suffices in many cases—a rectangular box corresponding closely to the shape of the transformed target pixel is used to

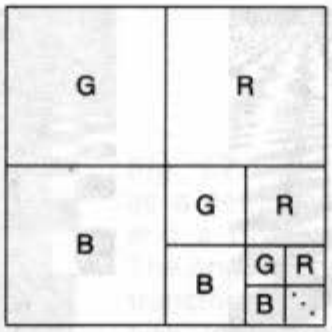


Fig. 17.10 A MIP map. The red, green, and blue channels of the original image fill three quarters of the MIP map. Each is filtered by a factor of 4, and the three resulting images fill up three quarters of the remaining quarter. The process is continued until the MIP map is filled.

compute a filtered pattern value for the pixel. The scheme is based on the algebraic identity $(x + a)(y + b) - (x + a)\,y - x\,(y + b) + xy = ab$. Interpreted geometrically, this says that the area of the small white rectangle in Fig. 17.11 can be computed by taking the area of the large rectangle and subtracting the areas of both the vertically and the horizontally shaded rectangles, and then adding back in the crosshatched rectangle (which has been subtracted twice). By taking the source image and creating a new image, whose value at pixel (x, y) is the sum of all the values in the source image in the rectangle with corners $(0, 0)$ and (x, y), we create a *summed area table*, S. We can now compute the sum of the pixels in the rectangle with corners at (x, y) and $(x + a, y + b)$, for example, by taking $S[x + a, y + b] - S[x + a, y] - S[x, y + b] + S[x, y]$.

Glassner [GLAS86] observes that if the transformed pixel is not approximately an aligned rectangle, then summed area tables may blur the result excessively. He therefore develops a system in which the excess area in the aligned bounding box for the pixel is systematically trimmed, in order to provide a more accurate estimate of the filtered source image at the point. This requires detecting the geometry of the inverse-mapped target pixel relative to its bounding box.

Heckbert [HECK86a] proposes a system using both the Feibush-Levoy-Cook method and MIP maps. He maps the target pixel's filter support (which is supposed to be circular,

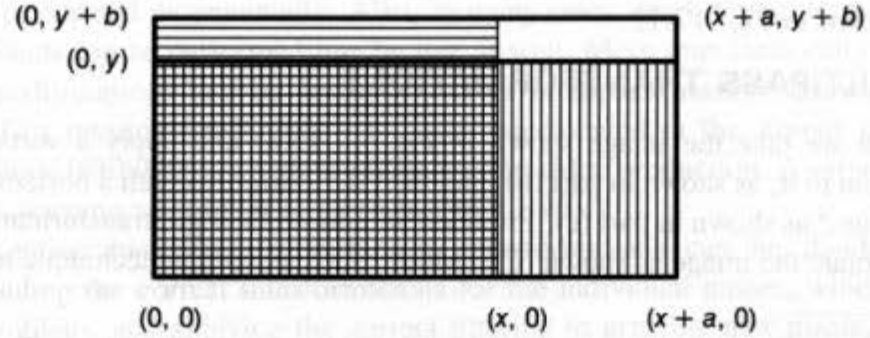


Fig. 17.11 The area of the small white rectangle in the image is computed by subtracting the horizontally and vertically shaded areas from the area of the large rectangle, and then adding back in the area of the crosshatched rectangle.

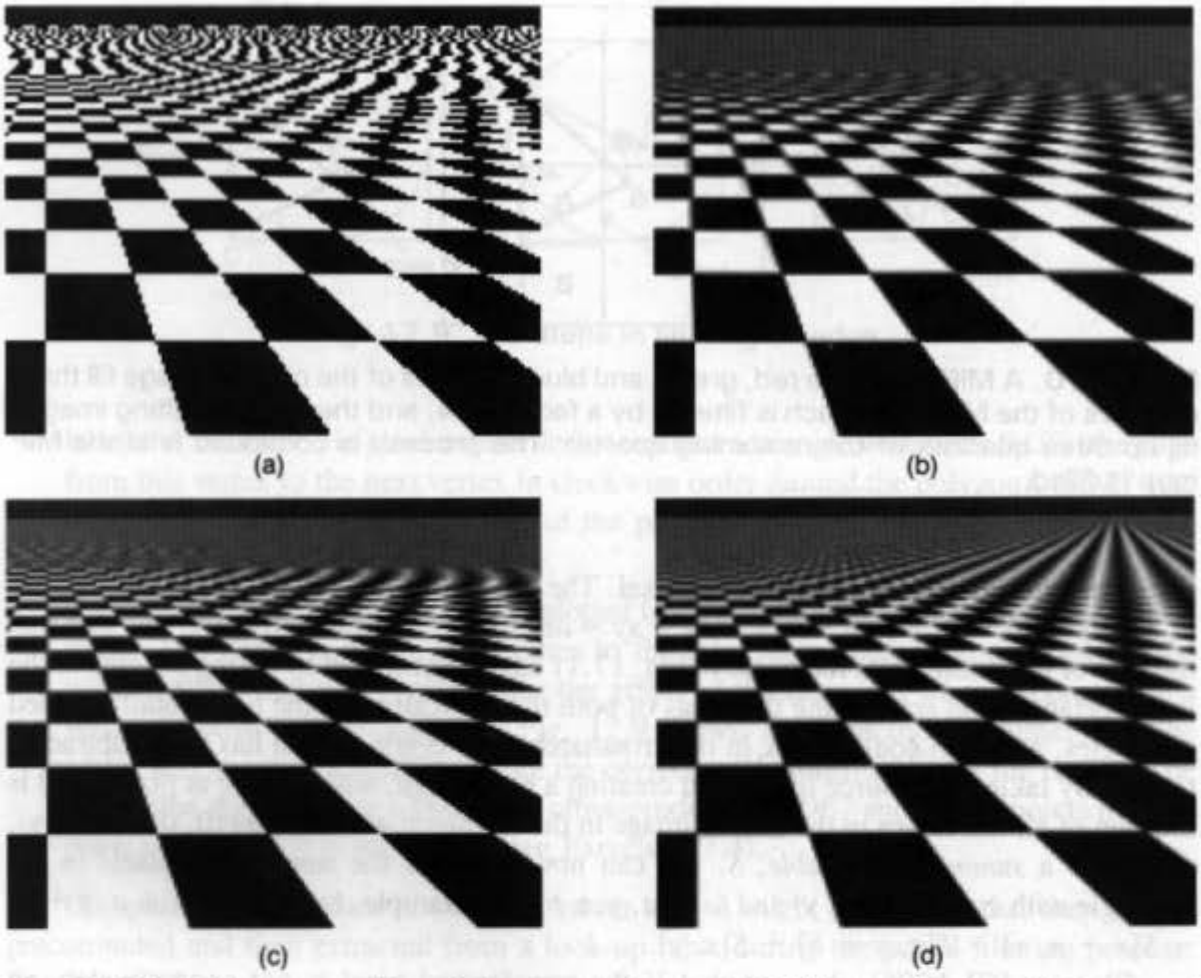


Fig. 17.12 (a) Point sampling of the source image. (b) MIP map filtering. (c) Summed area table filtering. (d) Elliptical weighted average using MIP maps and a Gaussian filter. (Courtesy of P. Heckbert.)

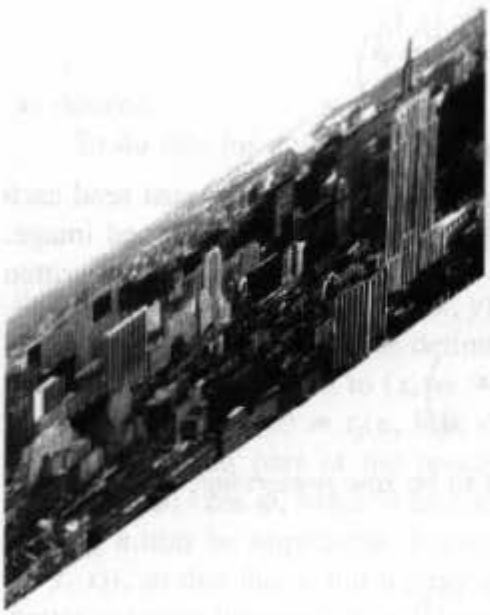
and is defined by a quadratic function) to an elliptical region in the source image (defined by a different quadratic). Depending on the size of this region in the source image, an appropriate level in a MIP map for the source image is selected, and the pixels within it are collected in a weighted sum over the elliptical region. This weighted sum is the value assigned to the target pixel. This combines the accuracy of the Feibush-Levoy-Cook technique with the efficiency of the MIP map system. A comparison of pattern-mapping results is shown in Fig. 17.12.

17.5 MULTIPASS TRANSFORMATIONS

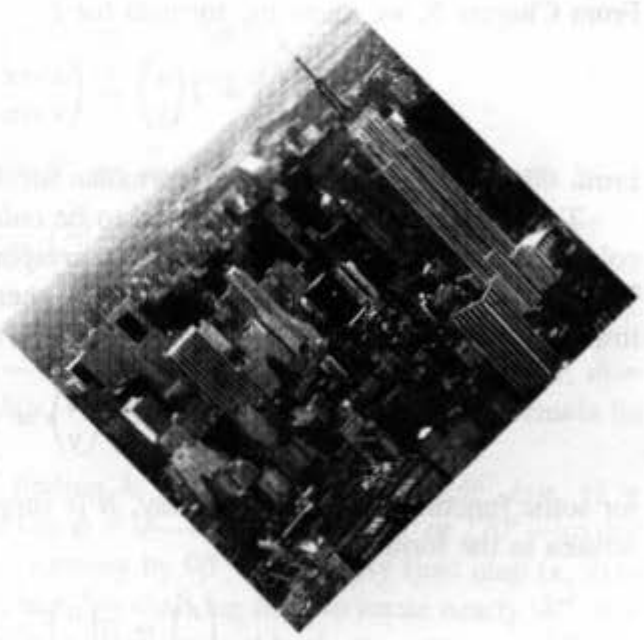
Suppose that we take the image shown in Fig. 17.13(a) and apply a vertical shearing transformation to it, as shown in part (b), and then we follow this with a horizontal shearing transformation,[6] as shown in part (c). Provided we choose the right transformations, the net effect is to rotate the image as shown [CATM80]. Such a two-pass technique may be much

[6]The two shearing transformations are actually shear-and-scale transformations. The first takes a column of pixels and translates and compresses it in the vertical direction. The second does the same for a row of pixels.

(a)



(b)



(c)

Fig. 17.13 A rotation may be expressed as a composition of a column-preserving and a row-preserving transformation. (a) The original image. (b) A column-preserving transformation has been applied to the image. (c) A row-preserving transformation has been applied to the second image. (Courtesy of George Wolberg, Columbia University.)

faster to compute than a direct application of the rotation transformation, since it operates on one vertical or horizontal line of pixels at a time, and the computations within each such line can be performed incrementally. Also, in many cases, the filtering necessary to avoid aliasing artifacts can be performed line by line as well. More important still is that a wide class of transformations can be implemented as multipass transformations [CATM80, SMIT87]. This multipass technique has been implemented in the Ampex digital optics (ADO) machine [BENN84], which is widely used in video production. A survey of this and other image warping techniques is given in [WOLB90].

Implementing two-pass (or multipass) transformations can be divided into two subtasks: finding the correct transformations for the individual passes, which is a purely algebraic problem, and applying the correct filtering to generate new pixels, which is an antialiasing problem. Since the second part will depend on the solution to the first, we begin by solving the first problem in the case of a rotation.

17.5.1 The Algebra of Multipass Transforms

To simplify the discussion, we will use three different sets of coordinates. The original image will be written in (x, y) coordinates, the vertically sheared image in (u, v) coordinates, and the final image in (r, s) coordinates. The first shearing transformation will be called A, the second B, and their composition, which is the rotation, will be called T. Thus,

$$\begin{pmatrix} u \\ v \end{pmatrix} = A \begin{pmatrix} x \\ y \end{pmatrix},$$

and

$$\begin{pmatrix} r \\ s \end{pmatrix} = B\begin{pmatrix} u \\ v \end{pmatrix} = B\left(A\begin{pmatrix} x \\ y \end{pmatrix}\right) = T\begin{pmatrix} x \\ y \end{pmatrix}.$$

From Chapter 5, we know the formula for T:

$$\begin{pmatrix} r \\ s \end{pmatrix} = T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \cos \phi - y \sin \phi \\ x \sin \phi + y \cos \phi \end{pmatrix}.$$

From this, we will determine the formulae for A and B.

The transformation A is supposed to be *column preserving*; that is, it must send each column of the original image into the corresponding column of the transformed image. Thus, if the pixel (x, y) is sent to (u, v) by A, then $u = x$. In other words, A must be written in the form

$$\begin{pmatrix} u \\ v \end{pmatrix} = A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ f(x, y) \end{pmatrix}$$

for some function f. In the same way, B is supposed to be *row preserving*, so B must be written in the form

$$\begin{pmatrix} r \\ s \end{pmatrix} = B\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} g(u, v) \\ v \end{pmatrix}$$

for some function g. To determine the formulae for A and B, we need to find the functions f and g.

Writing out the composite, we have

$$\begin{pmatrix} r \\ s \end{pmatrix} = B\begin{pmatrix} u \\ v \end{pmatrix} = B\left(A\begin{pmatrix} x \\ y \end{pmatrix}\right) = B\begin{pmatrix} x \\ f(x, y) \end{pmatrix} = \begin{pmatrix} g(x, f(x, y)) \\ f(x, y) \end{pmatrix}.$$

From this equation, we see that s and $f(x, y)$ are equal. Thus, the formula for s in terms of x and y gives the formula for $f(x, y)$: $f(x, y) = x \sin \phi + y \cos \phi$. Determining the formula for $g(u, v)$ is more complex. We know that, in terms of x and y, we can write $g(u, v) = x \cos \phi - y \sin \phi$. To write this in terms of u and v, we must solve for x and y in terms of u and v and substitute. Solving for x is easy, since we observed previously that $u = x$. Solving for y is slightly more difficult: $v = f(x, y) = x \sin \phi + y \cos \phi$, so $y = (v - x \sin \phi) / \cos \phi =$

$(v - u \sin \phi) / \cos \phi$. Substituting this result into the formula for $g(u, v)$ in terms of x and y, we get

$$g(u, v) = u \cos \phi - \frac{v - u \sin \phi}{\cos \phi} \sin \phi = u \sec \phi - v \tan \phi.$$

In summary, if we define

$$A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ x \sin \phi + y \cos \phi \end{pmatrix},$$

and

$$B\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} u \sec \phi - v \tan \phi \\ v \end{pmatrix},$$

then computing the composite gives

$$B\left(A\begin{pmatrix} x \\ y \end{pmatrix}\right) = \begin{pmatrix} x \cos \phi - y \sin \phi \\ x \sin \phi + y \cos \phi \end{pmatrix},$$

as desired.

To do this for a general transformation T, we must do exactly the same work. If

$$T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} t_1(x, y) \\ t_2(x, y) \end{pmatrix},$$

then we define $u = x$ and $v = f(x, y) = t_2(x, y)$. To define $g(u, v)$, we need to solve for y in terms of u and v, using these definitions—that is, to find a function h such that $(u, v) = (x, t_2(x, y))$ is equivalent to $(x, y) = (u, h(u, v))$. When we have found h, the formula for $g(u, v)$ is just $g(u, v) = t_2(u, h(u, v))$.

The difficult part of the process is finding h. In fact, in our example, $h(u, v) = (v - u \sin \phi) / \cos \phi$, which is undefined if $\cos \phi = 0$—that is, if $\phi = 90°$ or $270°$—so that finding h may be impossible. Fortunately, rotating by 90° is very easy (just map (x, y) to $(-y, x)$), so that this is not a problem. In fact, we shall see that, to rotate nearly 90°, it is better to rotate the full 90° and then to rotate a small amount back; thus, to rotate 87°, we would rotate 90° and then −3°. Algebraically, there is no difference between the two maps; at the pixel level, however, where filtering is involved, the difference is significant.

A rotation can also be broken into three transformations so as to avoid this *bottleneck* problem [PAET86; TANA86; WOLB90]. The decomposition for a rotation by ϕ is

$$\begin{bmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{bmatrix} = \begin{bmatrix} 1 & -\tan \phi/2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \sin \phi & 1 \end{bmatrix} \begin{bmatrix} 1 & -\tan \phi/2 \\ 0 & 1 \end{bmatrix}.$$

Note that each transformation involves a computation with one multiplication and one addition. Also, when $\phi > 90°$, we can do the rotation by first rotating by 180° and then by $180° - \phi$, so that the argument of the tangent function is never greater than 45°.[7]

[7]The tangent function is well behaved for angles near 0°, but has singularities at ±90°. Evaluating it for angles near 0° is therefore preferable.

To show that the multipass technique is not limited to rotations, let us factor a different map, which distorts a square into a trapezoid. (Such maps arise in the perspective transformations described in Chapter 6.) As an example, we take

$$T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x/(y+1) \\ y/(y+1) \end{pmatrix}.$$

Just as before, we wish to find functions

$$A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ f(x, y) \end{pmatrix} \quad \text{and} \quad B\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} g(u, v) \\ v \end{pmatrix}$$

such that $B\left(A\begin{pmatrix} x \\ y \end{pmatrix}\right) = T\begin{pmatrix} x \\ y \end{pmatrix}$. In this case, $v = f(x, y) = t_2(x, y) = y/(y+1)$. We need to find $g(u, v)$ so that $g(u, v) = x/(y+1)$. Solving the equation of f for y, we get $y = -v/(v-1)$. Thus (recalling that $u = x$), we can write $g(u, v) = u/(-v/(v-1)+1) = u/(-1/(v-1)) = u(1-v)$. Our two passes become

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} x \\ y/(y+1) \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} r \\ s \end{pmatrix} = \begin{pmatrix} u(1-v) \\ v \end{pmatrix}.$$

You should check that the composition of these transformations is really the original transformation T.

The technique has been generalized to handle other maps by Smith and colleagues [SMIT87]. Translation, rotation, scaling, and shearing all work easily. In addition, Smith considers functions of the form

$$T(x, y) = S(m(x)\, h_1(y),\ m(x)\, h_2(y)),$$

where S is a standard computer graphics transform—that is, a transformation of the plane by translation, scaling, rotation, and perspective transformations—and $m(x)$, $h_1(y)$ and $h_2(y)$ are arbitrary. He also considers maps T whose component functions $t_1(x, y)$ and $t_2(x, y)$ are bicubic functions of x and y, under the special hypothesis that T is injective (i.e., no two (x, y) points map to the same (r, s) point).

17.5.2 Generating Transformed Images with Filtering

When we transform an image by a row-preserving (or column-preserving) transformation, the source pixels are likely not to map exactly to the target pixels. For example, the pixels in a row might all be translated by $3\frac{1}{2}$ pixels to the right. In this case, we must compute values for the target pixels by taking combinations of the source pixels. What we are doing, in effect, is considering the values of the source pixels as samples of a function on a real line (the row); the values at the target pixels will be different samples of this same function. Hence, the process is called *resampling*.

The theoretically ideal resampling process is to take, for a given target pixel, a weighted average of the source pixels whose transformed positions are near it. The weights associated with each source pixel should be $\text{sinc}(kd)$, where d is the distance from the transformed source pixel to the target pixel and k is some constant. Unfortunately, this requires that every source pixel in a row contribute to every target pixel. As usual, we can instead work

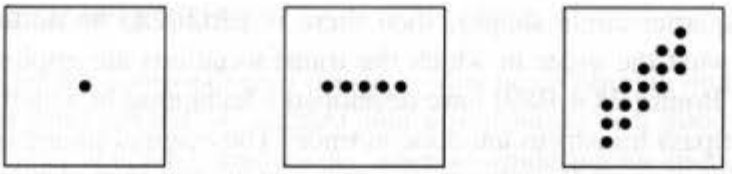


Fig. 17.14 The pixels that contribute to an output pixel in a two-pass rotation form a small area in (x, y) space. (a) A single pixel in (r, s) space; (b) The horizontal span of pixels in (u, v) space that contribute to the value of that pixel; (c) The pixels in (x, y) space that contribute to the values of the pixels in (u, v) space.

with various approximations to the sinc filter. The simplest is a box filter: Each source pixel is assumed to represent an interval in its row, and the endpoints of this interval are transformed. The contribution to the target pixel is the overlap of this transformed interval with the target pixel's interval multiplied by the value of the source pixel.

Using this method, each target pixel has a value that is a weighted average of a short span of source pixels (the length of the span depends on the exact transformation). In a two-pass rotation, a pixel in (r, s)-space has a value that is a weighted average of a horizontal span of (u, v)-pixels. Figure 17.14(a) shows a pixel in (r, s) space, and (b) shows the span of pixels in (u, v) space that contribute to the value of that pixel. Each of these (u, v) pixels, however, has a value that is an average of a vertical span of pixels in (x, y) space. Figure 17.14(c) shows these pixels in (x, y) space. Notice that the vertical spans in (x, y) space form a rhombus rather than a square, since the transformation from (x, y) to (u, v) is a shearing transformation.

We know from Chapter 14 that the pixels contributing to an output pixel really ought to form a circular shape[8] (i.e., the filter should be radially symmetric). If the rhombus is too different from a square, the filtering will begin to degenerate and will produce bad results. The result of such a separation of the filtering process into two component filters is discussed further in [MITC88].

In addition, we want to avoid the bottlenecking problem described previously, where many pixels in the source for a transformation contribute to each output pixel. In the case of a shear-and-scale operation, this occurs when the scale factor is small.

Thus, in doing two-pass rotations (or any other multipass transformation) we want to avoid extreme shearing or bottlenecking in our transformations. This is why rotating by 90° and then by $-3°$ is superior to rotating by 87°. In general, when we are constructing a two-pass transform, we can transform by rows and then columns, or by columns and then rows, or can rotate 90° before doing either of these. According to Smith, one of these approaches appears always to resolve the bottlenecking problem, at least for standard operations such as translation, rotation, scaling, and shearing [SMIT87]. Nonetheless, there are more general transformations where this technique may not succeed: If one portion of an image is rotated 90° while some other stays fixed (imagine bending a long thin

[8]This is particular to the case of a rotation. For a general transformation, the source pixels contributing to an output pixel should consist of those pixels that are transformed into a small disk about the target pixel; these pixels may or may not constitute a disk in the source image.

rectangle into a quarter-circle shape), then there is certain to be bottlenecking at some point, no matter what the order in which the transformations are applied.

Wolberg and Boult [WOLB89] have developed a technique in which two simultaneous versions of a multipass transform are done at once. The original image is first transformed by a row-preserving map and a column-preserving map, and then also is transformed by a 90° rotation and a different pair of maps. For both transformation sequences, the method records the amount of bottlenecking present at each pixel.

Thus, each output pixel can be computed in two different ways. Wolberg and Boult select, for each pixel, the route that has less bottlenecking, so that some portions of the image may be row–column transformed, whereas others are column-row transformed. Since the two sets of transformations can be performed simultaneously in parallel processors, this technique is ideally suited to implementation in hardware.

17.5.3 Evaluating Transformation Methods

There are several criteria for judging image-transformation algorithms. Filtering theory tells us that an image can be reconstructed from its samples, provided the original image had no high-frequency components. Indeed, from *any* set of samples, one can reconstruct *some* image with no high-frequency components. If the original image had no high frequencies, then we get the original back; if it did contain high frequencies, then the sampled image contained aliases, and the reconstructed image is likely to differ from the original (see Exercise 17.6).

So how should we judge a transformation algorithm? Ideally, a transformation algorithm would have the following properties:

- Translation by a zero vector should be the identity
- A sequence of translations should have the same effect as a single, composite translation
- Scaling up by a factor of $\lambda > 1$ and then scaling down by $1/\lambda$ should be the identity transformation
- Rotating by any sequence of angles totaling 360° should be the identity transformation.

Many workable algorithms clearly fail to satisfy any of these criteria. Weiman's algorithm fails on all but the fourth criterion. Feibush, Levoy, and Cook's algorithm fails on the first if a filter more than 1 pixel wide is used. Even Catmull and Smith's two-pass algorithm fails on all four criteria.

None of this is surprising. To resample an image, we ought to reconstruct it faithfully from its samples by convolving with a sinc filter. Thus, each new pixel ought to be a weighted average of *all* the pixels in the original image. Since all the methods are sensible enough to use filters of finite extent (for the sake of computational speed), all of them end up blurring the images.

There are many image-transformation methods not covered here. They each have advantages and disadvantages, mostly in the form of time–space tradeoffs. These are described in detail in an excellent survey by Heckbert [HECK86b].

17.6 IMAGE COMPOSITING

In this section, we discuss compositing of images—that is, combining images to create new images. Porter and Duff [PORT84] suggest that compositing is a good way to produce images in general, since it is fairly easy to do, whereas rendering the individual portions of the image may be difficult. With compositing, if one portion of the image needs alteration, the whole image does not need to be regenerated. Even more important, if some portions of an image are not rendered but have been optically scanned into memory instead, compositing may be the only way to incorporate them in the image.

We describe compositing using the α channel in Section 17.6.1, compositing using frame-buffer hardware in Section 17.6.2, the artificial generation of α values in Section 17.6.3, and an interface for image assembly in Section 17.6.4.

17.6.1 α-Channel Compositing

What sort of operations can be done in compositing? The value of each pixel in the composited image is computed from the component images in some fashion. In an *overlay*, the pixels of the foreground image must be given *transparency values* as well as whatever other values they may have (typically RGB or other color information). A pixel's value in the composited image is taken from the background image unless the foreground image has a nontransparent value at that point, in which case the value is taken from the foreground image. In a *blending* of two images, the resulting pixel value is a linear combination of the values of the two component pixels. In this section, we describe the Porter–Duff mechanism for compositing images using such combinations and transparency values [PORT84].

Suppose we have two images, one of a red polygon and one of a blue polygon, each on a transparent background. If we overlay the two images with the red polygon in front, then, at interior points of the front polygon, only the color red is visible. At points outside the front polygon but inside the back polygon, only blue is visible. But what about a pixel lying on the edge of the front polygon but inside the back polygon (see Fig. 17.15)? Here, the front polygon covers only part of the area of the pixel. If we color it red only, aliasing artifacts will result. On the other hand, if we know that the front polygon covers 70 percent

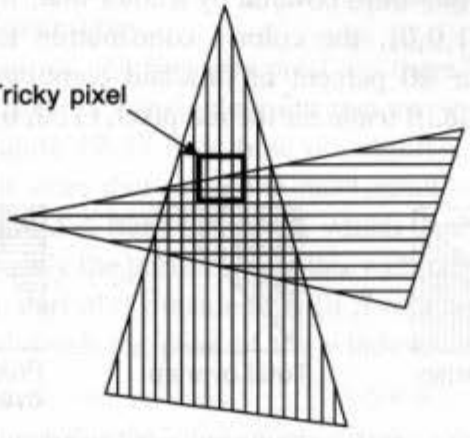


Fig. 17.15 Compositing operations near an edge: How do we color the pixel?

of the pixel, we can make the composited pixel 70 percent red and 30 percent blue and get a much more attractive result.

Suppose that as an image is produced, coverage information is recorded: The color associated with each pixel in the image is given an α *value* representing the coverage of the pixel. For an image that is to become the foreground element of a composited image, many of the pixels are registered as having coverage zero (they are transparent); the remainder, which constitute the important content of the foreground image, have larger coverage values (usually one).

To do compositing in a reasonable fashion, we need this α information at each pixel of the images being composited. We therefore assume that, along with the RGB values of an image, we also have an α value encoding the coverage of each pixel. This collection of α values is often called the α *channel* (see Section 17.7). Some types of renderers generate this coverage information easily, but it may be more difficult to generate for images that have been scanned into memory. We discuss this problem briefly in Section 17.6.3.

How do α values combine? Suppose we have a red polygon covering one-third of the area of a pixel, and a blue polygon that, taken separately, covers one-half of the area of the pixel. How much of the first polygon is covered by the second? As Fig. 17.16 shows, the first polygon can be completely covered, partly covered, or not covered at all. But suppose we know nothing more than the coverage information given. What is a *reasonable* guess for the amount of the first polygon covered by the second? Let us suppose that the area covered by the first is randomly distributed through the pixel area, and that the same is true of the second. Then any tiny spot's chance of being in the red polygon is $\frac{1}{3}$, and its chance of being in the blue polygon is $\frac{1}{2}$, so its chance of being in both is $\frac{1}{6}$. Notice that this value is exactly one-half of $\frac{1}{3}$, so that exactly one-half of the first polygon is covered by the second. This will be our general assumption: The area covered is distributed randomly across the pixel, so that the fraction of the first polygon covered by the second (within a particular pixel) is the same as the fraction of the whole pixel covered by the second polygon. The consequence of this assumption in practice is that compositing images with very fine detail that is parallel in the two images can have bad results. Porter and Duff report, however, that they have had no noticeable problems of this sort [PORT84].

Now, how do we compute the color of the pixel resulting from a 60—40 blend of these 2 pixels? Since the pixel is one-third covered by a color that, if it totally covered the pixel, would generate light of (1,0,0), the color's contribution to the light of the pixel is $(\frac{1}{3})(1,0,0)$. We want to take 60 percent of this and combine it with 40 percent of the other. We thus combine the RGB triple for the red pixel, (1, 0, 0), and the RGB triple for the

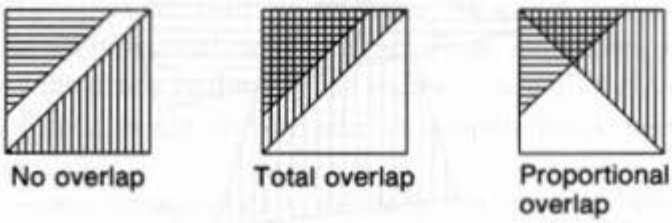


Fig. 17.16 The ways in which polygons can overlap within a pixel. In image composition, the first two cases are considered exceptional; the third is treated as the rule.

TABLE 17.1 AREAS AND POSSIBLE COLORS FOR REGIONS OF OVERLAP IN COMPOSITING

Region	Area	Possible colors
neither	$(1 - \alpha_A)(1 - \alpha_B)$	0
A alone	$\alpha_A(1 - \alpha_B)$	0, A
B alone	$\alpha_B(1 - \alpha_A)$	0, B
both	$\alpha_A\alpha_B$	0, A, B

blue pixel, (0, 0, 1), as follows: We say that

$$0.6\ (\tfrac{1}{3})(1,0,0) + 0.4\ (\tfrac{1}{2})(0, 0, 1) = (0.2, 0, 0.2)$$

is the resulting color.

Note that, whenever we combine 2 pixels, we use the product of the α value and the color of each pixel. This suggests that, when we store an image (within a compositing program), we should store not (R, G, B, α), but rather (αR, αG, αB, α) for each pixel, thus saving ourselves the trouble of performing the multiplications each time we use an image. Henceforth, when we refer to an RGBα value for a pixel, we mean exactly this. Thus, it should always be true that the R, G and B components of a pixel are no greater than that pixel's α component.[9] (In rare cases, we may want to consider pixels for which this condition is violated. Such pixels are effectively luminescent.)

Suppose now that two images, A and B, are to be combined, and suppose that we are looking at pixel P. If the α value of P in image A is α_A and the α value of P in image B is α_B, we can ask what fraction of the resulting pixel is covered by A only, what part by B only, what part by both, and what part by neither.

We have already assumed that the amount covered by both is $\alpha_A\alpha_B$. This means that $\alpha_A - \alpha_A\alpha_B$ is covered just by A, and $\alpha_B - \alpha_A\alpha_B$ is covered just by B. The amount left is 1 minus the sum of these three, which reduces algebraically to $(1 - \alpha_A)(1 - \alpha_B)$. In the composited image, the area that was covered by A alone might end up with color A or no color, and similarly for color B. The area that was covered neither by A nor B should end up with no color, and the area covered by both might end up with no color, color A, or color B. Table 17.1 lays out these possibilities.

How many possible ways of coloring this pixel are there? With three choices for the both-colors region, two for each of the single-color regions, and one for the blank region, we have 12 possibilities. Figure 17.17 lists these possibilities.

Of these operations, the ones that make the most intuitive sense (and are most often used) are A **over** B, A **in** B and A **held out by** B, which denote respectively the result of hiding B behind A, showing only the part of A that lies within B (useful if B is a picture of a hole), and showing only the part of A outside of B (if B represents the frame of a window, this is the part that shows through the pane of the window).

[9] In a fixed-point scheme for representing the colors, pixels whose α value is small have a more discrete realm of colors than do those whose α value is 1. Porter and Duff say that this lack of resolution in the color spectrum for pixels with small α values has been of no consequence to them.

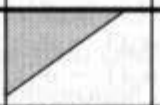
operation	quadruple	diagram	F_A	F_B
clear	(0, 0, 0, 0)		0	0
A	(0, *A*, 0, *A*)		1	0
B	(0, 0, *B*, *B*)		0	1
A **over** *B*	(0, *A*, *B*, *A*)		1	$1-\alpha_A$
B **over** *A*	(0, *A*, *B*, *B*)		$1-\alpha_B$	1
A **in** *B*	(0, 0, 0, *A*)		α_B	0
B **in** *A*	(0, 0, 0, *B*)		0	α_A
A **held out by** *B*	(0, *A*, 0, 0)		$1-\alpha_B$	0
B **held out by** *A*	(0, 0, *B*, 0)		0	$1-\alpha_A$
A **atop** *B*	(0, 0, *B*, *A*)		α_B	$1-\alpha_A$
B **atop** *A*	(0, *A*, 0, *B*)		$1-\alpha_B$	α_A
A **xor** *B*	(0, *A*, *B*, 0)		$1-\alpha_B$	$1-\alpha_A$

Fig. 17.17 The possibilities for compositing operations. The quadruple indicates the colors for the "neither," "A," "B," and "both" regions. Adapted from [PORT84]. (Courtesy of Thomas Porter and Tom Duff.)

In each case we can compute, as before, how much of each color should survive in the result. For example, in *A* **over** *B*, all the color from image *A* survives, while only a fraction $(1 - \alpha_A)$ of the color from *B* survives. The total color in the result is $c_A + (1 - \alpha_A)c_B$, where c_A denotes the color from image *A* (with α_A already multiplied in).

In general, if F_A denotes the fraction of the pixel from image *A* that still shows, and similarly for F_B, then the resulting color will be $F_A c_A + F_B c_B$. We must also compute the resulting α value: If fraction F_A of *A* is showing, and the original contribution of *A* is α_A, then the new contribution of *A* to the coverage is $F_A \alpha_A$. The same goes for *B*; hence, the total coverage for the new pixel is $F_A \alpha_A + F_B \alpha_B$.

A few unary operations can be performed on images. For example, the **darken** operation is defined by

$$\textbf{darken}(A, \rho) := (\rho R_A, \rho G_A, \rho B_A, \alpha_A) \qquad 0 \le \rho \le 1,$$

which effectively darkens a pixel while maintaining the same coverage. In contrast, the **fade** operator acts by

$$\textbf{fade}(A, \delta) := (\delta R_A, \delta G_A, \delta B_A, \delta \alpha_A) \qquad 0 \le \delta \le 1$$

which causes a pixel to become more transparent while maintaining its color (the color components are multiplied by δ because of the requirement that the colors must always be premultiplied by the α value).

A natural third operator (essentially a composite of these two) is the **opaque** operator, which acts on the α channel alone and is defined by

$$\textbf{opaque}(A, \omega) := (R_A, G_A, B_A, \omega \alpha_A).$$

As ω varies between 0 and 1, the coverage of the background by the pixel changes. Of course, if ω is made small, one of the color components may end up larger than the α component. For example, if ω is zero, then we get a pixel with color but a zero α component. Such a pixel cannot obscure anything, but can contribute light to a composited pixel; hence, it is called *luminous*. The possibility that color components can be larger than the α component requires that we clip the colors to the interval [0, 1] when reconstructing the true color (to reconstruct the color, we compute $(R/\alpha, G/\alpha, B/\alpha)$, and then clamp all three numbers to a maximum of 1).

One last binary operator, **plus,** is useful. In this operation, the color components and α components are added. Thus, to fade smoothly from image A to image B, we use

$$\textbf{fade}(A, t) \textbf{ plus } \textbf{fade}(B, 1 - t),$$

and let t vary from 1 to 0. Judicious combinations of these operations let us combine pictures in a great variety of ways.

We should examine the consequences of the assumptions made at the start: If a pixel is covered a fraction F_1 by one polygon and a fraction F_2 by another, then the first polygon (within the pixel) is covered a fraction F_2 by the second. If the two polygons happen to overlap in a reasonable way, this works fine, as in Figure 17.17. But if they happen to cross the pixel as parallel stripes, as in Fig. 17.18, then the assumption is invalid. When geometric entities are being composited, this problem is not particularly likely to occur.

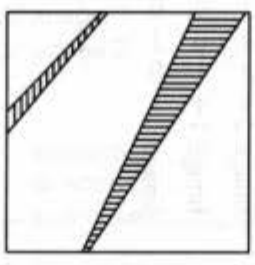


Fig. 17.18 A case where the overlap assumptions fail in compositing.

However, when repeated instances of an image are composited atop one another, it may happen frequently.

To illustrate the use of compositing, we describe the composition of the frame from the Genesis effect described in Chapter 20, and shown in Color Plate IV.14. In this case, there are four images composited: *FFire*, the particle systems in front of the planet; *BFire*, the particle systems in back of the planet; *Planet*, the planet itself; and *Stars*, the background star field. The composite expression for the image is [PORT84, p. 259]:

(*FFire* **plus** (*BFire* **held out by** *Planet*)) **over darken** (*Planet*, 0.8) **over** *Stars*.

The planet is used to mask out the parts of the particle systems behind it, and the results are added to the front particle systems. These are composited over a slightly darkened planet (so that the particles obscure the underlying planet), and the result is placed over the background star field.

17.6.2 Alternate Compositing Methods

Let us consider two other mechanisms for compositing. The first of these is to composite images in compressed form (e.g., bitmaps that are stored with 8 pixels per byte), by doing very simple operations (*A* **over** *B*, *A* **and** *B*, *A* **xor** *B*) on the compressed forms. These operations are simple to implement: For any two possible bytes, we use a bitwise operation in hardware to combine them. In a more sophisticated version, the algorithm can be extended to handle this sort of compositing on run-length–encoded bitmaps.

The other compositing technique uses the frame-buffer hardware to implement compositing. Consider a simple example. We usually think of a 3-bit-per-pixel bitmap as containing a single image with 3 bits per pixel. However, we can also think of it as containing two images, one with 2 bits per pixel and the other with 1 bit per pixel, or as three separate images, each with 1 bit per pixel. In any case, the look-up table is used to select or combine the separate images to form a single composite image displayed on the view surface. For instance, to display only image 2, the image defined by the high-order bit of each pixel value, we load the table as shown in Fig. 17.19. To display image 0, the image

Entry number (decimal)	Entry number (binary)			Contents of look-up table (decimal)
	Image 2	Image 1	Image 0	
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	7
5	1	0	1	7
6	1	1	0	7
7	1	1	1	7
				0 = black 7 = white

Fig. 17.19 Look-up table to display an image defined by the high-order bit of each pixel.

Entry number (decimal)	Entry number (binary)			Contents of look-up table (decimal)
	Image 2	Image 1	Image 0	
0	0	0	0	0
1	0	0	1	2
2	0	1	0	2
3	0	1	1	4
4	1	0	0	2
5	1	0	1	4
6	1	1	0	4
7	1	1	1	6

Fig. 17.20 Look-up table to display a sum of three 1-bit images.

defined by the low-order bit of each pixel, we load the table with a 7 in those locations for which the low-order bit is 1, and with a 0 for the other locations: 0, 7, 0, 7, 0, 7, 0, 7. If the displayed image is to be the sum of the three images and if each image that is "on" at a pixel is to contribute two units of intensity, then we load the table as shown in Fig. 17.20. If 1 of the 3 pixel bits is on, a 2 is placed in the table; if 2 of 3 are on, a 4; and if 3 of 3 are on, a 6.

As another example, think of each image as being defined on parallel planes, as in Fig. 17.21. The plane of image 2 is closest to the viewer; the plane of image 0 is farthest away. Thus, image 2 obscures both images 0 and 1, whereas image 1 obscures only image 0. This priority can be reflected in the look-up table, as shown in Fig. 17.22. In this case, image 2 is displayed at intensity 7, image 1 at intensity 5, and image 0 at intensity 3, so that images "closer" to the viewer appear brighter than those farther away. Where no image is defined, intensity 0 is displayed.

Yet another possibility is to use the look-up table to store a weighted sum of the intensities of two images, creating a double-exposure effect. If the weight applied to one image is decreased over time as the weight applied to the other is increased, we achieve the fade-out, fade-in effect called a *lap-dissolve*. When colored images are used, the colors displayed during the fade sequence depend on the color space in which the weighted sum is calculated (see Chapter 13).

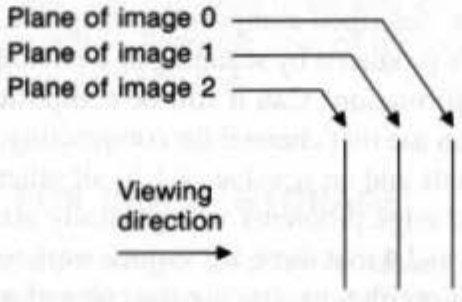


Fig. 17.21 Relation of three images to the viewing direction.

Entry number (decimal)	Entry number (binary)			Contents of look-up table (decimal)
	Image 2	Image 1	Image 0	
0	0	0	0	0 no image present
1	0	0	1	3 image 0 visible
2	0	1	0	5 image 1 visible
3	0	1	1	5 image 1 visible
4	1	0	0	7 image 2 visible
5	1	0	1	7 image 2 visible
6	1	1	0	7 image 2 visible
7	1	1	1	7 image 2 visible

Fig. 17.22 Look-up table to assign priorities to three 1-bit images.

Deciding how to load the look-up table to achieve a particular result can be tedious, especially if many images are used and the table has many entries. The Image Composition Language (ICL) [FOLE87c] allows the programmer to declare images (made up of one or more bit planes) as variables. The image to be displayed is described by a composition expression consisting of variables combined by arithmetic, relational, and conditional operations. The lap-dissolve is specified in ICL with the expression

$$newImage * t + oldImage * (1 - t),$$

in which the variables *oldImage* and *newImage* are images in the bit plane and t is a scalar variable varying from 0 to 1. The following composition expression adds red (the triplets in braces are RGB color specifications) to those values of an image c in the range [0.6, 0.8] and green to those values in the range [0.3, 0.5]:

if ($c < 0.8$) **and** ($c > 0.6$) **then** c + {1,0,0}
  else if ($c < 0.5$) **and** ($c > 0.3$) **then** c + {0,1,0}
  else c
endif endif

This short composition expression replaces a code segment that has a considerably larger number of lines.

17.6.3 Generating α Values with Fill Mechanisms

In the preceding section, we described compositing images that come equipped with an α channel. What if an image is produced by scanning of a photograph or is provided by some other source lacking this information? Can it still be composited? If we can generate an α channel for the image, we can use that channel for compositing. Even if we merely assign an α value of zero to black pixels and an α value of 1 to all others, we can use the preceding algorithms, although ragged-edge problems will generally arise.

Recall from Chapters 3 and 4 that there are various ways to *fill* regions of an image with new values. If we use a fill algorithm to alter not the color of a pixel but its α value, we can assign α values to various regions in the image. If the colors in the image represent foreground and background (e.g., a picture of a person standing in front of a white wall),

we can choose to fill the background region with its original color but assign it a reduced α value. Fishkin and Barsky give an algorithm for recognizing regions that consist of pixels that are either entirely or partially made up of some color [FISH84]. For the details of this algorithm, see Section 19.5.3.

Of course, applying a seed-fill algorithm (see Section 19.5.2) is bound to fail if the background is not a single connected piece. If we attempt to correct this difficulty by applying the Fishkin–Barsky criterion for similarity to the background to every pixel in the image, items in the foreground whose colors are close to the background color are treated incorrectly. (Imagine the person in our example, standing in front of a white wall. Is the white writing on his green T-shirt part of the background?) If we try to seed fill each separate piece of background, the task may be hopeless. (Imagine our person again and suppose that the background shows through his curly hair. There may be thousands of background regions.) Nonetheless, a soft-seedfill of the background to determine new α values makes a good preliminary step, and for simple images can improve substantially on the approach of assigning α values of 0.0 or 1.0 to every pixel.

17.6.4 An Interface for Image Assembly

How are the tools used for compositing and applying geometric transformations of images used in practice? An image like Color Plate IV.19 is described by a complex collection of operations. The compositing operations such as A **over** B can be described by a tree structure, where the leaf nodes are images and the internal nodes are operators with operands as child nodes. But before the images can be composited, they must be placed correctly. This requirement suggests an *image-assembly tree* structure, in which each internal node is either an image transformation or a compositing operation, and each leaf node is an image.

Such an image-assembly structure can be implemented in a convenient user interface, in which the user adds nodes or moves pieces of the tree with a mouse, and places a marker at some node to view the image described by that node and its children. This view can be structural, merely showing the relative positions and sizes of the child images within the parent. On a workstation with limited color capabilities, the view can be a dithered version of the true-color image; on a sophisticated workstation, it can be a full-sized version of the image in full color. The structural view can be extremely useful, since the user can edit the geometric transformations in this view by mouse dragging, eliminating the need to type exact coordinates for geometric transformations; of course, the ability to enter precise coordinates is also essential.

An image assembler of this sort has been developed by Kauffman [KAUF88a]. Far more sophisticated image assemblers form the core of the video processors that generate many of the special effects seen on television.

17.7 MECHANISMS FOR IMAGE STORAGE

When we store an image, we are storing a 2D array of *values*, where each value represents the data associated with a pixel in the image. For a bitmap, this value is a binary digit. For a color image, the value may be a collection of three numbers representing the intensities of the red, green, and blue components of the color at that pixel, or three numbers that are

indices into tables of red, green, and blue intensities, or a single number that is an index into a table of color triples, or an index into any of a number of other data structures that can represent a color, including CIE or XYZ color systems, or even a collection of four or five spectral samples for each color.

In addition, each pixel may have other information associated with it, such as the z-buffer value of the pixel, a triple of numbers indicating the normal to the surface drawn at that pixel, or the α-channel information. Thus, we may consider an image as consisting of a collection of *channels*, each of which gives some single piece of information about the pixels in the image. Thus, we speak of the *red, green,* and *blue channels* of an image.

Although this idea might seem contrary to good programming practice, in which we learn to collect information associated with a single object into a single data structure, it often helps to separate the channels for convenience in storage. However, some methods of image compression do treat the image as a 2D array, such as the quadtree and fractal encoding schemes described later, so separation into channels is inappropriate.

Before discussing algorithms for storing images as channels or arrays, we describe two important methods for storing pictures: use of the *metafile* and use of *application-dependent data*. Neither of these is, strictly speaking, an image format, but each is a mechanism for conveying the information that is represented in an image.

If an image is produced by a sequence of calls to some collection of routines, a metafile stores this sequence of calls rather than the image that was generated. This sequence of calls may be far more compact than the image itself (an image of the Japanese flag can be produced by one call to a rectangle-drawing routine and one call to a circle-drawing routine, but could take several MB to store as RGB triples). If the routines are sufficiently simple or are implemented in hardware, redisplaying a metafile image may be faster than redisplaying a pixel image. The term *metafile* is also used to refer to a device-independent description of a standardized data structure, such as the PHIGS data structure described in Chapter 7. To store an image in such a metafile, we traverse the current data structure and record the data structure in some device-independent fashion for redisplay later. This description may be not a sequence of function calls, but instead a textual transcription of some hierarchical structure.

The second storage scheme entails application-dependent data. If an application displays a particular class of images, it may be convenient to record the data from which these images were created, or even differences between the data and some standard set of data. If the images are all head-on views of human faces described as polygons, it may be simpler to store just a list of those polygons whose positions are different from their position in some standard facial image (and their new positions, of course). A more extreme version of this sort of condensation of information has been use in the Talking Heads project at the MIT Media Lab, in which only the positions of eyeballs, lips, and other high-level features are stored [BOLT84]. At this point, image description becomes more of a scene description, and properly belongs to the domain of modeling, rather than to that of image storage.

17.7.1 Storing Image Data

Now let us consider how to store the sort of image that consists of several channels of data. If our displays expect to be given information about an image in the form of RGB triples, it

may be most convenient to store the image as RGB triples. But if space is at a premium, as is often the case, then it may be worth trying to compress the channels in some way. Approaches to compression must be weighed against the cost of decompression: The more sophisticated the compression technique, the more likely decompression is to be expensive. Although all of these techniques apply equally well to any channel of information, our discussion will be couched in terms of color channels, since these are the ones most often present in images (z-buffer, normal vector, and other information being optional).

If an image has few colors and each color occurs many times (as in an image of a newspaper, in which there may be only black, dark gray, light gray, and white), it may be worthwhile to make a table of colors that occur (here, the table would have only four entries), and then to make a *single* channel that is an index into this color table. In our newspaper example, this single channel would need only 2 bits of information per pixel, rather than perhaps 8 bits per color per pixel; the resulting image is compressed by a factor of 12. In pictures with more colors, the savings are less substantial; in the extreme case where each pixel in the image is a different color, the look-up table is as large as the image would have been if stored as RGB triples, and the indices into the look-up table take even more space. Roughly speaking, indexing into a look-up table begins to be worthwhile if the number of colors is less than one-half the number of pixels. (Of course, if the hardware for displaying the image works by using look-up tables as well, it may be easier and faster to store the image in this fashion than as RGB triples. Typically, such hardware provides a modest space for the look-up table, about 8 to 12 bits per pixel.)

This single-channel approach still requires at least one piece of information per pixel. If the image has a great deal of repetition, it may be possible to compress it further by *run-length encoding* a channel. Run-length encoding consists of giving a count and a value, where the count indicates the number of times the value is to be repeated. The design of the Utah Raster Toolkit [PETE86] includes a number of improvements on this basic idea. For instance, the count, n, is an 8-bit *signed* integer (with values -128 through 127): a negative count indicates that n pixels' worth of unencoded data follow; a nonnegative count indicates that the next piece of information is the value to be used for $n + 1$ pixels. Further improvements might include reserving certain negative values for special meanings: -128 might indicate that the next few bytes of information give a scan line and position to which to jump (in order to skip the recording of large areas of background color), and -127 might be reserved to indicate a jump to the start of a specified scan line. Such a naive run-length–encoding scheme at worst adds 1 byte for every 126 values (a -126 indicating that 126 pixels worth of unencoded data follow), a cost of about 0.25 percent for an image with 8 bits per pixel for each of red, green, and blue. In the best case, an image in which all pixels have the same value, the compression would be by a factor of about 100: 128 pixels' worth of values compress to 1 pixel's worth (24 bits, in this example), but the count byte adds another 8 bits.

There are other clever formats for compressing channels. For example, we could store the value of each pixel from a bitmap in an integer (as a 0 or 1), but most of the bits in the integer would be wasted. Instead, we might store one pixel value in each bit (this is the origin of the term *bitmap*). If the image being represented contains regions filled with patterns whose width is a factor of 8, then we can perform a similar run-length encoding, in which the first byte gives a count, n, and the next byte gives a pattern to be repeated for the next $8n$ pixels of the bitmap. This method is less likely to generate savings than is the

ordinary run-length encoding for color images, since a block of 8 values must be repeated for any compression to take place.

Run-length encoding and other standard information-theory approaches such as Huffman encoding treat the image in channels, which can be imagined as linear arrays of values (although multiple channels can be considered a single large channel, so that we can run-length encode sets of RGB triples as well). Other methods treat the image as a 2D array of values, and hence can exploit any inter-row coherence. One of these techniques is based on the use of quadtrees.

The fundamental idea of the quadtree-based image description is that a region of an image may be fairly constant, and hence all pixels in the region can be treated as having the same value. Determining these near-constant regions is the core of the algorithm. This algorithm can be used either on a single component of the image, such as the 2D array of red values, or on the aggregate value associated with each pixel; for simplicity, we shall describe the algorithm for a single numerical component. The algorithm requires a mechanism for determining the mean value of the image in a region, and the extent of the deviations from the mean within a region.

The image is first considered as a whole. If the deviation from the mean in the image is sufficiently small (less than or equal to some nonnegative tolerance), then the image is reported as having a value equal to the mean, repeated over the entire image. (If the tolerance is set to zero, then the image really must be constant for this to occur.) If the deviation from the mean is not smaller than the tolerance, the mean of the image is recorded, the image is divided into quadrants, and the same algorithm is applied to each quadrant. The algorithm terminates because repeated subdivision of quadrants eventually breaks them into single-pixel regions, if necessary; for a single-pixel region, the deviation from the mean must be zero, and hence is less than or equal to any tolerance value.

We can improve the algorithm by recording not the mean of the image, but rather the means of the four quadrants whenever the image is subdivided, and the mean of the image when it is not. The advantage is that, when the image is redisplayed, if the quadtree is parsed breadth-first, the display may be constantly updated to show more and more refined images. The first image is four colored rectangles. Then, each rectangle is subdivided and its color is refined, and so on. In a system designed for scanning through a large number of images, this approach may be extremely convenient: after just a few bytes of information have been transmitted, a general sense of the image may begin to appear, and the user may choose to reject the image and to move on to the next one. This rapid detection of the sense of an image is especially useful if the images are transmitted over a low-bandwidth communications channel. Quadtree compression of images has been exploited by Knowlton, Sloan, and Tanimoto [KNOW80, SLOA79], and the algorithm has been further refined by Hill [HILL83]. Exercise 17.7 discusses other mechanisms for building a quadtree describing an image, some of which may be more efficient than the one described here.

17.7.2 Iterated Function Systems for Image Compression

A second image-compression algorithm is based on the notion of iterated function systems (IFSs). In this case, the compression factor can be extremely high, but the cost of

compressing the image tends to be large as well. The algorithm requires, for each image, that the user interactively solve a geometric problem, described later [BARN88a]. Also, like all nondestructive compression schemes, the pigeonhole principle[10] says that, if some images are compressed, others must be expanded by some modest amount (since there can be no one-to-one mapping of all n by k arrays to all p by q arrays where pq is less than nk). The advantage of the IFS technique is that images with substantial geometric regularity are the ones that are compressed, whereas those that look like noise are more likely to be expanded.

An *IFS code* is a finite collection of affine maps $\{w_1, \ldots, w_r\}$ of the plane to itself, together with a probability p_i associated with w_i. The maps must be *contractive*; that is, the distance between points must be reduced by the maps, on the average (the precise requirement is described in [BARN88a]). Recall that an affine map of the plane is given by a formula of the form

$$w\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}; \tag{17.1}$$

so it is entirely determined by six numbers, a, b, c, d, e, and f. Notice that these affine maps are just combinations of rotations, translations and scalings in the plane. The condition that they be contractive says that the scaling factors must be less than 1.

The next several pages give a rough description of how to produce a gray-scale image from an IFS code; the method is easily generalized to producing three gray-scale images from three IFS codes, that can then be used as the RGB components of a color image. ([BARN88a] uses a somewhat different scheme for encoding color.) This production of an image from an IFS code is essentially this decompression part of the IFS algorithm; we discuss it first.

Consider a rectangle, V, in the plane defining our image, and imagine V as divided into a rectangular grid whose subrectangles are V_{ij}, $i = 1, \ldots, n; j = 1, \ldots, k$. Choose a point (x_0, y_0) that remains fixed under one of the maps (say w_1, without loss of generality). We now proceed to apply the maps $w_1, \ldots, w_r$ in fairly random order (determined by the probabilities p_i), and watch where the point (x_0, y_0) is sent. We use the number of times it lands in V_{ij} for each i and j to determine the eventual brightness of pixel $[i, j]$ in the image. The pseudocode for this process is shown in Fig. 17.23.

Before the last step of this algorithm, each image$[i, j]$ entry indicates how often the starting point, in the course of being moved randomly by the affine maps, falls into the $[i, j]$th square of the image. For this number accurately to represent the *probability* of falling into the square over an infinite sequence of steps, the number of iterations of the algorithm must be very large: K should be a large multiple of the number of pixels in the image.

The effect of this algorithm is essentially to create a picture of the *attractor* of the IFS. The attractor is a set, A, with the property that, if all the affine maps are applied to A, and the results are combined, the result is A:

$$A = \bigcup_{i=1}^{r} w_i(A).$$

[10]The pigeonhole principle is the observation that, if more than m objects are placed in m boxes, then some box must contain more than one object.

```
void IFS (double image[MAX][MAX])
/* Given a collection of affine maps w_i, with associated probabilities p_i, */
/* which are global variables, generate a gray-scale image. */
{
    int x, y;                          /* A location in the plane */
    int i, j;                          /* Loop counters */
    int m;

    Initialize x and y to be a fixed point of w0;
    Initialize image[i][j] to 0 for all i and j;
    for (i = 0; i < K; i++) {
        double r = Random (0, 1);      /* A random number 0 <= r <= 1 */
        double total = p[0];           /* Probability tally */
        int k = 0;
        while (total < r) {
            k++;
            total += p[k];
        }
        apply (k, x, y);               /* Apply w_k to the point (x, y) */
        for (each i, j pair)
            if (LiesIn (x, y, i, j))   /* TRUE if (x, y) is in V_ij */
                image[i][j]++;
    }
    m = maximum of all image[i][j] entries;
    for (each (i, j) pair)
        image[i][j] /= m;
}  /* IFS */
```

Fig. 17.23 The iterated function system rendering algorithm.

The set A consists of the places to which (x_0, y_0) is sent in the course of iterating the maps. Some places are visited more often than others are, and the likelihood of a region being visited defines a probability measure on the set. The measure associated with a small region Q is p if a point, in the course of infinitely many iterations of the maps, spends a fraction p of its time in the region Q. It is this probability measure that we are using to associate values to pixels.

Since K must be so large, the time spent in reconstructing an image from an IFS code is substantial (although the process is highly parallelizable). What about creating an IFS code from an image? To do so, we must find a collection of affine maps of the plane to itself with the property that, after the affine maps have been applied to the original image, the union of the results "looks like" the original image. Figure 17.24 shows how to make a leaf by

Fig. 17.24 A leaf made as a collage. (© Michael Barnsley, *Fractals Everywhere*, Academic Press.)

creating four (slightly overlapping) smaller versions of the leaf and making a collage from them.

The collage theorem [BARN88a] guarantees that any IFS that uses these affine maps has an attractor that looks like the original image. Choosing the probability associated with w_j changes the brightness of the portion of the image coming from w_j. Still, to compress an image into an IFS code, a user must find a way to recreate the original image as a union of repeated subimages, each of which is an affine transform of the original. This is the previously mentioned geometric problem to be solved by the user. Barnsley has announced that this process can be automated [BARN88b]; until a mechanism for doing so is made public, the technique is hardly usable for compressing large numbers of images. It *is* usable for modeling interesting objects, however (see Chapter 20). Color Plate IV.1 shows an entire forest modeled with an IFS.

17.7.3 Image Attributes

When we store an image in the conventional manner as a collection of channels, we certainly must store information about each pixel—namely, the value of each channel at each pixel. Other information may be associated with the image as a whole, such as width and height, and any image-description format must include this kind of information as well. It is insufficient to allocate a few bytes at the start of the image description for width and height; experience has shown that other image attributes will arise. Typical examples are the space required for look-up tables, the depth of the image (the number of bitplanes it occupies), the number of channels that follow, and the name of the creator of the image. For accurate color reproduction of images on other devices, we may also want to record reference spectra for the pure red, green, and blue colors used to make the image, and some indication of what gamma correction, if any, has been applied.

The need to store such properties has prompted the creation of flexible formats such as RIFF [SELF79] and BRIM (derived from RIFF) [MEIE83], which are general attribute-value database systems. In BRIM, for example, an image always has a width, height, and creator, and also a "history" field, which describes the creation of the image and modifications to it. Programs using BRIM can add their own signature and timestamp to the history field so that this information is automatically kept up to date. Figure 17.25 shows a text listing of a typical BRIM header for an image.

```
TYPE (string, 5): BRIM
FORMAT (string, 11): FORMAT_SEQ
TITLE (string, 31): Molecular Modeling Intro Frame
NAME (string, 27): Charles Winston  447 C.I.T.
DATE (string, 25): Sun Oct  9 12:42:16 1988
HISTORY (string, 82): Sun Oct  9 12:42:16 1988 crw RAY/n/
00033: Sun Oct  9 13:13:18 1988 crw brim_convert
DESC (string, 21): A ray-traced picture
IMAGE_WIDTH (int, 1): 640
IMAGE_HEIGHT (int, 1): 512
BRIM_VERSION (int, 1): 1
CHANNEL_DESC (string, 21): RED GREEN BLUE ALPHA
CHANNEL_WIDTH (short, 4): 8 8 8 8
ENCODING (string, 4): RLE
```

Fig. 17.25 A BRIM header for an image.

Many image-handling packages have been developed. One of the most widely used is the Utah Raster Toolkit [PETE86], which was written in fairly portable C so that the same tools can be used on a number of different architectures. Particularly troublesome issues for designing such toolkits are the numbers of bytes per word and the ordering of bytes within a word.

17.8 SPECIAL EFFECTS WITH IMAGES

The image-processing techniques described in Section 17.3 can be applied to an image to generate interesting special effects. If an image is processed with a high-pass filter, only the small details of the image remain, while all slowly varying aspects are deleted. By processing an image with a derivative filter, we can arrange to highlight all points where sharp transitions occur. Filters that reduce all intensity values below a certain level to 0 and increase all other values to 1 can be used to generate high-contrast images, and so on.

A large number of video techniques can be applied to blur images, to fade them, to slide them off a screen in real time, and so on. Many of these effects are created by using the electronic hardware for generating video signals, and modifying the signals as they are being shown. These techniques all lie in the domain of electrical engineering rather than in that of computer graphics, although the combination of effects from both disciplines can be fruitful.

We conclude this chapter by describing a digital technique for simulating neon tubing. If we paint the shape of a neon tube onto a black background, using a constant-width (antialiased) brush of constant color, our image does not look particularly exciting. But suppose we filter the image with an averaging filter—each pixel becomes the average of its immediate neighborhood. If we do this several times, the edges of the band we have drawn become blurred. If we now filter the image with an intensity-mapping filter that brightens those pixels whose intensities are above some threshold and dims those pixels whose intensities are lower than that threshold, the result looks quite a lot like neon tube. (There is another way to produce the same effect; see the discussion of antialiased brushes in Section

19.3.4.) Compositing such an image using an ω value greater than 1 and an α value less than 1 can cause the "neon" to illuminate whatever it is placed over, while the tube remains partially transparent.

17.9 SUMMARY

We have discussed several techniques for storing images, including some that are organized by programming considerations (multiple channels, headers), and some that are motivated by compactness or ease of transmission (quadtrees, IFS encodings). There are many other image storage formats, including a number of commercial formats competing for the privilege of being the "standard." Given the differences of opinion about the amount of color information that should be stored (should it be just RGB or should it consist of multiple spectral samples?), and about what information should be present in an image (should z-buffer values or α values be stored?), we expect no universal image format to evolve for some time.

We have also discussed geometric transformations on images, including multipass algorithms with filtering and the necessity of performing the filtering during such transformations. The number of interesting effects possible with image transformations is quite surprising. Due to the filtering used in such transformations, however, repeated transformations can blur an image. Transforming bitmaps involves other difficulties, since no gray scale is available to soften the aliasing artifacts that arise in the transformations. The simplicity of the data, however, makes it possible to develop very fast algorithms.

We also have discussed image compositing, which has become an extremely popular tool for generating complex images. When the α channel is used, composited images show no seams at the points where the component images overlap, unless the component images have some high geometric correlation. If rendering speed increases to the point where regenerating images is inexpensive, compositing may cease to be as important a tool as it now is. But if progress in computer graphics continues as it has, with each new generation of hardware allowing more complex rendering techniques, we should expect the image quality to increase, but the time-per-image to remain approximately constant. Therefore, we should expect compositing to be in use for some time.

EXERCISES

17.1 Show that the product of the matrices

$$\begin{bmatrix} 1 & \tan(t) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\sin(t)\cos(t) & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \cos(t) \end{bmatrix} \begin{bmatrix} 1/\cos(t) & 0 \\ 0 & 1 \end{bmatrix}$$

is exactly

$$\begin{bmatrix} \cos(t) & \sin(t) \\ -\sin(t) & \cos(t) \end{bmatrix}$$

Use this result to show how to create a rotation map from shears and scales. Use this technique to describe a pixmap-rotation algorithm derived from the Weiman algorithm (see Section 17.4.1).

17.2 How would you create a Weiman-style translation algorithm? Suppose a pixmap has alternating

columns of black and white pixels. What is the result of translating this pixmap by $\frac{1}{2}$ pixel? What is the result of applying Weiman's scaling algorithm to stretch this image by a factor of 2? What do you think of these results?

17.3 When scaling a bitmap, you cannot perform averaging, as you can in the Weiman algorithm. What is a good selection rule for the value of the target pixel? Is majority rule best? What if you want to preserve features of the image, so that scaling an image with a black line in the middle of a white page should result in a black line still being present? Is the cyclic permutation of the Rothstein code still necessary?

17.4 Show that in the Feibush, Levoy and Cook filtering method (see Section 17.4.2), the correct sign is assigned to volumes associated with triangles in step 3 of the edge-filtering process. (Hint: Evidently *some* sign must be given to each triangle, and this sign is a continuous function of the shape of the triangle—two triangles that look alike will have the same sign. The sign changes only when you modify a triangle by passing it through a degenerate triangle—one where the vertices are collinear. Thus, to do this exercise, you need only to show that the sign is correct for two triangles, one of each orientation.)

17.5 This problem fills in the details of the edge-filtering mechanism in the Feibush, Levoy and Cook image-transformation algorithm (see Section 17.4.2.) Given a triangle within a rectangular region, with one vertex at the center, C, of the rectangle, and a function (drawn as height) on the rectangular region, the volume over the triangle may be computed in the following way:

1. Call the triangle ABC. Draw a perpendicular from C to the base of the triangle, AB, intersecting AB at the point D. Express the triangle as either the sum or the difference of the two triangles ACD and BCD.

2. Find the volume over the triangles ACD and BCD, and use these values to compute the volume over ABC.

3. Observe that, if the filter function is circularly symmetric, then the volume over ACD computed in step (2) is a function of only the length of the base, AD, and the height of the triangle, CD (the same is true for BCD, of course).

   a. Draw a picture of the situation described in step 1. Do this for two cases: angle ACB is less than 90° and angle ACB is greater than 90°.

   b. Find a condition on A, B, and D that determines whether ABD is the sum of the difference of ACD and BCD.

   c. Suggest a method for computing the volume above an arbitrary right triangle as described in step 3. Since the given function may not be integrable in elementary terms, consider Monte Carlo methods.

   d. Describe how you would arrange a table of widths and heights to store all the volumes computed in step 3 in a look-up table.

17.6 Consider the 1 by 3 image that is described as follows. The pixel centers in this image are at the points (−2,0), (0, 0), and (2, 0), and the values at these points are −1, 0, and 1. The Nyquist frequency for the image is 1, and the image can be considered as a sample of a unique image that is a linear combination of sine and cosine functions with frequency 1 or less. All the cosine terms are zero (you can see that they are by noting that the function is odd, in the sense that $f(-x) = -f(x)$ for all x).

   a. Compute the coefficients, a_k, of $\sin(kx)$ (for $0 \leq k < 1$) so that samples of $\Sigma a_k \sin(kx)$ give this image. Hint: $a_r \neq 0$ for only one value of k.

   b. The image might appear to be simply a gray-scale ramp (if we imagine −1 as black and 1 as

white). What would happen if we sampled the signal computed in part a at the points (−1, 0), and (1, 0). Would they interpolate the gray-scale ramp as expected?

This exercise shows the difficulty we encounter when we use exact reconstruction without filtering.

17.7 Assume that you are given an 2^k by 2^k gray-scale image with intensity values $0 \ldots 2^{n-1}$. You can generate a 2^{k-1} by 2^{k-1} image by condensing 2 by 2 regions of the image into single bits. In Section 17.7.1, we proposed doing the condensation by computing the mean of the region. There are other possibilities, however. Analyze the selection method for condensation, where *selection* means choosing one particular corner of the region as the representative value.

a. Assume that you want to transmit a 2^k by 2^k gray-scale image that has been completely condensed using selection. How many pixel values do you need to send? (Assume that the receiving hardware knows how to decode the incoming data, and can draw filled rectangles on a display.)

b. On what class of pictures will the selection method give bad artifacts until the image is nearly completely displayed? Would some other condensation rule work better for these images?

c. What other condensation rules would be preferable for sending black-and-white documents (such as typical pages from this book, which may contain figures)? Explain your choices.

17.8 (Note: To do this programming problem, you must have access to hardware that supports very rapid *bitBlt* operations on bitmaps.) Write a program that uses a 1-bit-deep α-buffer to composite bitmaps. Add such features as painting in a bitmap with an α-buffer (which determines where the paint "sticks" or shows). This problem is discussed extensively in [SALE85].

17.9 Suppose you were given a corrupted run-length encoded image, from which several bytes were missing. Could you reconstruct most of it? Why or why not? Suggest a run-length encoding enhancement that would make partial recovery of a corrupted image easier.

17.10 We saw in Section 17.5 how various transformations can be implemented in multiple passes.

a. If a pattern is mapped linearly onto a polygon and then projected onto the viewing plane, show that the composite map from the pattern map (x, y) coordinates to the image (u, v) coordinates has the form

$$(x, y) \rightarrow (u, v),$$

where

$$u = (Ax + By + C)/(Dx + Ey + F)$$
$$v = (Px + Qy + R)/(Sx + Ty + U)$$

b. Show how to factor this map into two passes as we did for the map

$$u = x/(y + 1),\ v = y/(y + 1).$$

This idea of using two-pass transforms for texture mapping was introduced by Catmull and Smith in [CATM80]. The names of variables for the coordinates here are chosen to agree with the convention used in Section 17.6 and not with the names of the coordinates used in describing texture mapping elsewhere.

18 Advanced Raster Graphics Architecture

Steven Molnar and Henry Fuchs

In this chapter, we discuss in more detail the issues of raster graphics systems architecture introduced in Chapter 4. We examine the major computations performed by raster systems, and the techniques that can be used to accelerate them. Although the range of graphics architectures and algorithms is wide, we concentrate here on architectures for displaying 3D polygonal models, since this is the current focus in high-end systems and is the foundation for most systems that support more complex primitives or rendering methods.

Graphics systems architecture is a specialized branch of computer architecture. It is driven, therefore, by the same advances in semiconductor technology that have driven general-purpose computer architecture over the last several decades. Many of the same speed-up techniques can be used, including pipelining, parallelism, and tradeoffs between memory and computation. The graphics application, however, imposes special demands and makes available new opportunities. For example, since image display generally involves a large number of repetitive calculations, it can more easily exploit massive parallelism than can general-purpose computations. In high-performance graphics systems, the number of computations usually exceeds the capabilities of a single CPU, so parallel systems have become the rule in recent years. The organization of these parallel systems is a major focus of graphics architecture and of this chapter.

We begin by reviewing the simple raster-display architecture described in Chapter 4. We then describe a succession of techniques to add performance to the system, discussing the bottlenecks that arise at each performance level and techniques that can be used to overcome them. We shall see that three major performance bottlenecks consistently resist attempts to increase rendering speed: the number of floating-point operations to perform

geometry calculations, the number of integer operations to compute pixel values, and the number of frame-buffer memory accesses to store the image and to determine visible surfaces. These demands have a pervasive influence on graphics architecture and give rise to the diversity of multiprocessor graphics architectures seen today. At the end of the chapter, we briefly discuss several unusual architectures, such as those for ray tracing, for true 3D displays, and for commercial flight simulators.

18.1 SIMPLE RASTER-DISPLAY SYSTEM

As described in Chapter 4, a simple raster-display system contains a CPU, system bus, main memory, frame buffer, video controller, and CRT display (see Fig. 4.18). In such a system, the CPU performs all the modeling, transformation, and display computations, and writes the final image to the frame buffer. The video controller reads pixel data from the frame buffer in raster-scan order, converts digital pixel values to analog, and drives the display.

It is important to remember that, if such a system has sufficient frame-buffer memory, has a suitable CRT display, and is given enough time, it can generate and display scenes of virtually unlimited complexity and realism. None of the architectures or architectural techniques discussed here enhance this fundamental capability (except for a few exotic 3D displays treated in Section 18.11). Rather, most work in graphics architecture concerns the quest for increased rendering speed.

In Chapter 4, we discussed two problems that limit the performance of this simple system: the large number of frame-buffer memory cycles needed for video scanout and the burden that image generation places on the main CPU. We now consider each of these problems in greater detail.

18.1.1 The Frame-Buffer Memory-Access Problem

In Chapter 4, we calculated the time between successive memory accesses when a low-resolution monochrome display is being refreshed. For a system with 16-bit words, the access rate is substantial—one memory access every 864 nanoseconds. Systems with higher-resolution color monitors require much higher memory speeds. For example, refreshing a 1280 by 1024 screen with 32-bit (one-word) pixels at 60 Hz requires that memory accesses occur every $1/(1280 \cdot 1024 \cdot 60) = 12.7$ nanoseconds. Even this is only the average memory access rate, not the peak rate, since pixels are not scanned out during horizontal and vertical retrace times [WHIT84]. A simple *dynamic random-access memory* (DRAM) system, on the other hand, has a cycle time of approximately 200 nanoseconds, a factor of 16 slower than the speed required. Clearly, something must be done to increase the bandwidth to frame-buffer memory.

The following sections discuss solutions that have been used by various system designers. Some of these provide only modest performance increases, but are sufficient for low-resolution systems. Others provide greatly increased memory bandwidth, but incur significant system complexity. We begin by reviewing briefly the fundamentals of DRAM

memories, since the characteristics of DRAMs strongly influence the set of solutions available.

18.1.2 Dynamic Memories

DRAMs are the memories of choice for most computer memory systems. *Static random-access memories* (SRAMs), which retain stored data indefinitely, can be made to run faster than DRAMs, which must be accessed every few milliseconds to retain data, but DRAMs are much denser and cheaper per bit. (DRAMs also require more complicated timing for reading and writing data, but these problems are easily solved with supporting circuitry.)

Figure 18.1 is a block diagram of a typical 1-Mbit DRAM chip. As in most DRAMs, single-bit storage elements are arranged in one or more square arrays (in this case, four arrays, each with dimension 512 by 512). Vertical *bit lines* transfer data to and from the storage arrays, one bit line for each column of each array. During read and write operations, one memory cell in each column is connected to its corresponding bit line. A *sense amplifier* attached to each bit line amplifies and restores the tiny signals placed on the bit line during read operations.

In a DRAM chip, read and write operations each require two steps. The first step is to select a row. This is done by asserting the *row address strobe* (RAS) while the desired row address is on the address inputs. The row decoder produces a 512-bit vector, whose bits are

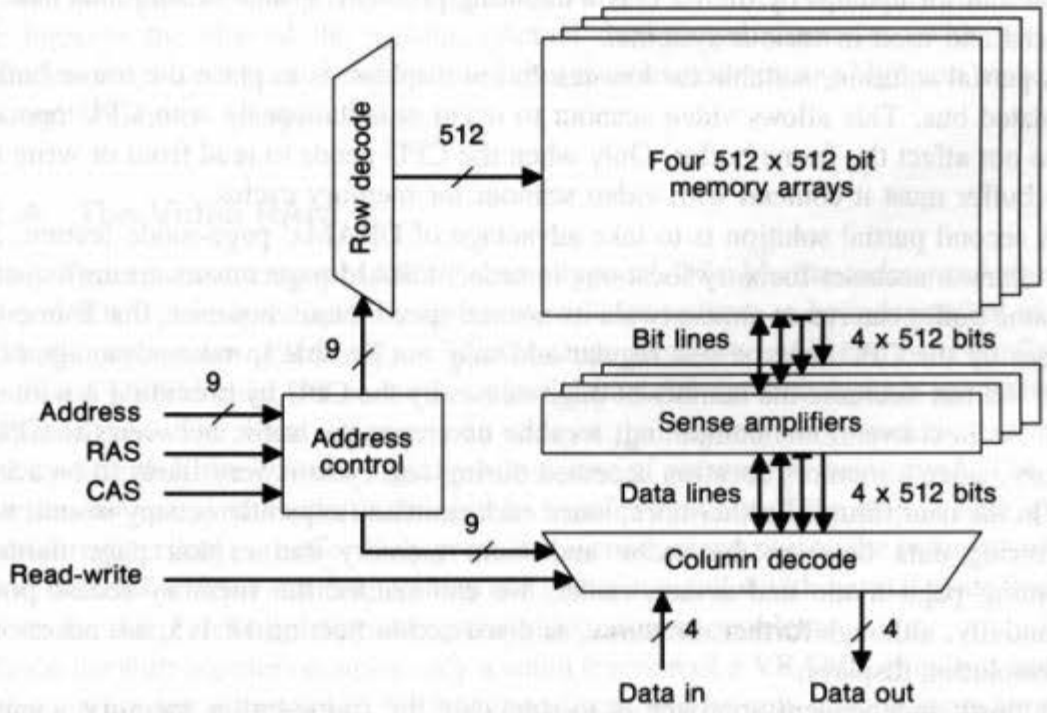


Fig. 18.1 A 1-Mbit (256K × 4) DRAM chip. An entire row is written to or read from each of the four memory arrays at the same time. The column decoder allows a particular element (column) of the selected row to be accessed.

0 everywhere except for a single 1 at the selected row. This bit vector determines which row's storage cells are connected to the bit lines and sense amplifiers.

The second step is to select a column, which is done by asserting the *column address strobe* (CAS) and read–write signal while the desired column address is on the address inputs. The column address selects a single bit from the active row of memory in each array. The selected bits are either buffered for output (during read operations) or set to the value on the data inputs (during write operations). Some DRAMs provide a faster access method called *page mode* for successive reads or writes to memory locations on the same row. In page mode, a row is selected just once, and successive columns are selected using the column address bits and CAS signal. In page mode, consecutive memory accesses require just one address cycle, instead of two. Thus, if adjacent pixels are stored in the same memory row, page mode can nearly double the bandwidth available for updating and displaying from the frame buffer.

Word widths greater than the number of data pins on a single memory chip can be accommodated by connecting multiple memory chips in parallel. With this arrangement, an entire word of data can be read from or written to the memory system in a single memory cycle. For example, eight 4-bit-wide DRAMs can be used to build a 32-bit memory system.

18.1.3 Increasing Frame-Buffer Memory Bandwidth

As we have seen, obtaining sufficient frame-buffer memory bandwidth both for video scanout and for updates by the CPU is a daunting problem. Different solutions have been proposed and used in various systems.

A partial solution, suitable for low-resolution displays, is to place the frame buffer on an isolated bus. This allows video scanout to occur simultaneously with CPU operations that do not affect the frame buffer. Only when the CPU needs to read from or write to the frame buffer must it contend with video scanout for memory cycles.

A second partial solution is to take advantage of DRAMs' page-mode feature. Since video scanout accesses memory locations in order, DRAM page misses are infrequent, so the frame buffer can run at almost twice its normal speed. Note, however, that frame-buffer accesses by the CPU may be less regular and may not be able to take advantage of page mode. We can decrease the number of page misses by the CPU by providing it with a data cache. As in conventional computing, a cache decreases the traffic between the CPU and memory, since a memory location accessed during one cycle is very likely to be accessed again in the near future. Furthermore, since cache entries frequently occupy several words, transferring data between the cache and main memory can exploit page mode. By combining page mode and a data cache, we can reduce the memory-access problem substantially, although further measures, as discussed in Section 18.1.5, are necessary for high-resolution displays.

A third, independent approach is to duplicate the frame-buffer memory, creating a *double-buffered* system in which the image in one buffer is displayed while the image in the other buffer is computed. Figure 18.2 shows one possible implementation, in which multiplexers connect each frame buffer to the system bus and video controller. Double-buffering allows the CPU to have uninterrupted access to one of the buffers while the video

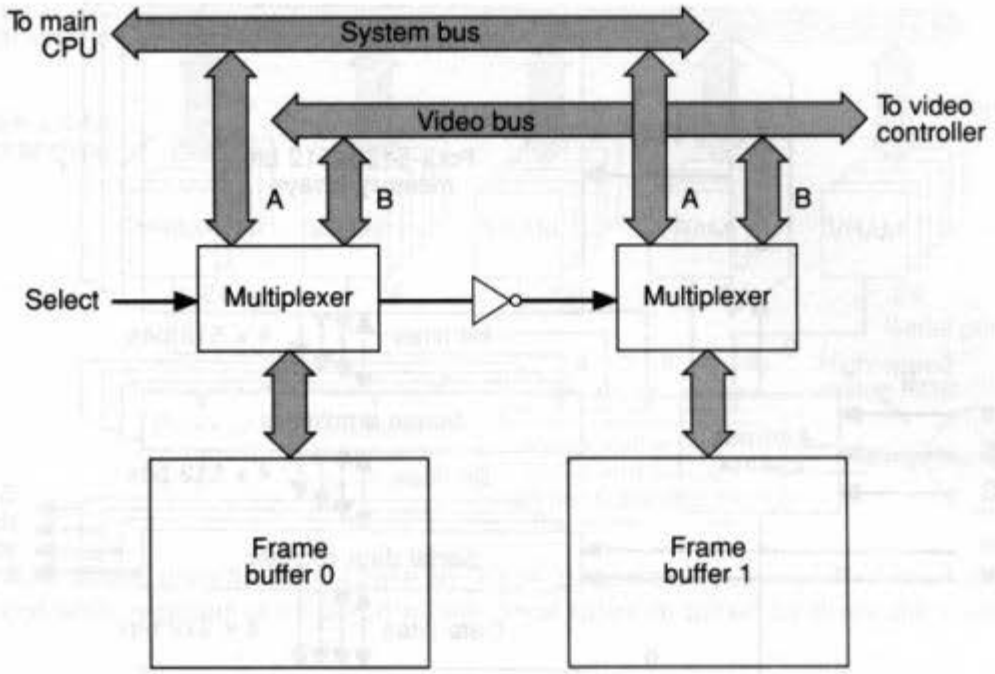


Fig. 18.2 Double-buffered frame buffer.

controller has uninterrupted access to the other. Double-buffering in this manner is expensive, however, since twice as much memory is needed as for a single-buffered display. Also, the multiplexers that provide dual access to the DRAMs require numerous chips, which increase the size of the system. (Actual dual-ported memories, with two fully independent read–write ports, could be used, but their low density and high cost make them impractical in almost all applications.)

18.1.4 The Video RAM

In 1983, Texas Instruments introduced a new type of DRAM, the *video random-access memory* (VRAM), designed specifically to allow video scanout to be independent of other frame-buffer operations [PINK83]. A VRAM chip, as shown in Fig. 18.3, is similar to a conventional DRAM chip, but contains a parallel-in/serial-out data register connected to a second data port. The serial register is as wide as the memory array and can be parallel loaded by asserting the transfer signal while a row of memory is being read. The serial register has its own data clock, enabling it to transfer data out of the chip at high speeds. The serial register and port effectively provide a second, serial port to the memory array. If this port is used for video scanout, scanout can occur in parallel with normal reads from and writes to the chip, virtually eliminating the video-scanout problem.

Since the shift register occupies only a small fraction of a VRAM's chip area and very few pins are needed to control it, VRAMs ideally should be only slightly more expensive than DRAMs. Unfortunately, the economics of scale tend to raise the price of VRAMs relative to DRAMs, since fewer VRAMs are produced (in 1989, VRAMs were roughly twice as expensive as DRAMs of the same density). In spite of this price differential, VRAMs are an excellent choice for many frame-buffer memories.

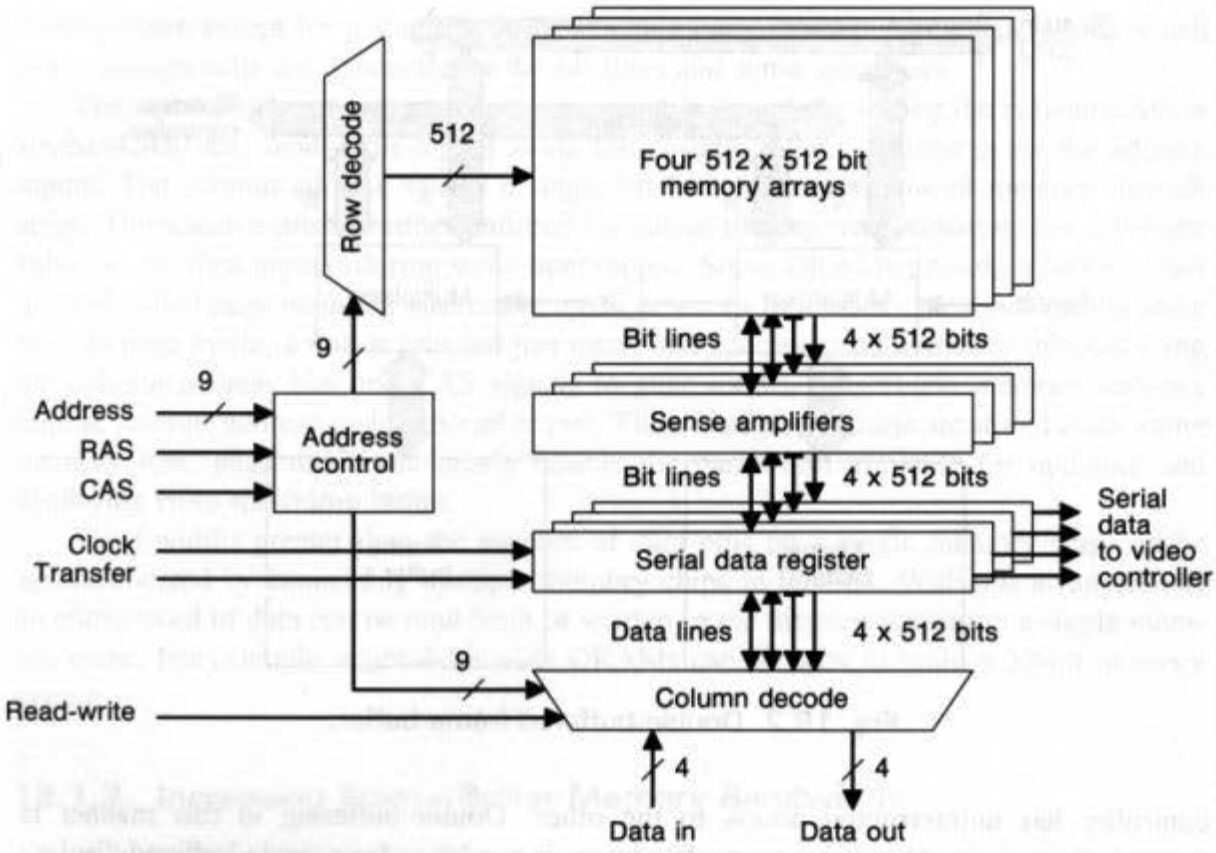


Fig. 18.3 Diagram of a 1-Mbit (256K × 4) VRAM chip. The serial data register provides a second (serial) port to the memory array.

18.1.5 A Frame Buffer for a High-Resolution Display

Figure 18.4 shows a typical frame-buffer design using VRAMs. The CPU has essentially unrestricted access to the frame-buffer memory. Only once every n cycles (where n is the dimension of the memory array) is a memory cycle required to load a new row into the VRAMs' serial registers. With screen resolutions up to 512 by 512 pixels, VRAMs can handle video scanout unassisted. With higher-resolution displays, the pixel rate exceeds the VRAMs' serial-port data rate of 30 MHz or less. A standard solution is to connect multiple banks of VRAMs to a high-speed off-chip shift register, as shown in the figure. Multiple pixels, one from each VRAM bank, can be loaded into the shift register in parallel and then shifted out serially at video rates. (Notice that this 1280- by 1024-pixel system, with five times the resolution of a 512 by 512 system, uses five-way multiplexing.) This shift register frequently is incorporated into a single chip containing look-up tables and digital-to-analog converters, such as Brooktree's Bt458 RAMDAC [BROO88]. In systems with 24 bits per pixel, such as the one shown in Fig. 18.4, three RAMDAC chips are typically used—one for each 8-bit color channel.

VRAMs provide an elegant solution to the frame-buffer memory-access problem. The bottleneck that remains in such systems is CPU processing speed. To generate a raster image (in 2D or 3D), the processor must calculate every pixel of every primitive. Since a typical primitive covers many pixels, and images typically contain thousands of primitives,

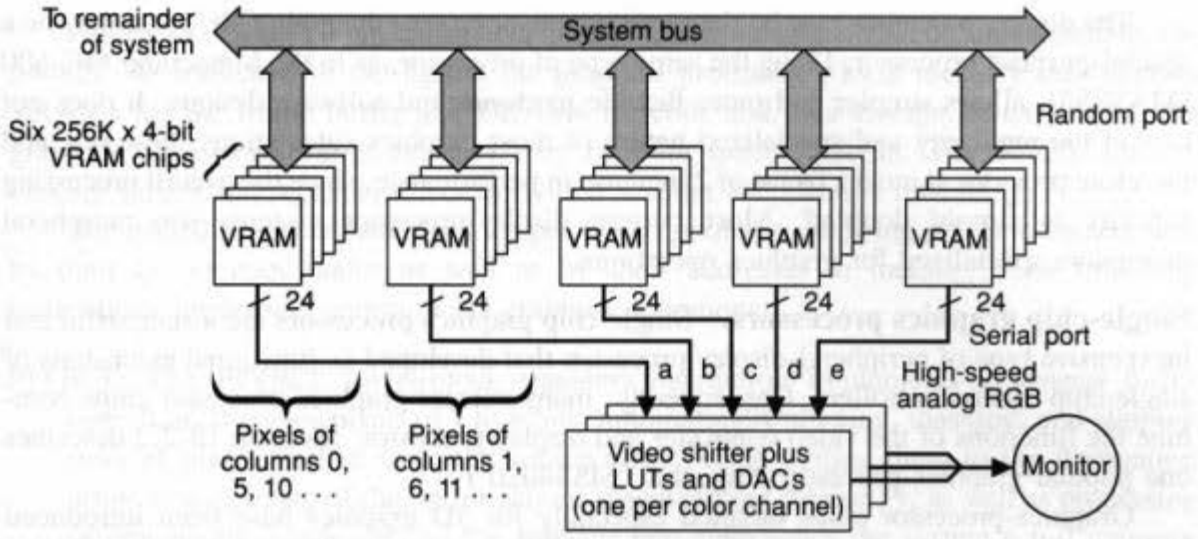


Fig. 18.4 Block diagram of a 1280- by 1024-pixel display using VRAMs. A parallel-in/serial-out shift register provides the high pixel rates required to drive the monitor.

generating a high-resolution image can require many millions of CPU operations. Because of this, even the fastest CPUs have difficulty rendering images quickly enough for interactive use.

18.2 DISPLAY-PROCESSOR SYSTEMS

To build a system with higher performance than the single-CPU system described in the previous section, we must either remove some of the graphics burden from the CPU or increase the CPU's ability to perform these tasks. Here we consider strategies that do not substantially increase the size or cost of the system (we consider larger, multiprocessing solutions in Sections 18.6 through 18.10). These strategies generally fall into three categories: (1) coprocessors that share the system bus with the main CPU, (2) display processors with their own buses and memory systems, and (3) integrated processors containing internal hardware support for graphics operations.

Coprocessors were discussed extensively in Chapter 4 (single-address-space (SAS) architectures). These have been found useful in 2D systems, but have not achieved wide use in 3D systems, largely because 3D systems require a larger set of operations than 2D systems and because bus contention in a SAS architecture severely limits system performance. The latter two approaches have been used successfully for 3D graphics, and we now discuss them further.

18.2.1 Peripheral Display Processors

Figure 4.22 shows a typical peripheral-display-processor system organization. The display processor generally has its own memory system (including the frame buffer) and a fast communication link to the main CPU. The main CPU sends display commands to the display processor, which executes them using resident software or firmware routines, or using specialized hardware.

The display processor may be the same processor type as the main CPU, or it may be a special-purpose processor. Using the same type of processor, as in the Masscomp MC-500 [MASS85], allows simpler and more flexible hardware and software designs. It does not exploit the regularity and specialized nature of most graphics calculations, however, and therefore provides at most a factor of 2 increase in performance, since the overall processing capacity is simply doubled. Most current display-processor systems use peripheral processors specialized for graphics operations.

Single-chip graphics processors. Single-chip graphics processors are a successful and inexpensive type of peripheral-display processor that developed as functional extensions of single-chip video controllers. Consequently, many current graphics-processor chips combine the functions of the video controller and display processor. (Section 18.2.2 describes one popular graphics processor chip, the TMS34020.)

Graphics-processor chips targeted especially for 3D graphics have been introduced only recently. A pioneering chip of this type is Intel's i860, which can be used either as a stand-alone processor with support for graphics or as a 3D graphics processor. Unlike many 2D graphics-processor chips, the i860 does not provide hardware support for screen refresh or video timing. The i860 will be discussed in depth in Section 18.2.4.

Programming the display processor. A major point of difference among display processor systems is where they store the model or database from which the image is generated: in the CPU's memory, or in the display processor's memory. Storing the database in the display processor's memory frees the main CPU from database traversal. Consequently, only a low-bandwidth channel is needed between the CPU and the display processor. However, such a system also inherits all the disadvantages of a structure database described in Chapter 7, particularly the inflexibility of a canonical, "hard-wired" database model.

Whether or not the display processor manages the database, it is an extra processor to program. Frequently, this program takes the form of a standard graphics library, such as PHIGS+, which satisfies the needs of many applications. Applications invariably arise, however, that require features not supported by the library. In such cases, users are forced either to program the display processor themselves (which may not even be possible), or to devise cumbersome workarounds using existing display-processor features.

18.2.2 Texas Instruments' TMS34020—A Single-Chip Peripheral Display Processor[1]

Texas Instruments' TMS34020 [TEXA89] is an example of a single-chip graphics processor designed to accelerate 2D displays on PCs and workstations. It can be paired with a TMS34082 floating-point coprocessor, which rapidly performs 3D geometric transformations and clipping needed for 3D graphics. Unlike graphics-processor chips that perform only graphics-specific operations, the 34020 is a fully programmable 32-bit processor that can be used as a stand-alone processor if desired.

[1]Material for this example was contributed by Jerry R. Van Aken of Texas Instruments, Inc.

Figure 18.5 shows a typical system configuration using the 34020 (and optionally the 34082). As indicated in the figure, the graphics processor's local memory may contain VRAMs for the frame buffer and DRAMs for code and data storage. Alternatively, the graphics processor's entire memory may be built from VRAMs. The 34020 contains on-chip timers and registers to control video scanout and DRAM refreshing.

The 34020 supports data types for pixels and pixmaps, allowing pixels to be accessed by their (x, y) coordinates as well as by their addresses in memory. The following instructions implement common 2D graphics operations:

PIXBLT: The PIXBLT (pixel-block transfer) instruction implements a general bitBlt operation. It copies pixels of 1 to 8 bits, automatically aligning, masking, and clipping rows of pixels fetched from the source array before writing them to the destination array. It uses many of the optimizations discussed in Chapter 19, as well as processing several pixels in parallel, so the transfer rate approaches the system's full memory bandwidth—up to 18 million 8-bit pixels per second. A special version of PIXBLT supports text by efficiently expanding packed 1-bit bitmaps into 8-bit pixmaps.

FILL: The FILL instruction fills a rectangular array of pixels with a solid color. Large areas are filled at rates approaching the memory-bus speed. Texas Instruments' 1-Mbit VRAM supports a special block-write mode that allows up to four memory locations to be written at once from the same on-chip color register. Using this block-write mode, the 34020 fills areas at up to 160 million 8-bit pixels per second.

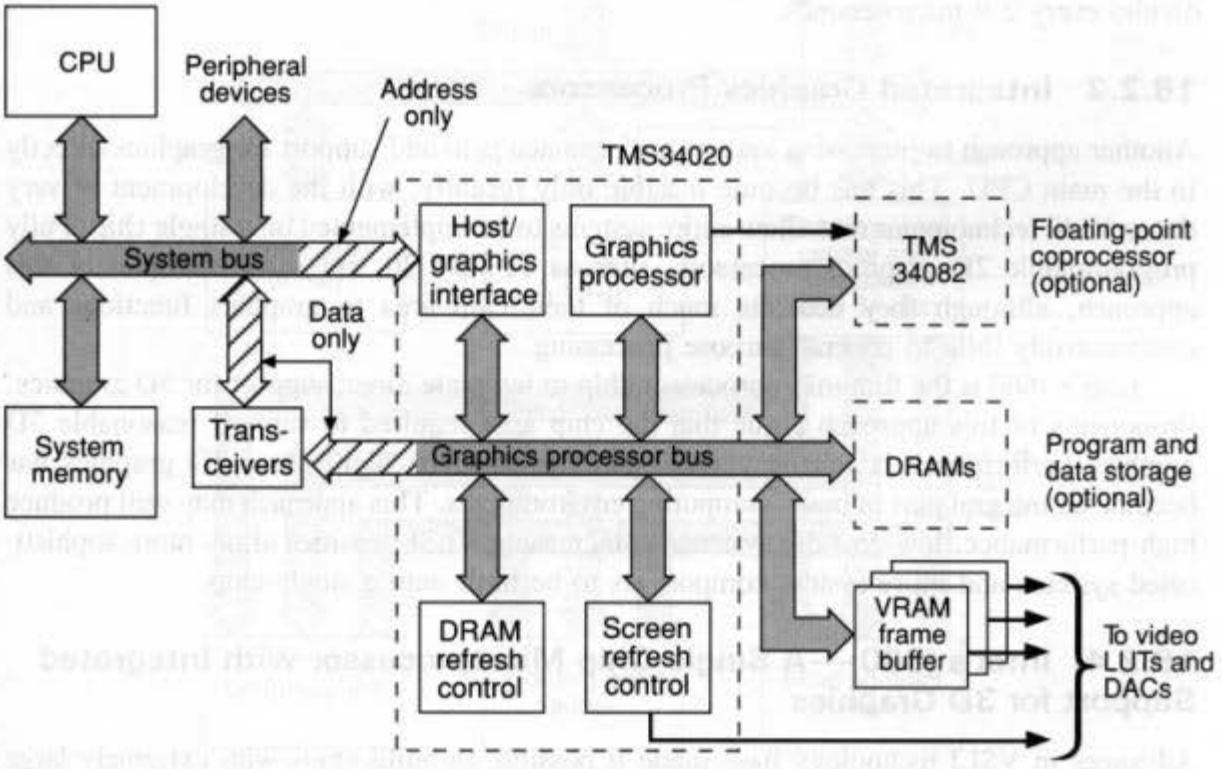


Fig. 18.5 Block diagram of the TMS34020 graphics-processor chip used as a peripheral display processor.

FLINE: The FLINE instruction draws 1-pixel-thick straight lines using a midpoint scan-conversion algorithm (see Section 3.2.2). The FLINE instruction draws up to 5 million pixels per second.

DRAV: The DRAV (draw-and-advance) instruction draws the pixel indicated by a pair of (x, y) coordinates, then increments the x and y coordinates in parallel. It is used in the inner loops of incremental algorithms for drawing circles and ellipses; for example, the inner loop of a midpoint circle routine (see Section 3.3.2) draws 2 million pixels per second.

Software for the 34020 is typically divided into time-critical graphics subroutines, written in assembly language, and other software, written in a high-level language. Software running on the 34020 communicates with the application program running on the main CPU.

The 34020 contains a 512-byte on-chip instruction cache and 30 general-purpose registers—enough memory to store the instructions and data required within the inner loop of many graphics subroutines. Once the instructions and data for the loop have been read from external memory, the memory bus is available exclusively for moving pixels to and from the frame buffer.

The TMS34082 floating-point coprocessor chip interfaces directly to the 34020, providing floating-point instructions and registers that increase the system's capabilities for 3D and other floating-point-intensive applications. The 34082 monitors the 34020's memory bus and receives floating-point instructions and data transmitted over the bus by the 34020. The 34082 can transform a 3D homogeneous point (including the homogeneous divide) every 2.9 microseconds.

18.2.3 Integrated Graphics Processors

Another approach to increasing system performance is to add support for graphics directly to the main CPU. This has become feasible only recently, with the development of very dense VLSI technologies that allow entire systems to be implemented on a single chip. Fully programmable 2D graphics processors, such as TI's 34020, are early examples of this approach, although they dedicate much of their chip area to graphics functions and comparatively little to general-purpose processing.

Intel's i860 is the first microprocessor chip to integrate direct support for 3D graphics. Proponents of this approach argue that the chip area required to support reasonable 3D graphics performance is relatively small and the payoff is high, since 3D graphics has become an integral part of many computing environments. This approach may well produce high-performance, low-cost 3D systems as increasing VLSI densities allow more sophisticated systems and more system components to be built onto a single chip.

18.2.4 Intel's i860—A Single-Chip Microprocessor with Integrated Support for 3D Graphics

Advances in VSLI technology have made it possible to build chips with extremely large numbers of transistors—more than 1 million in 1989. This allows high-performance CPU chips to include other features, such as caches, I/O controllers, and support for specialized

instructions. Intel's i860 microprocessor, also known as the 80860, [GRIM89] is an example of such a chip with features supporting 3D graphics. Figure 18.6 is a block diagram of the major data paths in the i860.

According to Intel, the i860 achieves the following benchmarks: 33 million VAX-equivalent instructions per second (on compiled C code), 13 double-precision MFLOPs (using the Linpack benchmark), and 500,000 homogeneous vector transformations [INTE89].

The i860's graphics instructions operate in parallel on as many pixels as can be packed into a 64-bit data word. For applications using 8-bit pixels, this means eight operations can occur simultaneously. For 3D shaded graphics, which generally requires 32-bit pixels, two operations can occur simultaneously. Parallel graphics instructions include:

- Calculating multiple linear interpolations in parallel
- Calculating multiple z-buffer comparisons in parallel
- Conditionally updating multiple pixels in parallel.

For systems using 8-bit pixels (where eight operations can occur in parallel), the i860 can scan convert and shade 50,000 Gouraud-shaded 100-pixel triangles per second.

The i860 may be used either as a peripheral display processor (with an 80486

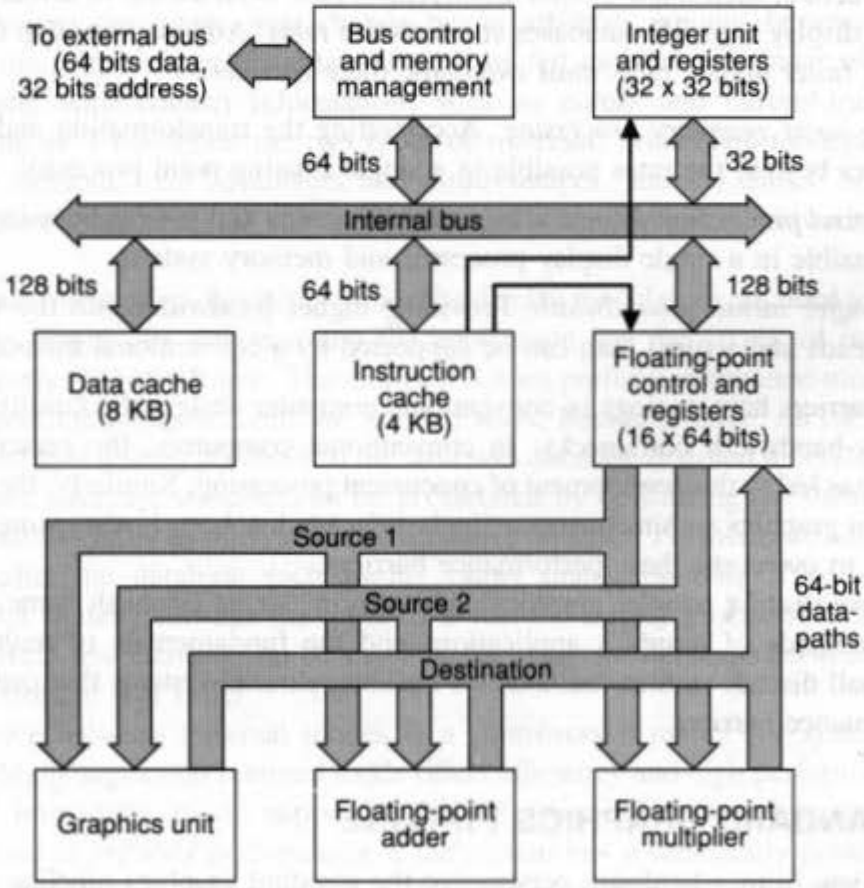


Fig. 18.6 Block diagram of Intel i860 microprocessor datapaths (adapted from [GRIM89]).

microprocessor as the main CPU, for example) or as a stand-alone processor. By combining front-end and back-end capabilities in a single processor, the i860 allows a powerful 3D graphics system to be built with very few parts—essentially the processor, memory, and the video system.

As VLSI densities continue to improve, we can expect to see even higher performance and more complex processors designed for graphics. Features commonly found in 2D graphics-processor chips, such as scanout and video-timing circuitry, may be included in such processors as well.

18.2.5 Three Performance Barriers

Systems can be built with a large range of performances using the techniques described up to this point. Low-end systems using a general-purpose microprocessor for the application processor and a 2D graphics processor are well suited for displays on PCs. Higher-performance systems using a powerful application processor and a high-speed 3D graphics processor, containing display-list memory and transformation hardware, are suitable for engineering and scientific workstations.

But how far can such designs be pushed? Let us imagine a display system made with the fastest available CPU and display processor, and a frame buffer containing the fastest available VRAMs. Such a system will achieve impressive—but still limited—performance. In practice, such *single-stream* architectures have been unable to achieve the speeds necessary to display large 3D databases at interactive rates. Architectures that display more primitives at faster update rates must overcome three barriers:

- *Floating-point geometry processing*: Accelerating the transformation and clipping of primitives beyond the rates possible in a single floating-point processor
- *Integer pixel processing*: Accelerating scan conversion and pixel processing beyond the rates possible in a single display processor and memory system
- *Frame-buffer memory bandwidth*: Providing higher bandwidth into the frame buffer (faster reads and writes) than can be supported by a conventional memory system.

These barriers have analogs in conventional computer design: the familiar processing and memory-bandwidth bottlenecks. In conventional computers, the presence of these bottlenecks has led to the development of concurrent processing. Similarly, the major focus of research in graphics architecture over the last decade has been investigating ways to use concurrency to overcome these performance barriers.

Before we discuss parallel graphics architectures, let us establish some groundwork about the demands of graphics applications and the fundamentals of multiprocessing. Later, we shall discuss various methods for building parallel systems that overcome these three performance barriers.

18.3 STANDARD GRAPHICS PIPELINE

Here we review from a hardware perspective the standard graphics pipeline discussed in Chapter 16. As has been mentioned, the rendering pipeline is a logical model for the computations needed in a raster-display system, but is not necessarily a physical model, since the stages of the pipeline can be implemented in either software or hardware.

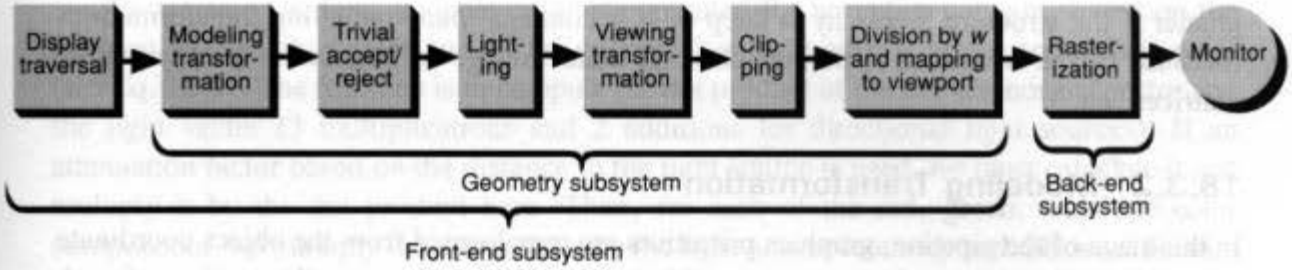


Fig. 18.7 Standard graphics pipeline for Gouraud- or Phong-shaded polygons.

Figure 18.7 shows a version of the rendering pipeline that is typical for systems using conventional primitives (lines and polygons) and conventional shading techniques (constant, Gouraud, or Phong). We shall discuss each stage of the pipeline in turn, paying particular attention to algorithms that have been used successfully in hardware systems and to those that may be useful in the future. At the end of this section, we estimate the number of computations needed at each stage of the graphics pipeline to process a sample database containing 10,000 polygons.

18.3.1 Display Traversal

The first stage of the pipeline is traversal of the display model or database. This is necessary because the image may change by an arbitrary amount between successive frames. All the primitives in the database must be fed into the remainder of the display pipeline, along with context information, such as colors and current-transformation matrices. Chapter 7 described the two types of traversal: immediate mode and retained mode. Both methods have advantages and disadvantages, and the choice between them depends on the characteristics of the application and of the particular hardware architecture used.

Immediate mode offers flexibility, since the display model does not need to conform to any particular display-list structure and the application has the luxury of recreating the model differently for every frame. The main CPU must perform immediate-mode traversal, however, expending cycles it could use in other ways. Retained mode, on the other hand, can be handled by a display processor if the structure database is stored in its local memory. Retained-mode structure traversal can be accelerated by optimizing the database storage and access routines or by using a dedicated hardware traverser. Furthermore, since the main CPU only edits the database each frame, rather than rebuilding it from scratch, a low-bandwidth channel between the main CPU and the display processor is sufficient. Of course, relatively few changes can be made to the structure database between frames, or system performance will suffer.

The choice between traversal modes is a controversial matter for system designers [AKEL89]. Many argue that retained mode offers efficiency and high performance. Others believe that immediate mode supports a wider range of applications and does not necessarily lead to reduced performance if the system has a sufficiently powerful CPU.

Unfortunately, it is difficult to estimate the processing requirements for display traversal, since they depend on the traversal method used and on the characteristics of the particular display model. At the very least, a read operation and a write operation must be performed for each word of data to be displayed. The processing requirements may be much

greater if the structure hierarchy is deep or if it contains many modeling transformations (because of the overhead in pointer chasing, state saving, concatenating transformation matrices, etc.).

18.3.2 Modeling Transformation

In this stage of the pipeline, graphics primitives are transformed from the object-coordinate system to the world-coordinate system. This is done by transforming the vertices of each polygon with a single transformation matrix that is the concatenation of the individual modeling transformation matrices. In addition, one or more surface-normal vectors may need to be transformed, depending on the shading method to be applied.

Constant shading requires world-space surface-normal vectors for each polygon. We compute these by multiplying object-space surface normals by the transpose of the inverse modeling transformation matrix. Gouraud and Phong shading require world-space normals for each vertex, rather than for each polygon, so each vertex-normal vector must be multiplied by the transpose inverse transformation matrix.

Let us compute the number of floating-point calculations required to transform a single vertex if Gouraud shading is to be applied. Multiplying a homogeneous point by a 4×4 matrix requires 16 multiplications and 12 additions. Multiplying each vertex normal by the inverse transformation matrix requires 9 multiplications and 6 additions (only the upper-left 3×3 portion of the matrix is needed—see Exercise 18.1). Therefore, transforming a single vertex with surface normal requires $16 + 9 = 25$ multiplications and $12 + 6 = 18$ additions.

18.3.3 Trivial Accept/Reject Classification

In the trivial accept/reject classification stage, primitives (now in world coordinates) are tested to see whether they lie wholly inside or outside the view volume. By identifying primitives that lie outside the view volume early in the rendering pipeline, processing in later stages is minimized. We will clip primitives that cannot be trivially accepted or rejected in the clipping stage.

To trivially accept or reject a primitive, we must test each transformed vertex against the six bounding planes of the view volume. In general, the bounding planes will not be aligned with the coordinate axes. Each test of a vertex against a bounding plane requires 4 multiplications and 3 additions (the dot product of a homogeneous point with a 3D plane equation). A total of $6 \cdot 4 = 24$ multiplications and $6 \cdot 3 = 18$ additions are required per vertex.

18.3.4 Lighting

Depending on the shading algorithm to be applied (constant, Gouraud, or Phong), an illumination model must be evaluated at various locations: once per polygon for constant shading, once per vertex for Gouraud shading, or once per pixel for Phong shading. Ambient, diffuse, and specular illumination models are commonly used in high-performance systems.

In constant shading, a single color is computed for an entire polygon, based on the position of the light source and on the polygon's surface-normal vector and diffuse color (see Eq. 16.9). The first step is to compute the dot product of the surface-normal vector and the light vector (3 multiplications and 2 additions for directional light sources). If an attenuation factor based on the distance to the light source is used, we must calculate it and multiply it by the dot product here. Then, for each of the red, green, and blue color components, we multiply the dot product by the light-source intensity and diffuse-reflection coefficient (2 multiplications), multiply the ambient intensity by the ambient-reflection coefficient (1 multiplication), and add the results (1 addition). If we assume a single directional light source, calculating a single RGB triple requires $3 + 3 \cdot (2 + 1) = 12$ multiplications and $2 + 3 \cdot 1 = 5$ additions. Gouraud-shading a triangle requires three RGB triples—one for each vertex.

Phong shading implies evaluating the illumination model at each pixel, rather than once per polygon or once per polygon vertex. It therefore requires many more computations than does either constant or Gouraud shading. These computations, however, occur during the rasterization stage of the pipeline.

18.3.5 Viewing Transformation

In this stage, primitives in world coordinates are transformed to normalized projection (NPC) coordinates. This transformation can be performed by multiplying vertices in world coordinates by a single 4×4 matrix that combines the perspective transformation (if used) and any skewing or nonuniform scaling transformations needed to convert world coordinates to NPC coordinates. This requires 16 multiplications and 12 additions per vertex. Viewing transformation matrices, however, have certain terms that are always zero. If we take advantage of this, we can reduce the number of computations for this stage by perhaps 25 percent. We will assume that 12 multiplications and 9 additions per vertex are required in the viewing transformation stage.

Note that if a simple lighting model (one that does not require calculating the distance between the light source and primitive vertices) is used, modeling and viewing transformation matrices can be combined into a single matrix. In this case only one transformation stage is required in the display pipeline—a significant savings.

18.3.6 Clipping

In the clipping stage, lit primitives that were not trivially accepted or rejected are clipped to the view volume. As described in Chapter 6, clipping serves two purposes: preventing activity in one screen window from affecting pixels in other windows, and preventing mathematical overflow and underflow from primitives passing behind the eye point or at great distances.

Exact clipping is computationally practical only for simple primitives, such as lines and polygons. These primitives may be clipped using any of the 3D clipping algorithms described in Section 6.5. Complicated primitives, such as spheres and parametrically defined patches, are difficult to clip, since clipping can change the geometric nature of the primitive. Systems designed to display only triangles have a related problem, since a clipped triangle may have more than three vertices.

An alternative to exact clipping is scissoring, described in Section 3.11. Here, primitives that cross a clipping boundary are processed as usual until the rasterization stage, where only pixels inside the viewport window are written to the frame buffer. Scissoring is a source of inefficiency, however, since effort is expended on pixels outside the viewing window. Nevertheless, it is the only practical alternative for clipping many types of complex primitives.

In the pipeline described here, all clipping is performed in homogeneous coordinates. This is really only necessary for z clipping, since the w value is needed to recognize vertices that lie behind the eye. Many systems clip to x and y boundaries after the homogeneous divide for efficiency. This simplifies x and y clipping, but still allows primitives that pass behind the eye to be recognized and clipped before w information is lost.

The number of computations required for clipping depends on how many primitives cross the clipping boundaries, which may change from one frame to the next. A common assumption is that only a small percentage of primitives (10 percent or fewer) need clipping. If this assumption is violated, system performance may decrease dramatically.

18.3.7 Division by w and Mapping to 3D Viewport

Homogeneous points that have had a perspective transformation applied, in general, have w values not equal to 1. To compute true x, y, and z values, we must divide the x, y, and z components of each homogeneous point by w. This requires 3 divisions per vertex. In many systems, vertex x and y coordinates must be mapped from the clipping coordinate system to the coordinate system of the actual 3D viewport. This is a simple scaling and translation operation in x and y that requires 2 multiplications and 2 additions per vertex.

18.3.8 Rasterization

The rasterization stage converts transformed primitives into pixel values, and generally stores them in a frame buffer. As discussed in Section 7.12.1, rasterization consists of three subtasks: *scan conversion*, *visible-surface determination*, and *shading*. Rasterization, in principle, requires calculating each primitive's contribution to each pixel, an $O(nm)$ operation, where n is the number of primitives and m is the number of pixels.

In a software rendering system, rasterization can be performed in either of two orders: primitive by primitive (object order), or pixel by pixel (image order). The pseudocode in Fig. 18.8 describes each of these two approaches, which correspond to the two main families of image-precision visibility algorithms: z-buffer and scan-line algorithms.

Most systems today rasterize in object order, using the z-buffer algorithm to compute

```
for (each primitive P)
    for (each pixel q within P)
        update frame buffer based on color
            and visibility of P at q;
```
(a)

```
for (each pixel q)
    for (each primitive P covering q)
        compute P's contribution to q,
            outputting q when finished;
```
(b)

Fig. 18.8 Pseudocode for (a) object-order and (b) image-order rasterization algorithms.

visibility. The z-buffer algorithm has only recently become practical with the availability of inexpensive DRAM memory. If we assume that visibility is determined using a z-buffer, the number of computations required in the rasterization stage still depends on the scan-conversion and shading methods. Scan-conversion calculations are difficult to categorize, but can be significant in a real system.

Constant shading requires no additional computations in the rasterization stage, since the polygon's uniform color was determined in the lighting stage. Gouraud shading requires red, green, and blue values to be bilinearly interpolated across each polygon. Incremental addition can be used to calculate RGB values for each succeeding pixel. Phong shading requires significantly more computation. The x, y, and z components of the surface normal must be linearly interpolated across polygons based on the normal vectors at each vertex. Since the interpolated vector at each pixel does not, in general, have unit length, it must be normalized. This requires 2 additions, an expensive reciprocal square-root operation, and 3 multiplications for every pixel. Then, the Phong illumination model must be evaluated, requiring a dot product, more multiplications and additions, and, in the general case, the evaluation of an exponential. Bishop and Weimer's approximation to Phong shading, as mentioned in Section 16.2.5, can reduce some of these calculations, albeit at the expense of realism for surfaces with high specularity. Other shading techniques, such as transparency and texturing, may be desired as well. The architectural implications of these techniques are discussed in Section 18.11.

In addition to the shading calculations, updating each pixel requires reading the z-buffer, comparing old and new z values, and, if the primitive is visible, writing new color and z values to the frame buffer.

18.3.9 Performance Requirements of a Sample Application

In Section 18.2, we mentioned the three major performance barriers for raster graphics systems: the number of floating-point operations required in geometry calculations and the number of pixel-oriented computations and frame-buffer accesses required in rasterization. To appreciate the magnitude of these problems, we shall estimate the number of computations needed at each stage of the rendering pipeline to display a sample database at a modest update rate. To make this estimate, we must define a representative database and application.

A sample database. We use triangles for primitives in our sample database, since they are common and their computation requirements are easy to estimate (complex primitives with shared vertices, such as triangle strips and quadrilateral meshes, are becoming increasingly popular, however. Exercise 18.2 explores the savings that can be achieved using these primitives.) We assume a modest database size of 10,000 triangles, a size that current workstations can display at interactive rates [AKEL88; APGA88; BORD89; MEGA89]. In addition, we assume that each triangle covers an average of 100 pixels.

The number of polygons that fall on clipping boundaries can vary widely from one frame to the next. For simplicity, we assume that no primitives need clipping in our sample application. This means we underestimate the amount of calculations required for clipping, but maximize the work required in succeeding stages. Image *depth complexity*, the average number of polygons mapping to a pixel, depends on the database as well as on the view. We

assume arbitrarily that one-half of the pixels of all the triangles are obscured by some other triangle.

We assume that an ambient/diffuse illumination model and Gouraud shading are to be applied to each primitive (this is the lowest common standard for current 3D systems, although Phong illumination combined with Gouraud shading is becoming increasingly popular). We assume a screen size of 1280 by 1024 pixels and an update rate of 10 frames per second, typical of current interactive applications, but far from ideal.

In summary, our sample application has the following characteristics:

- 10,000 triangles (none clipped)
- Each triangle covers an average of 100 pixels, one-half being obscured by other triangles
- Ambient and diffuse illumination models (not Phong)
- Gouraud shading
- 1280 by 1024 display screen, updated at 10 frames per second.

We cannot calculate all the computation and memory-bandwidth requirements in our sample application, since many steps are difficult to categorize. Instead, we concentrate on the three performance barriers: the number of floating-point operations for geometry computations, the number of integer operations for computing pixel values, and the number of frame-buffer accesses for rasterization.

Geometry calculations. For each frame, we must process $10{,}000 \cdot 3 = 30{,}000$ vertices and vertex-normal vectors. In the modeling transformation stage, transforming a vertex (including transforming the normal vector) requires 25 multiplications and 18 additions. The requirements for this stage are thus $30{,}000 \cdot 25 = 750{,}000$ multiplications and $30{,}000 \cdot 18 = 540{,}000$ additions.

Trivial accept/reject classification requires testing each vertex of each primitive against the six bounding planes of the viewing volume, a total of 24 multiplications and 18 additions per vertex. The requirements for this stage are thus $30{,}000 \cdot 24 = 720{,}000$ multiplications and $30{,}000 \cdot 18 = 540{,}000$ additions, regardless of how many primitives are trivially accepted or rejected.

Lighting requires 12 multiplications and 5 additions per vertex, a total of $30{,}000 \cdot 12 = 360{,}000$ multiplications and $30{,}000 \cdot 5 = 150{,}000$ additions.

The viewing transformation requires 8 multiplications and 6 additions per vertex, a total of $30{,}000 \cdot 8 = 240{,}000$ multiplications and $30{,}000 \cdot 6 = 180{,}000$ additions.

The requirements for clipping are variable; the exact number depends on the number of primitives that cannot be trivially accepted or rejected, which in turn depends on the scene and on the viewing angle. We have assumed the simplest case for our database, that all primitives lie completely within the viewing volume. If a large fraction of the primitives needs clipping, the computational requirements could be substantial (perhaps even more than in the geometric transformation stage).

Division by w requires 3 divisions per vertex, a total of $30{,}000 \cdot 3 = 90{,}000$ divisions. Mapping to the 3D viewport requires 2 multiplications and 2 additions per vertex, a total of 60,000 multiplications and 60,000 additions.

Summing the floating-point requirements for all of the geometry stages gives a total of 2,220,000 multiplications/divisions and 1,470,000 additions/subtractions per frame. Since a new frame is calculated every $\frac{1}{10}$ second, a total of 22.2 million multiplications/divisions and 14.7 million additions/subtractions (36.9 million aggregate floating-point operations) as required per second—a very substantial number.

Rasterization calculations and frame-buffer accesses. Let us now estimate the number of pixel calculations and frame-buffer memory accesses required in each frame. We assume that z values and RGB triples each occupy one word (32 bits) of frame-buffer memory (typical in most current high-performance systems). For each pixel that is initially visible (i.e., results in an update to the frame buffer), z, R, G, and B values are calculated (4 additions per pixel if forward differences are used), a z value is read from the frame buffer (1 frame-buffer cycle), the z values are compared (1 subtraction) and new z values and colors are written (2 frame-buffer cycles). For each pixel that is initially not visible, only the z value needs to be calculated (1 addition), and a z value is read from the frame buffer (1 frame-buffer cycle), and the two z values are compared (1 subtraction). Note that initially visible pixels may get covered, but initially invisible pixels can never be exposed.

Since we assume that one-half of the pixels of each triangle are visible in the final scene, a reasonable guess is that three-quarters of the pixels are initially visible and one-quarter of the pixels are initially invisible. Each triangle covers 100 pixels, so $\frac{3}{4} \cdot 100 \cdot 10{,}000 = 750{,}000$ pixels are initially visible and $\frac{1}{4} \cdot 100 \cdot 10{,}000 = 250{,}000$ pixels are initially invisible. To display an entire frame, therefore, a total of $(750{,}000 \cdot 5) + (250{,}000 \cdot 2) = 4.25$ million additions and $(750{,}000 \cdot 3) + (250{,}000 \cdot 1) = 2.5$ million frame-buffer accesses is required. To initialize each frame, both color and z-buffers must be cleared, an additional $1280 \cdot 1024 \cdot 2 = 2.6$ million frame-buffer accesses. The total number of frame-buffer accesses per frame, therefore, is 2.5 million + 2.6 million = 5.1 million. If 10 frames are generated per second, 42.5 million additions and 51 million frame-buffer accesses are required per second.

In 1989, the fastest floating-point processors available computed approximately 20 million floating-point operations per second, the fastest integer processors computed approximately 40 million integer operations per second, and DRAM memory systems had cycle times of approximately 100 nanoseconds. The floating-point and integer requirements of our sample application, therefore, are just at the limit of what can be achieved in a single CPU. The number of frame-buffer accesses, however, is much higher than is possible in a conventional memory system. As we mentioned earlier, this database is only modestly sized for systems available in 1989. In the following sections, we show how multiprocessing can be used to achieve the performance necessary to display databases that are this size and larger.

18.4 INTRODUCTION TO MULTIPROCESSING

Displaying large databases at high frame rates clearly requires dramatic system performance, both in terms of computations and of memory of bandwidth. We have seen that the geometry portion of a graphics system can require more processing power than a single CPU can provide. Likewise, rasterization can require more bandwidth into memory than a single memory system can provide. The only way to attain such performance levels is to

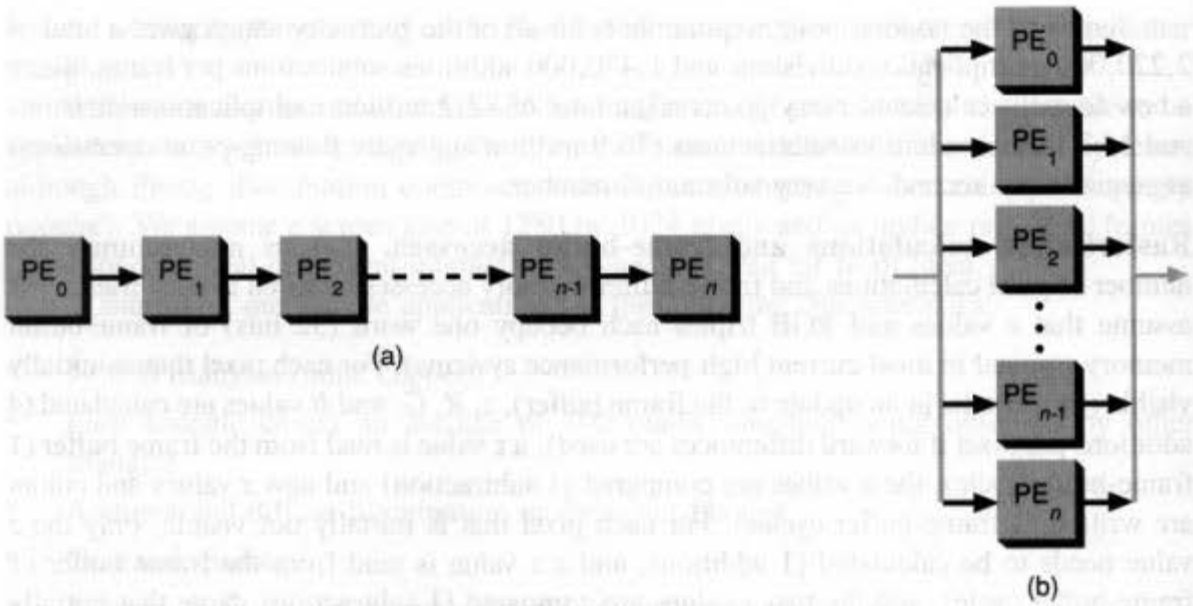


Fig. 18.9 Basic forms of multiprocessing: (a) pipelining, and (b) parallelism.

perform multiple operations concurrently and to perform multiple reads and writes to memory concurrently—we need concurrent processing.

Concurrent processing, or *multiprocessing*, is the basis of virtually all high-performance graphics architectures. Multiprocessing has two basic forms: *pipelining* and *parallelism* (we reserve the term *concurrency* for multiprocessing in general). A *pipeline processor* contains a number of *processing elements* (PEs) arranged such that the output of one becomes the input of the next, in pipeline fashion (Fig. 18.9a). The PEs of a *parallel processor* are arranged side by side and operate simultaneously on different portions of the data (Fig. 18.9b).

18.4.1 Pipelining

To pipeline a computation, we partition it into stages that can be executed sequentially in separate PEs. Obviously, a pipeline can run only as fast as its slowest stage, so the processing load should be distributed evenly over the PEs. If this is not possible, PEs can be sized according to the jobs they must perform.

An important issue in pipeline systems is *throughput* versus *latency*. Throughput is the overall rate at which data are processed; latency is the time required for a single data element to pass from the beginning to the end of the pipeline. Some calculations can be pipelined using a large number of stages to achieve very high throughput. Pipeline latency increases with pipeline length, however, and certain computations can tolerate only a limited amount of latency. For example, real-time graphics systems, such as flight simulators, must respond quickly to changes in flight controls. If more than one or two frames are in the rendering pipeline at once, the system's interactivity may be impaired, regardless of the frame rate.

18.4.2 Parallelism

To parallelize a computation, we partition the data into portions that can be processed independently by different PEs. Frequently, PEs can execute the same program. *Homogeneous* parallel processors contain PEs of the same type; *heterogeneous* parallel processors contain PEs of different types. In any parallel system, the overall computation speed is determined by the time required for the slowest PE to finish its task. It is important, therefore, to balance the processing load among the PEs.

A further distinction is useful for homogeneous parallel processors: whether the processors operate in *lock step* or independently. Processors that operate in lock step generally share a single code store and are called *single-instruction multiple-data* (SIMD) processors. Processors that operate independently must have a separate code store for each PE and are called *multiple-instruction multiple-data* (MIMD) processors.

SIMD processors. Because all the PEs in a SIMD processor share a single code store, SIMD processors are generally less expensive than MIMD processors. However, they do not perform well on algorithms that contain conditional branches or that access data using pointers or indirection. Since the path taken in a conditional branch depends on data specific to a PE, different PEs may follow different branch paths. Because all the PEs in a SIMD processor operate in lock step, they all must follow every possible branch path. To accommodate conditional branches, PEs generally contain an *enable register* to qualify write operations. Only PEs whose enable registers are set write the results of computations. By appropriately setting and clearing the enable register, PEs can execute conditional branches (see Fig. 18.10a).

Algorithms with few conditional branches execute efficiently on SIMD processors. Algorithms with many conditional branches can be extremely inefficient, however, since

```
statement 1;
if not condition then
  enable = FALSE;
statement 2;
toggle enable;
statement 3;
statement 4;
enable = TRUE;
statement 5;
statement 6;
```

Total operations:
10 if condition evaluates TRUE,
10 if condition evaluates FALSE

(a)

```
statement 1;
if condition then
  statement 2;
else
  begin
    statement 3;
    statement 4;
  end;
statement 5;
statement 6;
```

Total operations:
5 if condition evaluates TRUE,
6 if condition evaluates FALSE

(b)

Fig. 18.10 (a) SIMD and (b) MIMD expressions of the same algorithm. In a SIMD program, conditional branches transform into operations on the enable register. When the enable register of a particular PE is FALSE, the PE executes the current instruction, but does not write the result.

most PEs may be disabled at any given time. Data structures containing pointers (such as linked lists or trees) or indexed arrays cause similar problems. Since a pointer or array index may contain a different value at each PE, all possible values must be enumerated to ensure that each PE can make its required memory reference. For large arrays or pointers, this is an absurd waste of processing resources. A few SIMD processors provide separate address wires for each PE in order to avoid this problem, but this adds size and complexity to the system.

MIMD processors. MIMD processors are more expensive than SIMD processors, since each PE must have its own code store and controller. PEs in a MIMD processor often execute the same program. Unlike SIMD PEs, however, they are not constrained to operate in lock step. Because of this freedom, MIMD processors suffer no disadvantage when they encounter conditional branches; each PE makes an independent control-flow decision, skipping instructions that do not need to be executed (see Fig. 18.10b). As a result, MIMD processors achieve higher efficiency on general types of computations. However, since processors may start and end at different times and may process data at different rates, synchronization and load balancing are more difficult, frequently requiring FIFO buffers at the input or output of each PE.

18.4.3 Multiprocessor Graphics Systems

Pipeline and parallel processors are the basic building blocks of virtually all current high-performance graphics systems. Both techniques can be used to accelerate front-end and back-end subsystems of a graphics system, as shown in Fig. 18.11.

In the following sections, we examine each of these strategies. Sections 18.5 and 18.6 discuss pipeline and parallel front-end architectures. Sections 18.8 and 18.9 discuss pipeline and parallel back-end architectures. Section 18.10 discusses back-end architectures that use parallel techniques in combination.

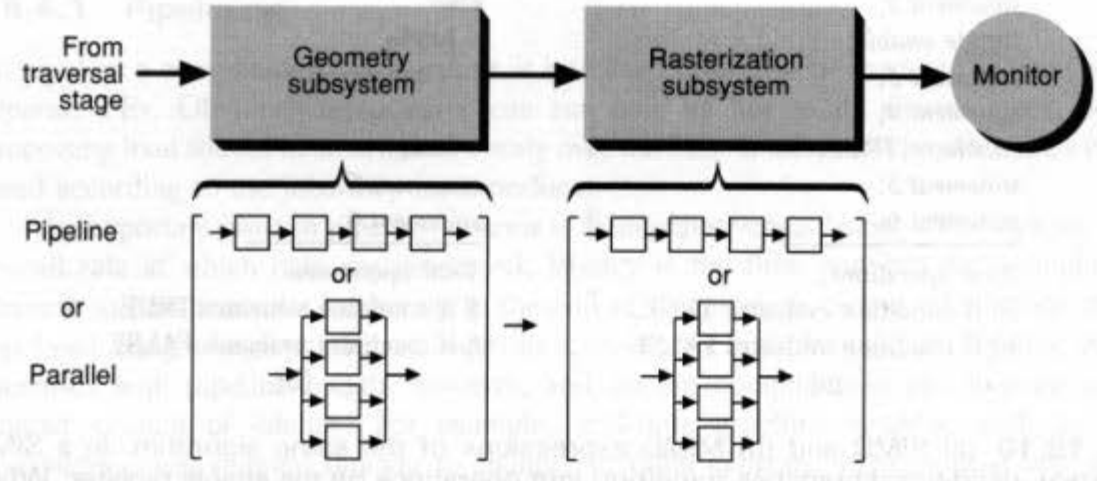


Fig. 18.11 Pipelining and parallelism can be used to accelerate both front-end and back-end portions of a graphics system.

18.5 PIPELINE FRONT-END ARCHITECTURES

Recall from Section 18.3 that the front end of a graphics display system has two major tasks: traversing the display model and transforming primitives into screen space. As we have seen, to achieve the rendering rates required in current applications, we must use concurrency to speed these computations. Both pipelining and parallelism have been used for decades to build front ends of high-performance graphics systems. Since the front end is intrinsically pipelined, its stages can be assigned to separate hardware units. Also, the large numbers of primitives in most graphics databases can be distributed over multiple processors and processed in parallel. In this section, we discuss pipeline front-end systems. We discuss parallel front-end systems in Section 18.6.

In introducing the standard graphics pipeline of Fig. 18.7, we mentioned that it provides a useful conceptual model of the rendering process. Because of its linear nature and fairly even allocation of processing effort, it also maps well onto a physical pipeline of processors. This has been a popular approach to building high-performance graphics systems since the 1960s, as described in [MYER68], a classic paper on the evolution of graphics architectures. Each stage of the pipeline can be implemented in several ways: as an individual general-purpose processor, as a custom hardware unit, or as a pipeline or parallel processor itself. We now discuss implementations for each stage in the front-end pipeline.

18.5.1 Application Program and Display Traversal

Some processor must execute the application program that drives the entire graphics system. In addition to feeding the graphics pipeline, this processor generally handles input devices, file I/O, and all interaction with the user. In systems using immediate-mode traversal, the display model is generally stored in the CPU's main memory. The CPU must therefore traverse the model as well as run the application. In systems using retained mode, the model is generally (but not always) stored in the display processor's memory, with the display processor performing traversal. Because such systems use two processors for these tasks, they are potentially faster, although they are less flexible and have other limitations, as discussed in Section 7.2.2.

Where very high performance is desired, a single processor may not be powerful enough to traverse the entire database with sufficient speed. The only remedy is to partition the database and to traverse it in parallel. This relatively new technique is discussed in Section 18.6.1.

18.5.2 Geometric Transformation

The geometric transformation stages (modeling transformation and viewing transformation) are highly compute-intensive. Fortunately, vector and matrix multiplications are simple calculations that require no branching or looping, and can readily be implemented in hardware.

The most common implementation of these stages is a single processor or functional unit that sequentially transforms a series of vertices. A pioneering processor of this type was the Matrix Multiplier [SUTH68], which could multiply a four-element vector by a

homogeneous transformation matrix in 20 microseconds. Other special-purpose geometry processors have been developed since then, most notably Clark's Geometry Engine, which can perform clipping as well (see Section 18.5.5). Recent geometry processors have exploited the power and programmability of commercial floating-point chips.

If pipelining does not provide enough performance, transformation computations can be parallelized in several ways:

- Individual components of a vertex may be calculated in parallel. Four parallel processors, each containing the current transformation matrix, can evaluate the expressions for x, y, z, and w in parallel.
- Multiple vertices can be transformed in parallel. If primitives are all of a uniform type—say, triangles—the three vertices of each triangle can be transformed simultaneously.
- Entire primitives can be transformed in parallel. If n transformation engines are available, each processor can transform every nth primitive. This technique has many of the advantages and disadvantages of parallel front-end systems, which we will discuss in Section 18.6.

18.5.3 Trivial Accept/Reject Classification

Trivial accept and reject tests are straightforward to implement, since they require at worst a dot product and at best a single floating-point comparison (or subtract) to determine on which side of each clipping plane each vertex lies. Because these tests require little computation, they are generally performed by the processor that transforms primitives.

18.5.4 Lighting

Like geometric transformation, lighting calculations are straightforward and are floating-point–intensive. A specialized hardware processor can calculate vertex colors based on a polygon's color and the light vector. More frequently, lighting calculations are performed using a programmable floating-point processor. In lower-performance systems, lighting calculations can be done in the same processor that transforms vertices. Note that if Phong shading is used, lighting calculations are deferred until the rasterization stage.

18.5.5 Clipping

Polygon clipping was once considered cumbersome, since the number of vertices can change during the clipping process and concave polygons can fragment into multiple polygons during clipping. Sutherland and Hodgman [SUTH74] showed that arbitrary convex or concave polygons can be clipped to a convex view volume by passing the polygon's vertices through a single processing unit multiple times. Each pass through the unit clips the polygon to a different plane. In 1980, Clark proposed unwrapping this processing loop into a simple pipeline of identical processors, each of which could be implemented in a single VLSI chip, which he named the *Geometry Engine* [CLAR82]. The Geometry Engine was general enough that it could transform primitives and perform perspective division as well.

Clipping using a Geometry Engine (or similar processor) can be performed either by a single processor that clips each polygon by as many planes as necessary, or by a pipeline of clipping processors, one for each clipping plane. The technique chosen affects the worst-case performance of the graphics system: Systems with only one clipping processor may bog down during frames in which large numbers of primitives need to be clipped, whereas systems with a clipping processor for each clipping plane can run at full speed. However, most of the clipping processors are idle for most databases and views in the latter approach.

General-purpose floating-point units recently have begun to replace custom VLSI transformation and clipping processors. For example, Silicon Graphics, which for many years employed custom front-end processors, in 1989 used the Weitek 3332 floating-point chip for transformations and clipping in their POWER IRIS system (described in detail in Section 18.8.2). The delicate balance between performance and cost now favors commodity processors. This balance may change again in the future if new graphics-specific functionality is needed and cannot be incorporated economically into general-purpose processors.

18.5.6 Division by *w* and Mapping to 3D Viewpoint

Like geometric transformation and lighting, the calculations in this stage are straightforward but require substantial floating-point resources. A floating-point divide is time consuming even for most floating-point processors (many processors use an iterative method to do division). Again, these stages can be implemented in custom functional units or in a commercial floating-point processor. In very high-performance systems, these calculations can be performed in separate, pipelined processors.

18.5.7 Limitations of Front-End Pipelines

Even though pipelining is the predominant technique for building high-performance front-end systems, it has several limitations that are worth considering. First, a different algorithm is needed for each stage of the front-end pipeline. Thus, either a variety of hard-wired functional units must be designed or, if programmable processors are used, different programs must be written and loaded into each processor. In either case, processor or functional-unit capabilities must be carefully matched to their tasks, or bottlenecks will occur.

Second, since the rendering algorithm is committed to hardware (or at least to firmware, since few systems allow users to reprogram pipeline processors), it is difficult to add new features. Even if users have programming support for the pipeline processors, the distribution of hardware resources in the system may not adequately support new features such as complex primitives or collision detection between primitives.

A final shortcoming of pipelined front ends is that the approach breaks down when display traversal can no longer be performed by a single processor, and this inevitably occurs at some performance level. For example, if we assume that traversal is performed by a 20-MHz processor and memory system, that the description of each triangle in the database requires 40 words of data (for vertex coordinates, normal vectors, colors, etc.), and that each word sent to the pipeline requires two memory/processor cycles (one to read it

from memory, another to load it into the pipeline), then a maximum of 20,000,000 / (2 · 40) = 250,000 triangles per second can be displayed by the system, no matter how powerful the processors in the pipeline are. Current systems are rapidly approaching such limits.

What else can be done, then, to achieve higher performance? The alternative to pipelining front-end calculations is to parallelize them. The following section describes this second way to build high-performance front-end systems.

18.6 PARALLEL FRONT-END ARCHITECTURES

Since graphics databases are regular, typically consisting of a large number of primitives that receive nearly identical processing, an alternate way to add concurrency is to partition the data into separate streams and to process them independently. For most stages of the front-end subsystem, such partitioning is readily done; for example, the geometric-transformation stages can use any of the parallel techniques described in Section 18.5.2. However, stages in which data streams diverge (display traversal) or converge (between the front end and back end) are problematic, since they must handle the full data bandwidth.

18.6.1 Display Traversal

Almost all application programs assume a single, contiguous display model or database. In a parallel front-end system, the simplest technique is to traverse the database in a single processor (serial traversal) and then to distribute primitives to the parallel processors. Unfortunately, this serial traversal can become the bottleneck in a parallel front-end system. Several techniques can be used to accelerate serial traversal:

- Traversal routines can be optimized or written in assembly code
- The database can be stored in faster memory (i.e., SRAM instead of DRAM)
- A faster traversal processor (or one optimized for the particular structure format) can be used.

If these optimizations are not enough, the only alternative is to traverse the database in parallel. The database either can be stored in a single memory system that allows parallel access by multiple processors (a shared-memory model), or can be distributed over multiple processors, each with its own memory system (a distributed-memory model).

The advantage of the shared-memory approach is that the database can remain in one place, although traversal must be divided among multiple processors. Presumably, each processor is assigned a certain portion of the database to traverse. Unfortunately, inherited attributes in a hierarchical database model mean that processors must contend for access to the same data. For example, each processor must have access to the current transformation matrix and to other viewing and lighting parameters. Since the data bandwidth to and from a shared-memory system may not be much higher than that of a conventional memory system, the shared-memory approach may not provide enough performance.

In the distributed-memory approach, each processor contains a portion of the database in its local memory. It traverses its portion of the database for each frame and may also perform other front-end computations. Distributing the database presents its own problems, however: Unless the system gives the application programmer the illusion of a contiguous database, it cannot support portable graphics libraries. Also, the load must be balanced

over the traversal processors if system resources are to be utilized fully. Hierarchical databases exacerbate both of these problems, since attributes in one level of a hierarchy affect primitives below them, and structures deep in a hierarchy may be referenced by multiple higher-level structure calls.

The following two sections examine two ways to distribute a hierarchical database over multiple processors: by structure, where each traversal processor is given a complete branch of the structure hierarchy; or by primitive, where each traversal processor is given a fraction of the primitives at each block in the hierarchy.

Distributing by structure. Distributing by structure is outwardly appealing, since state-changing elements in the structure apparently need to be stored only once. This can be an illusion, however, since multiple high-level structures may refer to the same lower-level substructure. For example, a database containing several cars, each described by a separate *car* structure, can be distributed by assigning each *car* structure to a separate processor. However, if each *car* structure refers to a number of *wheel* structures, *wheel* structures must also be replicated at every processor.

Load balancing among processors is also difficult. Since primitives in a structure are likely to be spatially coherent, changing the viewpoint or geometry within a scene may cause entire portions of the structure to be clipped or to reappear. Maintaining even loading among the multiple processors would require reassigning portions of the database dynamically.

Distributing by primitive. Distributing by primitive is costly, since the entire hierarchical structure of the database and any state-changing commands must be replicated at each processor. Structure editing is also expensive, since changes must be broadcast to every processor. Load balancing, however, is automatic. Since objects in a hierarchical database typically contain a large number of simple primitives (e.g., polygons forming a tiled surface), these primitives will be scattered over all the processors, and each processor will have a similar processing load.

Parallel display traversal is a relatively new technique. In 1989, the highest-performance architectures were just approaching the point where serial traversal becomes insufficient, and only a few systems had experimented with parallel traversal [FUCH89]. Neither of the distribution techniques for hierarchical databases that we have described is ideal. Compared to geometry processing, which easily partitions into parallel tasks, display traversal is much more difficult. Nevertheless, parallel traversal is likely to become increasingly important as system performance levels increase.

18.6.2 Recombining Parallel Streams

The transition between the front-end and back-end portions of the rendering pipeline is troublesome as well. In a parallel front-end system, the multiple streams of transformed and clipped primitives must be directed to the processor or processors doing rasterization. This can require sorting primitives based on spatial information if different processors are assigned to different screen regions.

A second difficulty in parallel front-end systems is that the ordering of data may change as those data pass through parallel processors. For example, one processor may transform two small primitives before another processor transforms a single, larger one. This does not

matter for many graphics primitives and rendering techniques. Certain global commands, however, such as commands to update one window instead of another or to switch between double buffers, require that data be synchronized before and after the command. If a large number of commands such as these occurs, some type of hardware support for synchronization may be necessary. A Raster Technologies system [TORB87] incorporates a special FIFO into each PE that stores tag codes for each command and allows commands to be resynchronized after they have been processed in separate PEs.

18.6.3 Pipelining versus Parallelism

We have seen that both pipelining and parallelism can be used to build high-performance front-end subsystems. Although pipelining has been the predominant technique in systems of the last decade, parallelism offers several advantages, including reconfigurability for different algorithms, since a single processor handles all front-end calculations, and more modularity, since PEs in a parallel system can be made homogeneous more easily than in a pipeline system. Because the performance of a pipeline system is limited by the throughput of its slowest stage, pipelines do not scale up as readily as do parallel systems. Parallel systems, on the other hand, require more complicated synchronization and load balancing and cannot use specialized processors as well as can pipelined systems. Both designs are likely to be useful in the future; indeed, the highest-performance systems are likely to combine the two.

18.7 MULTIPROCESSOR RASTERIZATION ARCHITECTURES

Recall that the output of the front-end subsystem is typically a set of primitives in screen coordinates. The rasterization (back-end) subsystem creates the final image by scan converting each of these primitives, determining which primitives are visible at each pixel, and shading the pixel accordingly. Section 18.2.4 identified two basic reasons why simple display-processor/frame-buffer systems are inadequate for high-performance rasterization subsystems:

1. A single display processor does not have enough processing power for all the pixel calculations.
2. Memory bandwidth into the frame buffer is insufficient to handle the pixel traffic—even if the display processor could compute pixels rapidly enough.

Much of the research in graphics architecture over the past decade has concerned ways to overcome these limitations. A great variety of techniques has been proposed, and many have been implemented in commercial and experimental systems. In this section, we consider low-cost, moderate-performance architectures that cast conventional algorithms into hardware. In Sections 18.8 and 18.9, we consider ways to improve performance by adding large amounts of parallelism to speed the calculation of the algorithm's "inner loop." In Section 18.10, we consider hybrid architectures that combine multiple techniques for improved efficiency or even higher performance. Figure 18.12 summarizes the concurrent approaches we shall discuss here.

Rasterization algorithm	*Architectural technique*		
	Serial pipeline	Highly parallel	Hybrid
Object order	Pipelined object order	Image parallel	Virtual buffer/ virtual processor
z-buffer, depth-sort, and BSP-tree algorithms	Polygon/edge/span-processor pipeline	Partitioned image memory Logic-enhanced memory	Parallel virtual buffer
Image order	Pipelined image order	Object parallel	
Scanline algorithms	Scan-line pipeline	Processor per primitive pipeline Tree-structured	Image composition

Fig. 18.12 Taxonomy of concurrent rasterization approaches.

18.7.1 Pipelined Object-Order Architectures

A direct way to add concurrency to rasterization calculations is to cast the various steps of a software algorithm into a hardware pipeline. This technique has been used to build a number of inexpensive, moderately high-performance systems. This approach can be used with either of the two main rasterization approaches: object order (*z*-buffer, depth-sort, and BSP-tree algorithms) and image order (scan-line algorithms). We consider object-order rasterization now and image-order rasterization in Section 18.7.2.

Object-order rasterization methods include the *z*-buffer, depth-sort, and BSP-tree algorithms (the *z*-buffer is by far the most common in 3D systems). The outer loop of these algorithms is an enumeration of primitives in the database, and the inner loop is an enumeration of pixels within each primitive. For polygon rendering, the heart of each of these algorithms is rasterizing a single polygon.

Figure 18.13 shows the most common rasterization algorithm for convex polygons. This algorithm is an extension of the 2D polygon scan-conversion algorithm presented in Section 3.6, using fixed-point arithmetic rather than integer arithmetic. Delta values are used to calculate the expressions for x, z, R, G, and B incrementally from scan line to scan line, and from pixel to pixel. We shall describe each step of the algorithm.

Polygon processing. Computations performed only once per polygon are grouped into this stage. The first step is to determine the initial scan line intersected by the polygon (this is determined by the vertex with the smallest y value). In most cases, the polygon intersects this scan line at a single pixel, with two edges projecting upward, the *left* and *right* edges. Delta values are calculated for x, z, R, G, and B for each edge. These delta values are sometimes called *slopes*.

Edge processing. Computations performed once for each scan line are grouped here. Scan lines within each primitive are processed one by one. The delta values computed previously are used to calculate x, z, R, G, and B values at the intersection points of the left and right edges with the current scan line (P_{left} and P_{right} in the figure). A contiguous sequence of pixels on a scan line, such as those between P_{left} and P_{right}, is called a *span*. Delta

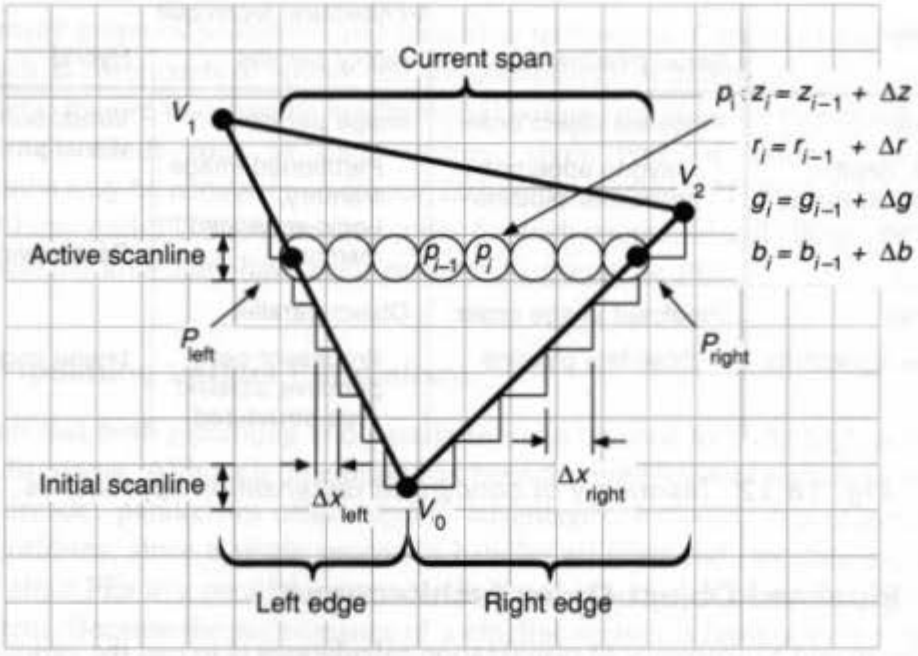


Fig. 18.13 Rasterizing a triangle. Each vertex (V_0, V_1, and V_2), span endpoint (P_{left} and P_{right}), and pixel (p_0, p_1, etc.) has z, R, G, and B components.

values for incrementing z, R, G, and B from pixel to pixel within the span are then calculated from the values at P_{left} and P_{right}.

Span processing. Operations that must be performed for each pixel within each span are performed here. For each pixel within the span, z, R, G, and B values are calculated by adding delta values to the values at the preceding pixel. The z value is compared with the previous z value stored at that location; if it is smaller, the new pixel value replaces the old one.

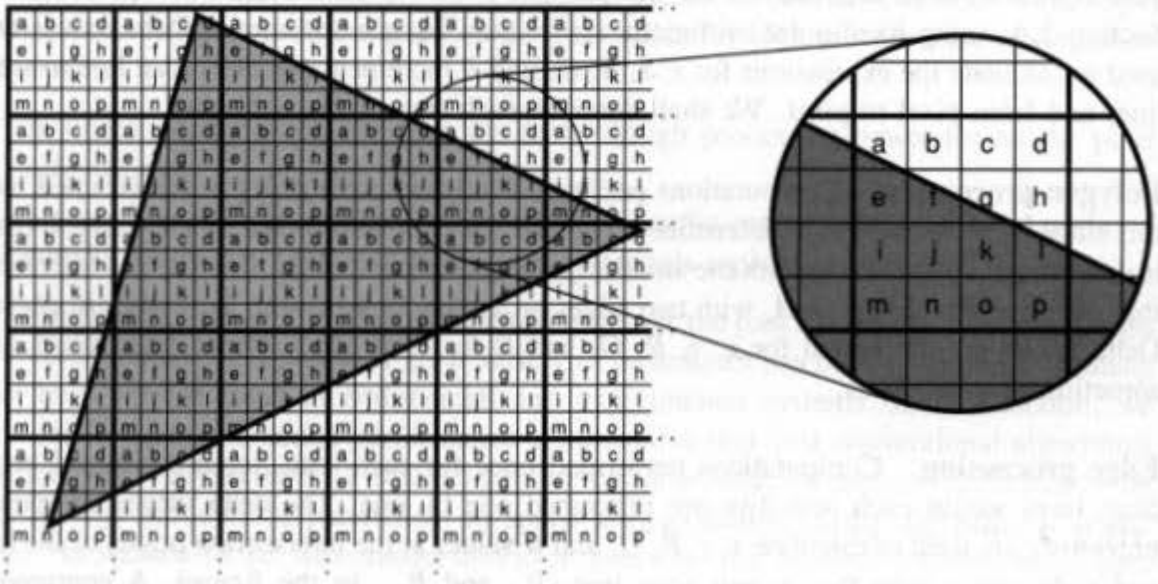


Fig. 18.14 A 4 × 4 interleaved memory organization. Each memory partition ("a" through "p") contains one pixel from each 4 × 4 block of pixels.

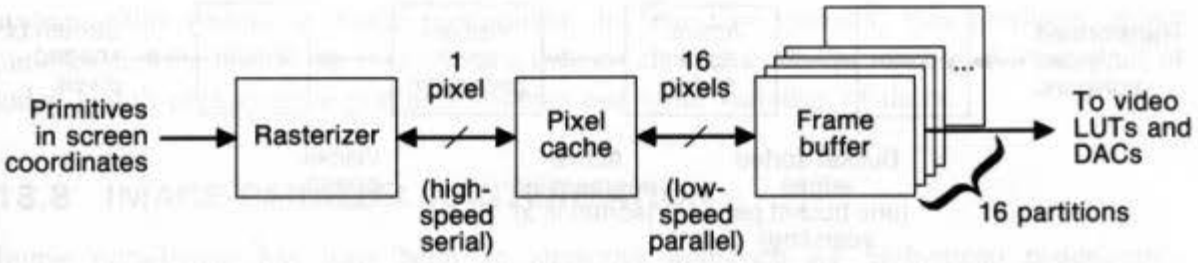


Fig. 18.15 A pixel cache matches bandwidth between a high-speed serial link to the rasterizer and a low-speed parallel link to the frame buffer.

A pipelined system containing one PE for each of the preceding three steps generates images dramatically faster than does a general-purpose display processor. In fact, it may generate pixels faster than a standard frame-buffer memory can handle. The Hewlett-Packard SRX [SWAN86], which uses this approach, generates up to 20 million pixels per second, approximately twice the speed of a typical VRAM memory system. In such systems, the rasterization bottleneck is access to frame-buffer memory.

Pixel cache. Pixels can be read and written faster if the frame buffer has some degree of parallel access. One way to accomplish this is to divide the memory into multiple—say, 16—partitions, each of which contains every fourth pixel of every fourth scan line, or perhaps every sixteenth pixel in every scan line (see Fig. 18.14). In this way, 16 pixels can be read or written in parallel. This technique, called *memory interleaving*, is also used in general-purpose CPU memory design.

A *pixel register* or *pixel cache* containing 16 pixels can be inserted between the rasterization pipeline and the interleaved image memory [GORI87; APGA88], as in Fig. 18.15. A cache allows the rasterizer to access individual pixels at high speed, assuming that the pixel is already in the cache. Multiple pixel values can be moved in parallel between the cache and the frame buffer at the slower speeds accommodated by the frame buffer.

As in any cache memory unit, performance depends on *locality of reference*, the principle that successive memory accesses are likely to occur in the same portion of memory. Erratic access patterns cause a high percentage of cache misses and degrade performance. For polygon rendering, the access pattern can be predicted precisely, since the extent of the polygon in screen space and the order in which pixels are generated are known before the pixels are accessed. Using this information, a cache controller can begin reading the next block of pixels from the frame buffer while the previous block of pixels is processed [APGA88].

Enhancing a rasterization subsystem with this kind of parallel-access path to frame-buffer memory may well increase the system throughput to the point where the bottleneck now becomes the single-pixel path between the rasterizer and the pixel cache. A logical next step is to enhance the rasterizer so that it can generate multiple pixel values in parallel. We consider such *image-parallel* architectures in Section 18.8.

18.7.2 Pipelined Image-Order Architectures

The alternative to object-order rasterization methods is *image-order* (or *scan-line*) rasterization, introduced in Section 15.4.4. Scan-line algorithms calculate the image pixel by pixel, rather than primitive by primitive. To avoid considering primitives that do not

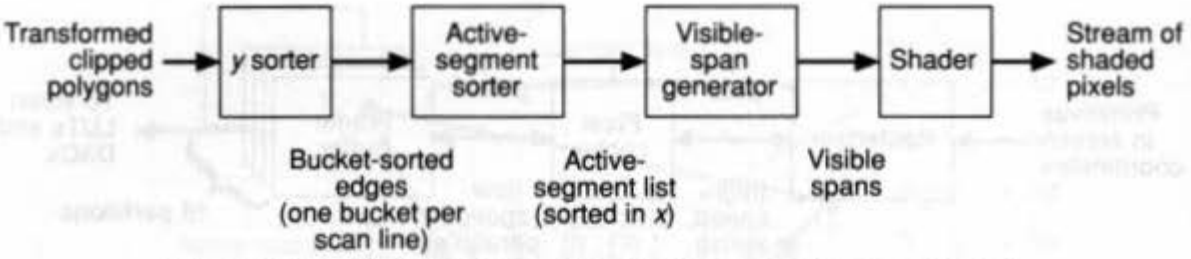


Fig. 18.16 Block diagram of a pipelined scan-line rasterizer.

contribute to the current scan line, most scan-line algorithms require primitives to be transformed into screen space and sorted into *buckets* according to the first scan line in which they each appear.

Scan-line algorithms can be implemented in hardware using the same approach as object-order algorithms: by casting the steps of the software algorithm into a pipelined series of hardware units. Much of the pioneering work on hardware scan-line systems was done at the University of Utah in the late 1960s [WYLI67; ROMN69; WATK70].

Figure 18.16 is a block diagram of a typical scan-line rasterizer. The *y sorter* places each edge of each polygon into the bucket corresponding to the scan line in which it first appears. The *active-segment generator* reads edges from these buckets, maintaining a table of active edges for the current scan line. From this table, it builds a list of active segments (a segment is a span within a single polygon), which is sorted by the x value of the left endpoint of each segment. The *visible-span generator* (called the *depth sorter* in the Utah system) traverses the active segment list, comparing z values where necessary, and outputs the sequence of visible spans on the current scan line. The *shader* performs Gouraud shading on these spans, producing a pixel stream that is displayed on the video screen.

Notice that no frame buffer is needed in this type of system, provided that the system can generate pixels at video rates. The original Utah scan-line system generated the video signal in real time for a modest number (approximately 1200) of polygons. However, since the rate at which pixels are generated depends on local scene complexity, a small amount of buffering—enough for one scan line, for example—averages the pixel rate within a single scan line. A double-buffered frame buffer allows complete independence of image-generation and image-display rates. This architecture was the basis of several generations of flight-simulator systems built by Evans & Sutherland Computer Corporation in the 1970s [SCHA83].

18.7.3 Limits of Pipeline Rasterization and the Need for Parallelism

Two factors limit the speedup possible in a pipeline approach. First, most rasterization algorithms break down easily into only a small number of sequential steps. Second, some of these steps are performed far more often than are others, particularly the steps in the inner loop of the rasterization algorithm. The processor assigned to these steps, therefore, becomes the bottleneck in the system.

The inner loop in an object-order (z-buffer) system is calculating pixels within spans; the inner loop in an image-order (scan-line) system is processing active edges on a scan line. For rasterization to be accelerated beyond the level possible by simple pipelining, these inner-loop calculations must be distributed over a number of processors. In z-buffer

systems, this produces image parallelism; in scan-line systems, this produces object parallelism. The following two sections discuss each of these approaches. Virtually all of today's high-performance graphics systems use some variation of them.

18.8 IMAGE-PARALLEL RASTERIZATION

Image parallelism has long been an attractive approach for high-speed rasterization architectures, since pixels can be generated in parallel in many ways. Two principal decisions in any such architecture are (1) how should the screen be partitioned? (into rows? into columns? in an interleaved pattern?), and (2) how many partitions are needed? In the following sections, we shall describe the most heavily investigated alternatives, discussing the advantages and disadvantages of each. Also, we shall identify which schemes are approaching fundamental limits in current architectures. Note that, because an image-parallel system rasterizes in object order, a frame buffer is required to store intermediate results.

18.8.1 Partitioned-Memory Architectures

Two obvious partitioning strategies are to divide pixels into contiguous blocks (Fig. 18.17a) [PARK80] and to divide them into an interleaved checkerboard pattern (Fig. 18.17b) [FUCH77a]. In either approach, a processor (or PE) is associated with each frame-buffer partition. Such organizations increase a graphics system's computation power by providing parallel processing, and its memory bandwidth by providing each PE with a separate channel into its portion of frame-buffer memory. During rasterization, polygons are transferred from the front end to the PEs in parallel, and each PE processes primitives in its portion of the frame buffer.

Contiguous partitioning. In the contiguous-region partitioning scheme, primitives need to be processed in only those regions in which they may be visible. These regions can be determined rapidly using geometric extents. If primitives are small compared to the region size, each primitive is likely to fall into a single region. Large primitives may fall into

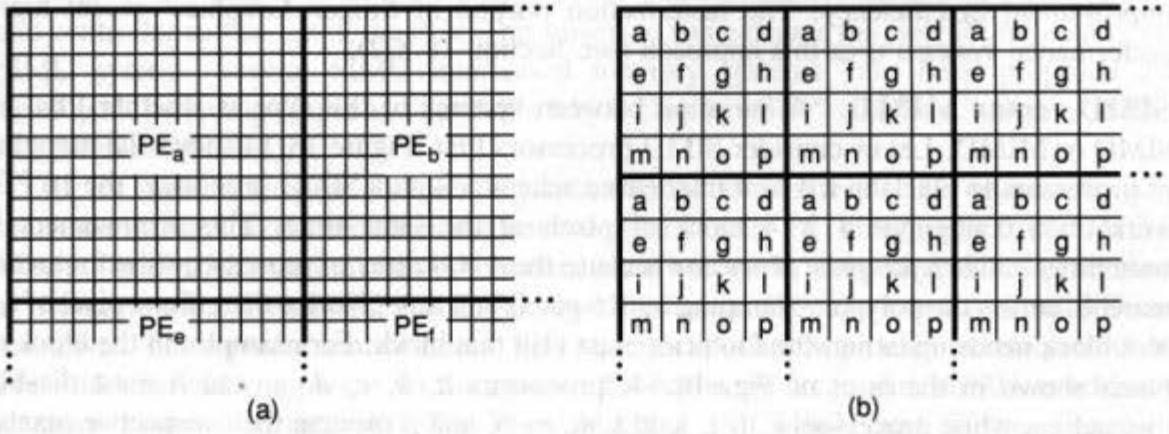


Fig. 18.17 Two schemes for frame-buffer partitioning. In (a), processors are assigned contiguous blocks of pixels; in (b), processors are assigned pixels in an interleaved pattern.

multiple regions. If the region size is chosen appropriately, the number of primitives handled by each processor will be approximately m/p, where m is the number of primitives in the database and p is the number of processors. Note, however, that if the viewpoint is chosen so unfortunately that all of the primitives fall into a single screen region, one processor must rasterize all of the primitives, and system performance decreases dramatically. In a contiguous region system, the frame rate is determined by the number of primitives in the busiest region.

Interleaved partitioning. Interleaved partitioning, on the other hand, achieves a better balance of workload, since all but the tiniest polygons lie in all partitions of the frame buffer. Since each processor handles every primitive (although only a fraction of its pixels), this scheme is less efficient in the best case than is the contiguous region approach. However, its worst-case performance is much improved, since it depends on the *total* number of primitives, rather than on the number in the busiest region. Because of this intrinsic load balancing, interleaved systems have become the dominant partitioned-memory architecture.

The polygon scan-conversion algorithm described in Section 18.7.1 requires set-up calculations to determine delta values and span endpoints before pixel computations can begin. These calculations need be performed only once per polygon or once per span, and can be shared among a number of PEs. The first proposed interleaved memory architectures [FUCH77a; FUCH79] contained no provision for factoring out these calculations from the PEs (see Fig. 18.18). Since each PE had to perform the entire rasterization algorithm for every polygon, many redundant calculations were performed.

Clark and Hannah [CLAR80] proposed an enhancement of this architecture to take advantage of calculations common to multiple PEs. In their approach, two additional levels of processors are added to perform polygon and edge processing. A single polygon processor receives raw transformed polygon data from the front-end subsystem and determines the polygon's initial scan line, slopes of edges, and so on. Eight edge processors (one per column in an 8×8 grid of pixel processors) calculate x, z, R, G, and B values at span endpoints. The edge processors send span information to the individual PEs (span processors), which interpolate pixel values along the span. The added levels of processing allow the PEs to perform only the calculations that are necessary for each pixel—a large improvement in efficiency. The rasterization portion of Silicon Graphics' recent high-performance systems uses this approach (see Section 18.8.2).

SIMD versus MIMD. A variation between systems of this type is whether PEs are SIMD or MIMD. Let us consider SIMD processors first. Figure 18.14 shows the mapping of processors to pixels in a 4×4 interleaved scheme. With a SIMD processor, the 16 PEs work on a contiguous 4×4 block of pixels at the same time. This arrangement is sometimes called a *footprint processor* because the 4×4 array of processors (the footprint) marches across the polygon, stamping out 16 pixels at a time. Notice that, if any pixel of a 4×4 block needs updating, the footprint must visit that block. For example, in the block of pixels shown in the inset of Fig. 18.14, processors a, b, c, d, g, and h must disable themselves, while processors e, f, i, j, k, l, m, n, o, and p process their respective pixels.

A disadvantage of SIMD processors is that they do not utilize their PEs fully. This occurs for two reasons. First, many of the PEs map to pixels outside the current primitive if

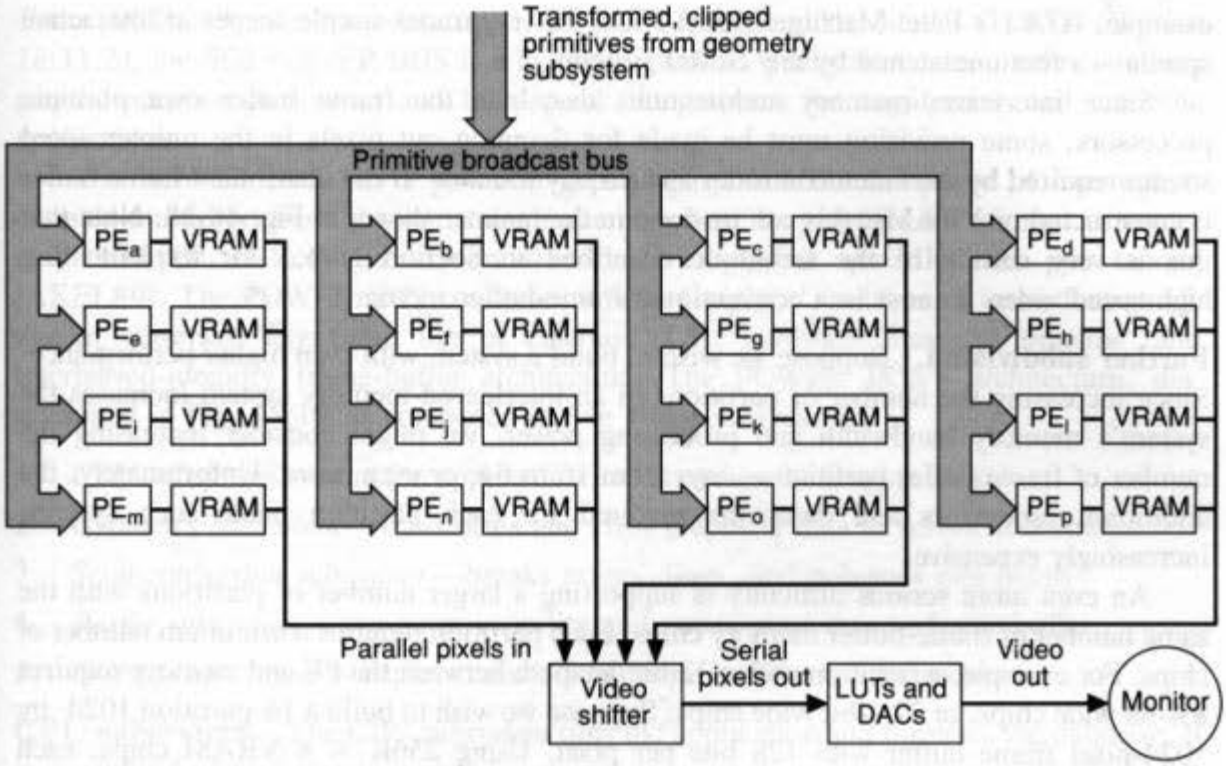


Fig. 18.18 Block diagram of a typical interleaved memory system. Each PE is responsible for one pixel in every 4 × 4 block of pixels.

small primitives are being rendered. For example, PEs in a 4 × 4 footprint processor rasterizing 100-pixel polygons map to pixels within the triangle as little as 45 percent of the time [APGA88]. Second, the choice of rasterization algorithm affects PE utilization. As remarked in Section 18.4.2, algorithms containing few conditional branches (including rasterizing convex polygons with Gouraud shading) can be implemented quite efficiently. Algorithms containing many conditional branches or using complicated data structures (such as rendering curved-surface primitives, texturing, shadowing, or antialiasing with a list of partially covering polygons) can be extremely difficult to make efficient. SIMD processors, however, can be built inexpensively and compactly, since a single code store and controller suffice for all the PEs. This offsets to some degree the poor PE utilization in a SIMD system. Several SIMD interleaved memory systems have been proposed and developed, including Gupta and Sproull's 8 by 8 Display [GUPT81b] and Stellar's GS2000 (see Section 18.11.3).

If we wish to support complex algorithms or to eliminate the idle processor cycles indicated in Fig. 18.14, we can add a control store to each PE, changing it from SIMD to MIMD. In a MIMD system, PEs do not need always to work on the same 4 × 4 pixel block at the same time. If each PE has FIFO input buffering, PEs can even work on different primitives. The separate control stores, FIFO queues, and hardware required to synchronize PEs add size and complexity to the system. Examples of successful MIMD interleaved-memory systems include AT&T's Pixel Machine [MCMI87] and Silicon Graphics' POWER IRIS (Section 18.8.2). Such systems compete well against SIMD systems on more complicated types of primitives or with more complicated rendering algorithms. For

example, AT&T's Pixel Machine Model PXM 924 ray traces simple scenes at interactive speeds—a feat unmatched by any SIMD system.

Since interleaved-memory architectures distribute the frame buffer over multiple processors, some provision must be made for scanning out pixels in the uninterrupted stream required by the video controller and display monitor. If the distributed frame buffer is constructed of VRAMs, this can be done in the manner shown in Fig. 18.18. Note that this is very similar to the technique described in Section 18.1.5 for implementing high-speed video scanout in a conventional frame-buffer memory.

Further subdivision. Suppose we wish to build a system with even higher performance. Since increasing the number of partitions in an interleaved-memory system increases the system's memory bandwidth and processing power, we might consider increasing the number of frame-buffer partitions—say, from 16 to 64, or even more. Unfortunately, the additional processors and datapaths required for each partition make such systems increasingly expensive.

An even more serious difficulty is supporting a larger number of partitions with the same number of frame-buffer memory chips. Each partition requires a minimum number of chips. For example, a partition with a 32-bit datapath between the PE and memory requires 8 4-bit wide chips, or 32 1-bit wide chips. Suppose we wish to build a 16-partition 1024- by 1024-pixel frame buffer with 128 bits per pixel. Using 256K × 4 VRAM chips, each partition requires 8 256K × 4 VRAM chips, so $16 \cdot 8 = 128$ chips are needed to support all 16 memory partitions. This is the exact number required to store the pixel data.

Suppose, however, that we increase the number of partitions from 16 to 64 (an 8 × 8 footprint). Although we still need only 128 memory chips to store the pixel data, we need $64 \cdot 8 = 512$ memory chips to support the PE–memory bandwidth. The extra 384 memory chips are needed only to provide communication bandwidth—not for memory. This is an extra expense that continues to grow as we subdivide the frame buffer further.

Increasing the density of memory parts from 1 Mbit to 4 Mbit exacerbates this problem even further. For example, if 1Mbit × 4 VRAM memory chips are used in the example mentioned above, 512 chips are still needed, even though each one contains sixteen times the memory actually required. Current systems such as the Silicon Graphics' POWER IRIS GTX (described in the next section), which uses 20 frame-buffer partitions, are already at the bandwidth limit. A way to ameliorate this problem would be for memory manufacturers to provide more data pins on high-density memory parts. Some 4-Mbit DRAM chips have eight data pins, rather than four, which helps somewhat, but only reduces the bandwidth problem by a factor of 2.

18.8.2 Silicon Graphics' POWER IRIS 4D/240GTX—An Interleaved Frame-Buffer Memory Architecture[2,3]

Silicon Graphics' POWER IRIS 4D/240GTX [AKEL88; AKEL89] uses many of the techniques described in this chapter. Like a number of its competitors, including the Ardent

[2]Material for this example is adapted from [AKEL88] and [AKEL89].

[3]In 1990, Silicon Graphics announced POWERVISION (similar to the GTX) that renders 1 million Gouraud-shaded triangles/sec and with 268 bits/pixel for antialiasing and texturing.

Titan [BORD89], the Megatek Sigma 70 [MEGA89], and the Stellar GS2000 (Section 18.11.3), the SGI POWER IRIS is a high-end graphics workstation, designed to combine general-purpose processing and high-speed 3D graphics for engineering and scientific applications.

The POWER IRIS has a powerful general-purpose CPU composed of four tightly coupled multiprocessors sharing a single memory bus. Its graphics subsystem can render over 100,000 full-colored, Gouraud-shaded, z-buffered quadrilaterals per second [AKEL89]. The POWER IRIS continues Silicon Graphics' tradition of immediate-mode display traversal, aggressive use of custom VLSI, hardware front-end pipeline, and interleaved-memory frame-buffer architecture. The POWER IRIS's architecture, diagrammed in Fig. 18.19, is composed of five major subsystems:

1. *CPU subsystem*—runs the application and traverses the display model
2. *Geometry subsystem*—transforms and clips graphical data to screen coordinates
3. *Scan-conversion subsystem*—breaks points, lines, and polygons into pixels
4. *Raster subsystem*—computes visibility and writes pixel data to frame buffer
5. *Display subsystem*—displays contents of frame buffer on color monitor.

CPU subsystem. The *CPU subsystem* runs the application and traverses the database. It is composed of four tightly coupled, symmetric, shared-memory multiprocessors. Hardware provides high-speed synchronization between processors, so parallelism can be achieved within a single process (although special programming constructs are required).

Geometry subsystem. The *geometry subsystem* transforms, clips, and lights primitives. It is composed of five floating-point processors arranged in a pipeline. Each of these processors, called a *geometry engine* (GE), contains an input FIFO, a controller, and a floating-point unit capable of 20 MFLOPS. Unlike Silicon Graphics' earlier Geometry Engine (see Section 18.5.4), the POWER IRIS' GEs are based on a commercial floating-point chip, the Weitek 3332.

The first GE transforms vertices and vertex normals. The second GE performs lighting calculations (supporting up to eight point light sources). The third GE performs trivial accept/reject clipping tests. The fourth GE performs exact clipping on primitives that cross clipping boundaries, and also does perspective division for all primitives. The fifth GE clips color components to maximum representable values, calculates depth-cued colors where necessary, and converts all coordinates to screen-space integers.

Scan-conversion subsystem. The *scan-conversion subsystem* rasterizes primitives using the pipeline approach described in Section 18.7.1, except that its spans are vertical columns of pixels, rather than the horizontal rows we have assumed so far (the only effect on the rasterization algorithm is that x and y coordinates are interchanged).

The single *polygon processor* sorts the vertices of each polygon from left to right in screen space. The sorted vertices are then used to decompose the polygon into vertically aligned trapezoids. The upper pair of vertices and the bottom pair of vertices of each trapezoid are used to calculate slopes for use by the edge processors.

The *edge processor* uses vertex and slope information to compute x, y, and z coordinates and color values for each pixel that lies on the top or bottom edges of each

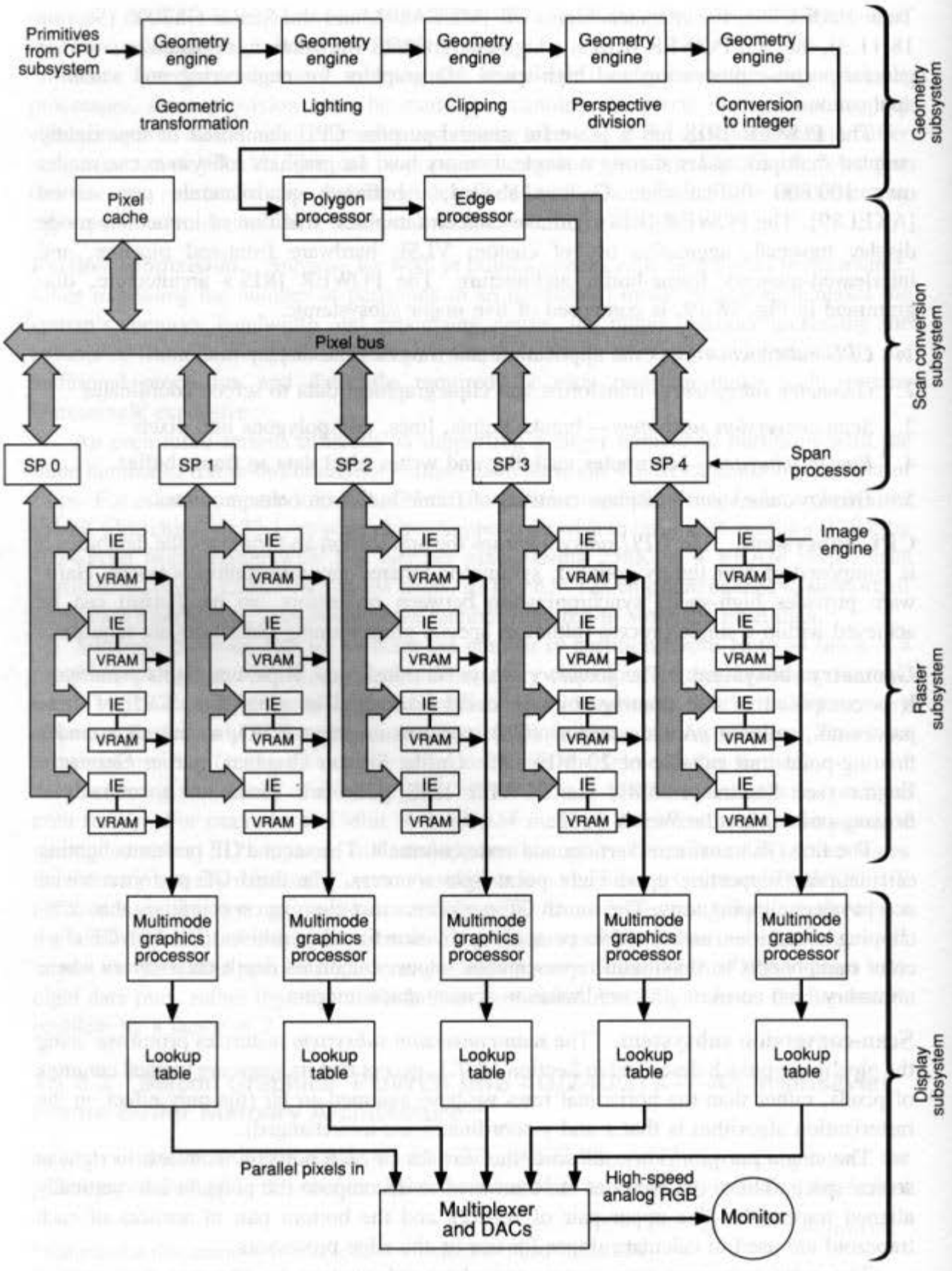


Fig. 18.19 Silicon Graphics' GTX system architecture (based on [AKEL88]).

trapezoid. In a given vertical column, a pair of pixels determines the top and bottom endpoints of a vertical span. These pixel pairs, together with slope information, are passed on to the span processors.

Each of the five parallel *span processors* is responsible for one-fifth of the columns of the display screen. For example, span processor 0 manages scan lines 0, 5, 10, and so on. Span processors calculate z, R, G, B, and α (for transparency and antialiasing) values for each pixel in the span. Because spans generated from a single polygon are adjacent, the processing load over span processors is approximately uniform.

The *pixel cache* buffers blocks of pixels during pixel copy operations so that the full bandwidth of the pixel bus can be used.

Raster subsystem. The *raster subsystem* takes pixel data generated by the span processors and selectively updates the image and z bitplanes of the frame buffer, using the results of a z comparison and α blending. The raster subsystem is composed of 20 *image engines*, each responsible for one-twentieth of the screen's pixels, arranged in a 4- $\times$ 5-pixel interleaved fashion. 96 bits are associated with each pixel on the screen: two 32-bit image buffers (r, G, B, and α), a 24-bit z-buffer, four overlay/underlay bitplanes, and four window bitplanes.

The overlay/underlay bitplanes support applications that use pop-up menus or windowing backgrounds. The window bitplanes define the display mode (single- or double-buffered, etc.) and window-masking information.

The 20 image engines work on pixels in parallel. Each can blend values based on the pixel's α value, allowing transparent or semitransparent objects to be displayed, and allowing supersampling antialiasing.

Display subsystem. The display subsystem contains five *multimode graphics processors* (MGPs), each assigned one-fifth of the columns in the display. The MGPs concurrently read image data from the frame buffer (together with window-display-mode bits), process them using the appropriate display mode (RGB or pseudocolor), and send them on to digital-to-analog converters for display.

The GTX's architecture is a good example of many of the techniques we have discussed so far. It provides high performance for polygon rendering at a reasonable cost in hardware. Because the GTX's rendering pipeline is highly specialized for graphics tasks, however, the system has difficulty with the advanced rendering techniques we shall discuss in Section 18.11, and its resources cannot be applied easily to nongraphics tasks. Section 18.11.3 discusses the architecture of Stellar's GS2000, which has complementary advantages and disadvantages.

18.8.3 Logic-Enhanced Memory

Since commercial memories may not support enough frame-buffer partitions, one might consider building custom memories with a large number of concurrently accessible partitions on a single chip. Since each (intrachip) partition must have its own connection to its associated (external) processor, extra pins must be added to each memory package to support these additional I/O requirements. Alternatively, multiple processors could be built

onto the chip itself. The first possibility—that of adding pins to memory chips—directly increases the memory bandwidth, but makes the chip package and associated circuit boards larger and more expensive, and also increases the power requirements. These packaging effects become progressively more severe as memory densities increase. (In the past two decades, the number of bits in a typical RAM has increased by a factor of 1000, while the size of the package and the number of pins have changed hardly at all.)

In this section, we shall concentrate on the second option—that of adding processing to multipartition memory chips. In the simplest schemes, only new addressing modes are provided, such as the ability to address an entire rectangle of memory pixels in parallel. At the other extreme, an entire microprocessor (including code store) could be provided for each internal partition of memory.

Before we describe specific logic-enhanced–memory approaches, let us consider the advantages and disadvantages of any logic-enhanced–memory scheme. First, adding logic or processing to memories has the potential to increase vastly the processing power within a system. By increasing the number of internal memory partitions and providing processing for each on the same chip, enormous processor/memory bandwidths can be achieved. Second, in custom VLSI chips, options become available that are impractical in board-level systems, since VLSI technology has an entirely different set of cost constraints for gates, wiring channels, and memory. Third, off-chip I/O bandwidth can potentially be reduced, since the only off-chip communication needed is to control the processor and to scan pixels out of the chip; this translates into fewer pins in the package and thus to a smaller package and less board space.

The principal disadvantages of an enhanced-memory approach are low memory densities and increased cost. With enormous production volumes, commercial DRAM manufacturers can afford to develop specialized, high-density fabrication capabilities and to incur large development costs to fine-tune their designs. Design and fabrication resources for custom memory chips, however, are generally more limited, resulting in densities lower than those of commercial RAMs. The price per chip is also high, since the costs for designing a custom VLSI chip are not offset by such large sales volumes. In spite of these disadvantages, at least one custom memory chip for graphics has become commercially successful—the VRAM. It remains to be seen whether other custom memory designs for graphics have sufficient market appeal to justify large-scale commercial development.

Pixel-Planes. An early and very general logic-enhanced–memory design is Pixel-Planes [FUCH81]. Pixel-Planes pushes frame-buffer subdivision to its extreme: It provides a separate processor for every pixel in the display. Each SIMD *pixel processor* is a 1-bit processor (ALU) with a small amount of memory. Figure 18.20 shows a block diagram of an enhanced-memory chip in Pixel-Planes 4, a prototype system completed in 1986 [EYLE88]. Its design is similar to that of the VRAM chip of Fig. 18.3, only here the 1-bit ALUs and associated circuitry replace the video shifter. Each enhanced-memory chip contains 128 pixels (columns in the memory array), and each pixel contains 72 bits of local memory (rows within the column).

Pixel-Planes' performance is not based simply on massive parallelism. If it was, each PE would have to perform all the operations for scan conversion independently, resulting in many redundant calculations and a grossly inefficient system. Rather, Pixel-Planes uses a

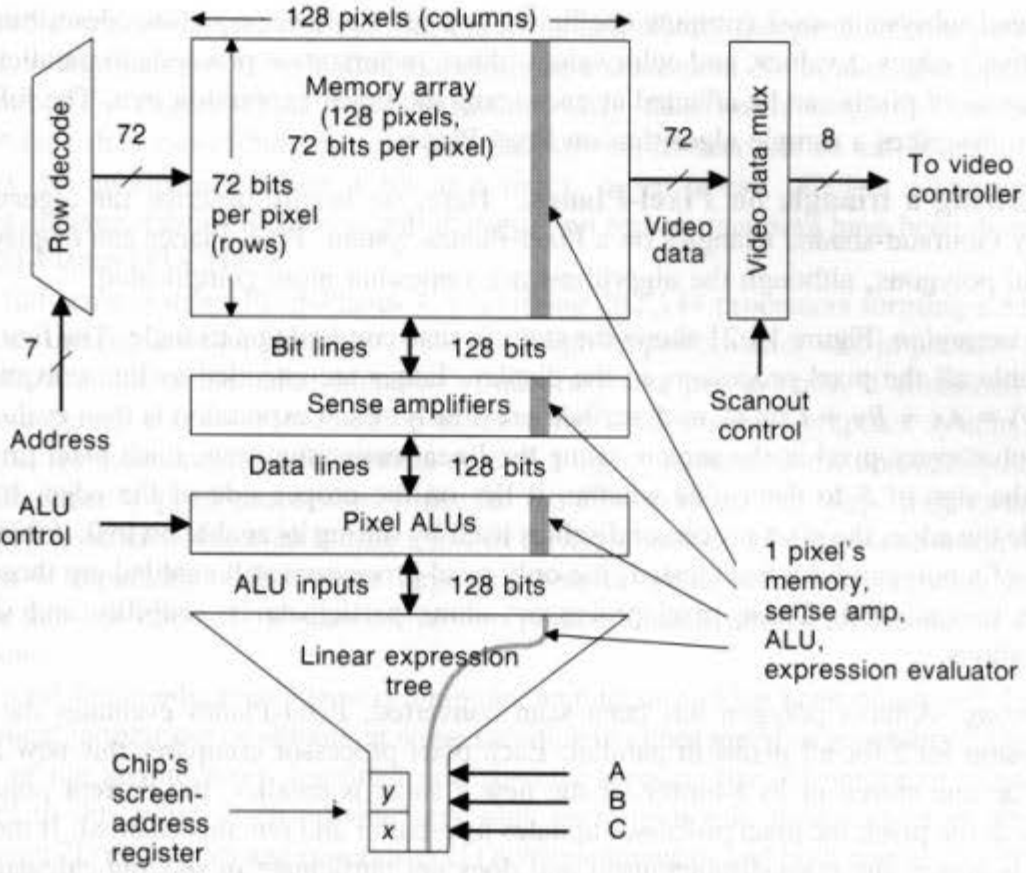


Fig. 18.20 Pixel-Planes 4 logic-enhanced–memory chip.

global computing structure called a *linear expression tree* that evaluates linear expressions of the form $F(x, y) = Ax + By + C$ for every pixel (x, y) of the screen in parallel [FUCH85]. A, B, and C floating-point coefficients are input to the tree; each pixel receives its own value of F in its local memory, 1 bit per clock cycle (approximately 20–30 cycles are required for each linear expression). The linear expression tree is especially effective for accelerating rasterization calculations, since many of these can be cast as linear expressions. For example,

- Each edge of a convex polygon can be described by a linear expression. All points (x, y) on one side of the edge have $F(x, y) \geq 0$; all points on the other side of the edge have $F(x, y) \leq 0$.
- The z value of all points (x, y) within a triangle can be described as a linear expression.
- R, G, and B color components of pixels in a Gouraud-shaded triangle can be described as linear expressions.

Double-buffering is implemented within Pixel-Planes chips by providing a *video-data multiplexer* that reads pixel values from specific bits of pixel memory while the image is computed in the remaining bits. Video data are scanned out of the chip on eight video data pins.

Displaying images on Pixel-Planes requires modifying the algorithms we have assumed so far. In addition to transforming and clipping primitives in the usual manner, the

front-end subsystem must compute coefficient sets for the linear equations describing each primitive's edges, z values, and color values. Also, rasterization proceeds in parallel, since large areas of pixels can be affected at once using the linear expression tree. The following section describes a sample algorithm on Pixel-Planes.

Rasterizing a triangle on Pixel-Planes. Here, we briefly describe the algorithm to display Gouraud-shaded triangles on a Pixel-Planes system. Pixel-Planes can display more general polygons, although the algorithms are somewhat more complicated.

Scan conversion. Figure 18.21 shows the steps in scan converting a triangle. The first step is to enable all the pixel processors in the display. Edges are encoded as linear expressions $F(x, y) = Ax + By + C = 0$, as described previously. Each expression is then evaluated in parallel at every pixel in the screen, using the linear expression tree. Each pixel processor tests the sign of F to determine whether it lies on the proper side of the edge. If it lies outside the edge, the pixel processor disables itself by setting its enable bit to 0. After all the edges of a polygon have been tested, the only pixel processors still enabled are those lying within the polygon. These pixel processors alone participate in visibility and shading calculations.

z-buffering. After a polygon has been scan converted, Pixel-Planes evaluates the linear expression for z for all pixels in parallel. Each pixel processor compares this new z value with the one stored in its z-buffer. If the new z value is smaller, the current polygon is visible at the pixel; the pixel processor updates its z-buffer and remains enabled. If the new z value is larger, the pixel disables itself and does not participate in shading calculations.

Gouraud shading. The linear expressions for R, G, and B components of the color are evaluated for each pixel in parallel by the linear expression tree. Pixel processors that are still enabled write the new color components into their pixels' color buffers.

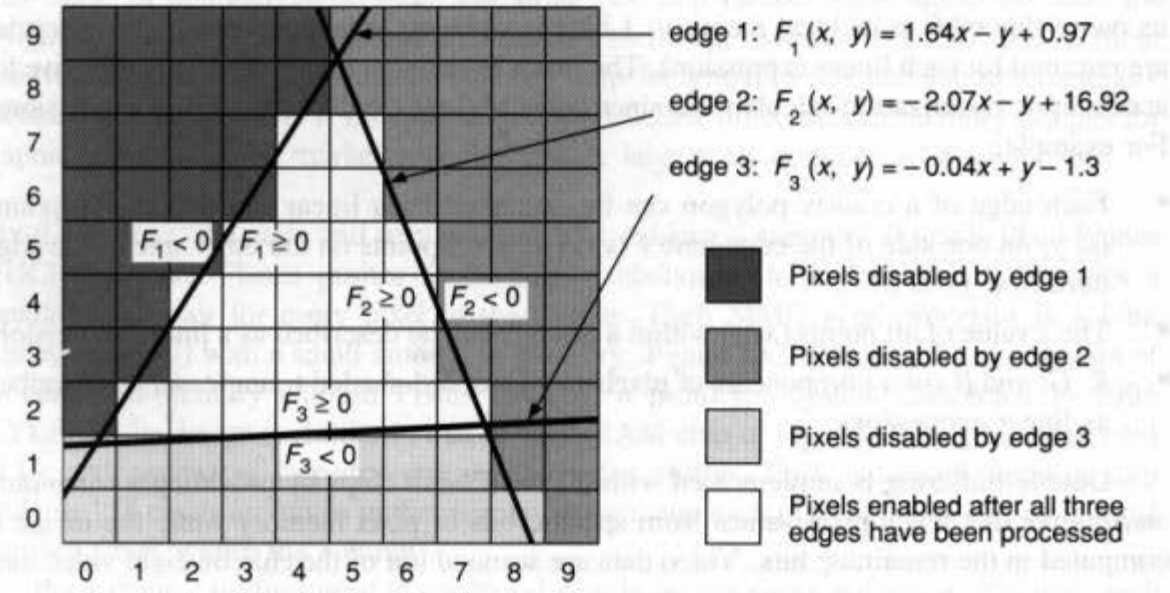


Fig. 18.21 Rasterizing a triangle on Pixel-Planes.

Note that scan conversion in Pixel-Planes is completely independent of a polygon's size, so a large polygon is rasterized as rapidly as a small one. Note, also, that operations that cannot be expressed as linear equations may take much longer to execute on Pixel-Planes than those that can (e.g., a quadratic expression can be calculated in pixel memory by multiplying values 1 bit at a time). Nevertheless, efficient algorithms for drawing spheres, casting shadows, antialiasing, and texture mapping have been developed for Pixel-Planes [FUCH85].

A full-scale system, Pixel-Planes 4, containing 262,144 processors forming a 512- by 512-pixel image, was completed in 1986. Although its performance was impressive for its time (40,000 Gouraud-shaded triangles of arbitrary size per second), it contained 2048 custom memory chips—a prohibitive expense for a commercial graphics system. This highlights the fundamental disadvantage of such a highly parallel SIMD approach: the very low utilization of pixel processors. Since all the PEs work in lock step, they cannot be retargeted to other tasks, even if only a few of them are still calculating useful results. This problem is especially severe when large numbers of small polygons are being drawn, because the first steps of the scan-conversion process disable almost all the screen's pixel processors.

Several logic-enhanced–memory graphics architectures have been developed that are more frugal in their use of silicon, at some sacrifice in either speed or generality. Although neither of the architectures described next directly supports the 3D rendering techniques assumed to this point, both provide very high performance in their respective domains (displaying 2D rectangles and generating 2D halftone images), and both provide insight into the potential of the logic-enhanced–memory approach.

Rectangle area-filling memory chip. Whelan proposed modifying the row and column addressing in the 2D memory-cell grid of a typical RAM to allow an entire rectangular region to be addressed at once [WHEL82]. Minimum and maximum row and column addresses specify the left, right, top, and bottom boundaries of the region. One write operation can store a single data value in every location within the region. This allows upright, constant-shaded rectangles to be rasterized in very few clock cycles—just enough to specify the four address values and the constant data.

Scan Line Access Memory. Demetrescu designed a more complicated chip called a Scan Line Access Memory (SLAM) for rasterizing more general 2D primitives [DEME85]. Like VRAMs and Pixel-Planes, SLAM takes advantage of the fact that, internally, a RAM reads or writes an entire row of its memory array in one cycle. Figure 18.22 shows a block diagram of a single SLAM chip. Each chip contains 256×64 bits of frame-buffer memory. Each row of memory corresponds to 1 bit in a scan line of pixels. In a system with k bits per pixel, a SLAM chip can store up to $64/k$ scan lines of 256 pixels each. In each memory cycle, a SLAM chip can read or write one row of memory from its memory array. By specifying appropriate x_{min} and x_{max} values, one can address any contiguous span of pixels on the current scan line, allowing fast polygon scan conversion. Video scanout is accomplished using a display shifter in exactly the same manner as a VRAM chip.

In a single clock cycle, either a row address, an x_{min} value, an x_{max} value, or a 16-bit repeating data pattern (for specifying halftone patterns) can be specified. Concurrent with

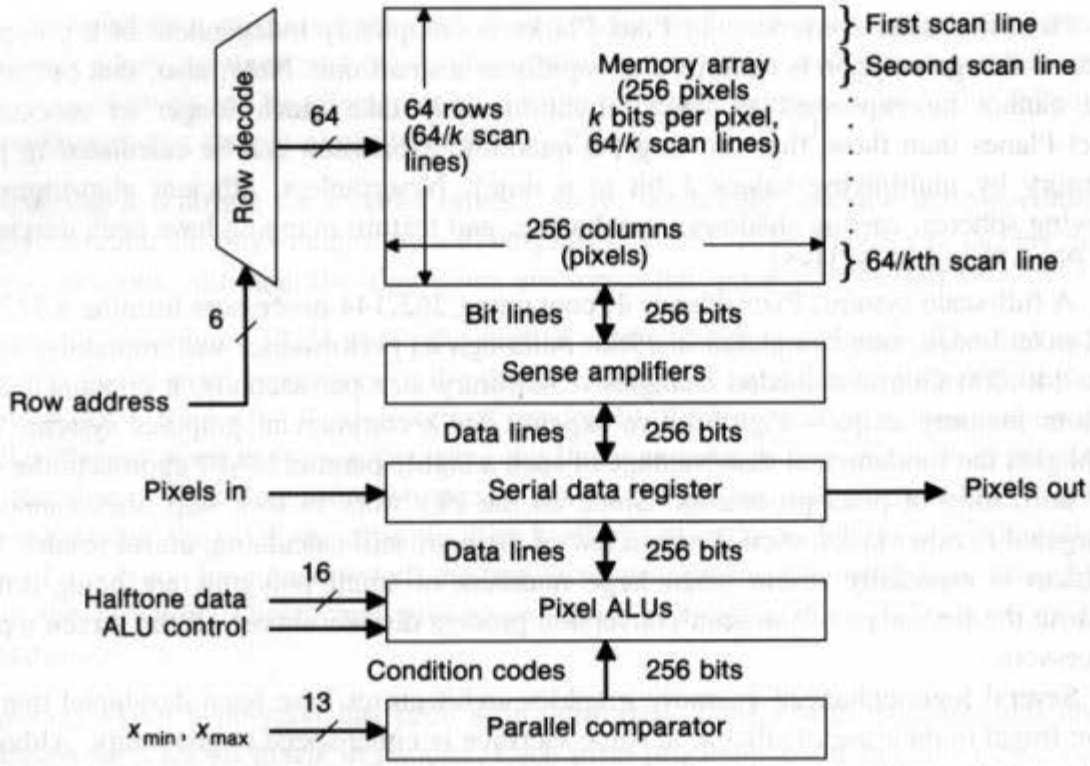


Fig. 18.22 Block diagram of a SLAM chip (for a system configured with k bits per pixel).

any one of these commands, SLAM can write the current scan-line segment into its memory array and can optionally increment the row address. Since, for many primitives, row addresses and halftone patterns need to be specified only once (for the initial scan line), succeeding scan lines can be processed rapidly. SLAM therefore can scan convert a convex polygon covering n scan lines in $2n + 2$ cycles (four cycles to specify the row address, x_{min}, x_{max}, and the halftone pattern for the initial scan line, and two cycles to specify x_{min} and x_{max} for each of the remaining $n - 1$ scan lines).

A SLAM system is composed of a number of SLAM chips (the number depends on the dimensions of the display screen). Figure 18.23 shows a SLAM system for updating a 512 by 512 monochrome display screen. Systems with more bits per pixel require proportionately more SLAM chips. SLAM can also be extended to display Gouraud-shaded polygons by adding a Pixel-Planes–style linear-expression tree. However, this enhanced version of SLAM would require approximately the same amount of hardware as Pixel-Planes and would suffer the same low utilization of PEs, since the pixels in any given (small) primitive would likely be contained in just a few SLAM chips.

Although both Whelan's architecture and the original SLAM architecture use less hardware than does Pixel-Planes, they do not offer the generality necessary to render realistic 3D images. Pixel-Planes and the enhanced version of SLAM do offer this generality, but suffer poor PE utilization. It would be useful to gain the performance of these processor-per-pixel architectures, but with higher PE utilization for small primitives. Section 18.10.1 examines a way to accomplish this using enhanced-memory arrays smaller than the full screen size.

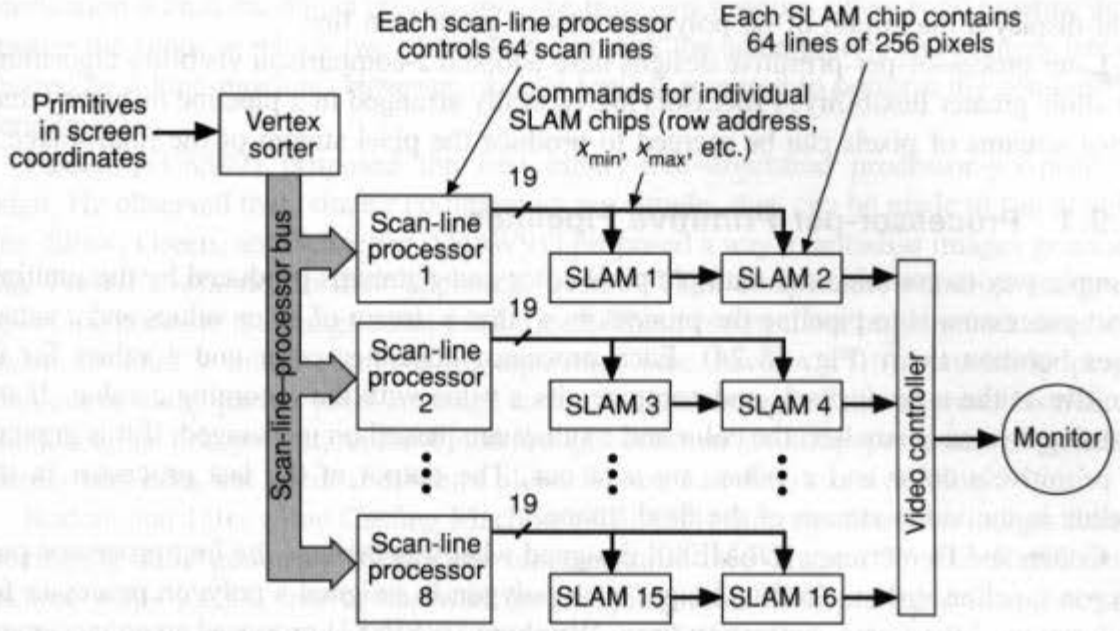


Fig. 18.23 SLAM system for updating a 512- by 512-pixel monochrome display.

18.9 OBJECT-PARALLEL RASTERIZATION

So far, we have focused on image-parallel architectures. The *object-parallel* family of parallel architectures parallelizes the inner loop of image-order (generally scan-line) algorithms. In an object-parallel architecture, multiple primitives (objects) are processed in parallel, so that final pixels may be generated more rapidly.

The usual object-parallel approach is to assign primitives (either statically or dynamically) to a number of homogeneous *object processors*, each of which can generate an entire image containing its primitive(s). During rasterization, each object processor enumerates the pixels of the display in some specific order (generally scan-line), generating color, z, and possibly partial-coverage values for its primitive(s). The pixel streams from each of the object processors are then combined to produce a single pixel stream for the final image. Although any number of primitives may be assigned to each object processor, most designs allocate a single primitive per processor. The advantage is that each processor can perform a well-defined task and thus can be reproduced inexpensively.

General Electric's NASA II. A pioneering real-time processor-per-primitive system was General Electric's NASA II flight simulator [BUNK89], delivered to NASA in 1967. The NASA II contained a number of hardware units called *face cards*, each of which rasterized a single polygon at video rates. At any given instant, each face card would process the same pixel.

The NASA II used a depth-sort visibility algorithm, rather than a z-buffer. The output of each face card included a bit indicating whether or not the pixel was covered by its polygon, the pixel color, and the polygon priority number. This information was fed into a priority multiplexer so that, at each pixel, the color of the highest-priority visible polygon was output. Since face cards were expensive, they were reassigned to new polygons when their polygons no longer intersected the current scan line. The NASA II system

could display a maximum of 60 polygons on any given scan line.

Later processor-per-primitive designs have adopted z-comparison visibility algorithms that allow greater flexibility. Processors are typically arranged in a pipeline or binary tree, so that streams of pixels can be merged to produce the pixel stream of the final image.

18.9.1 Processor-per-Primitive Pipelines

A simple way to combine the multiple pixel color and z streams produced by the multiple object processors is to pipeline the processors so that a stream of color values and z values passes between them (Fig. 18.24). Each processor generates color and z values for its primitive at the current pixel, and compares its z value with the incoming z value. If the incoming z value is smaller, the color and z values are passed on unchanged; if it is greater, the primitive's color and z values are sent out. The output of the last processor in the pipeline is the video stream of the final image.

Cohen and Demetrescu [DEME80] designed what was perhaps the first processor-per-polygon pipeline system. In this design, each polygon is assigned a polygon processor for the duration of the frame-generation time. Weinberg [WEIN81] proposed an enhancement of this architecture to generate antialiased images. Instead of passing pixel color and z values between adjacent processors, Weinberg's design passes, for each pixel, an arbitrarily long packet of polygon-fragment information relating to that pixel. Each polygon processor compares z and edge information for its polygon with the incoming packet describing the pixel. The packet is updated to take the new polygon into account, and is forwarded to the next polygon processor. A set of filtering engines at the end of the pipeline calculates the single color value for each pixel from the (final) packet associated with that pixel.

A team at Schlumberger proposed a design combining aspects of the NASA II, Cohen–Demetrescu, and Weinberg designs [DEER88]. Their Triangle Processor and Normal Vector Shader uses a pipeline of triangle processors that passes along surface-normal and polygon-color data, rather than actual pixel colors. Triangles are assigned to processors only during scan lines in which they are active in the same manner as in the NASA II. A pipeline of Normal Vector Shaders computes the Phong lighting model for pixels emerging from the triangle-processor pipeline. Because lighting and shading calculations are delayed until the end of the pipeline, Phong calculations need to be performed only once for each pixel in the final image, rather than once for each pixel of every polygon—a substantial savings.

18.9.2 Tree-Structured Processor-per-Primitive Architectures

An alternative to pipelining object processors is to arrange them in parallel and to use a binary tree of compositors to merge the color and z streams. This approach allows

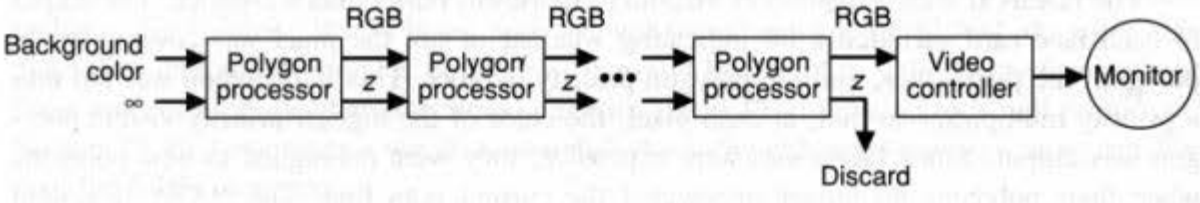


Fig. 18.24 Processor-per-polygon pipeline system.

rasterization within the object processors to be truly synchronous (there is no pipeline delay between the times at which two processors compute the same pixel), and reduces latency incurred by a long pipeline. However, special logic is required to perform the composition operation.

Fussell [FUSS82] proposed the first binary-tree-structured processor-per-primitive design. He observed that, since z comparators are simple, they can be made to run at video rates. Shaw, Green, and Schaeffer [SHAW91] proposed a way to antialias images generated using Fussell's scheme. In their approach, custom VLSI *compositors* combine images at each of the nodes in the image-composition tree. These compositors implement a simplified version of Duff's image-composition algorithm (see Section 17.6), providing proper treatment of many partial-pixel-coverage cases. A disadvantage of this approach is that the antialiasing is not perfect; color bleedthrough between abutting polygons is possible, whereas this does not occur if z values alone are used to determine visibility.

Kedem and Ellis's Ray Casting Machine [KEDE84] directly rasterizes images from a constructive solid geometry (CSG) tree representation (see Chapter 7). The Ray Casting Machine maps a CSG tree to hardware by assigning each CSG primitive to a processor called a *primitive classifier* and each operator to a processor called a *classification combiner*.

The image is traversed pixel by pixel in raster-scan order. For each pixel, primitive classifiers compute the segment of the ray through that pixel that is interior to their respective CSG primitives. These segments are then passed upward to the classification combiners, which perform set operations on them using the techniques described in Section 15.10.3. Segments are split and combined as needed. The updated segments are passed upward in the tree until they reach the root of the tree.

The set of segments emerging from the root of the classification-combiner tree describes the intersection of the current ray with the entire CSG object. The near endpoint of the nearest segment contains the z value of the visible surface at that pixel. This value is used to compute a color value for the pixel using any desired lighting model. If all the segments for a pixel, rather than just the closest one, are considered, the Ray Casting Machine can calculate the volume of the object or other geometric quantities about the CSG object. A prototype Ray Casting Machine completed in 1988 computes the surfaces of CSG objects with 32 primitives in near real time, although shading calculations, which are currently performed by the host computer, take several seconds.

18.9.3 Object Parallelism versus Image Parallelism

Although object-parallel architectures are simple and appealing, they have received much less attention than have the image-parallel techniques discussed previously; a number of experimental object-parallel systems have been proposed, but few have led to commercial products. Several factors may be responsible:

- Object-parallel systems typically require specialized processors. This implies heavy reliance on custom VLSI chips, making system design difficult and expensive. Image-parallel systems, on the other hand, place more reliance on frame-buffer memory, which can be built with commercial parts such as VRAMs.
- The specialized nature of object processors limits the types of primitives that can be displayed and the shading algorithms that can be used.

- Object-parallel systems have poor overload characteristics. Generally, object-parallel systems perform at full speed as long as there are enough object processors. Special provisions must be made to handle large databases, and performance generally decreases rapidly.

In addition to these factors, system designers have so far been able to increase system performance using image parallelism alone, so the designers of commercial systems by and large have not had to confront the challenges of object-parallel systems. Nevertheless, as discussed in Section 18.8, image parallelism may be reaching a point of diminishing returns, and object parallelism may appeal to the designers of future systems. The following section discusses another promising approach toward increasing system efficiency and performance: building hybrid-parallel systems that combine aspects of the approaches we have discussed so far.

18.10 HYBRID-PARALLEL RASTERIZATION

We have seen that when image parallelism and object parallelism are pushed to the extreme, systems with low utilization result. This poor utilization can be reduced if object-order and image-order rasterization techniques are used in combination. Such hybrid-parallel systems frequently are more complex than are the systems we have discussed so far, but they can be much more efficient. They also provide us with one more layer of parallelism that can be used to build still higher-performance systems.

18.10.1 Virtual Buffers and Virtual Processors

A major drawback of highly parallel architecture designs, both object-parallel and image-parallel, is the low utilization of each PE. As discussed in Section 18.8.2, image-parallel architectures with many partitions can have extremely poor PE utilization. For example, in a fully instantiated processor-per-pixel system such as Pixel-Planes 4, PEs may be doing useful work less than 1 percent of the time. Similarly, in an object-parallel architecture such as Cohen and Demetrescu's processor-per-polygon pipeline, PEs may be actively rasterizing their polygons less than 1 percent of the time.

One way to achieve higher utilization is to build only a fraction of the hardware, but to allocate its resources dynamically around the screen as they are needed. Two variants of this technique exist: *virtual buffers* (for image-parallel systems) and *virtual processors* (for object-parallel systems). Like classic virtual memory, both attempt to increase a system's apparent physical resources by reallocating resources dynamically as needed.

Virtual buffers. In a virtual-buffer system [GHAR89], the screen is divided (conceptually) into a number of regions of uniform size, and a parallel rasterization buffer the size of a region computes the image one region at a time. Since this buffer is small, it can be built using fast or custom processing/memory at a reasonable price. A full-sized conventional frame buffer is generally provided to store the final image.

A region can be a single scan line, a horizontal or vertical band of pixels, or a rectangular area. Virtual buffers differ from interleaved-memory "footprint" processors in one critical respect: A virtual buffer remains in one screen region until *all* the primitives for

that region are processed, whereas an interleaved-memory footprint processor moves about the image to rasterize each primitive, typically returning to the same region more than once.

Virtual processors. A virtual-processor system is outwardly similar to a scan-line virtual-buffer system; in both cases, the image is computed one scan line at a time. A virtual-processor system, however, uses object-parallel PEs, rather than image-parallel PEs. Since object processors are needed for only the primitives active on a single scan line, the number of object processors can be a small fraction of the number of primitives. General Electric's NASA II and Deering's Triangle Processor, discussed in Section 18.9, are both virtual-processor systems.

Complete versus incremental bucket sorting. Because virtual-buffer/virtual-processor systems visit a region only once, bucket sorting is required. The shape of the region influences the nature of the bucket sort. In virtual-buffer systems with rectangular regions, primitives can be stored in the buckets of all the regions in which they appear, since most primitives lie entirely within one region. We call this *complete bucket sorting*. In systems with scan-line–sized regions, complete bucket sorting is impractical, since a single primitive typically covers more than one scan line. *Incremental bucket sorting*, in which a primitive is stored only in the bucket associated with its initial scan line, is generally the method of choice.

The virtual-buffer/virtual-processor approach has several advantages and disadvantages. The most important advantage is that it makes possible the rasterization speed of a fully instantiated image-parallel system with a fraction of the hardware. It also decreases traffic to and from the frame buffer, since each region's pixels are written only once. Finally, no full-screen z-buffer is needed.

Virtual-buffer/virtual-processor systems have two major disadvantages, however. First, extra memory is required to buffer the scene during bucket sorting. If transformed primitives require the same amount of storage as do primitives in object coordinates, this approximately doubles the memory requirements in the front end. It also places a hard limit on the number of primitives that can be displayed in a single frame. These disadvantages offset, to some degree, the advantage of not needing a full-screen z-buffer. Another important disadvantage is that latency is added to the display process. Even though bucket sorting can be pipelined with rasterization, bucket sorting for one frame must be completed before rasterization of that frame can begin. This increases the system's latency by one frame time, which can be detrimental in real-time systems.

Systolic Array Graphics Engine (SAGE). We now consider a system that uses the virtual-buffer approach (we discussed several virtual-processor systems in Section 18.9). SAGE is a scan-line virtual-buffer system that uses a 1D array of pixel processors implemented in VLSI [GHAR88]. Like other scan-line systems, it generates the image one scan line at a time. However, SAGE uses an image-parallel z-buffer algorithm to rasterize primitives within a single scan line.

In addition to the array of pixel processors, SAGE contains auxiliary processors to maintain the active-edge list and to break polygons into spans (see Fig. 18.25). A *polygon manager* maintains the list of active polygons as scan lines are processed (at each new scan line, it adds polygons from the next bucket and deletes polygons that are no longer active).

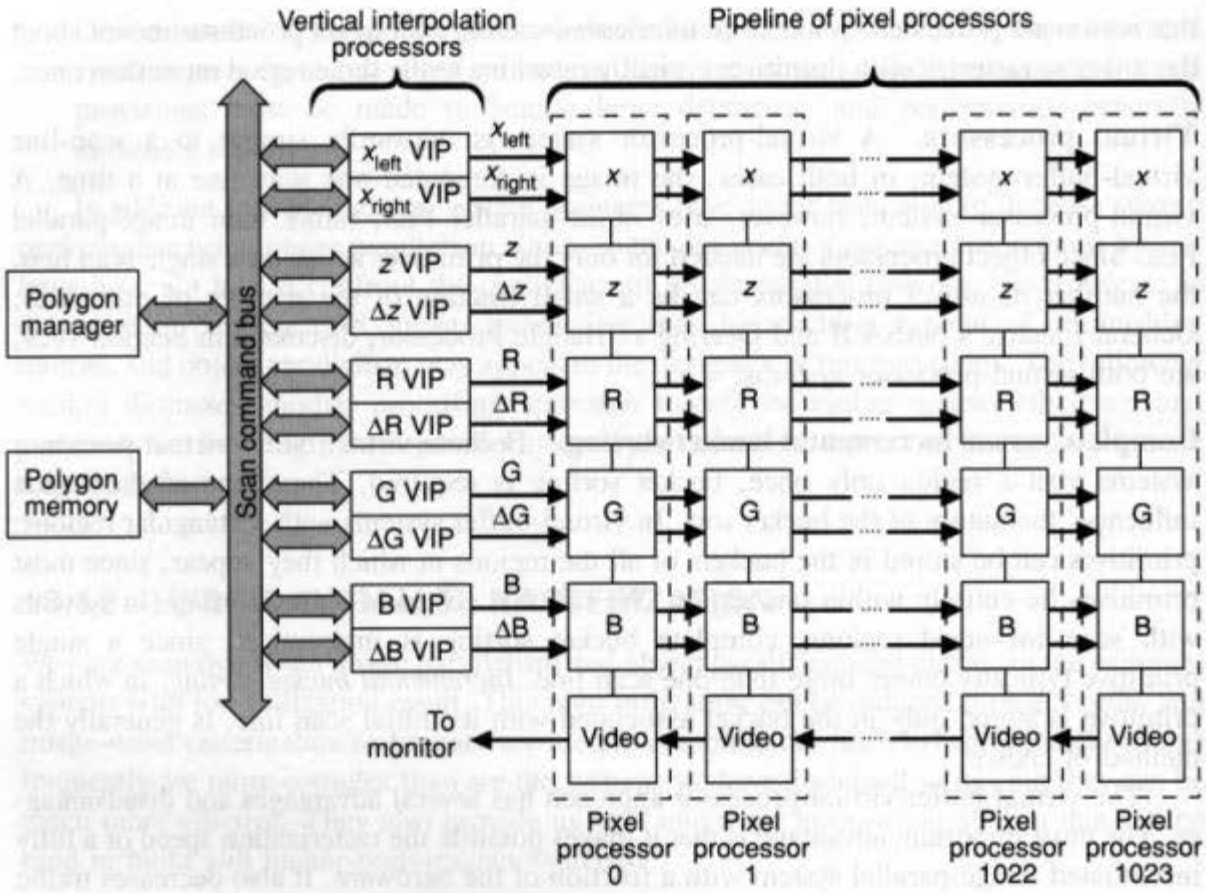


Fig. 18.25 Block diagram of a complete SAGE system.

It also calculates delta values for computing span endpoints on successive scan lines. *Vertical interpolation processors* use these delta values to compute span endpoints for each polygon active on the current scan line. So that many accesses to polygon memory can be avoided, these processors contain internal memory for storing delta values and x, z, R, G, and B values for up to 512 active polygons.

After these set-up operations, the vertical interpolation processors load the endpoints of each active polygon's current span into the pipeline of pixel processors. When a pixel processor receives a span, it compares its local x value against the span's endpoint values. If the pixel lies within the span, the processor calculates z and color values for its pixel (again using the span-endpoint values). If the new z value is smaller than the value stored at the pixel, the z and color values for the pixel are updated. Pixel values are scanned out for display using a serial shifting scheme similar to that used in VRAMs. The video stream flows in the opposite direction from that of the stream of spans.

18.10.2 Parallel Virtual-Buffer Architectures

Another level of parallelism is available in virtual-buffer systems that use rectangular regions and complete bucket sorting. Since complete bucket sorting allows a region to be rasterized without any knowledge of preceding regions (unlike incremental bucket sorting, which requires knowledge of the active-polygon list from the preceding region), a system

that uses complete bucket sorting can have multiple rasterization buffers. These rasterization buffers can work in parallel on different screen regions, allowing even faster systems to be built.

In such a system, each buffer is initially assigned to a region. When it has processed all the primitives in the region's bucket, it is assigned to the next region needing processing. This provides an additional coarse-grained level of image parallelism that can be used regardless of the rasterization method used by the individual virtual buffers.

Pixel-Planes 5. Pixel-Planes 5, which was under construction in 1989 [FUCH89], is one of the first systems to use parallel virtual buffers. Pixel-Planes 5 uses 10 to 16 logic-enhanced memory rasterization buffers that are 128 pixels on a side. Each buffer is a miniature Pixel-Planes 4 array, and is capable of rasterizing all primitives falling into a 128 by 128 region. These rasterizers can be assigned dynamically to any of the 80 regions in a 1280- by 1024-pixel screen.

Rasterization in Pixel-Planes 5 occurs in two phases. First, the front-end processors (16 to 32 floating-point processors) transform and sort primitives into screen regions. Sorting is done by checking a primitive's screen extent against the 128-pixel–aligned region boundaries; primitives that cross region boundaries are placed in the buckets of all regions affected (this occurs approximately 20 percent of the time for 100-pixel triangles). After bucket sorting is complete, the multiple rasterizers process regions in parallel. When a rasterizer finishes a region, it transfers its newly computed image to the appropriate part of a conventional frame buffer and begins processing primitives from an unassigned region (see Fig. 18.26). All communication between system components is performed over a 1.28-gigabyte-per-second token ring.

Pixel-Planes 5 achieves much higher processor utilization than did previous Pixel-Planes systems, since 128 by 128 regions are much closer to the size of most primitives than is a single 512 by 512 region. Since virtual-buffer rasterizers can be assigned to screen regions dynamically, system resources can be concentrated on the regions that need them the most.

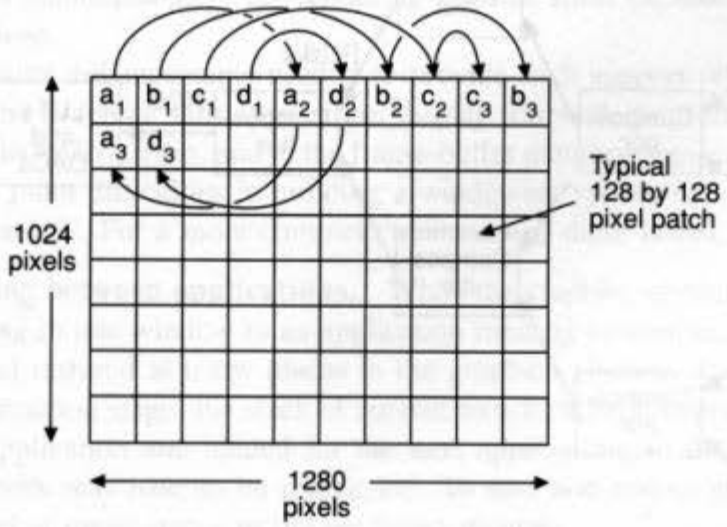


Fig. 18.26 Dynamic assignment of rasterizers to regions in Pixel-Planes 5. (Reprinted with permission from [FUCH89]. © ACM.)

Parallel virtual-buffer systems do present two difficulties. First, transferring buckets to multiple rasterization buffers in parallel requires a high-performance data network between the front-end and rasterization subsystems, as well as sophisticated control and synchronization software [ELLS89]. Second, in some images, most of the primitives in the database can fall into a single region, making the extra layer of parallelism useless. The primitives in the overcrowded region could be allocated to more than one rasterizer, but then the multiple partial images would have to be combined. Although this complicates the rasterization process, it can be done by compositing the multiple images into one buffer at the end of rasterization.

18.10.3 Image-Composition Architectures

The notion of combining images after rasterization can be used to build a second type of multilevel parallel architecture, *image-composition* or *composite* architectures [MOLN88; SHAW88]. The central idea is to distribute primitives over a number of complete rendering systems. The multiple renderers are synchronized so they use identical transformation matrices and compute the same frame at the same time. Each renderer then computes its partial image independently and stores that partial image in its own frame buffer.

Video scanout from each frame buffer occurs in the normal way, except that z-buffer contents are scanned out as well. Scanout processes in each frame buffer are synchronized so that each frame buffer scans out the same pixel at the same time. A tree or pipeline of compositors combines the RGB and z streams from each renderer using the technique

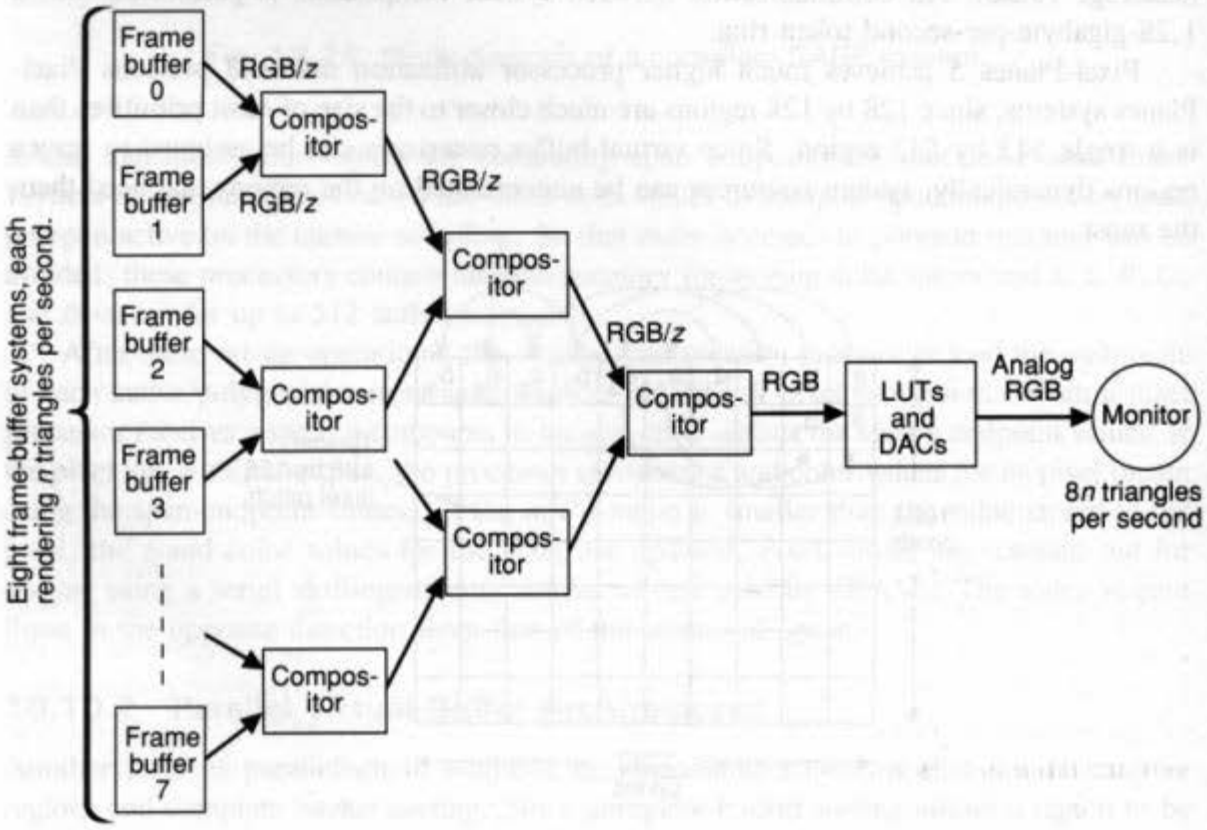


Fig. 18.27 An image-composition system composed of eight individual renderers.

described in Section 18.9.2. Figure 18.27 shows a composite system for displaying $8n$ triangles per second built from eight renderers, each of which can display n triangles per second.

This technique can be exploited to build systems of arbitrarily high performance by using a large number of parallel renderers. The main difficulties with this approach are the need to distribute the database over multiple processors, which incurs all the difficulties of parallel front ends (described in Section 18.6.1); aliasing or erroneous pixels caused by the image-composition operation; and a lack of flexibility, since image composition restricts the class of rasterization algorithms that can be implemented on the machine. Nevertheless, this approach provides an important way to realize systems of extremely high performance.

18.11 ENHANCED DISPLAY CAPABILITIES

This section discusses the architectural implications of a variety of enhancements to the standard Gouraud-shaded polygon-rendering algorithm. Increasing numbers of these features are judged necessary or desirable by suppliers and users. At the end of this section, we discuss a few aspects of flight simulators, the systems that traditionally have included the greatest number of advanced features.

18.11.1 Support for Multiple Windows

As mentioned in Chapter 10, the ability to display multiple overlapping windows controlled by separate applications has become a necessity in current workstations. Without hardware support for this capability, the overall speed of the graphics system may suffer seriously. For example, operations such as pushing one window behind another, popping a window so that all of it becomes visible, and dragging a window around the screen, all require the ability to copy pixels rapidly from one frame-buffer region to another, and, in some cases, to regenerate part or all of an image. Also, if multiple images are to be generated in different windows simultaneously, the rendering pipeline must be able to switch rapidly between applications.

The architectural enhancements needed to provide such support often cut across the design of the entire graphics subsystem: from display traversal, to geometric transformation, to clipping, to rasterization, and to the frame-buffer memory organization. We discuss here some of the main difficulties in building a windowing system that supports multiple interactive applications. For a more complete treatment of these issues, see [RHOD89].

Context switching between applications. When the graphics system switches from an application running in one window to an application running in another, state information must be saved and restored at many places in the graphics pipeline. For example, in the modeling-transformation stage, the stack of current transformation matrices must be saved for the current application and loaded for the next application; in the video controller, look-up table entries may have to be reassigned. To save and restore state rapidly, local memory is needed at many stages in the rendering pipeline.

Having a fixed amount of local storage, of course, limits the number of applications that can run simultaneously. If fast context switching is not possible, graphics systems

generally switch between windows as infrequently as possible (perhaps only a few times per second), causing a noticeable hesitation in the movement of objects in the various windows.

Clipping to nonconvex screen windows. When windows overlap, pixels must be written only to the exposed portions of their respective windows. This requires clipping pixels to window boundaries that may not be convex. Several solutions to this problem are possible.

Writing an entire window to offscreen memory. This allows the rendering pipeline to be largely isolated from windowing issues. A fast pixel copy can transfer exposed portions of the window to the actual frame buffer after each frame update. This requires extra processing for the pixel copy and adds latency to the image-generation process, both of which can seriously hamper interactivity. Alternatively, the video system can assemble the video stream on the fly from disjoint portions of video memory, as is done in Lexidata's Lex 90 system [SUYD86] or Intel's 82786 graphics coprocessor chip [SHIR86]. This approach does not add latency, but may restrict the locations and number of windows, since assembling the video image on the fly from different memory locations places heavy demands on the memory system.

Selectively updating screen pixels. A second alternative is to compute all the pixels for each window, but to update only those pixels in exposed portions of the window. This is analogous to scissoring, described in Chapter 3. To make this operation fast, hardware support is generally needed. A popular approach is to use window ID bits [AKEL88]. Here, a few (generally 4 to 8) additional bits are associated with each pixel. These bits specify a unique ID for each window. When the frame buffer is updated, hardware compares the window ID of the pixel to be written with the window ID stored in the frame buffer. The frame-buffer pixel is updated only if the window IDs agree.

Clipping within the rendering pipeline. The main disadvantage of both of the preceding schemes is that they generate many pixels that may never be displayed. The alternative is to clip within the rendering pipeline, so that the only pixels computed are those that will actually be displayed. The main difficulty with this approach is that the exposed part of a window may not be a simple rectangle: It may have concavities caused by partially overlapping windows of higher priority, or it may even contain a hole consisting of a smaller, higher-priority window. Clipping to arbitrary boundaries such as these is computationally expensive. One solution is to tesselate the exposed region with a series of rectangular windows and to clip primitives to these simple window boundaries. Of course, this too affects system performance, since primitives must be processed multiple times, once for each rectangular region.

It is extremely difficult, in general, to provide full performance for all possible window configurations and desired actions. System designers typically begin with a list of desired windowing capabilities and try to find ways of implementing these without incurring exorbitant system costs. Most systems run at full speed only under certain simple situations, and take a big performance hit otherwise. For example, a system may run at full speed only when the rendering window is fully uncovered (or is mostly uncovered, or when its visible portion is a rectangular region), or its rendering speed may suffer when more than one window is being updated simultaneously.

18.11.2 Support for Increased Realism

Our discussion of graphics architecture thus far has primarily concerned the quest for speed. With the continued increase in speed of affordable systems over the past years, users have desired increasingly realistic images as well. Algorithms for generating realistic images are discussed in Chapters 14, 15, 16, and 20; we discuss here the impact realistic rendering has on graphics system architecture.

Antialiasing. As discussed in Chapter 14, the two main approaches to antialiasing use area sampling and point sampling. In area sampling, we need to know the fractional contribution of each primitive to each pixel (ideally, we would like to weight these contributions using an appropriate filter). Unfortunately, calculating the precise fractional contribution is very difficult for 3D images. Architectures that provide hardware support for antialiasing generally use some sort of supersampling scheme. Depending on the characteristics of the rasterization processors, uniform or adaptive supersampling is used.

Uniform supersampling. Uniform supersampling involves calculating the entire image at high resolution and combining multiple sample points at each pixel to determine the pixel's single color value. This appears to be the only feasible approach in SIMD image-parallel systems. Unfortunately, supersampling with n samples per pixel reduces system performance by a factor of n. For this reason, some systems of this type adopt a successive-refinement approach to antialiasing, in which a crude image is generated at full speed when high update rates are desired; when no changes in the image are pending, the system computes further sample points and uses them to refine the image [FUCH85].

Adaptive supersampling. Adaptive supersampling involves calculating the image at high resolution only where necessary. This can be performed most readily on image-order rasterization systems, where information is available on whether or not pixels are partially covered. A well-known adaptive supersampling technique useful for object-order rasterization is the A-buffer (see Section 15.7.3). The linked lists needed to store partially covering primitives are feasible only in MIMD systems that have an arbitrary amount of memory available for each pixel.

Transparency. As mentioned in Chapter 16, true transparency, including refractions and translucency, is difficult to model in software. Even simple models of transparency that exclude refraction and translucency are difficult to implement on many hardware systems. Image-parallel architectures in particular have difficulty with transparency, since the z-buffer algorithm does not handle multiple surfaces well (an extra z value is potentially needed for every transparent surface). Image-parallel systems with SIMD PEs, or systems with fixed amounts of storage per pixel, can generally display a few transparent objects by rasterizing in multiple passes—one pass for all the opaque primitives, and then a separate pass for each transparent surface [MAMM89]. If more than a few transparent surfaces are required, a screen-door transparency scheme generally must be used. MIMD systems may be able to represent pixels as linked lists of visible surfaces, allowing larger numbers of transparent objects to be displayed.

Object-parallel architectures handle transparency more gracefully, since they do not require intermediate storage for the entire image, and have all the potentially visible

polygons in hand when the color for each pixel is computed. The rasterizer, however, must be configured to allow multiple covering polygons and to perform the weighted-sum calculations needed to compute final pixel values.

Textures. Until recently, the only systems that performed texturing at interactive rates were multimillion-dollar flight simulators. Increasingly, however, different forms of texturing are becoming available on graphics workstations and even low-end systems. Textures can be simulated, of course, on any architecture using myriad small triangles with interpolated shading between vertices. This approach, however, places high demands on the display system and is feasible only in high-performance workstations or when simple textures are displayed.

The two traditional methods of texturing are texture maps and procedural textures (see Chapter 16). The drawback of both of these methods is that texturing is applied at the end of the rendering pipeline—after the image has already been converted into pixels. In many architectures, pixel data are not accessible by a general-purpose processor at this point. Flight simulators add an extra hardware stage to perform texture-map lookups. A few graphics workstations, such as the Stellar GS2000 (see Section 18.11.3), store the generated image in main memory, where it is accessible by the system's general-purpose processors.

In architectures in which large-grain MIMD processors have access to the computed image, the texture-map approach is the most appropriate. Texture coordinates, rather than colors, can be stored at each pixel. If each of these MIMD processors is provided with a copy of the texture maps, it can perform the texture-map lookup, replacing the texture coordinates with the appropriate colors. Note that simple texture-map lookup is generally insufficient here—some sort of multiresolution texture map or summed-area table is needed to avoid aliasing.

In architectures in which only fine-grain parallel processors have access to the computed image, procedural textures may be more feasible. Fine-grain processors may not contain enough memory to store the texture maps. If many PEs are available, however, they may be able to compute simple procedural texture models at interactive rates [FUCH89].

Shadows. Computing true shadows, as described in Section 16.4, exacts a high computational price; on many architectures, true shadows cannot be computed at all. For these reasons, few high-performance systems implement true shadows [FUCH85], although a number of techniques can be used to approximate shadows under certain conditions. A common technique used in commercial flight simulators and in a number of workstation systems is to generate shadows for the ground plane only, as described in Section 16.4.

Ray-tracing architectures. Ray tracing, described in Chapters 15 and 16, is a powerful rendering method that can generate extremely realistic images. Unfortunately, it requires a great deal of computation (a typical image can require minutes or hours to compute on a typical workstation). Fortunately, ray-tracing algorithms can be parallelized in several ways, many of which have analogs in conventional rendering:

- *Component parallelism.* Computations for a single ray can be parallelized. For example, reflection, refraction, and intersection calculations all require computing the

x, y, and z components of vectors or points. These three components can be calculated in parallel, resulting in a speedup by a factor of 3.

- *Image parallelism*. Ray–primitive intersections can be calculated in parallel in separate PEs, since the calculations for each ray are independent. To take advantage of this form of parallelism, however, PEs potentially need access to the entire database, since the ray tree for a particular ray may reach any portion of the database.
- *Object parallelism*. Primitives in the database can be distributed spatially over multiple PEs. Each PE, then, is responsible for all rays that pass through its region. It computes ray–object intersections if the ray hits an object, and forwards the ray to the next PE otherwise.

Many architectures for ray tracing have been designed using these techniques alone or in combination. Most use a large number of MIMD PEs, since ray-tracing typically requires a large amount of branching and random addressing.

The simplest type of image-parallel architecture assigns one or more rays to each PE and replicates the entire database at each PE. This technique was used in the LINKS-1 [NISH83a], built at Osaka University in 1983. The LINKS-1 has been used to compute numerous animation sequences. This technique was also used in SIGHT [NARU87], a more recent design done at Nippon Telegraph and Telephone. The SIGHT architecture also takes advantage of component parallelism in its TARAI floating-point unit.

The first proposed object-parallel ray-tracing architectures used uniform spatial subdivision to assign portions of the universe to PEs [CLEA83]. This resulted in poor efficiency for many scenes, since most of the primitives were clustered in a few regions. Other proposed architectures subdivided the scene adaptively to improve the load balance among PEs [DIPP84]. Although this approach improves PE utilization, it makes mapping rays to PEs much more difficult.

Since many ray-tracing architectures closely resemble commercial parallel computers, which offer advantages of lower cost and mature programming environments, research in parallel ray tracing has shifted largely to developing efficient algorithms for commercial multiprocessors, such as hypercubes, Transputer meshes, and the Connection Machine. Two chief concerns are achieving high utilization and balanced load among PEs, particularly when distributed database models are used.

An image-parallel ray-tracing algorithm has been developed for Thinking Machines' SIMD Connection Machine [DELA88], in which the database is repeatedly broadcast to all of the PEs, which perform ray–object intersections in parallel.

Implementations have also been reported on shared-memory multiprocessors, such as the BBN Butterfly [JENK89]. Here, the database does not need to be stored at each PE or broadcast repeatedly; instead, PEs request portions of the database from the memory system as needed. Unfortunately, contention for shared memory resources increases with the number of PEs, so only modest performance increases can be achieved in such systems.

Nemoto and Omachi [NEMO86], Kobayashi and Nakamura [KOBA87], Scherson and Caspary [SCHE88], and others (see [JEVA89]) have proposed various methods for adaptively assigning objects to PEs and for passing rays between those PEs. AT&T's Pixel Machine, a MIMD multiprocessor with PEs based on digital signal-processor chips, ray

traces simple images at interactive rates [POTM89]. Such systems offer a dramatic glimpse of the performance possible in parallel ray tracers of the future.

18.11.3 Stellar's GS2000—A Tightly Integrated Architecture that Facilitates Realistic Rendering

Systems such as Silicon Graphics' POWER IRIS (Section 18.8.2) use dedicated hardware to compute images rapidly. Many of these calculations, particularly in the geometric transformation and other front-end stages, are similar to the types of processing needed in many types of scientific computing. The Stellar GS2000 (similar to its predecessor, the Stellar GS1000 [APGA88]) seeks to make its computing resources available both for image generation and for accelerating other compute-intensive jobs that may run on the machine.

Figure 18.28 shows a block diagram of the GS2000 architecture. The Multi-Stream Processor is a single high-performance processor that simultaneously executes instructions from four independent instruction streams; its peak processing rate is 25 million instructions per second. The vector floating-point processor performs scalar and vector floating-point operations; it can process a maximum of 40 million floating-point operations per second. The rendering processor uses a setup processor to perform polygon-processing calculations and a 4 × 4 SIMD footprint processor to perform pixel operations. The GS2000 renders 150,000 pseudocolored, Gouraud-shaded, z-buffered, 100-pixel triangles per second and 30,000 Phong-shaded polygons per second.

All the GS2000's processing and memory resources are organized around a central 512-bit-wide communication structure called the DataPath. The DataPath has an unusual design in which all of the I/O connections and registers for a single bit of the datapath are built onto a custom gate-array chip (each DataPath chip contains circuitry for 16 bits, so a total of 32 are needed).

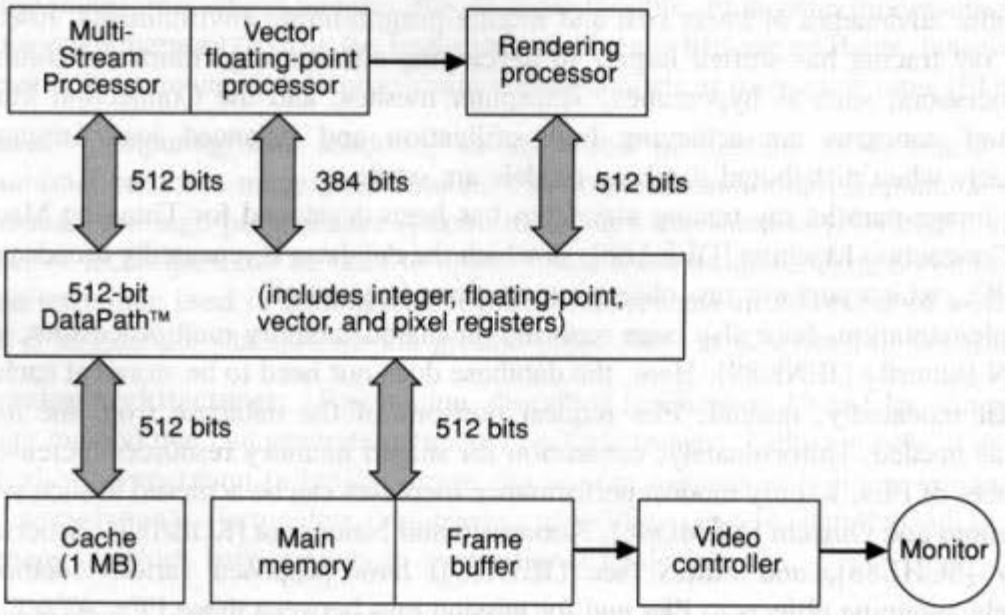


Fig. 18.28 Block diagram of the Stellar GS2000 (based on [APGA88]).

In a typical graphics application, the Multi-Stream Processor traverses a structure database stored in main memory; it then feeds data to the vector processor, which performs geometry calculations. The vector processor sends transformed primitives to the rendering processor, which calculates the pixels of the image. Rather than storing images directly in a frame buffer, the rendering processor stores them in main memory as *virtual pixel maps*, visible portions of which are copied to the appropriate portion of the frame buffer for display.

Because virtual pixel maps are stored in main memory and are accessible to the Multi-Stream Processor and vector processor, a variety of postprocessing operations (as well as image processing and other operations) can be performed. For example, to display textures, the rasterizing unit generates texture indices in pixels, rather than final colors. Later, one of the general-purpose processors passes over the half-generated image and substitutes proper color values for each pixel by table lookup of texture indices. Also, overlapping windows can be implemented easily, since the contents of each screen window are always available in main memory. The main disadvantages of virtual pixel maps are the extra bandwidth and time required to copy image data from main memory to the frame buffer.

18.11.4 Support for Advanced Primitives

We have focused so far on architectures that display polygons rapidly. Increasing demand is being placed on systems to handle other types of primitives. Many complex-surface primitives, such as spline patches and mesh primitives, can be converted into polygons with little time penalty. For other types of primitives, however, such conversions can be either time-consuming or difficult. For example, converting a CSG object to boundary representation is time-consuming, and can result in a large expansion in the amount of data; polygonalizing a volume dataset is slow as well, and tends to obscure data and to produce undesirable artifacts. When more complicated primitives are displayed directly, many of these difficulties can be avoided; in many cases, system performance can be increased as well.

Curved surfaces. As discussed in Chapter 11, polygons are simple, regular primitives that are convenient to display (especially in hardware). Unfortunately, since they are very low-level, they are inconvenient for modeling and lead to inaccuracies, especially for complex, curved surfaces. Other representations for surfaces, such as Bezier patches and NURBS, are more convenient to specify when modeling, but require complicated processing to render directly.

Most hardware systems that display high-order primitives decompose the primitives into polygons for display purposes. For many types of surface primitives (particularly Bezier patches and NURBS), this decomposition can be done rapidly in hardware using forward-difference engines [LIEN87]. In some cases, however, it may be faster to display curved surfaces directly, especially in systems with programmable parallel rasterization processors. When this book was written, few systems provided this capability [MCLE88]. It remains to be seen whether systems of the future will continue to tile such surfaces with polygons, or whether they will render curved surfaces directly.

Volume data. Volume rendering—the direct rendering of data represented as 3D scalar fields (discussed in Section 20.6)—is becoming an important branch of computer graphics. In many ways, it has an even greater need for hardware acceleration than does polygon rendering, since volume datasets are generally much larger than are polygon datasets (a typical dataset from a CT scanner might be $64 \cdot 256 \cdot 256 = 4.2$ million voxels, whereas few polygon datasets contain more than 1 million polygons). Furthermore, voxel calculations are simpler than polygon calculations.

The first architectures designed to accelerate the display of volume datasets classified voxels as either "occupied" or "empty," and displayed occupied voxels. This approach minimizes the amount of processing but obscures data in the interior of the object and produces a "sugar-cube" effect in the generated image. Phoenix Data Systems' Insight system [MEAG85] uses this approach: It interactively displays volume datasets encoded as octrees. Kaufman's Cube architecture [KAUF88b] provides special hardware for accessing in parallel the entire row of voxels corresponding to a single pixel. In addition, this hardware can determine the nearest visible voxel in the row in logarithmic time. Views from arbitrary angles are implemented by rotating the dataset in Cube's 3D memory.

More recent volume-rendering methods assign partial transparencies (or opacities) to each voxel, rather than using binary classification, and attempt to eliminate sampling artifacts (see Section 20.6). Since more voxels contribute to each pixel than when binary classification is used, these algorithms substantially increase image-generation time.

Volume-rendering architectures based on these algorithms generally use variations of the image- or object-parallel approaches we have seen before. Image-parallel architectures assign PEs to pixels or to groups of pixels [LEVO89]. Object-parallel architectures generally partition the voxels into contiguous 3D regions and assign to each a separate PE [WEST89]. During rendering, PEs compute their voxels' contributions to each pixel in the image as though their voxels were the only ones in the dataset. The contributions from multiple regions are then composited using these aggregate colors and opacities.

The primary advantage of the object-parallel approach is that it divides the database among PEs. The disadvantage is that pixel values calculated in one PE may be obscured by pixels calculated in another PE whose voxels lie nearer to the eyepoint. Image parallelism has complementary advantages and disadvantages: Each PE processes only voxels that are potentially visible, but each PE needs access to the entire (very large) database.

Dynamic Digital Displays' parallel Voxel Processor system is an example of an object-parallel architecture [GOLD88]. A hypercube architecture could also be used in an object-parallel approach by assigning each voxel in each image slice (or even in the entire database) to a separate PE. Levoy proposed a hybrid system using Pixel-Planes 5, in which voxel shading is done in an object-parallel fashion and the image is generated in an image-parallel fashion [LEVO89]. The Pixar Image Computer computes red, green, and blue components of an image in parallel [LEVI84]. Because volume rendering is a relatively new area, it is difficult to predict what architectures (or even algorithms) will be dominant in the future.

Constructive-solid-geometry (CSG) architectures. As we saw in Chapter 12, CSG is one of the more popular techniques for modeling solid objects. Chapter 15 described ways to display a CSG object directly from that object's binary-tree representation. The two

popular approaches for direct CSG rendering are generalizations of the image-order and object-order rasterization techniques described earlier in this chapter.

Image-order CSG rasterization. This approach was described in Section 18.9.2. Kedem and Ellis's Ray Casting Machine is a hardware implementation of this algorithm. Their current system displays small CSG objects at interactive rates.

Object-order CSG rasterization. Chapter 15 also described object-order (or depth-buffer) CSG rasterization algorithms that are generalizations of the z-buffer algorithm. CSG depth-buffer algorithms can be implemented on any architecture that has enough memory for each pixel to store two z-buffers, two color buffers, and three 1-bit flags [GOLD86]. Many current high-performance workstations provide frame buffers with sufficient memory. If enough pixel-level processing is available, depth-buffer CSG display algorithms can display modest objects at interactive speeds [GOLD89]. If more memory per pixel is available, more complicated CSG objects can be displayed even faster [JANS87].

18.11.5 Support for Enhanced 3D Perception

Recall from Chapter 14 that 3D images displayed on a 2D screen contain only a few 3D depth cues: obscuration, kinetic depth effect, lighting, and occasionally shadows. The real 3D world provides such additional powerful cues as stereopsis and head-motion parallax. This section discusses architectures and architectural enhancements that seek to provide these extra cues.

Stereo display. Stereopsis can be achieved with a 2D display by computing separate images for the left and right eyes and channeling each image to the respective eye. Image pairs can be displayed on separate monitors (or in separate windows on the same monitor), or left and right images can be displayed in alternating frames. In the former case, image pairs can be viewed with a stereo viewer that optically channels the two images to a fixed point in front of the screen where the viewer must be positioned. Disadvantages of this scheme are that only one person at a time can view the scene, and that only one-half of the monitor's resolution is available (unless two monitors are used, in which case the cost of the system increases dramatically).

Multiplexing left and right images in time is a more popular technique. This requires displaying left and right images alternately in rapid succession, and blocking each eye's view while the other eye's image is being displayed. A graphics system displaying stereo images generally must have a frame buffer large enough to store four complete images—enough to double-buffer the image for each eye. Also, some scheme is needed to block each eye's view of the screen at the appropriate time. One approach uses a mechanical shutter synchronized to the frame buffer so that left and right images are displayed at the correct time [LIPS79]. Unfortunately, mechanical shutters can be heavy, noisy, and, in some cases, dangerous (one design used a rapidly spinning cylinder less than 1 inch from the eye).

A more popular mechanism is an electronic shutter that alternately polarizes light in one direction and then another. The electronic shutter may be the same size as and mounted in front of the display screen, or it can be smaller and worn on special goggles. In either case, lenses polarized in opposite directions are placed before each eye. When the polarization of the electronic shutter corresponds to the polarization of one of these lenses,

the screen becomes visible; when the polarization is in the opposite direction, the view is blocked. Placing the shutter in front of the display screen allows several users to view the image simultaneously, if each wears a pair of inexpensive, passive glasses. Large electronic shutters are expensive, however. Stereo displays with electronic shutters tend to be darker than mechanical ones as well, since the polarizing film transmits only a fraction of the light from the monitor.

Varifocal mirror. A *varifocal mirror* is an unusual display device that uses an oscillating mirror to display true 3D images. These images provide stereopsis and head-motion parallax without requiring the user to wear special headgear. The basic idea is to use a flexible mirror whose focal length can be changed rapidly, and to position it so that it reflects an image of the display monitor to the viewer (Fig. 18.29) [TRAU67; FUCH82; JOHN82]. When the mirror is vibrated with an ordinary loudspeaker, the mirror's focal length changes sinusoidally. This is generally done at a frequency of approximately 30Hz.

The mirror's periodically changing focal length makes the distance to the monitor appear to increase and decrease by several inches or more during each 30-Hz cycle. Points, lines, and similar data are displayed on a point-plotting (not raster-scan) display monitor. The perceived depth of a point is determined by its position in the display list: "near"

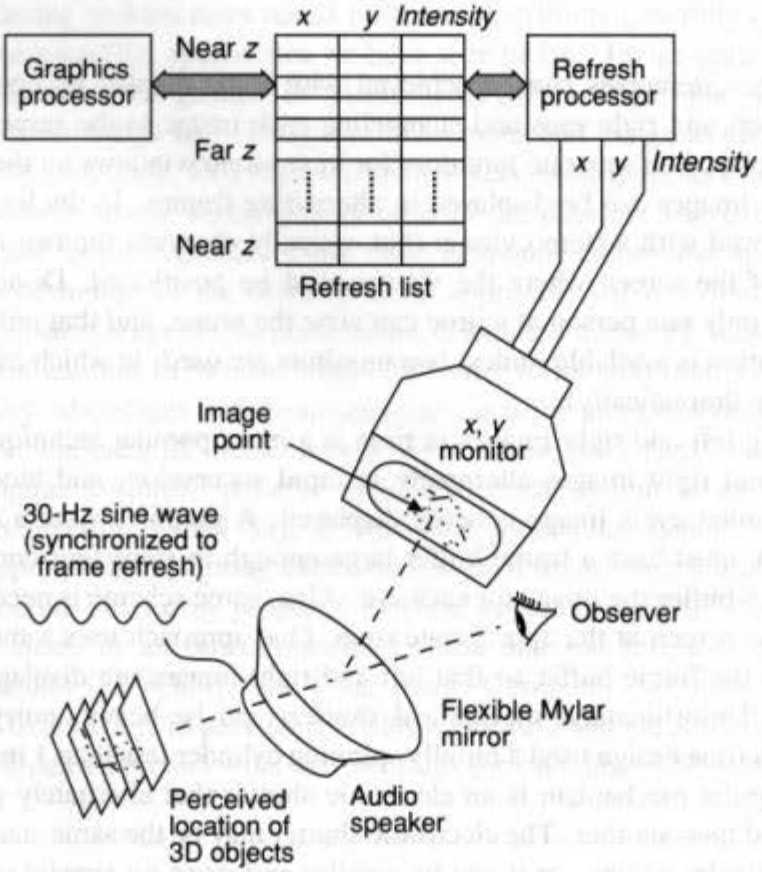


Fig. 18.29 Varifocal-mirror display system.

points are stored in the beginning or end of the refresh list; "far" points are stored in the middle of the refresh list. Note that there are two places in the list at which a point appears at the same depth—when the mirror is moving forward, and when it is moving backward.

Varifocal-mirror displays have several limitations. One drawback is that nearer objects do not obscure more distant ones, and thus only nonobscuring primitives such as points, lines, and transparent volume data can be displayed. A second difficulty is that only a limited amount of data can be displayed at a time—the amount that can be refreshed during a single mirror cycle. In spite of these limitations, the varifocal mirror is one of the very few true 3D display devices to be made into a commercial product, the Genisco SpaceGraph [STOV82].

Head-mounted display. In a seminal address at the 1965 International Federation for Information Processing Congress [SUTH65], Ivan Sutherland proposed that the *ultimate display* would be one that produced images and other sensory input with such fidelity that the observer could not tell the simulated objects from real ones. In 1968, he showed a prototype display that was worn on the head and that demonstrated the most important property of the *ultimate display*—it allowed the user to walk around in a *virtual world*. Specifically, the system comprised the following:

1. Headgear with two small display devices, each optically channeled to one eye
2. A tracking system that allowed the computer system to know the precise location of the user's helmet (and thus head) at all times
3. A hand-held wand, whose position was also tracked by the system, that allowed the user to reach out to grab and move objects in the virtual environment
4. A real-time graphics display system that constantly regenerated the images to the display devices as the user moved, giving the user the illusion of walking around "virtual" objects in the room.

This system, with its rich 3D cues of head-motion parallax and stereopsis and its simple, direct 3D manipulation of objects inside the virtual world, convincingly demonstrated to many people that the conventional way of interacting with 3D scenes using a desktop CRT is an unsatisfying, constrained mechanism that keeps the user outside the CRT's window to the 3D virtual world. Unfortunately, several technical problems have prevented the head-mounted display from reaching full effectiveness—indeed, they have kept it to a level discouragingly close to the capabilities of Sutherland's 1968 prototype [FISH86; CHUN89]. These technical problems include:

1. Developing headgear with high-resolution displays that allows a wide-screen view of the graphics display screen superimposed on the real world (a difficult optical problem)
2. Developing a tracking system for the helmet and hand that has the range of a room with 1-millimeter resolution and response time of a few milliseconds or less
3. Designing a graphics system that generates at least 30 frames per second with minimal latency (low latency is a particularly important concern here, since latency in a head-mounted display can induce motion sickness [DEYO89]).

The most visible and widespread use of head-mounted displays has been for heads-up displays in cockpits of military aircraft. Heads-up displays, however, give only auxiliary information to the user, rather than creating a virtual world, complete with objects that can be directly manipulated. In 1989, two companies, Autodesk and VPL Research, introduced commercial head-mounted display systems based on off-the-shelf technology. Proponents of such systems predict that these kinds of systems will be the twenty-first century graphics equivalents of Sony Walkman personal stereos, to be used not only for real-time 3D applications but also for general portable interactive computing.

Digital holography.[4] Holography is another method for displaying true 3D images without using special headgear or tracking the viewer's location. Traditional holograms are produced by exposing photographic film simultaneously to laser light scattered from the object to be recorded and to a reference beam from the same laser. The interference patterns recorded on the film encode the object's appearance from a range of viewpoints. The hologram is viewed by illuminating it with laser light from the opposite direction.

Holograms of imaginary objects can be produced by simulating the laser-interference process on a computer and writing the computed fringes onto high-resolution film. Unfortunately, holograms produced by this technique can require 10^{12} Fourier transforms (each of which requires a large number of multiplications and additions) and need a VLSI-type electron-beam writer to inscribe the results on film [DALL80; TRIC87]. For these reasons, this technique is currently too expensive for almost all applications. Fully computed holograms, however, contain more information than is needed by the human eye. A promising approach is to reduce these calculations to the number actually needed. This is the subject of ongoing research.

Holographic stereograms, which are built up from a sequence of computer-generated perspective views, are attractive in the meantime. In a holographic stereogram, a sequence of approximately 100 views from slightly differing side-to-side viewpoints is projected with laser light onto holographic film, each from the direction from which it was calculated. A second reference beam from the same laser overlaps the projection beam to record the view direction in an interference pattern. After exposure and processing, the film (now the holographic stereogram) is illuminated by a reference beam in the opposite direction. Image beams then diffract back in the directions from which they were projected. An eye moving from one view to the next perceives a smooth progression of perspective information that yields an impression of a solid object or 3D scene floating in the vicinity of the film. Combined with all the conventional monocular depth cues of 3D computer graphics, this holographic image gives a particularly effective sense of shape and space [BENT82].

Although holographic stereograms are much less expensive to produce than are traditional holograms, they still require a large amount of computation (dozens to hundreds of images for a single stereogram). Furthermore, the need to record the stereogram on photographic film adds expense and time to the image-generation process. Consequently, digital holography is unlikely to yield interactive 3D images in the near future, although it

[4]Material for this section was contributed by Stephen A. Benton of the MIT Media Laboratory.

may prove useful for recording 3D still images, just as photographic film records 2D still images today.

18.11.6 Real-Time Flight Simulators

The systems that "put it all together" to generate the most realistic simulation of 3D scenes for interactive tasks are the multimillion-dollar flight simulators. Flight simulation is not unique in being able to benefit from truly real-time systems. It is, however, the one application for which customers have consistently been willing to spend millions of dollars for a single system, largely because of the cost and danger of training pilots solely in actual airplanes. Because the community of flight-simulator users is fairly small, and because the systems tend to use proprietary, hardware-specific software provided by the manufacturer, detailed information about flight-simulator architectures has not appeared in the literature.

Early flight simulators include General Electric's NASA II (see Section 18.9) and Evans & Sutherland designs based on the scan-line systems developed at the University of Utah in the late 1960s. Some of these early flight-simulator systems could display 1000 or more polygons in real time, but all used simple shading methods and provided few image enhancements. Later systems have not substantially increased the number of primitives that can be displayed. For example, Evans & Sutherland's current high-end system, the ESIG-1000, displays only 2300 polygons at 60 Hz [EVAN89]. Rather, system developers have increased scene realism and reduced distracting artifacts by incorporating features such as antialiasing, haze and fog, point light sources, clouds, and filtered textures [SCHA83]. The effectiveness of these techniques can be seen in Color Plates I.5(a) and I.5(b).

Flight simulators from major manufacturers such as Evans & Sutherland, General Electric, McDonnell-Douglas, and Singer/Link all share several architectural themes: Since flight simulation involves predictable interactions with very large datasets, these systems tend to use more specialized processing than do other graphics systems. For example, custom processors are frequently built to manage the image database, to transform primitives, to rasterize the image, and to perform image-enhancement operations afterward. A typical simulator system is composed of a long pipeline of proprietary processors [SCHA83].

Certain simplifications can sometimes be made in a flight simulator that are not possible in more general graphics systems. For example, since a typical simulator dataset involves a small number of moving objects and a wide, unchanging backdrop, the generality of the z-buffer visibility algorithm may not be needed and a simpler depth-sort algorithm may suffice.

Flight simulators also must manage complex databases without hesitation. A typical database may represent a region 100 miles square. Detail needed for low-level flight cannot be displayed when the airplane is at 40,000 feet. This requires the system to maintain object descriptions with different levels of detail that can be swapped in and out in real time. The architecture also must handle overloading gracefully, since image complexity may increase drastically at the most crucial times, such as during takeoffs, during landings, and in emergency situations [SCHU80]. Frames must be generated at least 30 times per second—even in these situations.

18.12 SUMMARY

This chapter has provided an overview of the architectural techniques used to build high-performance graphics systems. We have seen that the computational demands of many interactive applications quickly surpass the capabilities of a single processor, and that concurrent processing of various kinds is needed to meet the performance goals of demanding 3D applications, such as computer-aided design, scientific visualization, and flight-simulation.

We have shown how the two basic approaches to concurrency—pipelining and parallelism—can be applied to accelerate each stage of the display process, together with the advantages and limitations of each of these choices. The design of any real graphics system (indeed, any complex system in general) represents a myriad of compromises and tradeoffs between interrelated factors, such as performance, generality, efficiency, and cost. As a result, many real systems use combinations of architectural techniques we have described.

The current state of hardware technology also plays an important role in deciding what architectural techniques are feasible. For example, the memory-intensive architectures that now dominate the field would have been impractical just ten years ago before inexpensive DRAMs became available. In the future, we can expect rapidly improving technology to continue to improve graphics system performance.

The designs of future systems will be complicated by demands not just for rendering standard primitives such as points, lines, and Gouraud-shaded polygons, but also for rendering with more advanced capabilities—transparency, textures, global illumination, and volume data. These all complicate not only the calculations, but also the basic structure of the display process, making it more difficult to design systems with both high performance for the basic capabilities and sufficient generality to handle the advanced features.

EXERCISES

18.1 Section 5.6 showed that we can transform a plane equation by multiplying it by the transpose of the inverse point-transformation matrix. A surface-normal vector can be considered to be a plane equation in which the D component does not matter. How many multiplications and additions are needed to transform a surface-normal vector if the point-transformation matrix is composed of translations, rotations, and scales? (Hint: Consider the form of transformation matrix.)

18.2 A simple way to reduce the number of front-end calculations when displaying polygonal meshes is to use a mesh primitive, such as a triangle strip. A *triangle strip* is a sequence of three or more vertices, in which every consecutive set of three vertices defines a triangle. A triangle strip of $n + 2$ vertices, therefore, defines a connected strip of n triangles (whereas $3n$ vertices are needed to define n individual triangles). Estimate the number of additions/subtractions and multiplications/divisions required to display the sample database of Section 18.3.9 if the 10,000 triangles are contained in:

a. 5000 triangle strips, each containing 2 triangles
b. 1000 triangle strips, each containing 10 triangles
c. A single triangle strip containing 10,000 triangles.

What is the maximum speedup you could obtain in the front-end subsystem by converting a database of discrete triangles into triangle strips? Are there any disadvantages to using triangle strips?

18.3 Assume that the 10,000-triangle database described in Section 18.3.9 is displayed at 24 Hz on a pipelined graphics system with the following characteristics: 1280 by 1024 color display refreshed at 72 Hz, 32-bit color values, and 32-bit z values.

a. Estimate the data bandwidth between the following points of the display pipeline: (1) between display-traversal and modeling-transformation stages (assume that 24 32-bit words of data are required for each triangle in the object database); (2) between front-end and back-end subsystems (assume that 15 32-bit words are required for each transformed triangle); (3) between rasterizer and frame buffer; and (4) between frame buffer and video controller.

b. Repeat the calculations in part (a) for a database with 100,000 polygons with an average area of 10 pixels and the same overlap factor.

c. Repeat the calculations in part (a) for a database with 100,000 polygons with an average area of 100 pixels, but assume that only 10 percent of the pixels are initially visible.

18.4 Consider the pipelined object-order rasterization architecture described in Section 18.7.1. If separate processors are provided for polygon processing, edge processing, and span processing, all these operations can be overlapped. Assume that we have a sophisticated span processor that can process an entire pixel (i.e. compute its RGB and z values, compare z values, and update the frame buffer) in a single 250-nanosecond clock cycle. Ignoring the time required for clearing the screen between frames, calculate how many triangles per second this system can display under the following conditions:

a. 100-pixel triangles; negligible time for polygon and edge processing

b. 100-pixel triangles; 20 microseconds per triangle for polygon processing; negligible time for edge processing

c. 100-pixel triangles; 20 microseconds per triangle for polygon processing; 2 microseconds per scan line for edge processing (assume that a typical triangle covers 15 scan lines)

d. 10-pixel triangles; 20 microseconds per triangle for polygon processing; 2 microseconds per scan line for edge processing (assume that a typical triangle covers four scan lines)

e. 1000-pixel triangles; 20 microseconds per triangle for polygon processing; 2 microseconds per scan line for edge processing (assume that a typical triangle covers 50 scan lines).

18.5 A frame buffer is to be built with 32 bits per pixel, and access to the frame buffer is to be by single pixels (a 32-bit-wide memory system). What frame-buffer sizes with aspect ratios 1:1, 5:4, and 2:1 (up to a maximum dimension of 2048 pixels) are possible if the following commercial memory parts are used and no memory is to be wasted (i.e., all memory in the frame buffer should be used to store visible pixels):

a. 64K × 4 VRAMs (256 Kbit)

b. 256K × 4 VRAMs (1 Mbit)

c. 512K × 8 VRAMs (4 Mbit).

Answer the following questions assuming frame buffers of sizes 512 by 512, 1024 by 1024, and 2048 by 2048 (each refreshed at 60 Hz):

d. How frequently must the serial port of each memory chip be accessed during video scanout? (Assume that vertical and horizontal retrace times are neglible, and that VRAM outputs are not multiplexed.)

e. Given that the serial-port cycle time of the fastest VRAMs is about 35 nanoseconds, which of these frame buffers could be built? (Again, assume no multiplexing.)

f. Which frame buffers could be built if multiple pixels were read simultaneously and multiplexed as described in Section 18.1.5 (again assuming a VRAM cycle time of 35 nanoseconds)? How many pixels would have to be read at once for each of these frame buffers?

18.6 Consider the pipelined, object-order rasterization architecture described in Section 18.7.1.

a. Determine to what accuracy screen-space (x, y) vertex coordinates must be calculated (i.e., how many bits of precision are needed) if vertices are to be specified to within $\frac{1}{10}$ pixel of their true position in the following displays: a 320 by 200 PC display; a 1280 by 1024 workstation display; and a 1840 by 1035 high-definition TV display.

b. How many fractional bits are needed in left and right x slopes (calculated during polygon processing) if left and right span endpoints are to lie within $\frac{1}{10}$ pixel of their true position in all polygons displayable in each of the systems in part (a)? (Assume that vertex coordinates have been calculated with infinite precision and that additions are performed with perfect accuracy.)

c. What are the maximum and minimum possible values for Δx in each of the systems of part (a)? (Assume that horizontal edges have been recognized and removed before delta values are calculated.)

d. If fixed-point arithmetic is used, how many bits are needed to represent Δx values that can range from the minimum to maximum values calculated in part (c) and the precision calculated in part (b), for the three systems?

18.7 The performance of image-parallel architectures that partition the frame buffer into contiguous blocks of pixels is reduced if many primitives fall into more than one region. Assume that the display screen is divided into a number of regions of width W and height H and that a typical primitive covers a rectangular area of width w ($w \ll W$) and height h ($h \ll H$) on the display screen. Derive an expression in terms of W, H, w, and h for the average number of regions affected by a typical primitive, assuming that the primitive has an equal probability of appearing anywhere on the screen.

19
Advanced Geometric and Raster Algorithms

In Chapter 3, we described a number of methods for clipping and scan converting primitives. In this chapter, we begin by discussing more advanced clipping techniques. These are purely geometric techniques, to be applied to geometric primitives being clipped to geometrically defined regions.

Following these clipping algorithms, we reconsider the description and scan conversion of the primitives discussed in Chapter 3, beginning by analyzing the attributes associated with primitives. This analysis is necessary because attributes such as line style have arisen from diverse demands on raster graphics packages; some line styles, such as dotted lines, are cosmetic, some, such as the dot-dash lines used in mechanical drawings, are geometric. It will be important to understand the differences.

We next consider the criteria for selecting pixels in the scan-conversion process; different criteria sometimes lead to different choices. Then, after doing some analytic geometry, we give algorithms for noninteger lines, noninteger circles, and general ellipses, and discuss the perils of representing arbitrary curves as short line segments in an integer world. We then discuss antialiasing, including its application to rendering thick lines, polylines, and general curves. After this, we analyze the problems associated with drawing text both in bitmap graphics and with gray-scale antialiasing, and we examine some solutions. We also discuss a data structure that can be used to speed the manipulation of scan-converted primitives, especially for bilevel displays, and some techniques for making a fast copyPixel operation for bilevel displays (bitBlt). We conclude the chapter with three further topics: the management of overlapping windows, fill algorithms, and 2D page-description graphics. One example is Interpress [HARR88]; another is POSTSCRIPT

[ADOB85b], which is really more than a page-description model—it also offers the full functionality of a relatively complex programming language, so complex images may be described compactly through the use of notions of iteration and procedural definitions of image elements. Such page-description languages are now being used to provide not merely static image descriptions, but also screen descriptions for interactive graphics.

19.1 CLIPPING

Before giving details of specific clipping algorithms, we first discuss the general process of clipping, and then its specialized application to lines. As described in Chapter 2, *clipping* is the process of determining the portion of a primitive lying within a region called the *clip region*. The clip region is typically either a window on a screen or a view volume. The second case is handled by the Sutherland–Hodgman algorithm described in Chapter 3, so we concentrate on the first here. In the discussion in Chapter 3, the clip region was always a rectangle; because primitives are typically drawn on a rectangular canvas, this is an important special case. In multiple-window environments, such as the Macintosh operating system or the X Windows System, various rectangular windows overlap one another, and the clip region can be an arbitrary set of polygons with only horizontal and vertical edges. In systems such as POSTSCRIPT, a clipping region can be defined by an arbitrary set of outlines in the plane. Furthermore, the primitives being clipped may be 1D (e.g., lines) or 2D (e.g., filled polygons).[1] It is easy to think that once line clipping is solved, so is polygon clipping: Just clip all the edges of the polygon to the window and draw. This assumption fails, however, if the polygon completely encloses the clip window. Clipping 2D primitives is a more difficult problem than is clipping 1D primitives.

One method for drawing clipped primitives deserves discussion, although it is not actually a clipping method per se. It consists of computing the points that would be drawn on an infinite canvas, and then drawing only those points that actually lie within the clip region. This method (called *scissoring* in Chapter 3) has one drawback: the cost of rendering all of a primitive lying substantially outside the clip region. This drawback is offset, however, by scissoring's simplicity and generality, so the technique turns out to be a reasonable approach for many systems. If the time taken to draw a pixel in the frame buffer is long compared to the computation time for each pixel, then, while visible pixels are queued up to be drawn, other invisible ones may be computed. It is also simple to address clipping to multiple windows through this method: An application can maintain a notion of the current window, and the drawing algorithm can simply draw up to the border of this window, then pass control back to the application, which then passes the next window to the drawing algorithm. There are many applications, however, in which clipping to a single clip region is essential. We therefore discuss methods that take into account the geometry of the clip region before scan conversion.

[1] References to 1D and 2D primitives refer to the intrinsic geometry of the primitive: Position on a line can be specified with a single number; hence, it is said to be 1D. Position on a surface can be specified by two numbers; it is called 2D. Thus, even a helical curve in 3D is a 1D primitive.

19.1.1 Clipping Lines to Rectangular Regions

Clipping lines to upright rectangular regions is a purely geometric problem, in that it is completely independent of the size or even the existence of the pixels; it involves computing intersections of lines and rectangles in the Euclidean plane. In Chapter 3, we discussed the Cyrus–Beck/Liang–Barsky line-clipping algorithm. Nicholl, Lee, and Nicholl have created a better line clipper for this 2D case [NICH87]. Although the algorithm has a great many cases, the basic idea is simple enough that understanding one case lets us generate all the others.

Before discussing this algorithm, let us restate the problem: Given a collection of (zero-width) line segments and an upright clipping rectangle, find the endpoints of the (possibly empty) intersections of the line segments and the rectangle. Each line segment is given as a pair of endpoints, and the upright clipping rectangle is given by four equations: $x = x_{min}$, $x = x_{max}$, $y = y_{min}$, and $y = y_{max}$ (see Fig. 19.1). For convenience, we assume for the time being that the line segment to be clipped is neither vertical nor horizontal.

The most simple-minded algorithm computes the equation of the line containing the line segment, then computes all intersection points of this line with the clip-rectangle boundary lines. Except for degenerate cases, either zero or two of these points lie within the clip rectangle (see Fig. 19.2); if two, they are compared with the endpoints of the original segment to give the clipped segment. Of course, this comparison requires the computation of four intersections, even if the line segment is entirely within (or entirely outside) the clip region. Recall from Chapter 3 that the parametric algorithm instead computes the parameter values for the intersections of the line with the boundaries of the clip rectangle and compares these with the parameter values 0 and 1 to determine whether the segment lies within the clip rectangle, (the line is parameterized so that parameter values between 0 and 1 correspond to the segment). Only when the parameter values of intersections with the clip rectangle are computed does the algorithm go on to compute the intersection points.

The Nicholl–Lee–Nicholl (NLN) algorithm is based on an improvement of this simple delaying tactic. Consider a segment PQ that is to be clipped. We first determine where P lies. If we divide the plane into the same nine regions used in the parametric clipping algorithm (see Fig. 19.3), then P must lie in one of these regions (each boundary line is assigned to one of the regions it touches). By determining the position of Q relative to the

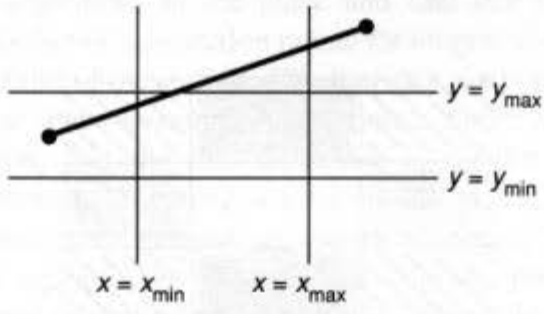


Fig. 19.1 The equations defining the clipping rectangle, and a typical line segment to be clipped.

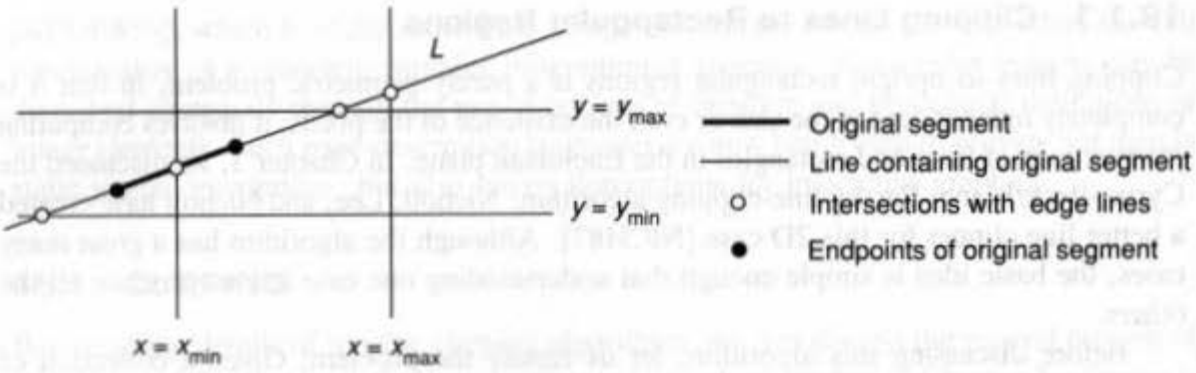


Fig. 19.2 Clipping by finding intersections with all clip-rectangle boundary lines.

lines from P to each of the corners, we can determine which edges of the clip rectangle PQ intersects.

Suppose that P lies in the lower-left–corner region, as in Fig. 19.4. If Q lies below y_{min} or to the left of x_{min}, then PQ cannot intersect the clip region (this amounts to checking the Cohen–Sutherland outcodes). The same is true if Q lies to the left of the line from P to the upper-left corner or if Q lies to the right of the line from P to the lower-right corner. Many cases can be trivially rejected by these checks. We also check the position of Q relative to the ray from P through the lower-left corner. We will discuss the case where Q is above this ray, as shown in Fig. 19.4. If Q is below the top of the clip region, it is either in the clip region or to the right of it; hence the line PQ intersects the clip region either at its left edge or at both the left and right edges. If Q is above the top of the clip region, it may be to the left of the ray from P through the top-left corner. If not, it may be to the right of the right edge of the clip region. This latter case divides into the two cases: Q is to the left of the line from P to the upper-right corner and to the right of it. The regions in Fig. 19.4 are labeled with the edges cut by a segment from P to any point in those regions. The regions are labeled by abbreviations; LT, for example, means "the ray from P to any point in this region intersects both the left and top sides of the clipping rectangle."

Assuming that we have a function LeftSide (*point*, *line*) for detecting when a point is to the left of a ray, and a function Intersect (*segment*, *line*) that returns the intersection of a

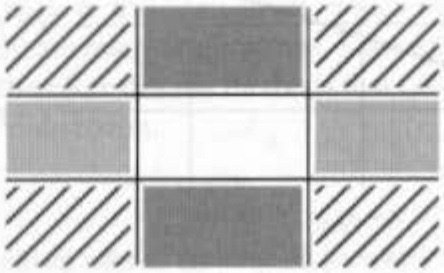


Fig. 19.3 The nine regions of the plane used in the Nicholl–Lee–Nicholl algorithm.

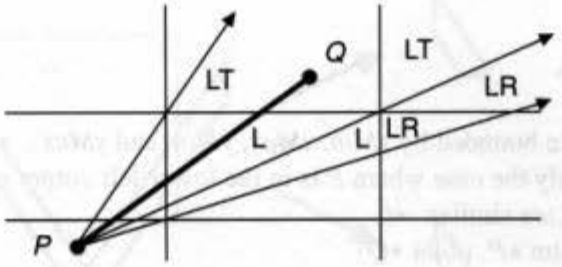


Fig. 19.4 The regions determined by the lines from P to the corners.

segment and a line, the structure of the algorithm for these cases is shown in Fig. 19.5. P and Q are records of type "point" and have x and y fields.

The interesting thing about this computation is the possibility of reusing intermediate results. For example, in computing whether Q is to the left of the ray from P to the upper-right corner (x_{max}, y_{max}), we must check whether

$$(Q.y - P.y)(x_{max} - P.x) - (y_{max} - P.y)(Q.x - P.x)$$

is positive (see Exercise 19.25). Similar computations are used for the other edges, and the numbers $Q.y - P.y$ and $Q.x - P.x$ appear in all of them, so these are kept once computed. The two products in this formula are also reused, so they too are recorded. For example, if Q is to the right of the line, when we compute the intersection of PQ with the right edge, the y coordinate is given by

$$P.y + (Q.y - P.y)(x_{max} - P.x)(1 / (Q.x - P.x)),$$

and the first product can be reused (the formula is just an application of the point-slope formula for a line). Since the reciprocal of $Q.x - P.x$ occurs in computing the intersection with the right edge, it too is stored. We leave it to you to make this particular code fragment as efficient as possible (Exercise 19.1). The remaining cases, where P is in the center region or in one of the side regions, are similar. Thus, it is worthwhile to recognize the symmetries of the various cases and to write a program to transform three general cases (P in the center, in a corner, or in an edge region) into the nine different cases.

Nicholl, Lee, and Nicholl present an analysis of the NLN, Cohen–Sutherland (CS), and Liang–Barsky(LB) algorithms in the plane and find that (1) NLN has the fewest divisions, equal to the number of intersection points for output, and (2) NLN has the fewest comparisons, about one-third of those of the CS algorithm, and one-half of those of the LB algorithm. They also note that—assuming subtraction is slower than addition, division is slower than multiplication, and the first difference is smaller than the second—their algorithm is the most efficient. Of course, unlike the others, NLN works only in 2D.

Clipping lines against more general regions is a special case of clipping generic primitives against such regions. We next discuss clipping polygons against arbitrary polygons. Clipping general primitives to the arbitrary regions defined in POSTSCRIPT and some other imaging models is described later in the chapter, since this is implemented by raster algorithms.

```
/* Clip PQ to a rectangle bounded by xMin, xMax, yMin, and yMax . */
/* This code handles only the case where P is in the lower-left corner of the */
/* region—other cases are similar. */
void PartialNLNclip(point *P, point *Q)
{
    boolean visible;        /* TRUE if clipped segment is nonempty */

    if (Q->y < yMin)
        visible = FALSE;
    else if (Q->x < xMin)
        visible = FALSE;
    else if (LeftSide (Q, ray from P to lower-left corner)) {
        if (Q->y <= yMax) {                                    /* Region L or LR */
            visible = TRUE;
            *P = Intersection (PQ, left edge of clip region);  /* Stores intersection in P */
            if (Q->x > xMax)                                   /* Region LR */
                *Q = Intersection (PQ, right edge of clip region);
        } else {
            /* Above top */
            if (LeftSide (Q, ray from P to upper-left corner))
                visible = FALSE;
            else if (Q->x < xMax) {                            /* First region LT */
                visible = TRUE;
                *P = Intersection (PQ, left edge of clip region);
                *Q = Intersection (PQ, top edge of clip region);
            } else if (LeftSide (Q, ray from P to upper-right corner)) {
                visible = TRUE;                                /* Region LT */
                *P = Intersection (PQ, left edge of clip region);
                *Q = Intersection (PQ, top edge of clip region);
            } else {                                           /* Region LR */
                visible = TRUE;
                *P = Intersection (PQ, left edge of clip region);
                *Q = Intersection (PQ, right edge of clip region);
            }
        }  /* else */
    } else
        /* Cases where Q is to the right of line from P to lower-left corner */
        ...
}  /* PartialNLNclip */
```

Fig. 19.5 Part of the Nicholl–Lee–Nicholl algorithm.

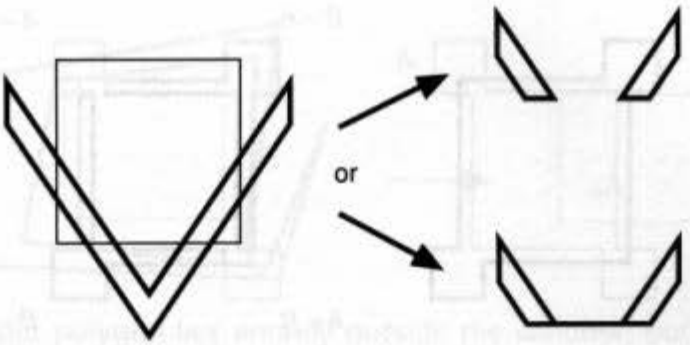


Fig. 19.6 Should the clipped V-shaped polygon contain the degenerate edge?

19.1.2 Clipping Polygons against Rectangles and other Polygons

In drawing a polygon in a rectangular region, we may wish to clip it to the region to save drawing time. For truly general clipping regions (where the interior of the region is specified, for example, by giving a canvas corresponding to a region on the screen, and for which the interior of the clip region consists of those points corresponding to black pixels in the canvas), scissoring is quite practical: We simply compute all the (rasterized) points of the primitive, then draw only those lying within the rasterized clip region. The shape algebra described in Section 19.7 can be used as well, to determine rapidly the regions of overlap between the pixels of the clipping shape and those of the primitive; this is the most efficient technique until the clip regions become extremely complex—for example, a gray-scale bitmap. Since the analytic algorithms for clipping are interesting in their own right, and have other applications beyond windowing systems (e.g., the visible-surface algorithm presented in Chapter 15), we cover two algorithms in detail: the Liang–Barsky (LB) [LIAN83] and Weiler [WEIL80] algorithms.

There is some difference of opinion on what constitutes the clipped version of a polygon. Figure 19.6 shows a polygon being clipped against a rectangular window and the two possible outputs, one connected and one disconnected. The Sutherland–Hodgman [SUTH74b] and Liang–Barsky algorithms both generate connected clipped polygons, although the polygons may have degenerate edges (i.e., edges that overlap other edges of the polygon, or whose length is zero). The Weiler algorithm produces nondegenerate polygons, which are therefore sometimes disconnected. Since the Weiler algorithm is designed to do somewhat more than clipping—it can produce arbitrary Boolean combinations of polygons—it is clear that these combinations may need to be represented by disconnected polygons if they are to be truly disjoint.[2] Figure 19.7 shows two polygons such that A − B and B − A cannot be simultaneously made into connected polygons (by adding just a single degenerate edge pair to each) without an intersection being introduced between them.

In practice, the degenerate edges in the output polygon may be irrelevant. If the polygon is used merely to define a filled area, then the degenerate edges have no area

[2]The Weiler algorithm also handles general polygons—that is, polygons with holes in them. These are described by multiple nested contours.

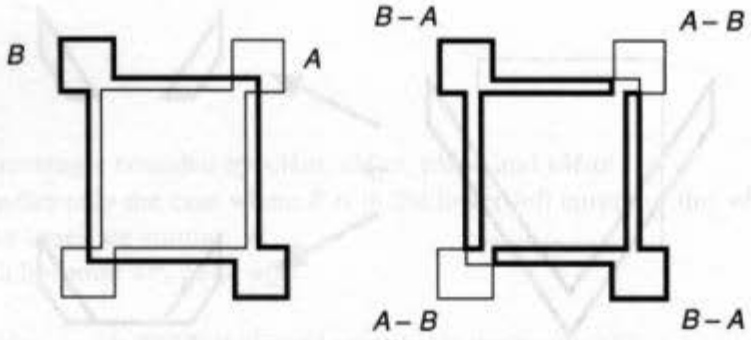


Fig. 19.7 The regions $A - B$ and $B - A$ cannot be made into connected polygons (using edges of the original polygons) without intersecting at least at a point.

between them, and hence cause no problems. If the polygon is used to define a polyline, however, the degenerate edges must be removed; see Exercise 19.2.

19.1.3 Clipping against Rectangles: The Liang–Barsky Polygon Algorithm

Let us begin with the Liang–Barsky (LB) algorithm. To distinguish between the polygon to be clipped and the rectangle against which the polygon is clipped, we call the upright rectangle the *window* and the polygon to be clipped the *input polygon*; the result of clipping is the *output polygon*. Each edge to be clipped is represented parametrically, and the intersections with the window edges are computed only when needed. In contrast to line clipping, however, an edge entirely outside the window can contribute to the output polygon. Consider the case where the input polygon entirely encloses the window: The output polygon is the boundary of the window, as shown in Fig. 19.8.

The mathematically inclined reader may object to the result in the case shown in Fig. 19.9: The interior of the input polygon misses the window entirely, but the LB algorithm produces a polygon that includes all edges of the window (although it includes each one once in each direction, as shown by the dotted lines; these are drawn slightly outside the window so as to be visible). Exercise 19.2 discusses algorithms for removing such excess edges to get a minimal form for a clipped polygon.

We assume the input polygon is given as a sequence of points $P_1, P_2, \ldots, P_n$, where the edges of the polygon are $P_1P_2, P_2P_3, \ldots, P_nP_1$. Each edge can be considered as a vector starting from P_i and going toward P_{i+1}, and this determines the parametric form $P(t) = (1 - t)P_1 + tP_2$. Values of t between 0 and 1 represent points on the edge. (To be more

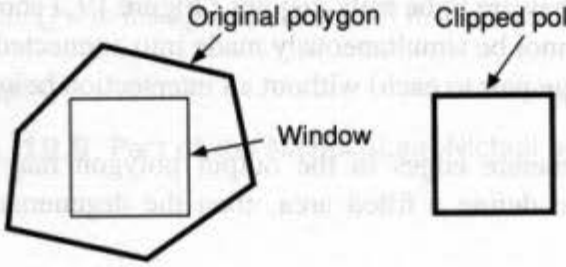


Fig. 19.8 An edge outside the window can contribute to the output (clipped) polygon.

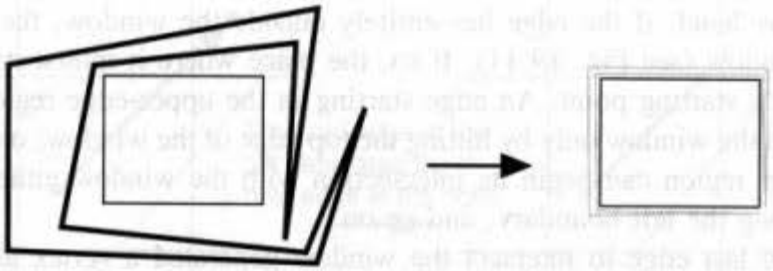


Fig. 19.9 The input polygon lies entirely outside the window, but the Liang–Barsky algorithm produces a nonempty result.

precise, we let values $0 < t \leq 1$ represent points on the edge, so that each edge fails to contain its starting point. Each vertex of the polygon is therefore contained in exactly one of the two edges that meet there. The choice to omit the starting point differs from that in Chapter 3, but it makes the explanation of the algorithm slightly simpler.) Other values of t represent points that are on the line containing the edge, but not on the edge itself. In the LB algorithm, we will consider one edge, P_iP_{i+1}, at a time, and let L_i denote the line containing P_iP_{i+1}.

We initially consider only diagonal lines—those that are neither horizontal nor vertical. Such a line must cross each of the lines that determine the boundary of the window. In fact, if we divide the plane into the nine regions determined by the edges of the windows, as in Fig. 19.10, it is clear that every diagonal line passes from one corner region to the opposite one. Each window edge divides the plane in two halfplanes. We call the one containing the window the *inside* halfplane. The nine regions in Fig. 19.10 are labeled by the number of inside halfplanes they lie in. The window is the only region lying in all four, of course. We call the regions at the corners (labeled "inside 2") *corner regions*, and the other outer regions (labeled "inside 3") *edge regions*.

Before we discuss details, we show the use of this algorithm by a few examples. First, if some portion of the edge (not just the line containing it) lies in the window, that portion must be part of the output polygon. The vertices this edge adds to the output polygon may be either the ends of the edge (if it lies entirely within the window) or the intersections of the edge with the window edges (if the endpoints of the edge lie outside the window), or there may be one of each.

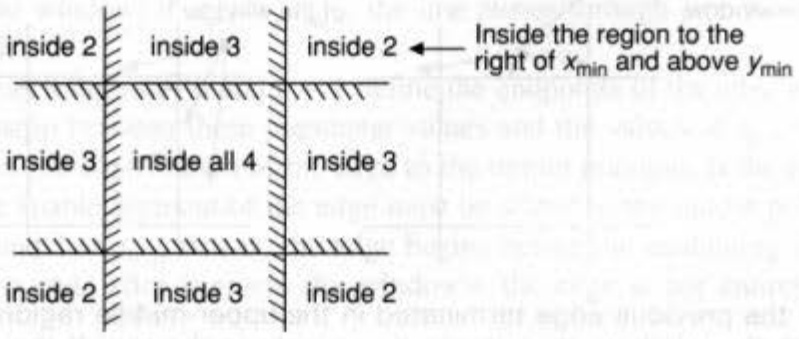


Fig. 19.10 The plane is divided into nine regions by the extended edges of the window. Each region is on the "inside" side of at least two edges.

On the other hand, if the edge lies entirely outside the window, the next edge may intersect the window (see Fig. 19.11). If so, the place where it intersects the window is determined by its starting point: An edge starting in the upper-edge region can begin its intersection with the window only by hitting the top edge of the window; one starting in the upper-left corner region can begin its intersection with the window either along the top boundary or along the left boundary, and so on.

Suppose the last edge to intersect the window generated a vertex at the top of the window, and the next edge to intersect the window will do so on the right edge of the window, as in Fig. 19.12. The output polygon will then have to contain the upper-right corner of the window as a vertex. Since we are processing the polygon one edge at a time, we will have to add this vertex now, in anticipation of the next intersection with the clip window. Of course if the next edge intersected the top edge of the window, this vertex would be redundant; we add it regardless, and handle the removal of redundant vertices as a postprocessing step. The idea is that, after processing of an edge, any intersection point added by the next edge must be able to be reached from the last vertex that we output.

In general, an edge that enters a corner region will add the corresponding corner vertex as an output vertex. Liang and Barsky call such a vertex a *turning vertex*. (The original algorithm operates in a slightly different order: Rather than adding the turning vertex when the edge enters the corner region, it defers adding the vertex until some later edge leaves the corner region. This cannot entirely remove the degenerate-edge problem, and we find that it makes the algorithm more difficult to understand, so we have used our alternative formulation.)

We now examine the various cases carefully, using the analysis of the parametric form of clipping in Chapter 3. The line L_i containing the edge P_iP_{i+1} crosses all four window boundaries. Two crossings are potentially entering and two are potentially leaving. We compute the parametric values of the intersection points and call them t_{in_1}, t_{in_2}, t_{out_1}, and t_{out_2}. Notice that t_{in_1} is the least of these, and t_{out_2} is the greatest, since every nonvertical, nonhorizontal line starts in a corner region and ends in a corner region. The other two values are in between and may be in either order. As noted in Chapter 3, if $t_{in\_2} \leq t_{out\_1}$, the

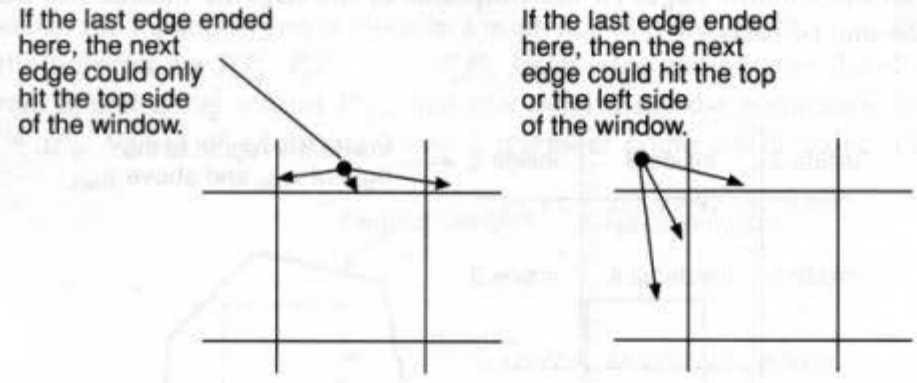


Fig. 19.11 If the previous edge terminated in the upper-middle region, and the next edge intersects the window, then it can do so only at the top. If the previous edge terminated in the upper-left region, and the next edge intersects the window, then it can do so only at the top edge or left edge.

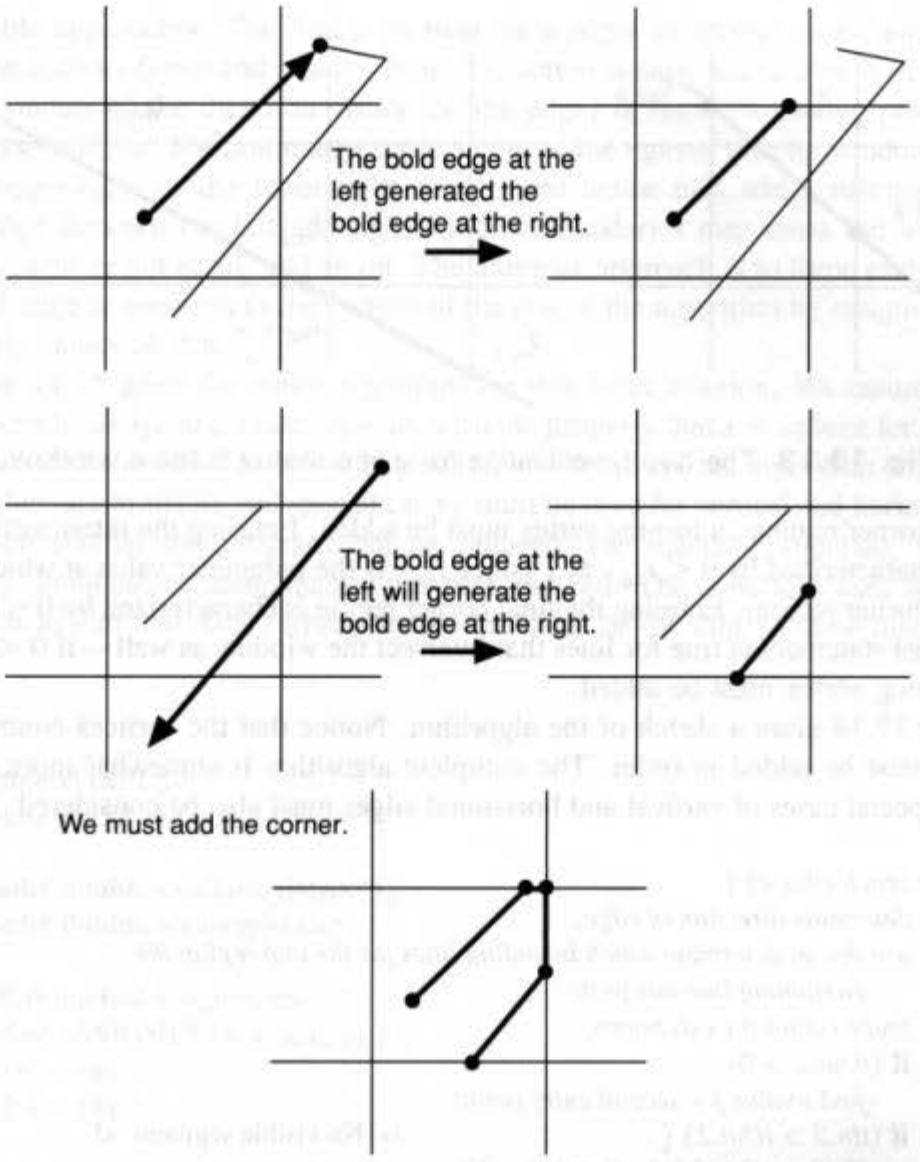


Fig. 19.12 A vertex must be added to the output polygon at the upper-right corner of the window: The next edge (after the one that fails to intersect the window) could intersect either the top or the right side. We must add a vertex that can reach all possible intersection points.

line intersects the window; if $t_{in\_2} > t_{out\_1}$, the line passes through a corner region instead (see Fig. 19.13).

The parameter values $t = 0$ and $t = 1$ define the endpoints of the edge within the line L_i. The relationship between these parameter values and the values of t_{in_1}, t_{in_2}, t_{out_1}, and t_{out_2} characterizes the contribution of the edge to the output polygon. If the edge intersects the window, the visible segment of the edge must be added to the output polygon. In this case, $0 < t_{out\_1}$ and $1 \geq t_{in\_2}$; that is, the edge begins before the containing line leaves the window and also ends after it enters the window—the edge is not entirely outside the window.

If the edge does not intersect the window, the line containing it starts in one corner region, passes through another, and terminates in a third. If the edge enters either of the

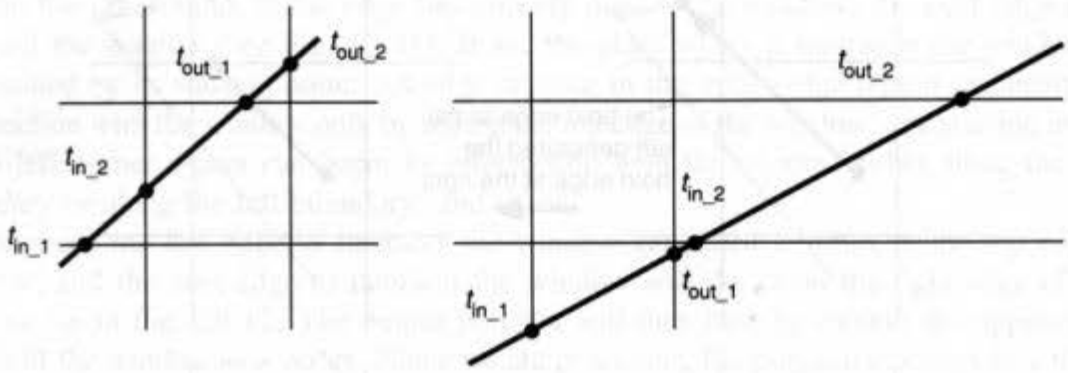


Fig. 19.13 The two possibilities for a line that crosses a window.

latter two corner regions, a turning vertex must be added. Entering the intermediate corner region is characterized by $0 < t_{out\_1} \leq 1$ (since t_{out_1} is the parameter value at which the line enters the corner region). Entering the final corner region is characterized by $0 < t_{out\_2} \leq 1$.

This last statement is true for lines that intersect the window as well—if $0 < t_{out\_2} \leq 1$, then a turning vertex must be added.

Figure 19.14 gives a sketch of the algorithm. Notice that the vertices contributed by each line must be added in order. The complete algorithm is somewhat more complex, since the special cases of vertical and horizontal edges must also be considered. There are

```
for (each edge e) {
    determine direction of edge;
    use this to determine which bounding lines for the clip region the
        containing line hits first;
    find t-values for exit points;
    if (tOut_2 > 0)
        find t-value for second entry point;
    if (tIn_2 > tOut_1) {                    /* No visible segment */
        if (0 < tOut_1 && tOut_1 <= 1)
            Output_vert (turning_vertex);
    } else {
        if (0 < tOut_1 && 1 >= tIn_2) {      /* There is some visible part. */
            if (0 <= tIn_2)
                Output_vert (appropriate side intersection);
            else
                Output_vert (starting vertex);
            if (1 >= tOut_1)
                Output_vert (appropriate side intersection);
            else
                Output_vert (ending vertex);
        }
    }
    if (0 < tOut_2 && tOut_2 <= 1)
        Output_vert (appropriate corner);
} /* for each edge */
```

Fig. 19.14 A sketch of the Liang–Barsky polygon-clipping algorithm.

two possible approaches. The first is to treat such edges as special cases, and simply to expand the code to detect and process them. Detection is easy: Either *deltaX* or *deltaY* (the two components of the direction vector for the edge) is zero. Processing proceeds on a case-by-case analysis. For example, a vertical edge to the right of the clip window may cross into the upper-right or the lower-right corner, and hence may add a turning vertex. A vertical edge between the left and right window boundaries may cross the window in a visible segment or not at all, and so on. The alternate approach is to force each vertical or horizontal edge to conform to the pattern of the rest of the algorithm by assigning entering and leaving values of $\pm\infty$.

Figure 19.15 gives the entire algorithm for this latter solution. We assume that real variables can be assigned a value, *infinite*, with the property that $x < infinite$ for all x unless $x = infinite$. Exercise 19.3 asks you to generate the details of the algorithm when no such infinite values are available and special casing must be used for vertical and horizontal lines. So that the size of the program can be reduced, the macro AssignTwo, which performs two simultaneous assignments, has been defined. The code also uses several variables (such as *Xin* and *Xout*, which denote the sides of the clip window through which

```
#define MAXPT 50;
#define MAX2 150;

typedef double smallarray[MAXPT];
typedef double bigarray[MAX2];

/* Perform two assignments. */
#define ASSIGNTWO( x, y, a, b) { \
    (x) = (a); \
    (y) = (b); \
}

/* Clip an n-sided input polygon to a window. */
void LiangBarskyPolygonClip(
        int n,
        const smallarray x, const smallarray y,     /* Vertices of input polygon */
        bigarray u, bigarray v,                     /* Vertices of output polygon */
        double xMax, double xMin,                   /* Edges of clip window */
        double yMax, double yMin,
        int *outCount)                              /* Counter for output vertices */
{
    double xIn, xOut, yIn, yOut;                    /* Coordinates of entry and exit points */
    double tOut1, tIn2, tOut2;                      /* Parameter values of same */
    double tInX, tOutX, tInY, tOutY;                /* Parameter values for intersections */
    double deltaX, deltaY;                          /* Direction of edge */
    int i;
```

Fig. 19.15 (*Cont.*)

```
x[n] = x[0];
y[n] = y[0];                                    /* Make polygon closed */
*outCount = 0;                                  /* Initialize output vertex counter */
for (i = 0; i < n; i++) {                       /* for each edge */
    deltaX = x[i + 1] - x[i];                   /* Determine direction of edge */
    deltaY = y[i + 1] - y[i];
    /* Use this to determine which bounding lines for the clip region the */
    /* containing line hits first. */
    if ((deltaX > 0) || (deltaX == 0 && x[i] > xMax))
        ASSIGNTWO (xIn, xOut, xMin, xMax)
    else
        ASSIGNTWO (xIn, xOut, xMax, xMin)
    if ((deltaY > 0) || (deltaY == 0 && y[i] > yMax))
        ASSIGNTWO (yIn, yOut, yMin, yMax)
    else
        ASSIGNTWO (yIn, yOut, yMax, yMin)
    /* Find the t values for the x and y exit points. */
    if (deltaX != 0)
        tOutX = (xOut - x[i]) / deltaX;
    else if (x[i] <= xMax && xMin <= x[i])
        tOutX = ∞;
    else
        tOutX = -∞;
    if (deltaY != 0)
        tOutY = (yOut - y[i]) / deltaY;
    else if (y[i] <= yMax && yMin <= y[i])
        tOutY = ∞;
    else
        tOutY = -∞;

    /* Order the two exit points. */
    if (tOutX < tOutY)
        ASSIGNTWO (tOut1, tOut2, tOutX, tOutY)
    else
        ASSIGNTWO (tOut1, tOut2, tOutY, tOutX)
    if (tOut2 > 0) {                            /* There could be output—compute tIn2. */
        if (deltaX != 0)
            tInX = (xIn - x[i]) / deltaX;
        else
            tInX = -∞;
        if (deltaY != 0)
            tInY = (yIn - y[i]) / deltaY;
        else
            tInY = -∞;
        if (tInX < tInY)
            tIn2= tInY;
        else
            tIn2 = tInX;
```

Fig. 19.15 (*Cont.*)

```
if (tOut1 < tIn2) {                    /* No visible segment */
    if (0 < tOut1 && tOut1 <= 1) {
        /* Line crosses over intermediate corner region */
        if (tInX < tInY)
            OutputVert (u, v, outCount, xOut, yIn);
        else
            OutputVert (u, v, outCount, xIn, yOut);
    }
} else {
    /* Line crosses through window */
    if (0 < tOut1 && tIn2 <= 1) {
        if (0 < tIn2) {                /* Visible segment */
            if (tInX > tInY)
                OutputVert (u, v, outCount, xIn, y[i] + tInX * deltaY);
            else
                OutputVert (u, v, outCount, x[i] + tInY * deltaX, yIn);

            if (1 > tOut1) {
                if (tOutX < tOutY)
                    OutputVert (u, v, outCount, xOut, y[i] + tOutX * deltaY);
                        else
                            OutputVert (u, v, outCount, x[i] + tOutY * deltaY, yOut);
                        }
                    } else
                        OutputVert (u, v, outCount, x[i + 1], y[i + 1]);
                }
            }
            if (0 < tOut2 && tOut2 <= 1)
                OutputVert (u, v, outCount, xOut, yOut);
        } /* if tOut2 */
    } /* for */
} /* LiangBarskyPolygonClip */
```

Fig. 19.15 The Liang–Barsky polygon-clipping algorithm.

the containing line for the segment enters and leaves) that could be eliminated. But these variables reduce the case structure of the code, and hence make the code more readable. The OutputVert routine (not shown) stores values in u and v and increments a counter.

19.1.4 The Weiler Polygon Algorithm

We now move to the problem of clipping one polygon to another arbitrary polygon, an issue that arose in the visible-surface computations in Section 15.7.2. Figure 19.16 shows several polygons to be clipped and the results of clipping. Notice that the clipped polygon may be disconnected and may be nonconvex even if the original polygon was convex.

The Weiler polygon algorithm [WEIL80] is an improvement on the earlier Weiler–Atherton algorithm [WEIL77], and it is based on the following observation. If we draw the edges of the clipping polygon, A, and the polygon to be clipped, B, in black pencil on a white sheet of paper, then the part of the paper that remains white is divided into disjoint

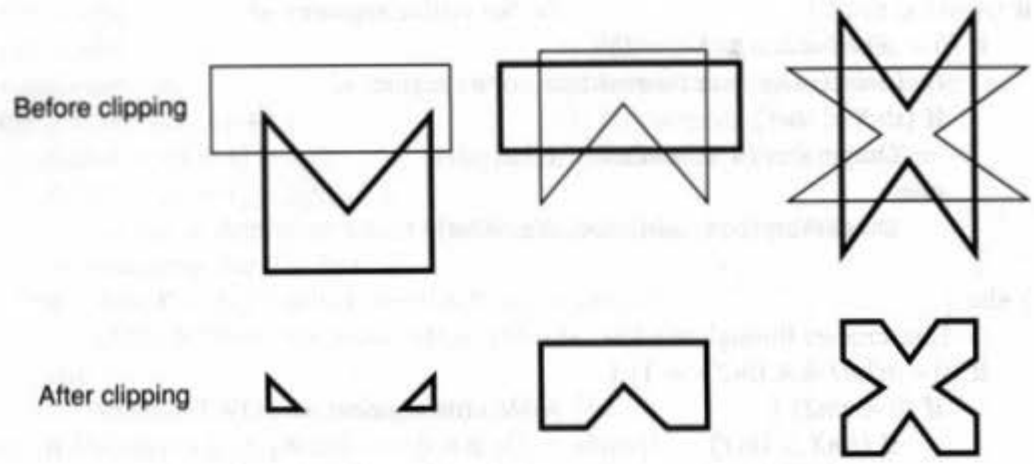


Fig. 19.16 Several examples of clipping polygons against polygons.

regions (if the polygon edges are thought of as borders on a map, then these regions are the countries). Each of these regions is entirely in A, entirely in B, entirely contained in both, or contained in neither. The algorithm works by finding a collection of closed polylines in the plane that are the boundaries of these disjoint regions. The clipped input polygon consists of the regions contained in both A and B. In this algorithm, the clipping polygon and the input polygon play identical roles; since we want the regions inside both, we shall refer to them as A and B from now on. (Notice the similarity between this approach to clipping and the polyhedral constructive solid geometry in Chapter 12. In fact, we might use the phrase *constructive planar geometry* to describe this polygon–polygon clipping.)

Before discussing the details of the algorithm, we consider one example that exhibits most of its subtlety. Figure 19.17(a) shows two intersecting polygons, A and B. In part (b), the intersections of the polygons have been added as vertices to each of the polygons; these new vertices are indicated by the dots in the figure. If two edges intersect in a single point (a *transverse* intersection), that point is added as a vertex. If they intersect in a segment, any

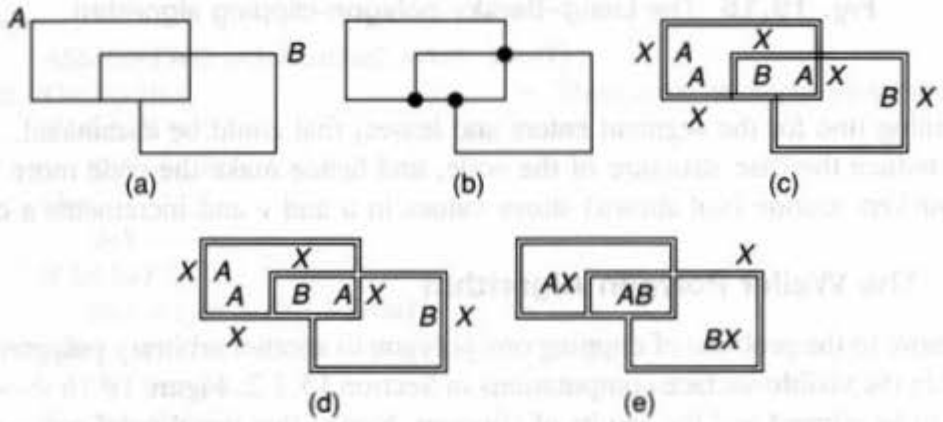


Fig. 19.17 The Weiler polygon algorithm applied to two polygons. (a) The two polygons A and B. (b) Intersections are added as vertices. (c) The polygons are drawn as doubled contours with labels. (d) The contours are reconnected so they do not cross. (e) The labels on the contours are collected into labels for the regions they bound.

vertex of one that lies in the intersection segment is added as a vertex of the other. In part (c), we double the edges of the polygons. Although these doubled polygons are drawn slightly displaced from the original polygon, they should be thought of as infinitely close to the original edge. The resulting curves are called *contours*. Each edge of a polygon contributes to two contours. We shall use the term *segment* for the parts of the contours that it contributes, and reserve the word *edge* for pieces of the original polygons. Once these contours are created, we label each segment of the inner contour of each polygon with the name of the polygon (A or B), those on the outer contour are labeled X. (Only some of the labels are shown; also, the segment where two edges overlap really has four contour segments, but only two are drawn.)

The idea is to rearrange the contours so that they form the borders of the disjoint regions we described, as is done in part (d). In part (e), the labels on each contour are collected to give a label for the region; these labels determine in which of the original polygons the region lies. For the intersection of two polygons, we are interested in only those regions labeled by both A and B (the diagram uses the label AB to indicate this). In reality, the doubled contours are generated when the polygons are first read into the algorithm; in an efficient implementation, the contours can be merged at the same time as the intersections are found.

The algorithm actually works for an arbitrary number of polygons; when more than two are intersected, there may be regions entirely contained in other regions (e.g., a square in a larger square). We will need to determine these containment relationships to complete the algorithm in such cases.

The three steps in the algorithm are setting up, determining the regions, and selecting those regions that are in both A and B. Setting up involves determining all intersections of edges of A with edges of B (since all such intersections will be vertices of the output polygon), then redefining A and B to include these intersection points as vertices. Standard algorithms from computational geometry can be used to determine all edge intersections [PREP85].

To determine the regions, we must rearrange the contours. We want no two contours to intersect. Thus, at each vertex where the two polygons intersect, an adjustment must be made. Suppose we have a transverse intersection of the polygons, as shown in Fig. 19.18(a). The corresponding contours are shown in Fig. 19.18(b). Contours are imple-

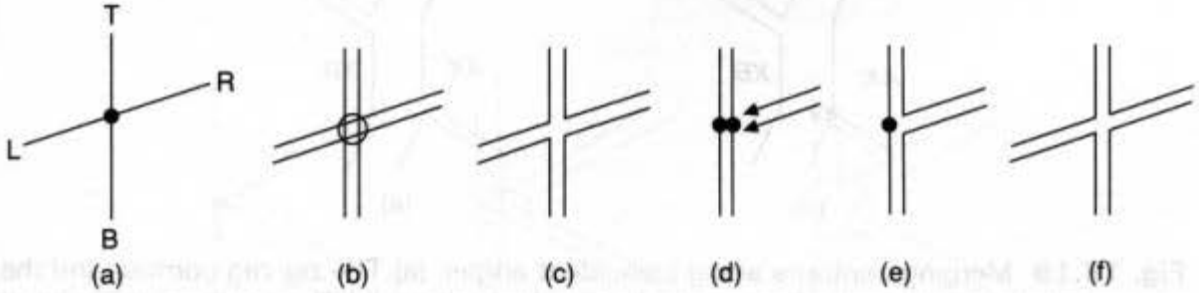


Fig. 19.18 Merging contours when edges cross. (a) The two polygons cross at a vertex. (b) The contours for the edges. (c) The final arrangement of contours. (d) Adding the contour segments contributed by edge R to the vertical contour segments. (e) The result of this addition. (f) The result after a similar merge of the segments from edge L.

mented as doubly linked lists, and we wish to rearrange the links to the pattern shown in Fig. 19.18(c). At a transverse intersection, we do this by taking the contour segments associated with the edges of one polygon and merging in the contour segments associated with the other polygon, a pair at a time. Thus, in Fig. 19.18(d), we start with the vertical contour segments, and we wish to merge the pair of segments associated with edge R. In Fig. 19.18(e), these segments have been merged into the vertical contour segments by rearranging links. The segments associated with the edges T and B have two sides (i.e., there are two vertical contours), and it is important to add the new segments to the proper side. We compare the remote vertex (the one not at the intersection point) of the edge R, to the vertical line (consisting of edges T and B), to determine which side R is on. After this attachment, the segments associated with edge L are left hanging. In Fig. 19.18(f), the segments associated with L have also been merged into the contour, so the resulting contours have no intersections at the vertex. Notice that this process involved only *local* information: we needed to know only the positions of the edges being considered, and did not need to traverse any more of the polygon.

Merging contours is more difficult at a nontransverse intersection, as shown in Fig. 19.19. In part (a), we see a nontransverse intersection of two polygons: They share a short vertical piece. Of course, in the actual data structures, this short vertical piece is an edge in each polygon. We call such edges *coincident*. The remarkable thing about coincident edges is that they are easy to process. Each edge contributes two segments to the contours, one on each side, as shown in Fig. 19.19(b), where one set of contours has been drawn within the other to make them distinct, and where a dotted segment shows the original shared edge. Each segment has an label (we have shown sample labels in Fig. 19.19b). The labels from each pair of segments on one side of the coincident edge are merged together, and the vertical segments corresponding to the coincident edge in one polygon are deleted. The

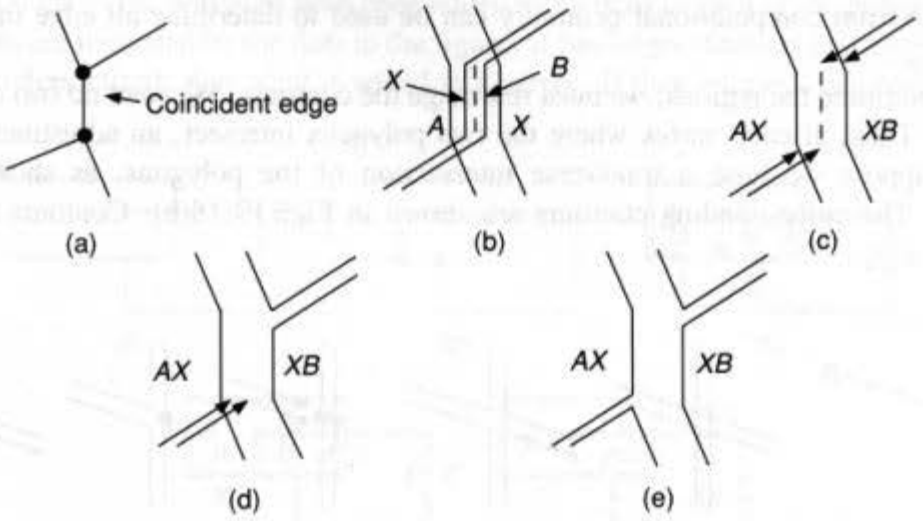


Fig. 19.19 Merging contours along coincident edges. (a) The zig-zag contour and the more vertical contour have short vertical coincident edges. (b) The contours associated with this arrangement of edges; the original vertical edge is shown as a dotted line. (c) The vertical segments from the zig-zag polygon have been deleted, and their labels merged with the labels of the other segments. (d) One adjoining edge's contour segments have been merged. (e) The other adjoining edge's contour segments have been merged as well.

resulting intermediate structure is shown in part (c) of the figure. The merged labels are shown on the remaining segments from the coincident edge. The arrows indicate dangling pointers. In parts (d) and (e), the segments corresponding to the dangling pointers are merged with the other contour just as before. If we process the contours by proceeding in order around one polygon, then we will always have at most one set of dangling pointers. If this set corresponds to a transverse intersection, it can be processed as before; if it corresponds to another coincident edge, it can be processed as this edge was.

There is another class of nontransverse intersections of polygons, as shown in Fig. 19.20(a). Since each of the diagonal edges intersects the vertical edge transversely, the process of merging contours is no more complex than in the original transverse case. The contour structure before and after merging is shown in parts (b) and (c).

Finally, we remark that, in processing intersections, the inside and outside contours associated with the original polygons are split into subcontours; since our output is derived from these subcontours, we must keep track of them. We do this record keeping by maintaining a reference to at least one segment of each contour. If we track all the segments processed at any intersection, this technique is guaranteed to provide such a list of *entry points*, although the list may have several entry points for some contours. In fact, for each intersection, we create a new contour for each segment, and set the startingPoint field of the contour to be the segment. We also set a backpointer (the contourPtr field) from the segment to the contour, which is used later.

These various tasks described lead us to the data structures for the Weiler algorithm: vertices, edges, and contours, shown in Fig. 19.21. An edge has two vertices and two *sides*. These sides are the segments that initially form the inner and outer contours for the polygons, and contribute to the output contours at the end. Thus, an edge side is what we have been calling a segment of a contour. Each edge side points to its clockwise and counterclockwise neighbors, and each edge side also has a list of owners (i.e., labels). Except in the case just described, the contourPtr field of an edge side is NULL.

When the polygons are first read in by the algorithm, they must be processed to establish the data structure given. Doing this efficiently is the task of Exercise 19.4. The second step is to take these data structures for A and B and to create a single data structure consisting of lots of contours, each contour being the boundary of one region. Since all the intersections between edges of A and B appear as vertices of these contours, we first compute these intersection points.

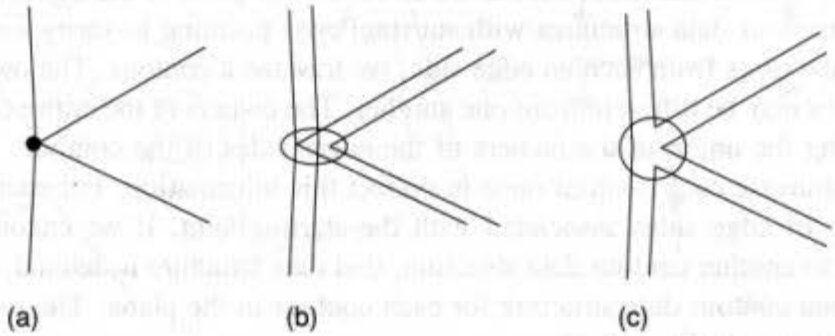


Fig. 19.20 Merging contours at a tangential intersection. (a) The edge configuration. (b) The initial contour configuration. (c) The final contour configuration.

```
#define SIDE 2      /* Each polygon edge has two sides */
#define END  2      /* Each edge has two ends, too */

typedef enum {CW, CCW} direction;
        /* Clockwise, counterclockwise directions for contours */

typedef struct {                    /* Exterior region, A, and B */
    unsigned int X:1;
    unsigned int A:1;
    unsigned int B:1;
} owners;

typedef struct {
    double x, y;
} vertex;

typedef struct contourStruct contour;
typedef struct edgeStruct edge;

struct contourStruct {
    owners belongsTo;               /* Filled in during traversal stage */
    struct {                        /* Where traversal of contour begins */
        edge *entryEdge;
        int entrySide;
    } startingPoint;
};

struct edgeStruct {
    vertex *vertices[END];          /* The ends of the edge */
    edge *edgeLinks[SIDE][2];       /* Double link structure */
    owners edgeOwners[SIDE];        /* Owners of inside and outside */
    contour *contourPtr[SIDE];      /* Points to any contour whose entry point */
                                    /* is this edge side */
};
```

Fig. 19.21 The data structures used in the Weiler algorithm.

We then merge the contours at these intersection points, as described previously, generating new contour data structures as we do so. At this point in the algorithm, we have a collection of contour data structures with startingPoints pointing to various edge sides; by following all the links from such an edge side, we traverse a contour. The owner labels on these edge sides may be different from one another. The owners of the entire contour can be found by taking the union of the owners of the edges sides of the contour.

We must traverse each contour once to collect this information. For each contour, we follow the list of edge sides associated with the startingPoint. If we encounter a vertex pointing back to another contour data structure, that data structure is deleted, so we are left with exactly one contour data structure for each contour in the plane. The pseudocode for this process is given in Fig. 19.22.

We now have a complete collection of contours, each corresponding to a single entry point. If the input polygons are simple closed curves in the plane, we are done: The output polygons are simple closed curves, and no output contour is contained in any other, so we

```
for (each contour c) {
    c.belongsTo = NULL;
    for (each edge-side pair e and s in c, beginning at c.startingPoint) {
        /* if edge side doesn't point to c, delete the contour to which it points */
        if (e.contourPtr[s] != &c)
            delete *e.contourPtr[s];
        c.belongsTo |= e.edgeOwners[s];
    }
}
```

Fig. 19.22 Pseudocode for merging contours.

can simply select those contours whose owner sets contain both A and B. In general, however, the problem is more subtle: The Weiler algorithm can be used to find any Boolean combination of A and B, and some of these combinations may have holes (if A is a square inside another square B, say, then $B - A$ has a hole). Also, A and B may have holes to begin with, and in this case even the intersection may not be nice. Since these cases are important in some applications, such as 3D clipping (see Chapter 15), we continue the analysis. We also note that the algorithm can be applied to polygons defined by such collections of contours, so that unions, differences, and intersections of polygons with holes can all be computed using this algorithm.

The contours we now have are disjoint, and each has an owner set indicating in which, if any, of the original polygons it is located. On the other hand, the regions bounded by these contours may well overlap (as in the case of the two nested squares). To determine the output of the algorithm, we must determine the nesting structure of the contours. We do so by storing the contours in a binary tree structure, in which the left child of a contour is a contour contained within it, and the right child is a contour at the same nesting depth as the parent. (We must first enlarge the data structure for a contour by adding two fields: a containedContour and a coexistingContour, each of which is a cptr—a pointer to a contour.) Figure 19.23 shows a collection of contours and an associated tree structure (this tree structure may not be unique).

Pseudocode for one algorithm for the construction of this tree is shown in Fig. 19.24.

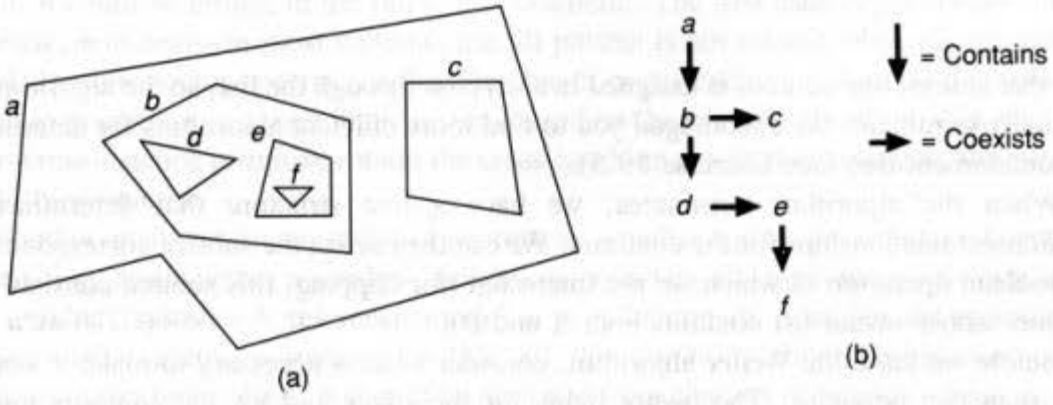


Fig. 19.23 (a) A collection of contours in the plane, and (b) an associated tree structure.

```
for (each unassigned contour c) {
    compare this contour with all the others on the unassigned list;
    if (it is inside one and only one, q) {
        remove c from the list;
        if (q.contained == NULL)
            q.contained = &c;
        else
            Insert (&c, &q);      /* Insert c into tree starting at q. */
    } else if (it is contained by more than one)
        skip this contour and go to the next one;
}

void Insert(contour *c, contour *q)
{
    if (q->contained == NULL)
        q->contained = c;
    else if (*c contains *(q->contained)) {
        c->contained = q->contained
        q->contained = c;
    } else if (*(q->contained) contains *c)
        Insert(c, q->contained);
    else if (q->contained->coexist == NULL )
        q->contained->coexist = c;
    else if (*c contains *(q->contained->coexist)) {
        c->contained = q->contained->coexist;
        q->contained->coexist = c;
    } else if (*(q->contained->coexist) contains *c) {
        Insert(c, q->contained->coexist);
    }
}  /* Insert */
```

Fig. 19.24 Pseudocode for building a containment tree structure from a collection of nested contours.

Note that at least one contour is assigned in each pass through the list, so the algorithm will eventually terminate. We encouraged you to find more efficient algorithms for determining the containment tree (see Exercise 19.5).

When the algorithm terminates, we have a tree structure that determines the containment relationships for the contours. We can then select the subtree corresponding to the Boolean operation in which we are interested (for clipping, this subtree consists of all contours whose owner list contains both A and B).

Before we leave the Weiler algorithm, consider what is necessary to make it work for more than two polygons. The owner fields for the edges and for the contours must be

enlarged to allow them to contain however many polygons we wish to combine at a time. Aside from this change, the algorithm remains essentially the same. Of course, it may be more efficient to compute the intersection of A, B, and C by computing the intersection of A and B, and then the intersection of this result with C, since the number of intersections of edges of C with edges of the intersection of A and B is likely to be smaller than is the number of intersections of C with all of A and B.

19.2 SCAN-CONVERTING PRIMITIVES

We now return to the topic of scan conversion, discussed in Chapter 3. Each primitive we scan convert has an underlying geometry (i.e., a shape) and certain attributes, such as line style, fill pattern, or line-join style.

The process of determining what pixels should be written for a given object is independent of any clipping, or of the write mode for the pixel-drawing operation. It does depend, however, on attributes that alter the object's shape, and on our criteria for pixel selection. We therefore begin by discussing attributes; then we discuss the geometry of lines and conics and the criteria that can be used for selecting pixels. We complete this section with a scan-conversion algorithm for general ellipses. We delay the consideration of text until after we have discussed antialiasing.

19.2.1 Attributes

Primitives are drawn with various attributes, as discussed in Chapter 3; these include line style, fill style, thickness, line-end style, and line-join style. These may be considered either *cosmetic* or *geometric* attributes. For example, if a line is to be drawn in 4-on, 1-off style, and it is scaled by a factor of 2, should it be drawn 8-on, 2-off? If your answer is yes, then you are treating the line style as a geometric attribute; if no, you are treating it as cosmetic. In this discussion, we assume that all line attributes are cosmetic. Since curves, polylines, circles, and ellipses are also intended to represent infinitely thin shapes, we extend the same assumption to them. What about rectangles, filled circles, and other area-defining primitives? As discussed in Chapter 3, when we apply a fill style to a primitive, we may choose to anchor it either to the primitive, to an arbitrary point, or to the bitmap into which the primitive is drawn. In the first case, the attribute is geometric; in the second, it could be either; in the third, it is cosmetic. The first case is not always entirely geometric, however—in most systems, the fill pattern is not rotated when the primitive is rotated. In some vector systems, on the other hand, cross-hatching (a form of patterning) is applied to primitives in a coordinate system based on the primitive itself, so that rotating a vector–cross-hatched primitive rotates the cross-hatching too; in these systems, the attribute is entirely geometric.

Whether attributes are cosmetic or geometric, whether a pixel lies at the grid center or grid crossing, and whether a window includes its boundary all constitute a *reference model* for a graphics system. A reference model is a collection of rules for determining the semantics of a graphics system, so that all the questions about ambiguities in the specification can be resolved. It should be designed from abstract principles, which are then embodied in the actual algorithms. It will be important to keep a reference model in mind

as we construct scan-conversion algorithms; only if we have defined this model clearly can we evaluate the success or correctness of the algorithm.

19.2.2 Criteria for Evaluating Scan-Conversion Algorithms

In drawing a line segment in Chapter 3, we drew the pixels nearest to the segment. For lines of slope greater than 1, we drew 1 pixel in each row, selecting 1 of the 2 pixels the segment passed between; when the segment passed exactly through a pixel, of course, we selected that pixel. The distance from a pixel to a segment can be measured in two ways: as the distance along a grid line, or as the perpendicular distance to the segment. Figure 19.25 shows, by similar triangles, that these distances are proportional, so that our choice of a distance measure is irrelevant.

In deriving the Gupta–Sproull antialiasing algorithm in Section 3.17.4, we found that, if the line equation was $Ax + By + C = 0$, then the perpendicular distance from a point (x, y) to the line was proportional to $F(x, y) = Ax + By + C$. This value, $F(x, y)$, is sometimes called the *residual* at the point (x, y), and the line can therefore be described as the set of all points in the Euclidean plane where the residual is zero. Thus residuals can also be used as a measure of distance to the line; the resulting choices of pixels are the same as with perpendicular distance or grid-line distance.

For circles, also, we can determine pixel choice by grid-line distance, perpendicular distance, or residual value. McIlroy has shown [MCIL83] that, for a circle with integer center and radius (or even one for which the square of the radius is an integer), the three choices agree. On the other hand, for circles whose center or radius fails to satisfy the assumption, the choices disagree in general and we must select among them. (For circles with half-integer centers or radii, the choices are not unique, because the choice made in tie-breaking cases may disagree, but McIlroy shows that there is always only 1 pixel that is a "closest" pixel by all three measures.)

For ellipses, the situation is even worse. Again, there are three ways to measure the amount by which a pixel fails to lie on the ellipse: the grid-line distance, the perpendicular distance, and the residual value. The grid-line distance is well-defined, but may lead to peculiar choices. In the case of the thin slanted ellipse in Fig. 19.26(a), for example, pixels

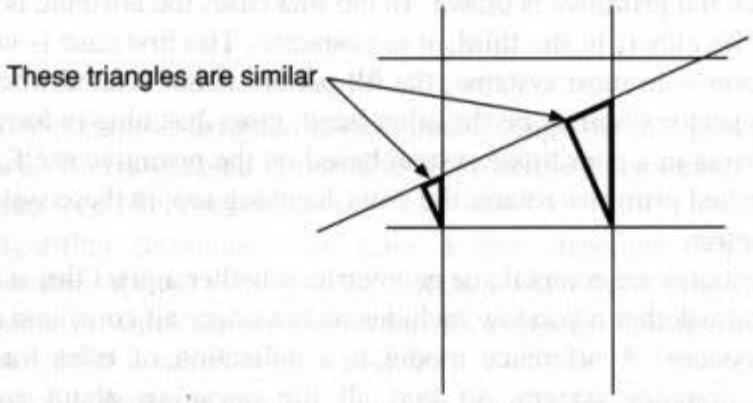


Fig. 19.25 Grid-line distance and perpendicular distance to a line are proportional, as the similar triangles show.

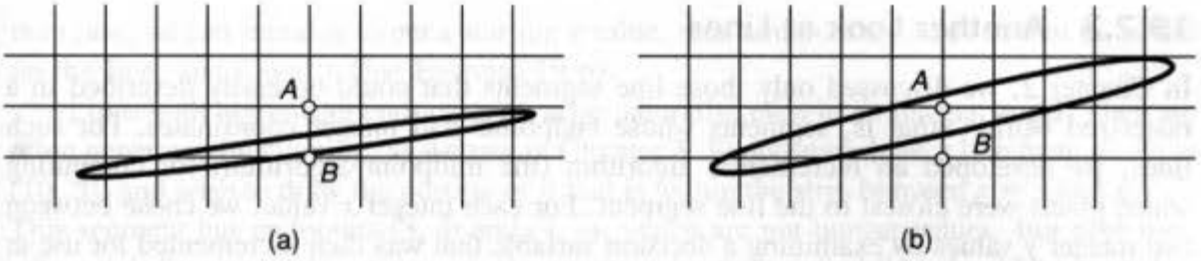


Fig. 19.26 Choosing ellipse points by different criteria. (a) *B* will be chosen in scan-converting both edges of the ellipse—should it be written twice? (b) In scan converting the bottom of the ellipse, pixel *A* will be chosen by both the grid-distance method and the residual method, even though *B* is evidently a better choice.

A and *B* are at the two ends of a grid line intersected by the upper side of the ellipse. Unfortunately, the bottom side of the ellipse also passes between them. Both sides are closer to *B* than to *A*, measured by grid-line distance. If we choose *B* in scan converting both edges, we will write the same pixel twice. The perpendicular distance to the ellipse is also well defined (it is the minimum of the distances between the point and all points of the ellipse), but a single point may be close to several points on the ellipse. For example, in scan converting the bottom side of the ellipse in Fig. 19.26(b), pixel *A* is closer to the ellipse than is pixel *B*, but we would like to draw pixel *B* regardless, since the *part* of the ellipse to which *A* is close happens to be unrelated to the part to which *B* is close. The residuals in this case determine the same (wrong) choice—the residual at *A* is less than the residual at *B*. Thus, there is no easy answer in choosing a measure of closeness for an ellipse. For tightly curved ellipses, all measures of closeness fail. We shall discuss ways to circumvent this difficulty later.

Part of the difficulty with using residuals as an error measure is implicit in the nature of residuals. For a circle, for example, it is true that the circle of radius R at the origin consists of all points (x, y) satisfying $x^2 + y^2 - R^2 = 0$, so $x^2 + y^2 - R^2$ is a reasonable measure of the extent to which the point (x, y) fails to lie on the circle. Unfortunately, the circle can also be defined as the set of points satisfying $(x^2 + y^2)^{1/2} - R = 0$, so residuals computed using $(x^2 + y^2)^{1/2} - \mathrm{R}$ could be used as well (but they are different from—not even proportional to—the first type of residual), as could residuals computed using $(x^2 + y^2)^a - R^{2a}$ for any positive value of a. Thus, use of residuals is really an arbitrary method of measurement.

Finally, for general curves, there is no clear choice. We still find the midpoint criterion convincing: In a place where the curve is near horizontal, we choose between vertically adjacent pixels by determining on which side of the midpoint the curve passes (this amounts to the grid-distance criterion), and we present an algorithm based on this criterion.

It is important, in designing a scan-conversion algorithm, to choose a way to measure error, and then to design the algorithm to minimize this error. Only if such a measure has been chosen can an algorithm be proved to be correct. It is also important to specify the point at which approximations are incorporated. An algorithm that approximates a curve by line segments, and then scan converts them, may have a very small error if the error is measured as distance to the line segments, but a very large error measured by distance to the curve.

19.2.3 Another Look at Lines

In Chapter 2, we discussed only those line segments that could be easily described in a rasterized world—that is, segments whose endpoints had integer coordinates. For such lines, we developed an incremental algorithm (the midpoint algorithm) for computing which pixels were closest to the line segment. For each integer x value, we chose between two integer y values by examining a decision variable that was then incremented for use at the next x value. Initializing this decision variable was easy, because the coefficients of the line equation ($Ax + By + C = 0$) were all integers, and the starting point of the segment, (x_0, y_0), was guaranteed to satisfy the line equation (i.e., $Ax_0 + By_0 + C = 0$).

Not all line segments have integer endpoints however. How can we draw on a raster device a line segment whose endpoints are real numbers? Suppose we wish to draw a line from (0, 0) to (10, 0.51) on a black-and-white raster device. Clearly, pixels (0, 0) through (9, 0) should be drawn, and pixel (10, 1) should be drawn. If we take the simple approach of rounding the endpoints to integer values, however, we draw pixels (0, 0) to (5, 0) and (6, 1) to (10, 1), which is completely different. Instead, we could compute the equation of the line:

$$1.0y - 0.051x + 0.0 = 0.0,$$

and apply the midpoint algorithm to this line. This approach requires using a floating-point version of the midpoint algorithm, however, which is expensive. Another possibility is to recognize that, if we multiply the line equation by 1000, we get

$$1000y - 51x + 0 = 0,$$

which is an integer line we could draw instead. We can use this approach in general, by converting floating-point numbers into fixed-point numbers and then multiplying by an appropriately large integer to give integer coefficients for the line.[3] In most cases, the multiplier determines the subpixel resolution of the endpoints of the line: If we wish to place endpoints on quarter-integer locations, we must multiply by 4; for tenth-integer locations, we multiply by 10.

Yet another problem arises if the first pixel to be drawn does not lie exactly on the line. Consider the line from (0, 0.001) to (10, 1.001). Its equation is

$$y - 0.1x - 0.001 = 0,$$

or

$$1000y - 100x - 1 = 0,$$

and the first pixel to be drawn is (0, 0), which does not lie exactly on the line. In this case, we need to initialize the decision variable by explicitly computing the value of the residual, $Ax + By + C$, at the starting point; none of the simplification from the original algorithm is possible here. Choosing the starting point in such a case is also a problem. If the actual endpoint is (x_0, y_0), we must choose an integer point near (x_0, y_0). For a line of slope less

[3] We must be careful not to multiply by too large an integer; overflow could result.

than one, we can round x_0 to get a starting x value, then compute a y value so that (x, y) lies on the line, and round y (see Exercise 19.6).

Lines with noninteger endpoints do arise naturally, even in an integer world. They are often generated by clipping, as we saw in Chapter 3. Suppose we have a line from (0, 0) to (10, 5), and wish to draw the portion of it that is within the strip between $x = 3$ and $x = 7$. This segment has endpoints $(3, \frac{3}{2})$ and $(7, \frac{7}{2})$, which are not integer values. But note that, when such lines are scan-converted, the equation for the original line should be used, lest roundoff error change the shape of the line (see Exercise 19.7).

In summary, then, we can take an arbitrary line segment, convert its endpoints to rational numbers with some fixed denominator, generate an integer equation for the line, and scan convert with the algorithm from Chapter 3 (after explicitly initializing the decision variable).

19.2.4 Advanced Polyline Algorithms

In creating rasterized polylines (unfilled polygons), there is little more to do than to use the line-drawing algorithm repeatedly. But the issue of a reference model arises here—we must decide what we want before we design the algorithm. Consider a polyline with a very sharp point, as shown in Fig. 19.27. In scan conversion of one edge, the pixels shaded horizontally are drawn. In scan conversion of the other edge, the pixels shaded vertically are drawn. The pixels drawn by both are cross-hatched. If this polyline is drawn in **xor** mode, the pixels drawn by both lines are **xor**ed twice, which is probably not what is wanted. There are two possible solutions: to draw the primitive into an offscreen bitmap in **replace** mode, and then to copy the resulting bitmap to the screen in **xor** mode, or to create a data structure representing the entire primitive and to render this data structure into the bitmap. The shape data structure described in Section 19.7 provides a means for doing this (and can accommodate patterned lines or polylines as well).

19.2.5 Improvements to the Circle Algorithm

We have an algorithm for generating circles rapidly (the midpoint algorithm). In Chapter 3, we discussed using this algorithm to draw thick circles and filled circles. The algorithm presented there, however, handled only circles with integer radii and integer centers. When the center or radius is noninteger, none of the symmetries of the original algorithm apply, and each octant of the circle must be drawn individually, as Fig. 19.28 shows.

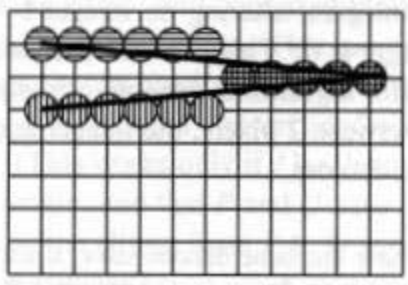


Fig. 19.27 A polyline with a sharp point may cause some pixels to be drawn more than once.

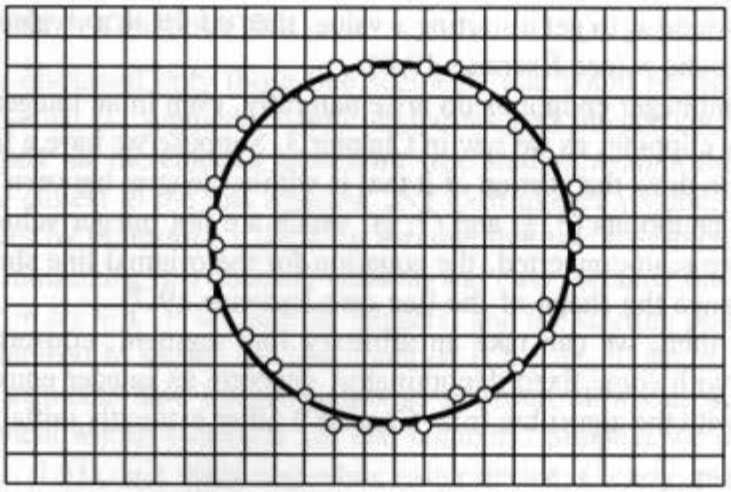


Fig. 19.28 This circle has noninteger center and radius, and each of its octants is different from the others, so no symmetries can be used to speed up scan conversion.

Unfortunately, an arbitrary circle (with real values for center and radius) cannot be converted to an integer conic. If the values of the center and radius are constrained to be rational numbers with a particular denominator,[4] then after computing (in floating point) the coefficients of the equation for the circle, we can multiply through by four times this denominator and round the coefficients. Applying a midpoint algorithm to the resulting equation with integer coefficients yields almost the same points as would result if the algorithm were applied to the original floating-point equation. To see this, suppose the center is $(h/p, k/p)$ and the radius is R. The equation of the circle is

$$S(x, y) = \left(x - \frac{h}{p}\right)^2 + \left(y - \frac{k}{p}\right)^2 - R^2 = 0.$$

Multiplying by $4p$, we get

$$4pS(x, y) = 4px^2 - 8hx + 4py^2 - 8ky + 4\left(\frac{h^2}{p} + \frac{k^2}{p} - pR^2\right) = 0.$$

Recall that, in the midpoint algorithm, we test the sign of the decision variable. Rounding the term in parentheses (the only noninteger) replaces a rational number by an integer, which alters the value of $4pS(x, y)$ by less than 1. Since we are evaluating S at points where one of x or y is an integer and the other is an integer plus $\frac{1}{2}$, the first few terms of the expression for $4pS(x, y)$ are integers; altering an integer by a number less than 1 cannot change its sign, unless the integer is 0.[5] Thus, we can use the rounded version of $4pS$ as our decision variable for a midpoint algorithm; the sole consequence is that, when the circle passes through the midpoint between 2 pixels, the integer and floating-point versions of the algorithm may make opposite choices.

---

[4] We choose to have all numbers have the same denominator, since the algebra is simpler. Since any two fractions can be put over a common denominator, this is not a severe restriction.

[5] We must, in designing a midpoint algorithm, decide whether 0 is positive or negative. Adding a small number to 0 can change this choice of sign.

The partial differences are the same as before, except that they have been multiplied by a constant $4p$. So, rather than incrementing by 2, we increment by $8p$. Initialization is more complex as well. The point resulting from adding $(0, R)$ to the center point is no longer an integer point, so we must choose some nearby point at which to start the circle. To draw the top-right eighth of the circle, we begin with the point at or just to the right of the top of the circle. We get the x coordinate by rounding up h/p to an integer, but then must compute the y coordinate explicitly and initialize the decision variable explicitly. We do these steps only once in the course of the algorithm, so their cost is not particularly significant. The code shown in Fig. 19.29 generates only one-eighth of the circle; seven more similar parts are needed to complete it.

19.2.6 A General Conic Algorithm

Chapter 3 presented an algorithm for scan converting ellipses whose axes are aligned with the axes of the plane. In this section, we describe an algorithm, developed by Van Aken [VANA89], for general conics, including ellipses with tilted axes, hyperbolas, circles, and parabolas. The algorithm is based on Pitteway's curve-tracking algorithm, which was published in 1967 [PITT67], just 2 years after Bresenham introduced his incremental line algorithm. At the time, Pitteway's algorithm received little attention, and much of it has been rediscovered several times.

Algorithms for conics have two separate elements: specifying the conic and performing the scan conversion. Since the general conic can be described as the solution to an equation of the form

$$S(x, y) = Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0,$$

the conic could be specified by the six coefficients A, B, C, D, E, and F. These are hardly intuitive, however, so we instead consider an alternative. We discuss only ellipses here, but similar techniques can be used for hyperbolas and parabolas (see Exercise 19.9).

A circle in the plane fits nicely into a unit square (see Fig. 19.30a). If an affine transformation (a linear transformation using homogeneous coordinates) is applied to the plane, the square is transformed to a parallelogram and the circle is transformed to an ellipse, as in Fig. 19.30(b). This is a generalization of SRGP's specification of an ellipse, where an aligned rectangle was used as a bounding box for an aligned ellipse. The midpoints of the sides of the parallelogram are points on the ellipse. Specifying these and the center of the parallelogram uniquely determines the parallelogram, and hence the ellipse. So our specification of an ellipse will consist of the center, J, and midpoints of the sides, P and Q, of a parallelogram. Observe that, if we can determine the coefficients in the case where J is the origin, then we can handle the general case: We just apply the simpler case to $P' = P - J$ and $Q' = Q - J$, scan convert, and add the coordinates of J to each output pixel before drawing it. (This works only if J has integer coordinates, of course.) We therefore assume that J is the origin, and that P and Q have been adjusted accordingly. We also assume that the short arc of the ellipse from P to Q goes counterclockwise around the origin; otherwise, we exchange P and Q.

To find the equation of the ellipse, we first find a transformation taking the points $\begin{bmatrix}1\\0\end{bmatrix}$ and $\begin{bmatrix}0\\1\end{bmatrix}$ to P and Q (it is given by a matrix whose first column is P and whose second column

```
void MidpointEighthGeneralCircle (
    int h, k;                          /* Numerators of x and y coordinates of center */
    int p,                             /* Denominator of both */
    double radius)                     /* Radius of circle */
{
  int x, y,                            /* Last point drawn */
    d,                                 /* Decision variable */
    A,D,E,F,                           /* Coefficients for equation */
    A2, A4,                            /* Multiples of A for efficiency */
    deltaE,                            /* d(x + 2,y - 1/2) - d(x + 1, y - 1/2) */
    deltaSE;                           /* d(x + 2,y - 3/2) - d(x + 1, y - 1/2) */
  double temp;

  /* Initialize coefficients x, y and differences */
  A = 4 * p;
  A2 = 2 * A;
  A4 = 4 * A;
  D = -8 * h;
  E = -8 * k;
  temp = 4 * (-p * radius * radius + (h*h + k*k) / (double)p);
  F = round (temp);                    /* Introduces error less than 1 */
  x = ceil (h / (double)p);            /* Smallest integer >= h/p */
  y = round (sqrt (radius*radius - (x - h/(double)p) * (x- h/(double)p)) +
      k/(double)p);
  /* The next line must be computed using real arithmetic, not integer; it */
  /* can be rewritten to avoid this requirement. */
  d = round (A * ((x + 1.0) * (x + 1.0) + (y - 0.5) * (y - 0.5)) +
          D * (x + 1.0) + E * (y - 0.5) + F);
  deltaE = A2 * (x + 1) + A + D;
  deltaSE = A2 * (x - y) + 5 * A + D - E;
  DrawPixel (x, y);
  while ((p * y - k) > (p * x - h)) {  /* While within this octant */
    if (d < 0) {                       /* Select E pixel. */
      d += deltaE;
      deltaE += A2;
      deltaSE += A2;
      x++;
    } else {
      d += deltaSE;                    /* Select SE pixel. */
      deltaE += A2;
      deltaSE += A4;
      x++;
      y--;
    }
    DrawPixel (x, y);
  } /* while */
} /* MidpointEighthGeneralCircle */
```

Fig. 19.29 The general midpoint-circle algorithm.

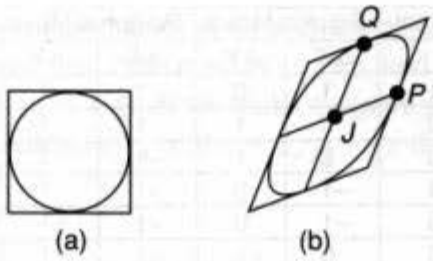


Fig. 19.30 (a) The bounding square for a circle. (b) The transformed square and the ellipse it bounds.

is Q). This transformation takes points on the unit circle, which have the form $\cos(t)\begin{bmatrix}1\\0\end{bmatrix} + \sin(t)\begin{bmatrix}0\\1\end{bmatrix}$, into points on the ellipse, which therefore have the form $\begin{bmatrix}x\\y\end{bmatrix} = \cos(t)P + \sin(t)Q$. Solving this equation for $\cos(t)$ and $\sin(t)$ gives each as a linear expression in x and y. By substituting these linear expressions into the identity $\cos^2(t) + \sin^2(t) = 1$, we can arrive at a quadratic expression in x and y that describes the ellipse (see Exercise 19.8). If we write P_x, P_y for the coordinates of P, and similarly for Q, the resulting coefficients are

$$A = P_y^2 + Q_y^2, \quad B = -2(P_xP_y + Q_xQ_y), \quad C = P_x^2 + Q_x^2,$$
$$D = 0, \quad E = 0, \quad F = -(P_xQ_y - P_yQ_x)^2.$$

We now translate the resulting ellipse to a new coordinate system centered at the point $-P$; that is, we replace the equation for the ellipse by a new equation:

$$A(x + P_x)^2 + B(x + P_x)(y + P_y) + C(y + P_y)^2 + D(x + P_x) + E(y + P_y) + F =$$
$$A'x^2 + B'xy + C'y^2 + D'x + E'y + F' = 0.$$

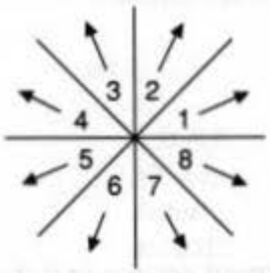
The resulting coefficients in this new coordinate system are

$$A' = A, \quad B' = B, \quad C' = C,$$
$$D' = 2Q_y(P_xQ_y - P_yQ_x) \quad E' = -2Q_x(P_xQ_y - P_yQ_x) \quad F' = 0.$$

The origin lies on this new conic, and if we scan convert this conic, but add (P_x, P_y) to each point before drawing it, we get the points of the original conic centered at the origin. Since A, B, and C are unchanged, we use D and E to denote the terms called D' and E' (since the original D and E were both zero).

Now, having derived the coefficients of the ellipse equation, we need to scan convert it. We divide the scan-conversion process into eight *drawing octants* (for the circle algorithm in Chapter 3, these corresponded to octants of the circle; here, they do not). The drawing octant indicates the direction in which the algorithm is tracking. In octant 1, the choice is between moving to the right and moving diagonally up and to the right. The moves to be made are classified as *square moves* or *diagonal moves*, depending on whether one coordinate changes or both do. Figure 19.31 shows the eight octants, a table indicating the directions of motion in each, and the corresponding arcs on a typical ellipse.

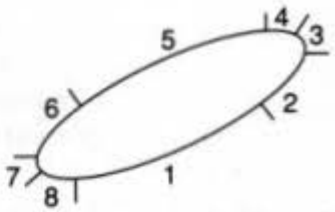
Our algorithm has two different loops—one for tracking in odd-numbered octants, terminated by reaching a diagonal octant boundary, and one for even-numbered octants, terminated by reaching a square octant boundary. To determine the starting octant, we observe that, at any point of a conic given by an equation $S(x, y) = 0$, the gradient of S,



(a)

Octant	Square Move		Diagonal Move	
	ΔX	ΔY	ΔX	ΔY
1	1	0	1	1
2	0	1	1	1
3	0	1	−1	1
4	−1	0	−1	1
5	−1	0	−1	−1
6	0	−1	−1	−1
7	0	−1	1	−1
8	1	0	1	−1

(b)



(c)

Fig. 19.31 (a) The eight drawing octants. (b) the corresponding directions of motion. (c) a typical ellipse.

which is $(2Ax + By + D, Bx + 2Cy + E)$, points perpendicular to the conic (this was used to determine the ellipse octants in the algorithm in Chapter 3). We use the coordinates of the gradient to determine the direction of motion (by rotating it 90° counterclockwise) and hence the drawing octant. Since our starting point is (0, 0), this gradient is just (D, E). The code fragment in Fig. 19.32 shows how the classification into octants is done.

Now, for each octant we must determine what the value of a decision variable, d, is, what its meaning is, and how to update it. When we move diagonally, we will update d by adding an increment v; for square moves, the increment will be called u. If $S(x, y) = 0$ is the equation of the ellipse, we define d by evaluating S at the midpoint of the segment between the next two possible pixel choices. Since $S(x, y) < 0$ inside the ellipse, and $S(x, y) > 0$ outside the ellipse, a negative value of d will indicate that the ellipse passes outside the midpoint, and we will want to choose the outer pixel. When d is positive, we choose the inner pixel. When d is zero, we must choose between the 2 pixels—Van Aken's choice (in odd-numbered octants) is to make a square step whenever d is negative, and a diagonal step otherwise. In even octants, he makes a diagonal step when d is negative, and a square step otherwise.

In octant 1, if we have just drawn pixel (x_i, y_i), we denote by d_i the value we use to decide between $(x_i + 1, y_i)$ and $(x_i + 1, y_i + 1)$. We write u_{i+1} and v_{i+1} for the quantities to be added to d_i in order to create d_{i+1}. The work to be done at each pixel is therefore (1) drawing the pixel, (2) choosing the next pixel to draw based on the value of d_i, (3) updating u_i and v_i to u_{i+1} and v_{i+1} on the basis of the choice made, (4) updating d_i to d_{i+1} by adding either u_{i+1} or v_{i+1}, and (5) checking for an octant change.

```
int GetOctant (int D, int E)
{
    if (D > 0 && E < 0)
        return (D < -E) ? 1 : 2;
    if ...      /* Handle remaining six cases */
} /* GetOctant */
```

Fig. 19.32 A function for determining the drawing octant from the components of the gradient.

Recall that d_{i+1} can be computed from d_i by a differencing technique. Suppose we are in the first drawing octant, have just drawn pixel (x_i, y_i), and have decided which pixel to choose next using $d_i = S(x_i + 1, y_i + \frac{1}{2})$. If we make a square move, then $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i$. The new decision variable d_{i+1} is $S(x_i + 2, y_i + \frac{1}{2})$; the difference between this and d_i is

$$\begin{aligned}
u_{i+1} &= d_{i+1} - d_i \\
&= A(x_i + 2)^2 + B(x_i + 2)(y_i + \tfrac{1}{2}) + C(y_i + \tfrac{1}{2})^2 \\
&\quad + D(x_i + 2) + E(y_i + \tfrac{1}{2}) + F \\
&\quad - [A(x_i + 1)^2 + B(x_i + 1)(y_i + \tfrac{1}{2}) + C(y_i + \tfrac{1}{2})^2 \\
&\quad + D(x_i + 1) + E(y_i + \tfrac{1}{2}) + F] \\
&= A[2(x_i + 1) + 1] + B(y_i + \tfrac{1}{2}) + D \\
&= A(2(x_i) + 1) + B(y_i + \tfrac{1}{2}) + D + 2A.
\end{aligned}$$

On the other hand, if we make a diagonal move, then d_i is $S(x_i + 2, y_i + \frac{3}{2})$, and the increment is

$$\begin{aligned}
v_{i+1} &= d_{i+1} - d_i \\
&= (2A + B)x_{i+1} + (B + 2C)y_{i+1} + A + B/2 + D + E \\
&= (2A + B)x_i + (B + 2C)y_i + A + B/2 + D + E + [2A + 2B + 2C].
\end{aligned}$$

If we let u_i denote $A[2(x_i) + 1] + B(y_i + \frac{1}{2}) + D$, then for the square move we see that $u_{i+1} = u_i + 2A$. Similarly, if v_i denotes $(2A + B)x_i + (B+ 2C)y_i + A + B/2 + D + E$, then for a diagonal move, $v_{i+1} = v_i + (2A + 2B + 2C)$. To keep correct values of u_i and v_i for both diagonal and square moves, we must update these values even if they are not used. Thus, for a square move,

$$\begin{aligned}
v_{i+1} &= (2A + B)x_{i+1} + (B + 2C)y_{i+1} + A + B/2 + D + E \\
&= (2A + B)(x_i + 1) + (B + 2C)y_i + A + B/2 + D + E \\
&= v_i + (2A + B);
\end{aligned}$$

for a diagonal move, $u_{i+1} = u_i + 2A + B$. We encourage you to work out a table that shows the increments in u_i and v_i for a square move or a diagonal move in each drawing octant (see Exercise 19.10). The algebra is tedious but instructive.

If $k_1 = 2A$, $k_2 = 2A + B$, and $k_3 = 2A + 2B + 2C$, then the update procedure for the us and vs can be described by the following rules.

Square move:

$$u_{i+1} = u_i + k_1, \qquad v_{i+1} = v_i + k_2.$$

Diagonal move:

$$u_{i+1} = u_i + k_2, \qquad v_{i+1} = v_i + k_3.$$

Let us now consider how to determine octant changes. We see that we leave drawing octant 1 when the gradient vector points down and to the right—that is, when it is a multiple of $(1, -1)$. In other words, we leave octant 1 when the sum of the two components of the gradient vector goes from being negative to being zero (see Fig. 19.33).

Now observe that the components of the gradient vector,

$$\left(\frac{\partial S}{\partial x}, \frac{\partial S}{\partial y}\right) = (2Ax + By + D, Bx + 2Cy + E),$$

can be expressed in terms of the values of u_i and v_i given previously:

$$\frac{\partial S}{\partial x} = u_i - \frac{k_2}{2}, \qquad \frac{\partial S}{\partial x} + \frac{\partial S}{\partial y} = v_i - \frac{k_2}{2}.$$

If we therefore check the sign of $v_i - k_2/2$, we can detect when we leave the first drawing octant. The corresponding check for leaving the second drawing octant is the sign of $u_i - k_2/2$.

Before we can write the actual code, however, we need to consider one last issue. When we go from drawing octant 1 to drawing octant 2, the definition of d_i changes, as do those of u_i and v_i; they still correspond to the increments for square moves and diagonal moves, respectively, but the square moves are now vertical rather than horizontal, and the value being updated is no longer $S(x_i + 1, y_i + \frac{1}{2})$ but rather $S(x_i + \frac{1}{2}, y_i + 1)$. Using primes to denote the values in octant 2, and unprimed symbols to denote the values in octant 1, we compute

$$\begin{aligned} d'_i - d_i &= S\left(x_i + \frac{1}{2}, y_i + 1\right) - S\left(x_i + 1, y_i + \frac{1}{2}\right) \\ &= \frac{v_i}{2} - u_i + \frac{3}{8}k_3 - \frac{1}{2}k_2, \\ v'_i - v_i &= [(2A + B)x_i + (B + 2C)y_i + \frac{B}{2} + C + D + E] \\ &\quad - [(2A + B)x_i + (B + 2C)y_i + A + \frac{B}{2} + D + E] \\ &= -A + C \\ u'_i - u_i &= v_i - u_i - \frac{k_2}{2} + \frac{k_3}{2}. \end{aligned}$$

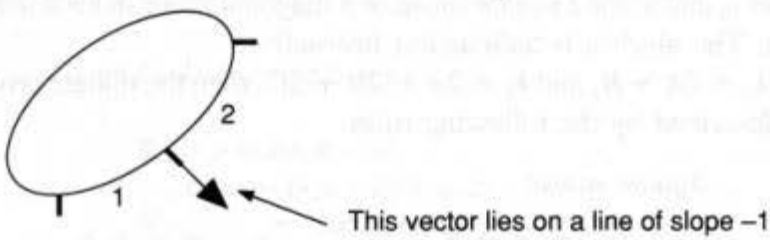


Fig. 19.33 While we draw in octant 1, the sum of the components of the gradient is negative. When we enter octant 2, the sum becomes positive.

The computations involved in deriving the first and third equations are straightforward but long. The variables for incrementing u_i and v_i need to be updated as well. Assuming you have worked out the table of increments, it is clear that $k_3' = k_3$, $k_2' = k_3 - k_2$, and $k_1' = k_1 - 2k_2 + k_3$.

We now have all the tools we need to generate the algorithm, at least for two octants. We include the coefficient F (the constant term for the conic) in the code, even though we are assuming it is zero; for a more general conic, whose center is not an integer, the starting point for the algorithm may not lie exactly on the conic, in which case F may be nonzero. We leave it to you to experiment with this case.

The code for the algorithm in Fig. 19.34 is followed by a procedure, Conjugate, which determines the coefficients for an ellipse (at the origin) from the endpoints of conjugate diameters. If the points are P and Q, then the parallelogram whose vertices are $P + Q$, $P - Q$, $-P - Q$, and $-P + Q$ bounds the ellipse; the ellipse is tangent to this parallelogram at the midpoints of the sides as in Fig. 19.30.

The code finishes by entering the ending octant, counting the number of steps in the square-step direction needed to reach the final pixel, then continuing the algorithm until that many steps have been taken. We leave the details of this step to you (see Exercise 19.13). Surprisingly, the code that updates the increments during the first two octant changes actually works for all octant changes.

One last issue remains in drawing ellipses. As shown in Fig. 19.35, sometimes a diagonal step in the algorithm takes the drawing point clear across to the other side of the ellipse—it changes several octants at once. When this occurs, the algorithm breaks down and marches away from the ellipse, as shown. It is remarkable that this is a form of aliasing—the signal $S(x, y)$ that we are sampling contains frequencies too high to be resolved by sampling at integer grid points.

Pratt has proposed a solution to this problem [PRAT85]. While the algorithm is tracking pixels in drawing octant 1, a jump across the ellipse causes the gradient vector to change direction radically. In fact, the gradient vector in drawing octant 1 always has a negative y component and positive x component. In jumping across the ellipse, we arrive at a point in octant 3, 4, 5, or 6. We determine the dividing line between these octants and octants 7, 8, 1, and 2, by setting the x component of the gradient to zero—that is, $2Ax + By + D = 0$. Since $u_i = 2Ax_i + By_i + D + A + B/2$, u_i can be used to detect such crossings. Similar checks can be generated for each octant. Notice that these checks need to be applied during only one type of move for each octant: one type of move steps away from the ellipse, and hence the check is not needed, but one steps toward it, and hence the check is needed. For further information on conic tracking algorithms see Exercise 19.14 and [PITT67; PRAT85; DASI89].

One other type of algorithm for conics has been developed by Wu and Rokne [WU89], but is suited for only gray-scale displays. Instead of scan converting one pixel at a time, it scan converts by blocks of pixels. Suppose we are stepping horizontally, and that the curvature is concave up. If we have just drawn pixel (x, y), and after two more steps we are drawing pixel $(x + 2, y + 2)$, then the intervening pixel must be $(x + 1, y + 1)$. Similarly, if after two more steps we are drawing $(x + 2, y)$, then the intervening pixel must be $(x + 1, y)$. Only if after two steps we draw pixel $(x + 2, y + 1)$ is there any ambiguity: Then, the intervening pixel may have been $(x + 1, y)$ or $(x + 1, y + 1)$. On a gray-scale

```
/* Draw an arc of a conic between the points (xs, ys) and (xe, ye) on the conic; */
/* the conic is given by Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0. If the conic is a */
/* hyperbola, the two points must lie on the same branch. */
void Conic (
        int xs, ys,                     /* Starting point */
        int xe, ye,                     /* Ending point */
        int A, B, C, D, E, F)           /* Coefficients */
{
    int x, y;                           /* Current point */
    int octant;                         /* Current octant */
    int dxsquare, dysquare;             /* Change in (x, y) for square moves */
    int dxdiag, dydiag;                 /* Change in (x, y) for diagonal moves */
    int d,u,v,k1,k2,k3;                 /* Decision variables and increments */
    int dSdx, dSdy;                     /* Components of gradient */
    int octantCount;                    /* Number of octants to be drawn */
    int tmp;                            /* Used to perform a swap */

    octant = GetOctant (D, E);          /* Starting octant number */
    switch (octant) {
        case 1:
            d = round (A + B * 0.5 + C * 0.25 + D + E * 0.5 + F);
            u = round (A + B * 0.5 + D);
            v = round (A + B * 0.5 + D + E);
            k1 = 2 * A;
            k2 = 2 * A + B;
            k3 = k2 + B + 2 * C;
            dxsquare = 1;
            dysquare = 0;
            dxdiag = 1;
            dydiag = 1;
            break;
        case 2:
            d = round (A * 0.25 + B * 0.5 + C + D * 0.5 + E + F);
            u = round (B * 0.5 + C + E);
            v = round (B * 0.5 + C + D + E);
            k1 = 2 * C;
            k2 = B + 2 * C;
            k3 = 2 * A + 2 * B + 2 * C;
            dxsquare = 0;
            dysquare = 1;
```

Fig. 19.34 (*Cont.*)

```
        dxdiag = 1;
        dydiag = 1;
        break;
    ...six more cases...
}  /* switch */

x = xe - xs;                          /* Translate (xs, ys) to origin. */
y = ye - ys;
dSdx = 2 * A * x + B * y + D;         /* Gradient at endpoint */
dSdy = B * x + 2 * C * y + E;
/* This determines the ending octant. */
octantCount = GetOctant (dSdx, dSdy) - octant;
if (octantCount <= 0) octantCount += 8;

/* Now we actually draw the curve. */
x = xs;
y = ys;
while (octantCount > 0) {
    if (octant & 1) {
        while (v <= k2 * 0.5) {
            DrawPixel (x, y);
            if (d < 0) {
                x += dxsquare;
                y += dysquare;
                u += k1;
                v += k2;
                d += u;
            } else {
                x += dxdiag;
                y += dydiag;
                u += k2;
                v += k3;
                d += v;
            }
        }  /* while v <= k2 * 0.5 */
        /* We now cross the diagonal octant boundary. */
        d = round (d - u - v * 0.5 - k2 * 0.5 + 3 * k3 * 0.125);
        u = round (-u + v - k2 * 0.5 + k3 * 0.5);
```

Fig. 19.34 (*Cont.*)

```
        v = round (v - k2 + k3 * 0.5);
        k1 = k1 - 2 * k2 + k3;
        k2 = k3 - k2;
        tmp = dxsquare;
        dxsquare = -dysquare;
        dysquare = tmp;
    } else {        /* Octant is even */
        while (u < k2 * 0.5) {
            DrawPixel (x, y);
            if (d < 0) {
                x += dxdiag;
                y += dydiag;
                u += k2;
                v += k3;
                d += v;
            } else {
                x += dxsquare;
                y += dysquare;
                u += k1;
                v += k2;
                d += u;
            }
        }  /* while u < k2 * 0.5 */
        /* We now cross over square octant boundary. */
        d += u - v + k1 - k2;          /* Do v first; it depends on u. */
        v = 2 * u - v + k1 - k2;
        u += k1 - k2;
        k3 += 4 * (k1 - k2);
        k2 = 2 * k1 - k2;
        tmp = dxdiag;
        dxdiag = - dydiag;
        dydiag = tmp;
    }  /* Octant is even */
    octant++;
    if (octant > 8) octant -= 8;
    octantCount--;
}  /* while octantCount > 0 */
Having entered last octant, continue drawing until you reach the last pixel;
}  /* Conic */
```

Fig. 19.34 (*Cont.*)

```
/* Specify and draw an ellipse in terms of the endpoints P = (xp, yp) and */
/* Q = (xq, yq) of two conjugate diameters of the ellipse. The endpoints are */
/* specified as offsets relative to the center of the ellipse, assumed to be the */
/* origin in this case. */
void Conjugate (int xp, int yp, int xq, int yq, int mode)
{
    int xprod, tmp, xe, ye, A, B, C, D, E, F;

    xprod = xp * yq - xq * yp;

    if (xprod != 0) {                    /* If it is zero, the points are collinear! */
        if (xprod < 0) {
            tmp = xp; xp = xq; xq = tmp;
            tmp = yp; yp = yq; yq = tmp;
            xprod = -xprod;
        }
        A = yp * yp + yq * yq;
        B = -2 * (xp *yp + xq * yq);
        C = xp * xp + xq * xq;
        D = 2 * yq + xprod;
        E = -2 * xq * xprod;
        F = 0;

        if (mode == FULL_ELLIPSE) {      /* Set starting and ending points equal */
            xe = xp; ye = yp;
        } else {                         /* mode == ELLIPSE_ARC; draw only the arc between P and Q */
            xe = xq; ye = yq;
        }
        Conic (xp, yp, xe, ye, A, B, C, D, E, F);
    }  /* if */
}  /* Conjugate */
```

Fig. 19.34 The general ellipse algorithm.

display, however, we can simply draw *both* these pixels at half intensity. Thus, we can step by 2 pixels at a time, reducing the computation substantially. At the junctions of drawing octants, the algorithm must be tuned so that it does not overstep, but this is not too difficult. The algorithm is clearly suited only for gray-scale displays, but illustrates how certain algorithms may be simplified when antialiasing is possible. We shall see this again in our discussion of text.

19.2.7 Thick Primitives

We are now ready to discuss thick primitives: thick lines, thick polylines, and thickened general curves (thick lines, circles, and ellipse arcs were discussed in Chapter 3; here we extend and refine the ideas presented there). The first step in scan converting thick primitives is to decide what is actually in the primitive as a geometric object—that is, to

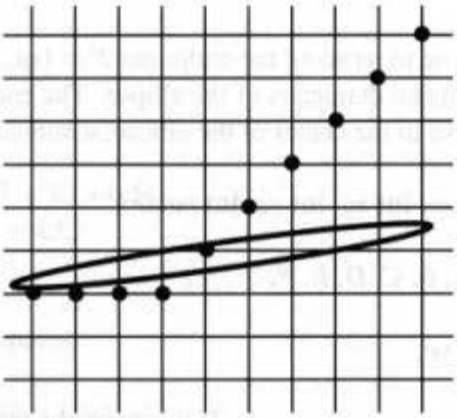


Fig. 19.35 The breakdown of the algorithm for a thin ellipse.

define the reference model for a thick object. For example (see Fig. 19.36), does a thick line look like a rectangle, or like a rectangle with semicircular endcaps, or like something else?

If we imagine the thick line as being drawn by a broad pen, we are in effect asking whether the pen has a flat or a round tip. More mathematically, we can describe the choice as one between the set of all points whose distance from the center line is less than one-half of the width of the thick line (this includes the rounded ends) and the set of all points whose distance from the center line along a normal (perpendicular) to the center line is less than one-half of the width of the thick line. A third mathematical description that will prove useful later is to consider two parallel copies of the center line, infinitely extended and then pushed off from the center line by the same amount in opposite directions. The thickened line consists of the area between these pushed-off lines, trimmed by two other lines through the endpoints of the original segment. If these trim lines are perpendicular to the original segment, the result is a rectangular thick line; if not, then the ends of the thick line are slanted (see Fig. 19.37). These slanted ends can be joined to make thick polylines; shaping the corners this way is called *mitering*.

A special case is to consider the line as being drawn with a pen with an aligned tip; that is, the tip is a line segment that is always either vertical or horizontal. A thick line drawn with such a pen looks like a rectangle if it is horizontal or vertical, and like a parallelogram otherwise. You should draw a few such lines, and determine why this is a bad way to draw thick lines. We mention this case only because it can be implemented quickly: We alter the midpoint line-drawing algorithm to place several pixels at once, rather than just a single one. Joints between horizontally and vertically drawn segments can be generated by extending the segments beyond the join points and then, in addition to the original segments, drawing the points lying in both extensions.

Fig. 19.36 A thick line may look like a rectangle or have rounded or slanted ends. Which is the correct version?

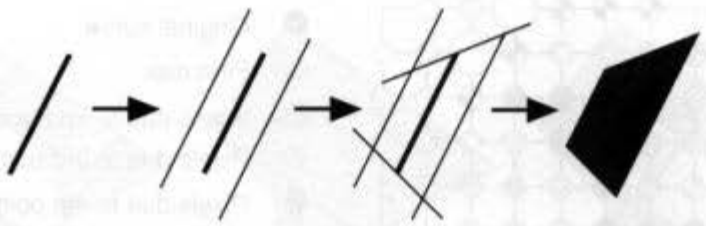


Fig. 19.37 We can define a wide class of thickened lines by pushing off parallel copies of a segment and trimming by arbitrary lines through the endpoints. These slanted ends can then be joined together to make thick polylines in a process called mitering.

Each of these classes of thick lines can be used to make thick polylines, as shown in Fig. 19.38. Butt-ended lines do not join well—they leave a notch at the joints (see Fig. 19.38a). Rounded-end lines join nicely—the exterior of the bend at each joint is a smooth curve, and the interior is a sharp angle (see Fig. 19.38b). Slanted-end lines (of the same thickness) join nicely only if the slant is the same on both sides—otherwise, the slanted edges of the two lines have different lengths. We can achieve the same slant on both sides by choosing the trim line to be midway between the two segments. If we miter-join lines that are nearly parallel, the miter point extends well beyond the actual intersection of the center lines (see Fig. 19.38d). This extension is sometimes undesirable, so trimmed miter joints are sometimes used to join lines. The trim on the miter is a line perpendicular to the miter line at some distance from the center-line intersection (see Fig. 19.38e).

These various methods of line joining are available in many drawing packages, including POSTSCRIPT, QuickDraw, and the X Windows System. Most such graphics packages provide not only a line-join style, but also a line-end style, so that, for example, we can use rounded ends but mitered joins for a polyline that is not closed.

There are several approaches to thickening general curves, including curves produced by freehand drawings or by other nonanalytic methods. To draw a thickened curve, we can use the method in Chapter 3 and simply convert the curve to many short line segments, each of which is then drawn thickened. If the segments are short enough and the resolution of the output device is high enough, the result may be adequate. It is important that the short line segments be constructed without rounding to integer coordinates (although the coefficients for the equations may be converted to integers by the methods discussed for scan converting general lines). If the endpoints of the segments were rounded to grid points, then there would be only a few possible choices for the segment slopes, and the curve would have a highly twisted shape.

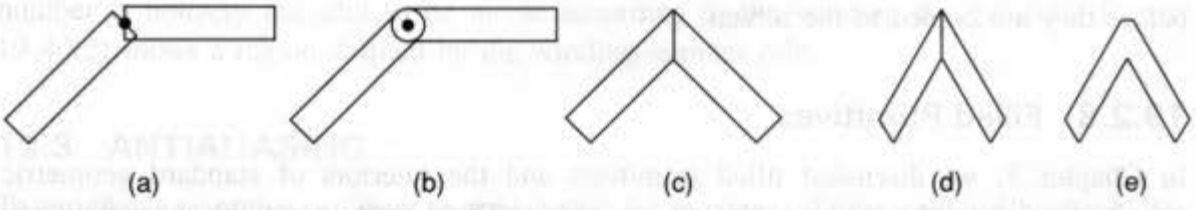


Fig. 19.38 Joining various classes of lines. (a) Butt-end lines mate poorly at joints. (b) Round-end lines mate nicely, but have curved segments, which may not be desired in some applications. (c) Mitered joints can be made by joining slant-end lines. (d) A miter at a joint where the angle is small may look bad. (e) Such joints can be trimmed.

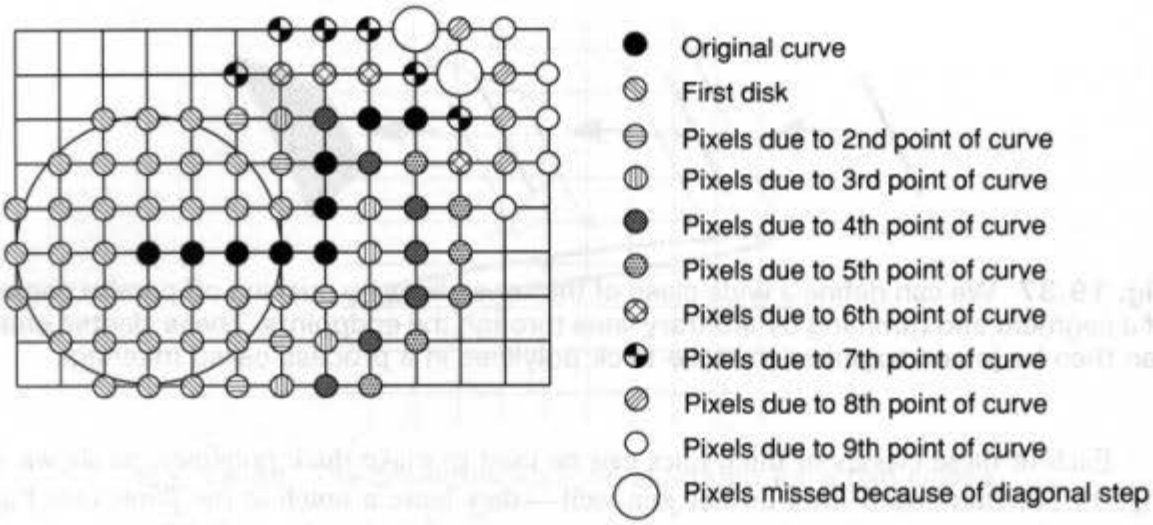


Fig. 19.39 A disk is drawn around the starting pixel of the thick line. After that, each pixel on the curve generates a semicircle. Notice the 2 pixels that were missed (shown as large circles in the figure).

A different approach is to draw the curve with a circular pen; Posch and Fellner [POSC89] describe a clever way to do this. Their method is of particular interest because it is designed for hardware implementation. The basic notion is first to scan convert the curve into a list of pixels that are adjacent diagonally, vertically, or horizontally (called *eight-way stepping*). This list of pixels is then expanded, so that any 2 pixels are adjacent horizontally or vertically (called *four-way stepping*). At the start of the curve, a filled circle (disk) is drawn. For each subsequent pixel, a half-circle (unfilled) is drawn, centered at the new pixel, and with its diameter perpendicular to the line from the previous pixel to the current one. Since there are only four possible directions of motion, there are only four possible half-circles to be drawn. Figure 19.39 shows the technique being applied to a curve using eight-way stepping instead, to show that, without the four-way stepping, some pixels are missed.

One difficulty with this method is that, for odd-thickness curves, the algorithm must be able to generate the points on a circle of half-integer radius centered on the original curve. These can be generated with a modified midpoint algorithm by multiplying the coefficients of the circle by 4. Another difficulty is that four-way stepping must be used to avoid gaps; unfortunately, this generates rougher curves. The algorithm also generates individual pixels instead of generating spans, so it is difficult to move the pixels to the screen quickly. The shape data structure described in Section 19.7 can be used to assemble the pixels into spans before they are copied to the screen.

19.2.8 Filled Primitives

In Chapter 3, we discussed filled primitives and the interiors of standard geometric primitives such as the circle, rectangle, or ellipse. These were reasonably unambiguous. The definition of interior for self-intersecting primitives is less obvious. We discussed two definitions for closed (possibly self-intersecting) polylines in Chapter 3. One is the even-odd or parity rule, in which a line is drawn from a point to some other point distant

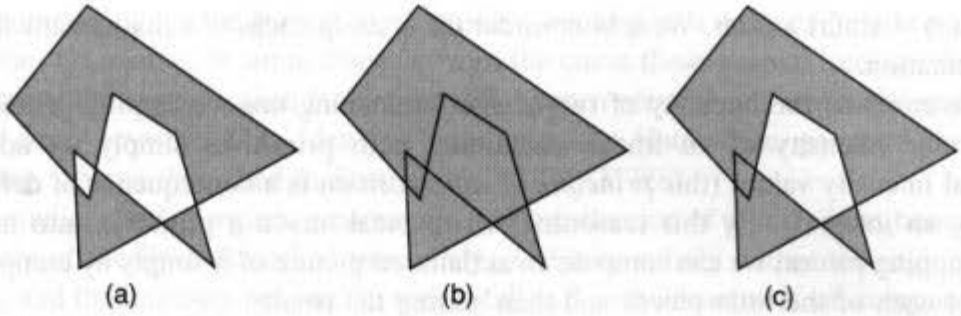


Fig. 19.40 (a) A polygon filled using the even-odd rule. (b) A polygon filled using the nonexterior rule. (c) A polygon filled using the nonzero winding rule.

from the polyline. If this line crosses the polyline an odd number of times, the point is *inside*, if not, it is outside. It is important that the test line pass through no vertices of the polyline, or else the intersection count can be ambiguous. The resulting interior region has a checkerboard appearance, as shown in Fig. 19.40(a). Another rule is the nonexterior rule, in which a point distant from the polyline is used. Any point that can be connected to this seed point by a path that does not intersect the polyline is said to be outside the polyline. The resulting interior region consists of everything one might fill in if asked to fill in whatever is within the curve. Put differently, if we think of the curve as a fence, the interior is the region within which animals can be penned up. See, for example, Fig. 19.40(b).

A third rule is the *nonzero winding rule*. The winding number of a point P with respect to a curve C that does not contain the point is defined as follows. Consider a point Q that travels once around C. The endpoint of a vector from P to Q, after normalization, travels along the unit circle centered at P. If we imagine the track of this endpoint as a rubber band, and let the band contract, it will end up wrapped about the circle some number of times. The winding number is the number of wraps (for clockwise wraps, the winding number is negative).

For closed polylines, this number can be computed much more easily. As before, we take a ray from the point out to a point far away from the polyline. The ray must hit no vertices of the polyline. If the ray is given parametrically as $P + t\mathbf{d}$, and $\mathbf{d} = \begin{bmatrix} x \\ y \end{bmatrix}$, we define $\mathbf{d}' = \begin{bmatrix} -y \\ x \end{bmatrix}$, which is just $\mathbf{d}$ rotated counterclockwise by 90°. Again, we count intersections of the ray with the polyline, but this time each intersection is assigned a value of $+1$ or -1 according to the following rule: If the dot product of the direction vector of the edge of the polyline with $\mathbf{d}'$ is positive, the value is $+1$, if it is negative, the value is -1. The sum of these values is the winding number of C with respect to P. Pixels for which the winding number is nonzero are said to be inside according to the winding-number rule. Figure 19.40(c) shows a region defined by the winding-number rule.

19.3 ANTIALIASING

To antialias a primitive, we want to draw it with fuzzy edges, as discussed in Chapter 3. We must take into account the filtering theory discussed there and in Chapter 14, where we saw that a signal with high-frequency components generates aliases when sampled. This fact indicates that we need to filter the signal before sampling it; the correct filter turned out to

be a $\text{sinc}(x) = \sin(x)/x$ filter. We now consider the consequences of a mathematical analysis of this situation.

If we compute the intensity of two images containing nonoverlapping primitives, we can find the intensity of an image containing both primitives simply by adding the individual intensity values (this *principle of superposition* is a consequence of defining the image by an integral). By this reasoning, if we break down a primitive into many tiny nonoverlapping pieces, we can compute an antialiased picture of it simply by computing the images of each of the little pieces and then adding the results.

Taking this approach to its logical limit, if we can draw an antialiased image of a single point, we can draw any antialiased primitive by representing it as a union of points, and summing (by an integral) the antialiased values for all these points to compute the values that should be drawn in the final image. We propose this not as a workable solution to the antialiasing problem, but as a motivation for various techniques. To avoid the technical difficulties of defining intensities for regions with infinitesimal areas, like points, we will instead speak of very small regions (dots).[6] What is the antialiased picture of such a dot? To compute the intensity at a pixel (x, y), we place a *sinc* filter over the pixel, and convolve it with a function ϕ that is 1 above the dot and 0 elsewhere. Mathematically, assuming the dot is at location (a, b) and the pixel is at (x, y), we get

$$I(x, y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \phi(t, s)\ \text{sinc}(\sqrt{(t - x)^2 + (s - y)^2})\ ds\ dt.$$

The integrand will be 0 except at points (t, s) that are close to (a, b); at those points it will be approximately $\text{sinc}(r)$, where r is the distance from (x, y) to (a, b)—that is, from the pixel to the dot—and the intensity at (x, y) is approximately proportional to $\text{sinc}(r)$. Thus, an antialiased dot (drawn in white on black) has a bright spot near its center and dim concentric rings at larger distances from the center. (These rings are related to the diffraction patterns produced when a light shines through a pinhole—see [BERK68].) Notice that, although $\text{sinc}(r)$ is sometimes negative, it is impossible to represent these values accurately, since we cannot draw a color darker than black. In the limit, as the dots get very small, the approximation to $\text{sinc}(r)$ becomes exact (although the values must be scaled by the inverse of the areas of the dots to prevent them from becoming 0).

If we now use this relation to antialias a line (which is just a collection of points), the result is a superposition of the individual antialiased patterns for the points[7]; the positive ripples from one point cancel the "negative" ripples from another, and the net result is a bright area near the line that fades gradually to black at greater distances. Furthermore, since a line drawn on a plane has a great deal of symmetry (it is invariant under translation along itself), the sum of the antialiased values is a function of the distance from the line and is independent of position along the line. This invariance suggests that, if we compute the profile of the intensity as we move away from the line along some perpendicular, we might be able to use the profile in a different way. Suppose the line is ever-so-slightly curved. If, at each point of the line, we lay down a copy of the intensity profile along the normal line to

[6]The mathematically inclined reader may use delta functions to make this precise.

[7]More precisely, the result is an integral of the contributions from the individual points.

the curve at that point, the sum of these profiles would give a decent filtered image of the curved line. Of course, at some distance from the curve these normal lines overlap, but if the line curvature is small, then the overlap will happen very far out, where the profile value is nearly zero anyway. This idea has been used by Hatfield to generate the general antialiasing scheme described in Section 19.3.3 [HATF89].

To antialias a filled region, we imagine it as a collection of points, or as being built up as a family of the lines described previously. The area solidly within the region has full intensity, and the intensity outside the region falls off with distance. If we compute what an antialiased halfplane looks like, we can antialias other regions by assuming that the edge of the region is (locally) very like a halfplane. In actual practice, however, we first draw antialiased lines or curves for the edges, then fill the region at full intensity.

19.3.1 Antialiasing Lines

In Chapter 3, we discussed the Gupta–Sproull method for generating antialiased lines. For each pixel near a line, the distance to the line is used to determine the brightness of the pixel. In an efficient implementation, this distance is converted to an integer value between 0 and some small number, say 16, and this number is used as an index into a table of gray-scale values. These gray-scale values were computed by determining the overlap between a conical filter and the region represented by the line. For lines of a fixed width, the possible overlaps of the filter and line can be computed a priori, but for general widths (and for rectangles, discussed in Section 19.3.5) it is preferable to compute the weighted overlap of the filter base with a halfplane. The overlap of the filter with a line can then be computed by subtracting its overlap with two slightly offset halfplanes (see Fig. 19.41).

Recall that the distance from the pixel center to the line must be *signed*, since the centers of a three-fourths–covered pixel and one-quarter–covered pixel are at the same distance from the side of the line. A particularly clever trick is available here: In the midpoint-line algorithm, the decision variable d determines whether the pixel center is on one side of the line or on the other. The value of this variable is closely related to the distance from the chosen pixel to the line. If we let $D(x, y) = ax + by + c$, then our decision variable at the point (x_i, y_i) is just $D(x_i, y_i - \frac{1}{2})$, and (x_i, y_i) lies on the line exactly when D is 0. Thus, the amount by which D fails to be 0 measures the signed distance from a (x_i, y_i) to the line. The implementation in Chapter 3 used exactly this fact to determine the distance to the line.

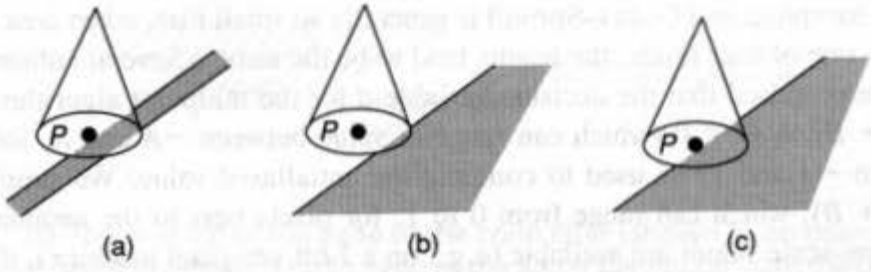


Fig. 19.41 The cone filter centered at P overlaps both sides of the thin line. We can compute the coverage of the filter over two halfplanes (b and c), one bounded by each side of the line, and subtract to obtain the actual line coverage (a).

A more complex indexing scheme is required to handle ends of lines. If the line is butt-ended, we must compute all possible overlaps of a corner of a square with the base of the cone filter, and we must store these in a table. This task is really part of a more general problem of drawing antialiased rectangles, and we postpone further discussion of it to Section 19.3.5 below.

The Gupta–Sproull method works for lines drawn in a foreground color on a background, but does not handle the case where a second line crosses the first. Consider two white lines drawn on a black background. If, during the drawing of the second line, the intensity for each pixel is computed as a mix of white with the background, two results are possible. If the current pixel value is used as the background intensity, then the point at which the lines cross will be overly bright. On the other hand, if the background intensity is taken to be black during the drawing of the second line, then points near the crossing of the two lines will not be bright enough.

The problem is compounded when color is introduced. First, a decision must be made about crossings. Is the color at the crossing point the color of the last line drawn? Or is it a mix of the color already there and the color last drawn? In essence, we must choose one of the compositing operations described in Chapter 17. Unfortunately, the assumption made in the Duff–Porter approach to compositing (that if a primitive covers a fraction α of a pixel, it covers a fraction α of any other primitive in that pixel) is false in the generic case of intersecting lines, so that using an α value compositing approach to choosing a color for the intersection pixel fails us here.

The Texas Instruments TI34020 chip provides a MAX operator that takes two pixel values, *src* and *dest*, and replaces *dest* with the larger of the two values. This use of the larger value to provide a value for pixels at intersections of antialiased colored lines has been a successful compromise.

An early paper by Barros and Fuchs [BARR79] describes a method for producing antialiased lines on a gray-scale device. The method is essentially a scan-line renderer for polygonal regions (which include 1-pixel-wide lines as a special case). All the polygons to be drawn are accumulated, and then the intensity of each pixel is computed by determining the portion of the pixel not covered by lines. A subdivision approach determines disjoint uncovered regions of each pixel, and the areas of such regions are then tallied.

Another approach to antialiasing of lines deserves mention, for the special case where the resolution of the device is very small, perhaps 2 bits (as in the NeXT machine). Here, the choice of pixel value is so coarse that Gupta–Sproull antialiasing is a waste of computation. A strict area-sampling technique works perfectly well instead. (The difference between area sampling and Gupta–Sproull is generally so small that, when area sampling is discretized to one of four levels, the results tend to be the same.) Several authors [PITT87; PITT80] have proposed that the decision variable d for the midpoint algorithm applied to the line $Ax + By + C = 0$ (which can range in value between $-A$ and B, for lines with slope between -1 and 1) be used to compute the antialiased value: We simply compute $(d + A)/(A + B)$, which can range from 0 to 1, for pixels next to the geometric line. If only a few gray-scale values are available (e.g., on a 2-bit-per-pixel machine), this approximation to the weighted overlap proves adequate. Because of its application to displays with few bits per pixel, this approach has been called *2-bit antialiasing*.

Finally, Hatfield's method [HATF89], described in detail in Section 19.3.3, can be applied rapidly to lines; in this case, it degenerates into an extension of a Gupta–Sproull scheme, in which the shape of the filter is defined at will.

19.3.2 Antialiasing Circles

Before we discuss antialiasing of circles, let us establish some terminology. A *disk* is a filled circle. A *circle* is the set of points at a fixed distance from a point in the plane, and hence is infinitely thin. When we refer to a *thick circle*, we mean the set of points whose distance to some center point lies between two values. The difference in the values is the thickness of the thick circle. Thus, a thick circle can be thought of as the (setwise) difference of a large disk and a slightly smaller one.

To antialias a circle drawn with a pen of unit thickness, we can do the following. For each point (x, y) near the circle, let $S(x, y)$ denote the decision variable in the circle algorithm. Then $S(x, y)$ is proportional to the signed distance of the point from the circle. This number, appropriately scaled, can be used as an index into a Gupta-Sproull-style look-up table, just as for lines. Recall that, with lines, we could take the overlap with two offset halfplanes and subtract. For circles, we can do a similar operation: We can compute the weighted overlap of the base of the filter with two concentric disks, and subtract the results to get the overlap with a thickened circle. With circles, however, there is a difference: The overlap of the base of the cone filter and a disk depends not only on the distance from the pixel to the edge of the disk, but also on the radius of the disk (see Fig. 19.42a).

We therefore need different tables for disks of different radii. Observe that, however, if we have a small region (i.e., a region contained in a disk of radius 2) near the edge of a disk whose radius is 10 pixels, as in Fig. 19.42(b), and another at the same distance from a halfplane, the coverage of the regions by the two primitives is similar (the difference is less than 0.004 for most distances). Thus, we can safely approximate all disks of radius 10 or greater as halfplanes, for the purpose of computing the pixel value in antialiasing, and can therefore use the intensity table developed in the straight-line antialiasing algorithm for

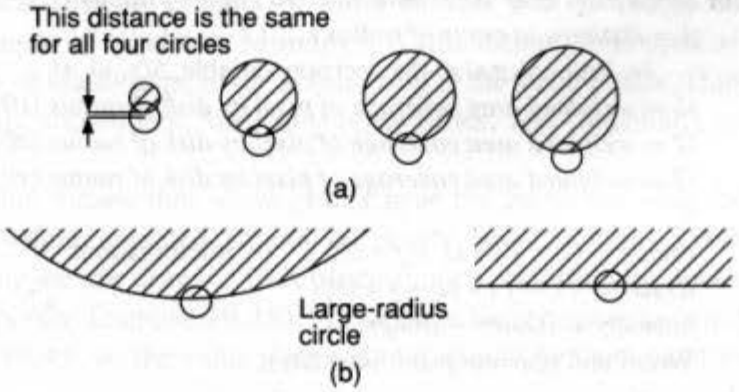


Fig. 19.42 (a) The overlap of the base of the cone filter (shown as an open circle) and a disk differs for disks of different radii, even at the same distances from the disks. (b) The overlap of the base of the filter (or any other small region) and a disk of radius 10 is about the same as the overlap of the filter base and a halfplane.

circles of large radii. Smaller circles need individual tables; one table for each integer-sized disk, with linear interpolation to fill in the gaps, gives adequate results.

Pseudocode for an antialised circle algorithm is shown in Fig. 19.43. The algorithm determines the intensity values for pixels near a unit-thickness circle. The method is to determine, for each pixel near the circle of radius r, by how much it overlaps a disk of radius $r + \frac{1}{2}$ and a disk of radius $r - \frac{1}{2}$, and to subtract the two overlap values to find the overlap with a circle of thickness 1 at radius r. Of course, all these overlap computations are *weighted*. The sole difficulty is that, if r is not an integer, the overlaps with the two disks must be linearly interpolated. For example, if $r = 6.25$, we need to know the overlap of the pixel (remember that the pixel represents a *round* area in weighted-area sampling) with the disk of radius 5.75 and with the disk of radius 6.75. We estimate the first number by computing the overlap with the disk of radius 5 and the disk of radius 6, then interpolating three-quarters of the way from the first to the second. The second number is estimated similarly.

Finally, to render a thick circle (say 5 or more pixels), we can draw two concentric circles and invoke a fill algorithm; for thinner circles, we can draw spans, as in Chapter 3.

```
/* Draw an antialiased circle of radius r. */
void AntialiasedCircle (double r)
{
    int r0;                          /* The integer part of (r + 1/2) */
    double f;                        /* The fractional part of (r + 1/2) */
    double i1,                       /* Intensity due to a disk of radius (r0 - 1) */
           i2,                       /* Intensity due to a disk of radius r0 */
           i3;                       /* Intensity due to a disk of radius (r0 + 1) */
    double iInner,                   /* Intensity due to a disk of radius (r0 - 0.5) */
           iOuter;                   /* Intensity due to a disk of radius (r - 0.5) */

    r0 = greater integer less than or equal to r + .5;
    f = 0.5 + r - r0;
    for (each pixel near the scan-converted circle of radius r) {
        d = distance to circle of radius r;
            /* Proportional to the decision variable, S(x, y) */
        i1 = weighted area coverage of pixel by disk of radius (r0 - 1);
        i2 = weighted area coverage of pixel by disk of radius (r0);
        i3 = weighted area coverage of pixel by disk of radius (r0 + 1);

        iInner = (1 - f) * i1 + f * i2;      /* Interpolations */
        iOuter = (1 - f) * i2 + f * i3;
        intensity = iOuter - iInner;
        WritePixel (current pixel, intensity);
    }
}  /* AntialiasedCircle */
```

Fig. 19.43 Pseudocode for the antialiased circle algorithm.

A somewhat simpler algorithm is based on two observations. The first is that the overlap of the filter with a thickened circle is (for circles with large enough radii) directly related to the distance between the center of the filter and the circle. The second is that this distance is directly related to the value of the residual at the filter center. It is easy to prove that, for a point (x, y) near the circle, the residual $F(x, y) = x^2 + y^2 - R^2$ is approximately $2Rs$, where s is the distance from (x, y) to the circle. Thus, by dividing the residual by twice the radius, we can compute the distance to the circle. This distance can be used as an index into a table of intensity values, and hence can be used to determine the intensities for points near the circle (see Exercise 19.26).

19.3.3 Antialiasing Conics

Considerable effort has been expended on the search for a good way to generate antialiased conics. Pitteway's 2-bit algorithm [PITT87] is fine for machines with a few bits per pixel, but does not extend well to larger numbers of bits per pixel; it makes a linear approximation to the weighted overlap that differs perceptibly from the correct value when displays with several bits per pixel are used. Field's circle and ellipse algorithm [FIEL86] can handle only circles and aligned ellipses with integer centers, and does only unweighted area sampling. If we can keep track of the curvature of the conic at each point, and if the curvature is not too great, then using a collection of Gupta–Sproull–style look-up tables for circles of the same curvature can be made to work, but at considerable cost.

Hagen [HAGE88] gives a general method for antialiasing lines, conics, and cubic curves that is based on using an antialiasing value related to the distance of a point from the curve as usual, but his mechanism for determining nearby pixels is clever. If we want to fuzz out a curve over several pixels, we must determine which pixels are close to the curve. When an incremental algorithm is used to compute the points of the curve, it is either stepping in a principally horizontal direction (choosing between the east and northeast pixels at each stage, for example), or in a principally vertical direction (e.g., choosing between the north and northeast pixels at each stage). When the curve-tracking algorithm is stepping in a vertical direction, we can find points near to the curve by stepping out horizontally, and vice versa. As described in Chapter 3, however, this process leaves "notches" in thick curves at the changes of quadrant. Moreover, if this technique is used to determine which pixels to antialias, it leaves gaps in the antialiasing at the same points. Hagen's solution is to widen horizontally sometimes, vertically other times, and diagonally at other times (see Fig. 19.44).

Of course, this means that some points near the curve are assigned intensity values twice. The two values assigned should be nearly equal, so this reassignment is not a problem (assuming we are drawing in **replace** mode).[8] A difficulty does arise, however, in tightly bent curves (see Exercise 19.18). A pixel can be very near to two different parts of a curve, as in Fig. 19.45, so the value assigned to it should be the *sum* of the values from the

[8]All the algorithms we present are best suited for drawing primitives on an empty canvas in **replace** mode. To draw multiple primitives, we can generally draw the primitive on a private canvas, and then copy it to the final canvas using a MAX operator or some compositing operation.

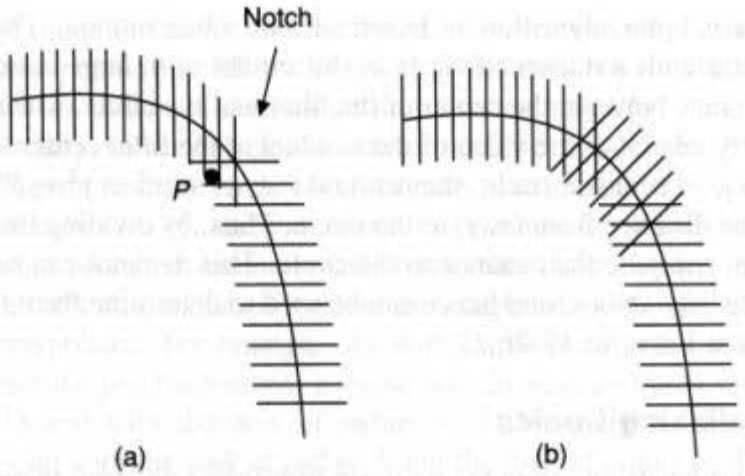


Fig. 19.44 Choosing where to compute antialiased values by determining which pixels are near a curve. If we always move either vertically or horizontally to determine nearby pixels, we get notches, as in (a). If we sometimes move diagonally, these notches can be filled in, as in (b).

two parts. Hagen notes this problem but presents no solution. Note that simply accumulating values in a pixel will not solve the problem: The pixel may be assigned a value twice either because of an overlap between a two different widenings of the curve (e.g., a pixel that is hit by both a horizontal and a diagonal widening, such as pixel P in Fig. 19.44), or because it is close to two different parts of the curve (e.g., pixel Q in Fig. 19.45).

Hagen's paper concludes by proposing a method for antialiasing of such curves using *width scans*: At each point, a line is drawn normal to the curve (using a scan-conversion algorithm for lines), and then an intensity is computed for each (sufficiently close) point along this normal line. This notion, described earlier, has been implemented and improved by Hatfield [HATF89]. He observes that, in scan converting a curve defined by an implicit function (such as $F(x, y) = x^2 + y^2 - R^2$ for the circle), when we compute the residue at each point P of the curve, we are computing various partial derivatives of the function. These in turn give the direction vector for a normal line to the curve.[9]

We can also find a good approximation to the place where a cubic curve crosses between two pixels by comparing the residuals at the pixels and doing a linear interpolation to infer where the residual is zero (see Fig. 19.46). If the residual is not zero at this point, iterating the procedure once will produce a very good estimate of the zero-crossing. (This technique is actually applicable to curves defined by higher-order polynomials as well.)

We can also compute, from the partial differences, a good approximation of the radius of curvature at each point. The formula for this is complex, but involves only the first and second derivatives of the implicit function F. The radius of curvature is the radius of the circle that best approximates the curve at the point; we therefore consider the curve to locally be a circle of that radius.

Thus, for each point on the scan-converted curve, we know the slope of the curve, the direction vector of the normal, the distance to the curve, and the radius of curvature of the

[9]Recall that we used the gradient in Chapter 3 to determine the normal vector.

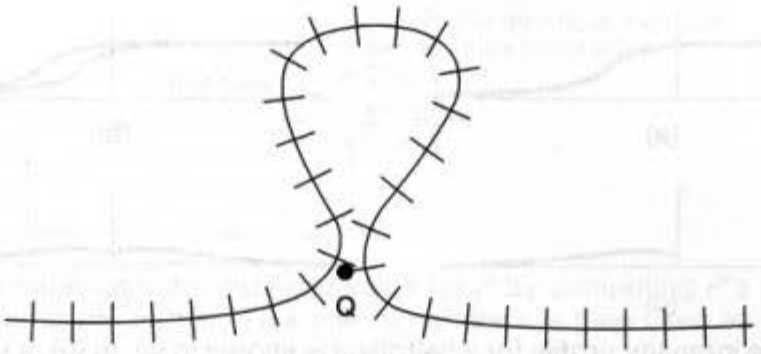


Fig. 19.45 On the inside of a tightly bent curve, points are close to two different pieces of the curve, and so should be brighter.

curve. Instead of horizontal, vertical, or diagonal stepping to widen the curve, we now scan convert the normal line to the curve, and compute the distance to the curve for each pixel. We use this distance as an index into an *intensity profile* describing the falloff of intensity as we move away from a circle whose radius is the current radius of curvature. To do exactly this would require computing (or providing by fine-tuning) an intensity profile for each possible radius of curvature, which is impractical. Fortunately, for radii greater than 10, Hatfield shows that assuming that the curve is straight at the point yields adequate results. For noninteger radii of curvature, indexing into adjacent (integer-radius) tables and interpolating can smooth out the results.

If we start with an intensity profile for an antialiased halfplane, we can compute the appropriate intensity profile for a narrow line or curve by shifting the profile slightly and subtracting. This corresponds to taking a halfplane and subtracting out a slightly shifted halfplane to generate values for a thin line. This technique works well for shifts as small as $\frac{1}{2}$ pixel (see Fig. 19.47). For shifts smaller than this, the lines look faint, but not thin. Of course, the total intensity of the pixels described by this profile curve is proportional to the area under the curve, so, to get comparably intense narrow curves, we must multiply the intensity profile by a number greater than 1. Hatfield reports that choosing the profile so as to maintain the intensity of the line without smearing is still more a matter of art than of science.

We can treat thick lines in a similar fashion, by simply adding up a collection of offset intensity profiles. For very thick curves, it is best to antialias the outside of the curve, and just to area fill the inside.

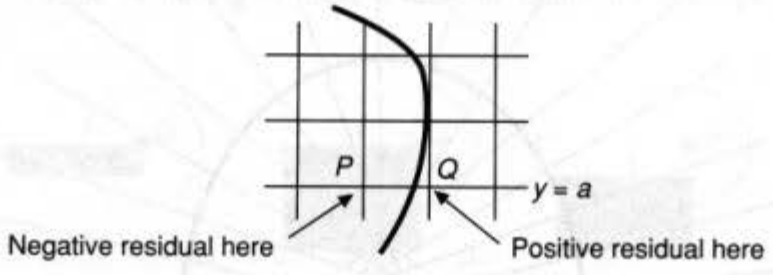


Fig. 19.46 The curve passes between the grid points P and Q, and hence the residuals at P and Q have opposite signs. If the residual at P is -10 and the residual at Q is 5, we can estimate that the curve crosses at a point two-thirds of the way from P to Q.

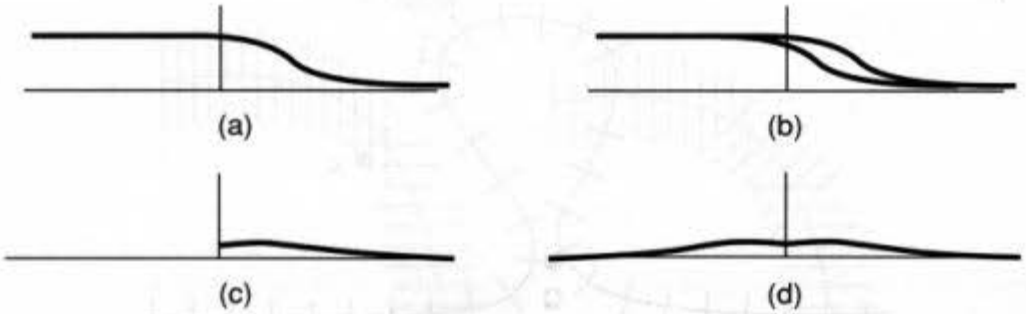


Fig. 19.47 The intensity profile for a halfplane is shown in (a). In (b), a second copy has been slightly displaced from it. In (c), the right half of the difference between them is shown; in (d), we see the full profile for a narrow line.

Note that the various lines along which the antialiasing is done may intersect if the width of the intensity profile is greater than the radius of curvature (see Fig. 19.48). The resulting intensities must be accumulated rather than just stored in the output pixmap.

19.3.4 Antialiasing General Curves

Methods for antialiasing general curves are few. One approach is to consider the curve as a sequence of very short straight lines and to use polygon antialiasing methods [TURK82; CROW78]. Another approach, described by Whitted [WHIT83], actually allows painting with an *antialiased brush*. The brush (a square array of pixels) is created at a very high resolution (it is 16 times as dense in each direction as the canvas into which the paint will be applied), and then is filtered to eliminate high-frequency components (the filter reduces the frequencies to one-sixteenth of the high-resolution sampling frequency; since this high-resolution brush eventually determines the values of pixels in a low-resolution bitmap, this filtering is necessary to avoid aliasing). The path along which the brush is dragged is also stored at high resolution (16 times the resolution in each direction). For each point on the path, those pixels in the high-resolution brush that correspond exactly to pixels in the

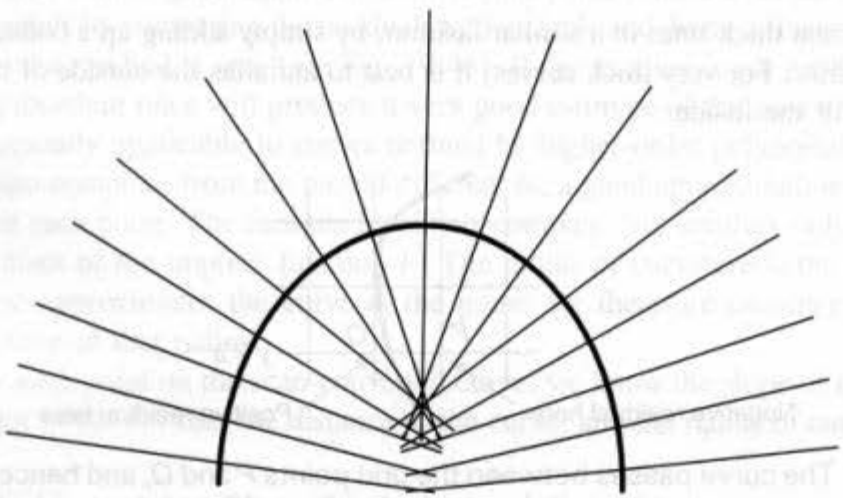


Fig. 19.48 When the radius of curvature is small but the intensity profile is wide, different points of the curve may contribute to the same output pixel.

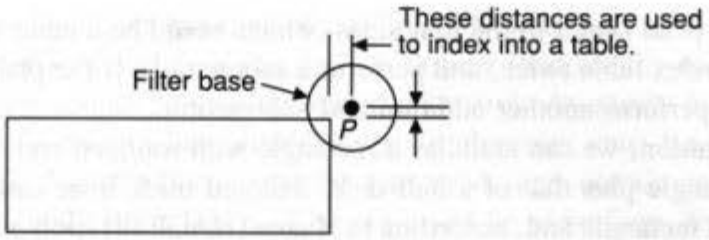


Fig. 19.49 We determine the intensity value for P by computing P's distances from each of the edges at the corner. This pair of numbers is then used as an index into a look-up table.

low-resolution output pixmap are copied. Furthermore, each pixel in the brush and the output image is given a z value, and a z-buffer algorithm is used to control pixel copying. If the brush is given any nonflat (e.g., hemispherical) profile, then it does not overwrite itself each time it is moved slightly. Furthermore, α values (coverage values) like those described in Section 17.6.1 are used to enhance the appearance of edges. The results of this technique are impressive (see Color Plate IV.2), but the computations for the image are costly, especially in terms of memory, since each image pixel needs R, G, B, and z values, and each brush pixel requires R, G, B, z, and α values. On the other hand, the preparation of brushes, although expensive, is done only once per brush, and catalogs of brushes can be stored for later use. In particular, brushes that draw lines of various widths can be generated so as to create smooth curves, although this technique becomes impractical for very wide curves.

19.3.5 Antialiasing Rectangles, Polygons, and Line Ends

Gupta and Sproull [GUPT81a] present a method for generating antialiased line ends as well as lines. To begin with, computing antialiasing values for pixels near the corner of a large rectangle can be done explicitly, just as it was for lines. In this case, however, it depends on two numbers—the distances to each of the nearby sides—instead of on just the distance to the line. The overlap of a rectangle and the filter base for a pixel are shown in Fig. 19.49. What if the rectangle is very thin, however, as in Fig. 19.50? The filter base for pixel Q hits three sides of the rectangle. We can compute an antialiasing value for Q by subtraction: We compute its filter base's weighted overlap with two rectangles whose difference is the small rectangle, and subtract the two numbers to get the weighted overlap with the small rectangle. Thus, instead of having to make a table of values for overlaps for all possible

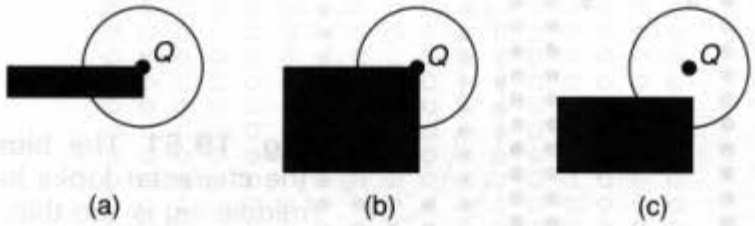


Fig. 19.50 The intensity value for a pixel near the end of a thin rectangle (or a thick line) can be computed by subtraction. Here, intensity for pixel Q in (a) is obtained by subtraction of (c) from (b).

distances from the pixel center to the four sides (which would be a table with four indices), we can use a two-index table twice, and perform a subtraction. If the pixel overlaps all four sides, we need to perform another addition and subtraction.

In a similar fashion, we can antialias a rectangle with rounded ends by computing the intensity of a rectangle plus that of a half-disk. Mitered thick lines can be treated by the methods used for a rectangle and, according to Hagen [HAGE88], polygons can be handled similarly. The intensity of a point near a corner of a polygon is computed by lookup into the table for the corner of a rectangle. If the angle is very different from 90°, the results are not particularly good. Hagen uses two additional tables, for 45° and 135° angles, as representative of acute and obtuse angles, and obtains good results with little effort.

19.4 THE SPECIAL PROBLEMS OF TEXT

The algorithms given so far have generated bitmap or antialiased versions of various geometric primitives, but we have not yet dealt with text. Text is a highly specialized entity, and our earlier techniques are usually not sufficient. In Chapter 3, we discussed using a font cache to store characters that could then be copied directly to the bitmap, but we also observed certain limitations of this approach: A different cache may be needed for each size of text, and the intercharacter spacing is fixed. Furthermore, although versions of bold or italic text can be created from this font cache, they are usually unsatisfactory.

Consider the letter "m." How do we create a bitmap version of an "m" from a precise description of it, given as a geometric drawing, by a font designer? Figure 19.51 shows a

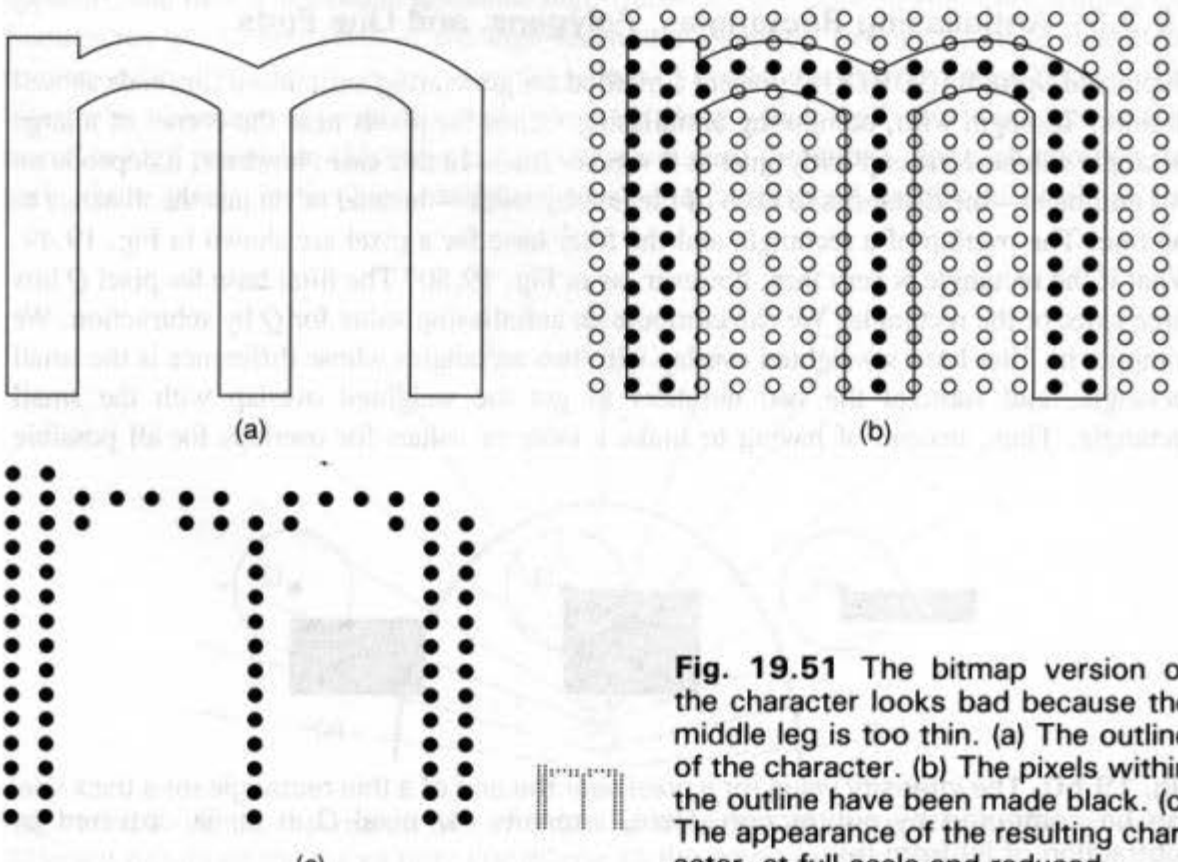


Fig. 19.51 The bitmap version of the character looks bad because the middle leg is too thin. (a) The outline of the character. (b) The pixels within the outline have been made black. (c) The appearance of the resulting character, at full scale and reduced.

drawing and a bitmap representation of an "m." Notice that, even with very careful scan-conversion, the middle leg of the "m" is thinner than are the outer two.

Thus, characters cannot be scan converted on a stroke-by-stroke basis. The geometry of character shape is deeply interrelated—certain elements must have the same width, certain topological characteristics must be preserved, and certain relationships among different characters in the same font must be preserved (in some fonts, for example, a width of a vertical segment of a capital "H" may need to be the same as the width of the hole in a capital "O"). Even if one manages to create characters with the right geometric relationships within and among then, there are artifacts that must be avoided. For example, in generating a capital "O" from a font-designer's plan, we must create two concentric oval arcs. When these are scan-converted to generate pixels representing the letter at some point size, the results can be disastrous. Figure 19.52 shows a letter "O" on a grid and the results of the scan-conversion process. The top of the "O" pokes just above a scan line, so that exactly 1 pixel on that scan line is turned on. The resulting character, which is said to have a *pimple*, is unattractive and difficult to read. Bitmap font-generating software must avoid such artifacts. Various vendors have developed rule-based software for satisfying geometric and typographical considerations such as these, but fonts must still be adjusted by hand, especially at very low resolutions. Such problems as these are only the most obvious. A great many others also arise, and addressing these is the task of the specialized field of *digital typography*; for further information, see [RUBE88].

In many ways, antialiased text is simpler than single-bit-per-pixel text. As long as the character is going to be antialiased, its location is irrelevant (antialiasing happens in the continuous, not in the discrete, world), so subpixel specification of character locations is possible, which is needed for high-quality output. Also, such artifacts as pimples, holes, and uneven-width stems disappear automatically. The problem that does remain is memory. Fonts are typically described by sequences of spline curves giving an outline for each character. Precomputing the appearance of all possible characters in a font (at several different subpixel locations, called *phases*) requires an immense amount of storage. Naiman and Fournier [NAIM87] estimate that, for a typical screen pixel density with a resolution of 8 bits per pixel, storing Roman, bold, italic, and bold-italic versions of two 128-character fonts at five point sizes and eight phases in both the vertical and horizontal directions requires over 50 MB of storage. Having 64 different phases may seem excessive to anyone

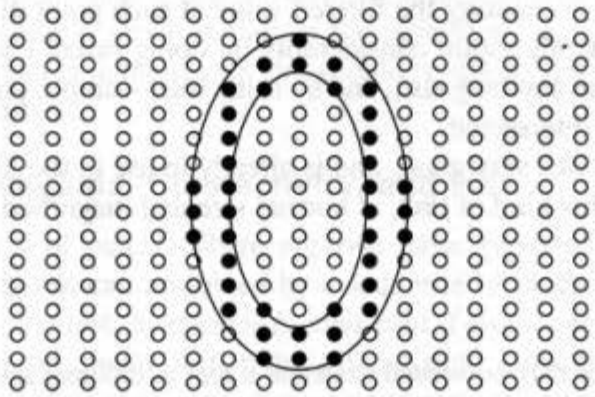


Fig. 19.52 Scan converting a letter "O" can generate a pimple at the top of the letter.

who has not tried to produce high-quality text output. But recall that we are talking about two fonts in only five sizes. Even ignoring phases, if we expand this to 10 fonts in 10 sizes, we soon approach the same 50 MB.

Another way to generate a character from such a family is to scale and translate the spline curves until they represent the character in the correct position,[10] and then to compute the antialiased image of the character at this position. This approach requires recomputing the antialiasing for each character every time it is used, and is computationally impractical unless clever methods for accelerating the process are discovered. Naiman and Fournier have developed just such a method. They decompose each character into subpieces much like the shape data structure described in the next section: each character is represented by a union of rectangles. This decomposition is done by scan converting the character at some very large point size to generate a *master* for it. This master is then broken into rectangles, and all smaller-sized characters are derived from the master by filtering.

At the most basic level, the filtered character is generated by convolving the master character (an $m \times m$ array of 0s and 1s, where m is typically much larger than the largest size character to be produced) with a filter function, which we represent as an $f \times f$ array of numbers whose sum is 1.0. The convolution is done by selecting an array of sample points within the master character array. If the output character is to be a $g \times g$ array (where g is less than m, of course), the sampling grid is also $g \times g$, and the space between samples is m/g. Notice that the sampling grid can be placed in m/g different positions in each of the horizontal and vertical directions (these offsets are the phases of the character). At this point, a copy of the $f \times f$ filter array is placed at each of the points in the sampling grid, and a value for the sample point is computed as follows: For each point in the filter array, the value of the filter is multiplied by the value in the master character bitmap at that point, and these results are summed. The resulting $g \times g$ array of samples is the filtered character (see Fig. 19.53).

Naiman and Fournier observe that computing the filtered value at each pixel is unnecessarily expensive. They represent the filter by a summed area table, as in Section 17.4.3; it is then easy to compute the filtered value of any rectangle. Since the master character is decomposed into rectangular pieces, they simply loop through all the rectangles in the master, compute overlaps with the filter box at the sample point, and determine the contribution to the intensity at the sample point by using summed area-table lookup. These intensities are accumulated over all rectangles in the master; the final result is the intensity of the pixel. Although this calculation must be done for each sample, it is still much faster on the average than is computing the filtered value at each point directly. We can improve the performance of the algorithm considerably by doing extent checking on the rectangle and the filter box before anything else, and by using inter–sample-point coherence in the list of rectangles that are intersected.

Antialiased text looks very good, and is already in use in the YODA display developed at IBM [GUPT86]. This kind of text, at normal size and magnified, is show in Color Plate IV.3.

---

[10]Slightly different curves may be needed to define a font at different sizes—scaling is not always sufficient.

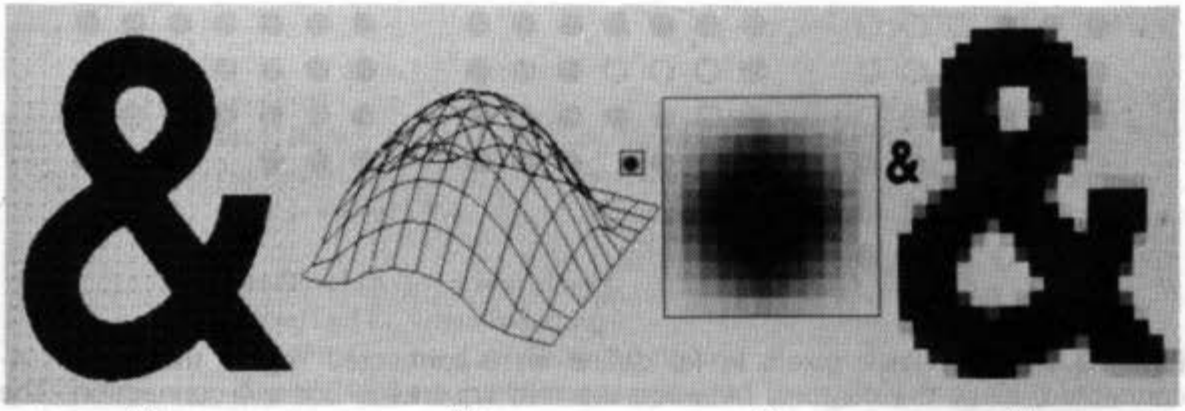


Fig. 19.53 (a) The master character overlaid with the sample grid. (b) A picture of the continuous filter. (c) A gray-scale representation of the filter, enlarged. (d) The filtered character, also enlarged. Courtesy of Avi Naiman, University of Toronto.

19.5 FILLING ALGORITHMS

Sometimes, after drawing a sequence of primitives, we may wish to color them in, or we may wish to color in a region defined by a freehand drawing. For example, it may be easier to make a mosaic pattern by creating a grid of lines and then filling it in with various colors than it is to lay down the colored squares evenly in the first place. Note that, when the first technique is used, no 2D primitives are drawn: We use only the 2D areas that happen to make up the background after the lines are drawn. Thus, determining how large a region to color amounts to detecting when a border is reached. Algorithms to perform this operation are called *fill algorithms*. Here we discuss *boundary fill*, *flood fill*, and *tint fill*. The last of these is a more subtle algorithm, being a type of *soft fill*. In it, the border of a region is determined not by the point at which another color is detected, but rather by the point at which the original color has faded to zero. Thus, a red region that fades to orange and then yellow (as might occur in the inside of a yellow circle drawn on a red canvas with antialiasing) can be converted to blue using tint fill. The orange areas are partly red, so their red component is replaced by blue; the result is a blue area, fading to green and bounded by yellow.

In all the filling algorithms we discuss, the value to be assigned to the interior pixels is called *newValue*. Following Fishkin and Barsky [FISH84], each algorithm can be logically divided into four components: a *propagation method*, which determines the next point to be considered; a *start procedure*, which initializes the algorithm; an *inside procedure*, which determines whether a pixel is in the region and should be filled; and a *set procedure*, which changes the color of a pixel.

19.5.1 Types of Regions, Connectivity, and Fill

A *region* is a collection of pixels. There are two basic types of regions. A region is *4-connected* if every 2 pixels can be joined by a sequence of pixels using only up, down, left, or right moves. By contrast, a region is *8-connected* if every 2 pixels can be joined by a sequence of pixels using up, down, left, right, up-and-right, up-and-left, down-and-right, or down-and-left moves. Note that every 4-connected region is also 8-connected.

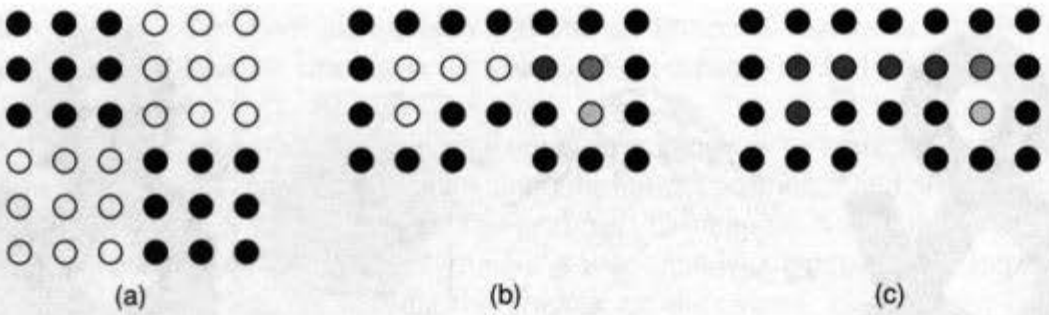


Fig. 19.54 The black pixels in (a) define an 8-connected region that is not 4-connected, since the diagonal between the two squares is not a 4-connection. The white pixels in (b) form the interior of the region if the region is interior-defined. The light- and dark-gray pixels are also in the interior if the region is boundary-defined by black pixels. If we try to fill this region with dark gray, starting at a white pixel, the result may be the one shown in (c), where the pixels on the extreme right were left unfilled because of the dark-gray pixel between them and the starting point.

A region can be defined in two ways. For each, we use a starting pixel, P. The *interior-defined* region is the largest connected region of points whose value is the same as that of the P. A *boundary-defined* region is the largest connected region of pixels whose value is *not* some given *boundary value*. Since most fill algorithms work recursively, and the recursion terminates when the next pixel already has the new color, problems may arise if the new value appears within a boundary-defined region, since some branch of the recursion may return prematurely (see Fig. 19.54).

Algorithms that fill interior-defined regions are called *flood-fill* algorithms; those that fill boundary-defined regions are called *boundary-fill* algorithms. Since both start from a pixel within the region, they are sometimes both called *seed-fill* algorithms.

19.5.2 The Basic Filling Algorithms

The most primitive propagation method is to move from the starting pixel in all four or all eight directions and to apply the algorithm recursively. For the FloodFill and BoundaryFill procedures given here, we have already determined the inside of the region, and the pixel-setting routine is just a call to WritePixel. Figure 19.55 presents the code for the 4-connected versions of the algorithms; the code for the 8-connected version has 8 recursive calls instead of four in each procedure.

These procedures, although simple, are highly recursive, and the many levels of recursion take time and may cause stack overflow when memory space is limited. Much more efficient approaches to region filling have been developed [LIEB78; SMIT79; PAVL81]. They require considerably more logic, but the depth of the stack is no problem except for degenerate cases. The basic algorithm in these approaches works with spans. These spans are bounded at both ends by pixels with value *boundaryValue* and contain no pixels of value *newValue*. Spans are filled in iteratively by a loop. A span is identified by its rightmost pixel; at least one span from each unfilled part of the region is kept on the stack.

The algorithm proceeds as follows. The contiguous horizontal span of pixels containing the starting point is filled. Then the row above the just-filled span is examined from right to left to find the rightmost pixel of each span, and these pixel addresses are

```
void FloodFill4 (
        int x, int y,                   /* Starting point in region */
        color oldValue,                 /* Value that defines interior */
        color newValue)                 /* Replacement value, must differ from oldValue */
{
    if (ReadPixel (x,y) == oldValue) {
        WritePixel (x, y, newValue);
        FloodFill4 (x, y - 1, oldValue, newValue);
        FloodFill4 (x, y + 1, oldValue, newValue);
        FloodFill4 (x - 1, y, oldValue, newValue);
        FloodFill4 (x + 1, y, oldValue, newValue);
    }
} /* FloodFill4 */

void BoundaryFill4 (
        int x, int y,                   /* Starting point in region */
        color boundaryValue,            /* Value that defines boundary */
        color newValue)                 /* Replacement value */
{
    color c = ReadPixel (x,y);
    if (c != boundaryValue &&           /* Not yet at the boundary... */
        c != newValue) {                /* Nor have we been here before... */
        WritePixel (x, y, newValue);
        BoundaryFill4 (x, y - 1, boundaryValue, newValue);
        three other cases;
    }
} /* BoundaryFill4 */
```

Fig. 19.55 The flood-fill and boundary-fill algorithms.

stacked. The same is done for the row below the just-filled span. When a span has been processed in this manner, the pixel address at the top of the stack is used as a new starting point; the algorithm terminates when the stack is empty. Figure 19.56 shows how a typical algorithm works. In Fig. 19.56(a), the span containing the starting point has been filled in (the starting point is shown with a hollow center), and the addresses of numbered pixels have been saved on a stack. The numbers indicate order on the stack: 1 is at the bottom and is processed last. The figure shows only part of the region-filling process; we encourage you to complete the process, step by step, for the rest of the region (see Exercises 19.19 and 19.20).

This algorithm can be further improved by avoiding redundant examinations of adjacent scan lines [SMIT79]. If, after a scan line has been filled, the line just above it is scanned for new span, and all the new spans have extents that lie entirely within the extents of the current scan line's spans (called the *shadow* of the current scan line), then during

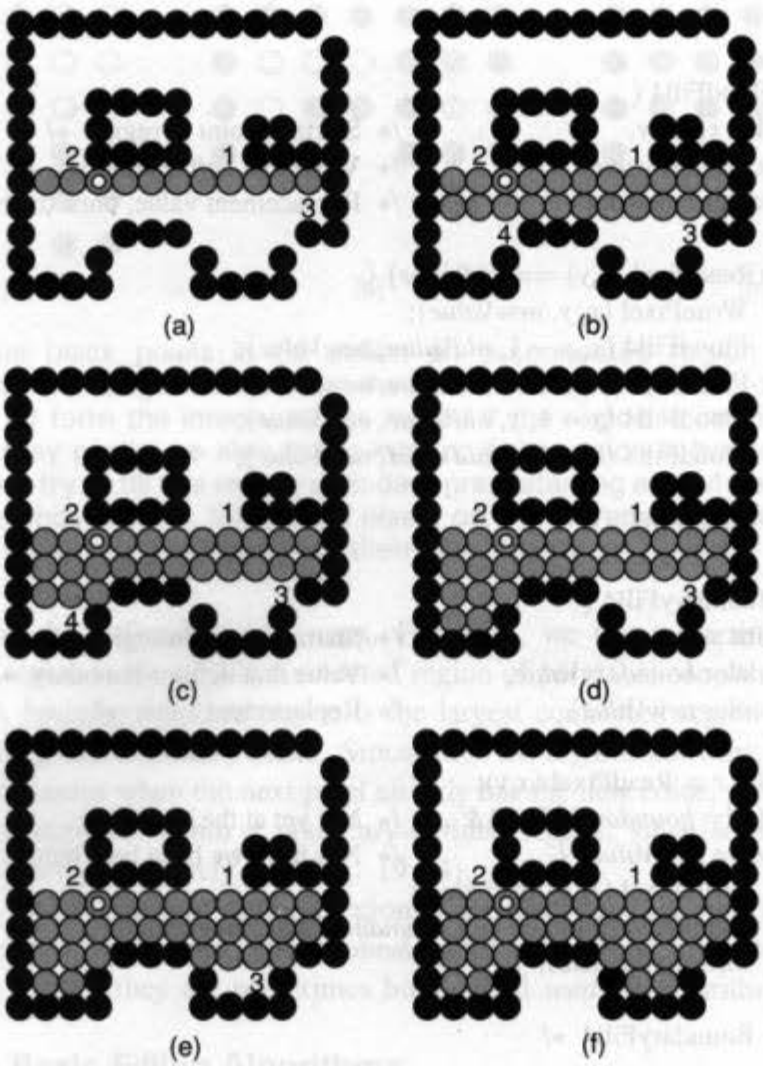


Fig. 19.56 The progress of the recursive fill algorithm. (a) The region with one span of pixels filled. The starting pixel has a hollow center. (b) through (f) Filling of subsequent spans.

processing of the upper scan line, the one beneath it does not need to be scanned for new spans. Also, the process of scanning a line for the spans can be combined with the process of filling the pixels, so multiple reads of each pixel are not necessary.

Better still, the entire process of scanning the lines above and below the current line can be accelerated at the cost of slightly greater memory use. After filling the current span of pixels, the line above is searched for a pixel that is connected to the current span, and this seed pixel is pushed onto the stack, *along with the endpoints of the current span* (henceforth called the *parent span*). The fill procedure is then invoked by starting at the seed pixel. If, after the fill procedure, the span that was filled does not extend past the ends of the parent span, scanning of the shadow of the parent span continues until another seed pixel is found, and this seed is used for starting a fill. In many cases, the span above the current span will be as large or larger than the current span, so that scanning for additional seed points is unnecessary [LEVO82].

19.5.3 Soft-Filling Algorithms

Soft filling is used to fill a region whose edge is blurred for some reason, typically antialiasing. It has been investigated by several authors [LEVO78; SMIT79], and Fishkin and Barsky have greatly extended Smith's techniques [FISH84]. We assume that the region is initially rendered in some foreground color against some other background color. If the region to be filled has an associated α value, as in Section 17.6.1, we can use it to detect the points inside the region and determine how they should be set: The *newValue* is mixed with the background color according to the fraction α, as is discussed in [LEVO78]. But many regions may require recoloring even though an α value is unavailable. Following Fishkin and Barsky, let us make three assumptions:

1. The region is rendered in a foreground color, F, against a background color, C. Each pixel in the image is a convex combination of F and C; that is, $P = tF + (1 - t)C$.
2. We have a region-traversal algorithm that visits each point once. The improved seed-filling algorithm can be tuned to do this; alternatively, we can mark each visited pixel with a flag.
3. The colors F and C are known and are not equal.

The basic algorithm for SoftFill is shown in Fig. 19.57.

Only the second step of this algorithm is difficult. Assuming that the colors are expressed as RGB triples, and writing $F = (F_R, F_G, F_B)$, and similarly for C and P, we know that, for some t, the following equations hold (by assumption 1):

$$P_R = tF_R + (1 - t)C_R, \qquad P_G = tF_G + (1 - t)\,C_G, \qquad P_B = tF_B + (1 - t)\,C_B.$$

If $F_R \neq C_R$, we can solve for t using just the first equation. If not, we can use the second, or even the third. Two problems arise. What if all six values on the right-hand sides

```
void LinearSoftFill (
        regionType region,
        color F,        /* Foreground color */
        color C,        /* Background color */
        color N)        /* New foreground color */
{
    for (each pixel in the region) {
        color P = color value for the pixel;
        find t so that P = tF + (1 - t) C;
        replace P with tN + (1 - t) C;
    }
} /* LinearSoftFill */
```

Fig. 19.57 The basic soft-fill algorithm.

are pairwise equal? What if the t values obtained by solving the three equations differ? The first situation occurs only if F and C are identical, which is ruled out by assumption 3. (Note that this assumption is entirely natural: If you were given a picture of a polar bear in a snowstorm and were asked to redraw the polar bear in brown, how would you know where the bear stopped and the snow began?) The second situation is more serious. In the strictest mathematical sense, it cannot happen, because P was formed as a linear combination of F and C. Nonetheless, in an integer world (which is how color values are typically expressed), there is roundoff error. The best inference of the value of t will come from the equation with the greatest difference between the F and C values. Thus, a slightly more robust algorithm is given in Fig. 19.58.

This algorithm has the disadvantage that it allows filling against a single background color only. We would like to be able to fill a picture of a frog sitting on a red-and-black checkerboard. Fishkin and Barsky [FISH84] address this problem by using some linear algebra, as follows. In an n-dimensional linear space, any sufficiently general $n + 1$ points determine a coordinate system. (All that is required is that no $(n - 1)$–dimensional affine subspace contain them all. For example, in 3D, four points are sufficiently general if no plane contains all of them.) If the points are $v_0, v_1, \ldots, v_n$, then any point p in the space

```
void LinearSoftFill (
        regionType region,
        color F,        /* Foreground color */
        color C,        /* Background color */
        color N)        /* New foreground color */
{
    int i;
    int d;

    /* Initialization section */
    find the i that maximizes |F_i−C_i| over i = R, G, B;
    d = |F_i−C_i|;

    /* Inside Test */
    for (each pixel) {
        color P = color value for the pixel;
        int t = (P_i − C_i) / d;
        if (t > some small value) {
            /* Setting pixel value */
            replace P with tN + (1 − t) C;
        }
    }
}   /* LinearSoftFill */
```

Fig. 19.58 The more robust soft-fill algorithm.

can be written uniquely as a combination

$$p = v_0 + t_1(v_1 - v_0) + t_2(v_2 - v_0) + \ldots + t_n(v_n - v_0),$$

where the t_i are real numbers. This is typically done with v_0 being the origin of the space and $v_1, \ldots, v_n$ being a basis. By assuming that pictures drawn in several colors have pixels whose values are *linear* combinations of those colors, Fishkin and Barsky observe that each pixel value lies in an affine subspace of the color space. In the case of a foreground color drawn against a background color, this subspace is a line—the line consisting of all color triples between the foreground color and the background color in RGB space. If the foreground color is drawn against two background colors, then the subspace is a plane determined by the locations of the three colors in RGB space (unless the three colors all lie on a line, a degenerate case; this is an example of a set of points being insufficiently general). If the foreground color is drawn against three background colors, then the subspace is the entire RGB space, unless all four lie in a plane (another degeneracy). Notice that the analysis can go no further: *Any* five points in RGB space lie in a 3D affine space, and hence constitute a degenerate case. (If the colors are represented by n spectral samples, where $n > 3$, then more complex cases can be handled by similar methods.)

So let us consider the case of a foreground color, F, drawn against two background colors, C and D. Each pixel in the image is a convex combination of these:

$$P = t * F + s * C + (1 - t - s) * D.$$

The problem is to determine the values of t and s, so we can replace the pixel with the color

$$P = t * N + s * C + (1 - t - s) * D.$$

Writing the first equation in terms of R, G, and B components yields three simultaneous equations in two unknowns. Just as in linear fill, we can use any two of these to determine s and t. If two of the vectors $F - C$, $F - D$ and $C - D$ are close to parallel, however, then using the corresponding two equations produces greater roundoff error. The other problem with linear fill was the possibility that the system of equations had no solution. The analogous problem in this case occurs when the points F, C, and D are colinear in color space. Thus, for the algorithm to be effective, it is important that the colors be in general position in RGB space (i.e., that no three of them lie on a line in that space). This requirement can be seen intuitively as follows. Imagine a background composed of various shades of red, pink, and white, all of which are convex combinations of red and white. Now imagine a foreground image drawn in a salmon color—pink with just a little yellow. It is difficult for us to determine where the salmon ends and the pink begins. The algorithm has troubles making this type of discrimination as well.

We conclude with an extension that does not appear to be in the literature. If a foreground object is rendered in two colors on a background of another two colors, we can recolor the foreground object in a different two colors. Thus, we can change a red-and-green checkerboard lying on a zebra into a yellow-and-orange checkerboard. The mechanism is identical to the four-color extension of the fill algorithm just described. Suppose the two

foreground colors are E and F and the background colors are C and D. We can solve the equation

$$P = rE + sF + tC + (1 - r - s - t)D$$

for the values r, s, and t (provided the colors C, D, E, and F are in general position). If we now wish to replace E and F by M and N, we simply set

$$P = rM + sN + tC + (1 - r - s - t)D.$$

An application of this mechanism is the following. A red sphere illuminated by a white light is rendered with careful shading, so the resulting object has smoothly varying colors that are combinations of red and white. This image is composited onto a blue-and-green checkerboard. We now decide that the sphere should have been a bluish-green sphere illuminated by a greenish-blue light. We can perform the preceding operation and generate the new image. Notice that this cannot be done by two applications of the more basic fill algorithm—after one substitution, the general-position condition no longer holds.

19.6 MAKING copyPixel FAST

In all our scan-conversion algorithms, we have tried to maintain scan-line coherence, since copying pixels (or writing pixels in any mode) is fastest when done for many pixels on a scan line at the same time. In the important case of 1-bit graphics, it is also a lot easier to copy a whole word's worth of bits at once. Setting individual bits in a word is almost as expensive as setting all the bits in the word. Thus, in a screen memory organized by words, doing a pixel-by-pixel operation is about n times slower than is doing a word-by-word operation, where there are n bits per word. (Even with 8-bit color, a 32-bit processor can copy 4 pixels at a time.) In discussing this operation in Chapter 3, we simply used SRGP_copyPixel, saying that its implementation was system-dependent. In general, a copyPixel procedure, when applied to 1-bit images, is known as bitBlt (for "bit block-transfer"); in this section, we discuss optimizing the bitBlt procedure, following Pike's description of a fast bitBlt routine for an MC68000 microprocessor [PIKE84].

We want to implement a bitBlt function that supports clipping to a rectangle, arbitrary write modes, and texturing. Texturing is essentially patterning during bitBlt. A window manager can use texturing when a window becomes inactive. By texturing the window with a stipple texture, the window manager can inform the user that the window must be activated before it can be used. Doing this stippling in the course of a bitBlt is much faster than is redrawing all the primitives in the window with a stipple pattern. Thus, we want to define a procedure as shown in Fig. 19.59.

The region to be copied is a rectangle of the same size as *rect* with origin at *pt* in the source bitmap. The target region is the one specified by *rect* in the *destination* bitmap. The texture is a $w \times w$ array of bits, where w is the size of a word on the machine. We assume that w is 32 and that a texture is actually represented by an array of 32 words.

Implementing this procedure is straighforward, but involves a fair number of cases. If the texture is all 1s, we want to avoid applying it. If either the source rectangle or destination rectangle lies partially or completely outside the corresponding bitmap, we want to avoid accessing those regions. The two bitmaps may be the same, and the source and

```
void bitBlt(
        bitmap source,          /* Source bitmap */
        point pt,               /* Corner of region to be copied */
        texture tex,            /* Texture to apply during copying */
        bitmap destination,     /* Target bitmap */
        rectangle rect,         /* Location of target region */
        writeMode mode);
```

Fig. 19.59 The procedure declaration for bitBlt.

destination rectangles may overlap, so we must do our copying in a nondestructive order. And finally, each write mode may require special handling, since some may be easily implemented by single machine instructions, whereas others may require multiple instructions, especially when they operate on partial words.

It is fortunate that C provides direct access to processor memory, since the bitBlt operation involves operations on individual bits, of course, and doing this as directly as possible can vastly increase the efficiency. The first version of the algorithm that we'll describe uses bit-shifting, bit-masking, pointer arithmetic, and pointer comparison extensively, and by doing so achieves a *reasonable* speed—one which would be impossible in languages that did not provide such direct memory access. Nonetheless, as we'll see, even this is not fast enough for practical use, and the ultimate version of bitBlt must be implemented in assembly language. This represents a standard design trade-off: that for the core of a time-critical system, one must make things as efficient as possible, while operations at the periphery may often be less efficient but more easily maintained or modified.

The basic data structure is the bitmap. To take advantage of word-by-word operations, we must arrange the bitmap as an array of words. We also want to be able to create bitmaps whose size is not a multiple of 32. Our bitmaps are therefore records with three fields: a pointer to an array of words (see Fig. 19.60); a rectangle describing a subset of the bits

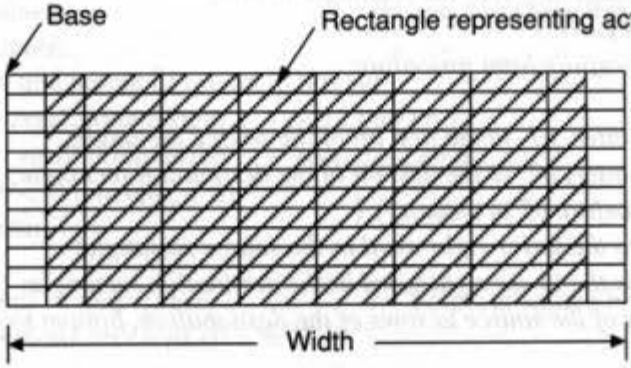


Fig. 19.60 The bitmap is an array of words, containing a rectangle that represents the actual bits of the bitmap.

represented by that array; and an integer, *width*, that says how many bits there are in each group of words storing a row of the rectangle. If the edges of the rectangle are aligned with the word boundaries, this value is just the width of the rectangle in pixels; if not, it is somewhat greater. Figure 19.61 gives the pseudocode for the basic bitBlt algorithm. Unfortunately, the actual code is quite a lot messier. The basic bitBlt code in Pascal is shown in Fig. 19.62.

Pike remarks that a C version of this code takes about 8 minutes to scroll an 800- by 1024-pixel bitmap horizontally, on an 8-MHz MC68000 processor. He also describes several improvements. The first is to eliminate the many calls to MoveBit by constructing a word of bits and moving it all at once, using word-at-a-time logical instructions available in C and assembler. This change gives a speedup of a factor of 16—the same scrolling takes only about 30 seconds.

When we bitBlt a large region on the screen with the basic code, we call the MoveBit routine hundreds of thousands of times. Even if we move only words, we still must do so thousands of times. Executing the smallest possible amount of code in this inner loop is thus extremely desirable. Notice that, when the source and destination bitmaps have the same offsets, much of the bit shifting can be avoided. If regions are to be copied off of and onto some bitmap (typically the screen), keeping the offscreen bitmaps with the same offset as their onscreen correspondents can save a great deal of time.

Since moving large regions on the screen is an extremely common activity, accelerating this process in any way possible is worthwhile. Since all that is really involved is detecting this special case and then calling (repeatedly) the assembler instruction to move a 32-bit word, indirectly, and to increment pointers, we can do nothing more than to hope that the instruction is very fast.

To accelerate bitBlt substantially, we must move to assembler language. The trick here is to look at the regions moved and the mode used, and to determine from these whether to scan left to right or right to left, whether to scan top to bottom or bottom to top, whether any rotation of bits is necessary, and so on. From this information, a fast bitBlt routine can generate optimal assembler code to execute the actual bitBlt, and then can jump to that

```
clip the rectangles to the source and destination bitmaps;
find the width and height of the rectangle in bits;
if (either is negative)
    return gracefully from procedure;

compute a pointer, p1, to the first bit of the source rectangle;
compute a pointer, p2, to the first bit of the destination rectangle;
if (p1 comes before p2 in memory) {
    move p1 to the lower right word in the source rectangle;
    move p2 to the lower right word in the destination rectangle;
    copy rows of the source to rows of the destionation, bottom to top;
} else
    copy rows of the source to rows of the destination, top to bottom;
```

Fig. 19.61 Pseudocode for the basic bitBlt algorithm.

```
typedef struct {                        /* This is not the rectangle type of Chapter 2 */
    point topLeft, bottomRight;
} rectangle;

typedef struct {
    char *base;                         /* Pointer to first word of memory for the bitmap */
    int width;
    rectangle rect;
} bitmap;

typedef struct {
    unsigned int bits:32;
} texture;

typedef struct {
    char *wordptr;
    int bit;
} bitPointer;

void bitBlt(
        bitmap map1,                    /* Source bitmap */
        point point1;                   /* Corner of region to be copied */
        texture tex;                    /* Texture to apply during copying */
        bitmap map2;                    /* Target bitmap */
        rectangle rect2;                /* Location of target region */
        writeMode mode)
{
    int width;
    int height;
    bitPointer p1, p2;

    /* Clip the source and destination rectangles to their bitmaps. */
    clip x-values;                      /* See procedure that follows */
    clip y-values;
    /* width and height of region in bits */
    width = rect2.bottomRight.x - rect2.topLeft.x;
    height = rect2.bottomRight.y - rect2.topLeft.y;
    if (width < 0 || height < 0)
        return;

    p1.wordptr = map1.base;             /* Points at source bitmap */
    p1.bit = map1.rect.topLeft.x % 32;
```

Fig. 19.62 (*Cont.*)

```
/* And the first bit in the bitmap is a few bits further in */
/* Increment p1 until it points to the specified point in the first bitmap */
IncrementPointer (p1, point1.x - map1.rect.topLeft.x + map1.width *
                  (point1.y - map1.rect.topLeft.y));

/* Same for p2—it points to the origin of the destination rectangle */
p2.worldptr = map2.base;
p2.bit = map2.rect.topLeft.x % 32;
IncrementPointer (p2, rect2.topLeft.x - map2.rect.topLeft.x +
                  map2.width * (rect2.topLeft.y - map2.rect.topLeft.y));
if (p1 < p2) {
    /* The pointer p1 comes before p2 in memory; if they are in the same bitmap, */
    /* the origin of the source rectangle is either above the origin for the */
    /* destination or, if at the same level, to the left of it. */
    IncrementPointer (p1, height * map1.width + width);
    /* Now p1 points to the lower-right word of the rectangle */
    IncrementPointer (p2, height * map1.width + width);
    /* Same for p2, but the destination rectangle */
    point1.x += width;
    point1.y += height;
    /* This point is now just beyond the lower right in the rectangle */
    while (height-- > 0) {
        /* Copy rows from the source to the target bottom to top, right to left */
        DecrementPointer (p1, map1.width);
        DecrementPointer (p2, map2.width);
        temp_y = point1.y % 32;        /* Used to index into texture */
        temp_x = point1.x % 32;
        /* Now do the real bitBlt from bottom right to top left */
        RowBltNegative (p1, p2, width, BitRotate (tex[temp_y],temp_x),mode);
    } /* while */
} else {      /* if p1 ≥ p2 */
    while (height-- > 0) {
        /* Copy rows from source to destination, top to bottom, left to right */
        /* Do the real bitBlt, from top left to bottom right */
        RowBltPositive (same arguments as before);
```

Fig. 19.62 (*Cont.*)

code. The idea is that this assembler code is so brief and efficient that it can be cached and executed very quickly. For example, the screen-scrolling described previously is executed in 0.36 seconds—an impressive speedup! The assembly code generated to scroll the whole screen is a masterpiece of compactness: It consists of only eight instructions (assuming that various registers have been set up beforehand).

```
            increment pointers;
        } /* while */
    } /* else */
} /* bitBlt */

void ClipValues (bitmap *map1, bitmap *map2, point *point1, rectangle *rect2)
{
    if (*point1 not inside *map1) {
        adjust *point1 to be inside *map1;
        adjust origin of *rect2 by the same amount;
    }
    if (origin of *rect2 not inside *map2) {
        adjust origin of *rect2 to be inside *map2;
        adjust *point1 by same amount;
    }
    if (opposite corner of *rect2 not inside *map2)
        adjust opposite corner of *rect2 to be inside;
    if (opposite corner of corresponding rectangle in *map1 not inside *map1)
        adjust opposite corner of rectangle;
} /* ClipValues */

void RowBltPositive(
    bitPtr p1, bitPtr p2,              /* Source and destination pointers */
    int n,                             /* How many bits to copy */
    char tword,                        /* Texture word */
    writeMode mode)                    /* Mode to blt pixels */
{
    /* Copy n bits from position p1 to position p2 according to mode. */
    while (n-- > 0) {
        if (BitIsSet (tword, 32))      /* If texture says it is OK to copy... */
            MoveBit (p1, p2, mode);    /* then copy the bit. */
        IncrementPointer (p1);
        IncrementPointer (p2);
        RotateLeft (tword);            /* Rotate bits in tword to the left. */
    } /* while */
} /* RowBltPositive */
```

Fig. 19.62 The bitBlt algorithm (with some special cases omitted).

The code for more complex cases is more elaborate but is equally efficient. We *could* design bitBlt by using massive switch statements or case statements to execute all possible combinations of rectangles, textures, modes, and so on. In fact, the entire program could be written as a loop in the form "for each row, for each word, do the following: If the word is a partial word, do . . .; if it needs a texture, do . . .; and so on." The disadvantage with

this is that all the cases must be examined, even for the simplest case to be executed. All the assembler code for the massive **if** and **switch** statements would never fit into a tiny instruction cache. Since the loop is executed many times, the cost of loading the code into the instruction cache might far outweigh the cost of actually doing the work of bitBlt. Alternatively, the code might be written as a massive nested **if** statement, in which each possible case is optimally coded and occurs as a branch in the decision tree. Then when this case occurred, that tiny fragment of optimal code could be loaded into the instruction cache and executed very quickly. The problem with this approach is that there are a great many cases; Pike estimates that there are about 2000 different cases, taking about 150 bytes on the average. This makes the code for bitBlt approach 1 MB, which is clearly excessive. So, instead, we have the bitBlt routine collect bits of assembler code, which are then used to execute the loop. Each case in the nested **if** statement contributes its own fragment to the final code to be executed. Of course, since this code must be collected in the data space, instead of in the address space of the processor, the instruction cache must be informed that certain bytes are no longer valid, so that it will be certain to reload the code instead of using the code from the last bitBlt that executed. All this must be combined with a certain amount of knowledgeable juggling to determine whether to separate out a case and to hard code it (e.g., bitBlt for very small rectangles—less than one full word).

Implementors of such systems should look for hardware support as well. The more of bitBlt we implement in microcode, the easier it is to make the rest extremely efficient.

19.7 THE SHAPE DATA STRUCTURE AND SHAPE ALGEBRA

The *shape* data structure has been developed to make raster operations (especially clipping) more efficient [DONA88; GOSL89; STEI89]. It lies somewhere on the boundary between the geometries of the Euclidean plane and the rasterized plane, in that shapes are used to represent raster approximations of regions that are typically defined by geometric constructions, but have been scan-converted into rasterized representations. The advantages of shapes over bit masks in defining regions are twofold: shapes are typically smaller data structures, and Boolean operations on shapes are fast. Furthermore, shapes are organized in a way that takes advantage of scan-line coherence, so that line-by-line processing of shapes can be made fast. Other methods for implementing faster raster operations have been developed as well, including implementations of the shape data structure using run-length encoding [STEIN89], and quadtreelike systems [ATKI86].

Before giving a precise definition of a shape, let's consider an example. The U-shaped region shown in Fig. 19.63(a) can be broken into several rectangular pieces, as shown in part (b).

In fact, any collection of pixels can be broken into a disjoint union of rectangular regions in the plane. In the most extreme case, for example, we can place a small square around each pixel. The shape data structure is designed to describe regions in the rasterized plane as lists of rectangles. More precisely, a shape consists of a list of intervals along the y axis and, for each of these, a list of intervals on the x axis. Each (y interval, x interval) pair represents the rectangle that is the Cartesian product of the two intervals. The region in Fig. 19.63(a), for example, would be decomposed into the rectangles $[0, 10] \times [0, 2]$, $[0, 3] \times [2, 4]$, and $[7, 10] \times [2, 4]$, as shown in Fig. 19.63(b); these would be stored in two groups,

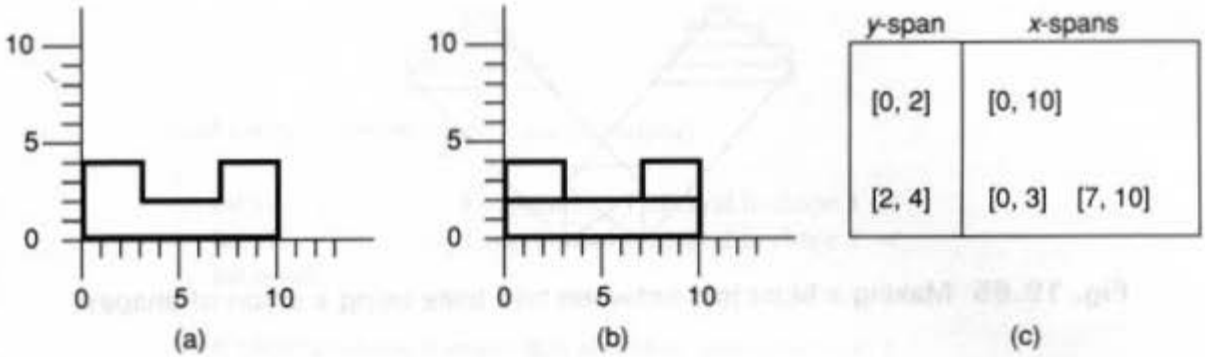


Fig. 19.63 (a) A region. (b) Its division into rectangles. (c) The shape data structure for the region.

corresponding to the y intervals [0, 2] and [2, 4]. The first group would have one x interval and the second would have two, as shown in Fig. 19.63(c). Note, however, that although this data structure is extremely efficient for rectangular regions and simple regions such as the one in Fig. 19.63(a), for more general regions it becomes less efficient. For a filled circle, for example, there is a rectangle for each horizontal scan line, so the structure becomes just a list of spans.

If we create a shape for a scan-converted primitive, and also have a shape for the region within which we want to draw the primitive (typically a window), we can find the shape of the clipped version of the primitive by taking the intersections of the shapes. Furthermore, creating a shape for the scan-converted primitive may be easy, since many scan-conversion algorithms work in some scan-line order. For example, the polygon scan-conversion algorithm in Chapter 3 used an active-edge table, which it then scanned to generate horizontal spans. These spans are exactly the rectangles needed for the shape structure. A similar technique can be used for a region with curved edges. Since curves can have multiple x values for a single y value, it is important to break them into pieces with the property that for each y value, there is only one x value on the curve segment (see Fig. 19.64), and then to scan convert these with an algorithm like that for the active-edge table, described in Chapter 3.

Shapes can also be used to generate joined lines or polylines without the problems caused by repeated **xor** operations on the same pixel. Here, the entire shape is constructed and then is drawn once in **xor** mode. Figure 19.65 shows how a blunt line join can be represented as a union of shapes (see also Exercise 19.21).

One of the principal virtues of shapes is the ease with which they are combined under Boolean operations (recall from Section 15.10.3 that Boolean operations on intervals in a

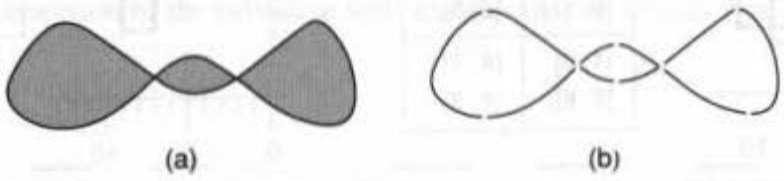


Fig. 19.64 The curved outline for the area shown in (a) must be broken into curve segments, as shown in (b), before it can be used to generate a shape structure for the area.

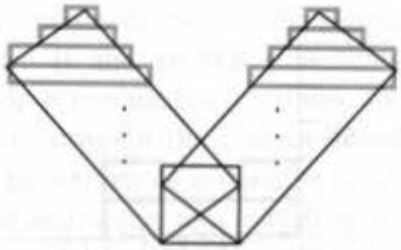


Fig. 19.65 Making a blunt join between two lines using a union of shapes.

line were used in ray tracing to implement constructive solid-geometry operations). Consider the intersection of the two shapes shown in Fig. 19.66(a). The shape structures for both are given in the table, and the intersection and its shape structure are shown in part (b). How can the intersection be computed from the original shapes?

Notice that the intersection of two overlapped rectangles is always a rectangle. Thus, we can compute the intersection of two shapes by searching for overlapping rectangles. We first check to see that the y extents of the shapes overlap (if not, there is no intersection). Then, starting with the first y interval for each shape (i.e., the one with lowest y value), we compare y intervals. If they overlap, we compare x intervals to generate output. If not, we increment the one with the lower y value to the next y interval in its data structure, and repeat. Assuming we have a function, Overlap, that compares two y intervals for overlap, returning a code characterizing the type of overlap, the pseudocode for this operation is as given in Fig. 19.67.

The processing of overlapping x intervals is exactly the same: If two x intervals overlap, the intersection is output and one or both of the two intervals is updated. Thus, the entire algorithm rests on processing the overlaps to determine a result code, then deciding what to do depending on the result code.

The six cases of x-interval overlap are shown in Fig. 19.68. In the first case, shown in part (a), the two segments are disjoint. No output is generated, and the update process consists of taking the next segment in shape2. The second case is the same, as shown in part (b), except that the next segment is taken from shape1. In the third case, in part (c), there is an overlap of the segments given by the minimum end of the segment from shape1 and the maximum end of the segment from shape2. Updating entails taking the next segment from

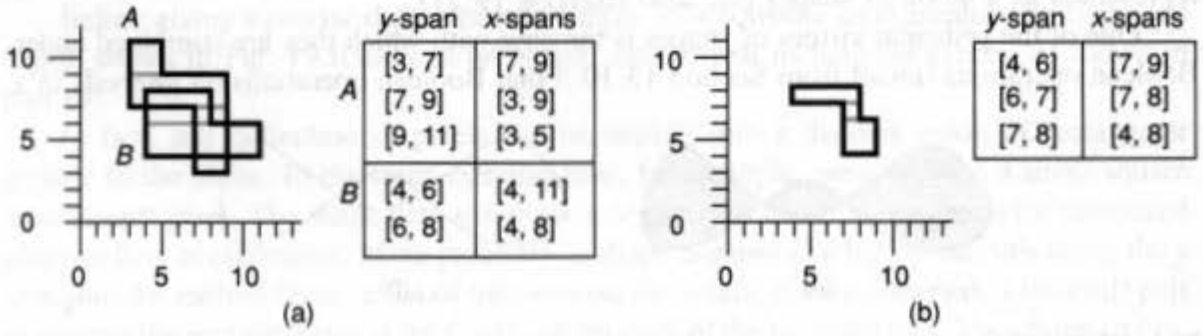


Fig. 19.66 The intersection of two shapes. (a) The two shapes, and their associated data structures. (b) The intersection, and its data structure.

```
void Intersect (shape shape1, shape shape2)
{
    int y11, y12;      /* Endpoints of y interval in shape 1 */
    int y21, y22;      /* Endpoints of y interval in shape 2 */
    int result;

    if (neither shape is empty && bounding boxes overlap) {
        y11 = lowest y-value for shape1;
        y12 = other end of first y-interval in shape1;
        same for y21, y22;

        while (still have y-interval in both shapes) {
            result = Overlap (y11, y12, y21, y22);
            if (result says overlap occurred) {
                output overlap of y-range, determined from code;
                output overlapping x-ranges if any;
                if no x-output generated, delete the y-overlap just output
            }
            update one or both y-intervals based on result;
        }
    } else      /* if */
        do nothing;
} /* Intersect */
```

Fig. 19.67 The algorithm for intersecting two shapes.

shape2. The remaining cases are similar. The algorithms for union and difference are based on the same consideration of the overlap structure (see Exercise 19.22).

The shape algebra can be extended to a combined intersect-and-fill routine to be used in drawing primitives in a clip region. Instead of merely generating the x and y ranges of the shapes, we take each rectangular piece in the output and draw it as a filled rectangle (using a fast copyPixel procedure).

This shape algebra has been used in implementing the NeWS and X11/NeWS window systems. Three optimizations are worth considering. First, in some cases, the shapes representing a primitive are cumbersome (slanted lines are a good example), and, to compute the intersection of the primitive with another shape, we may find it simpler to do

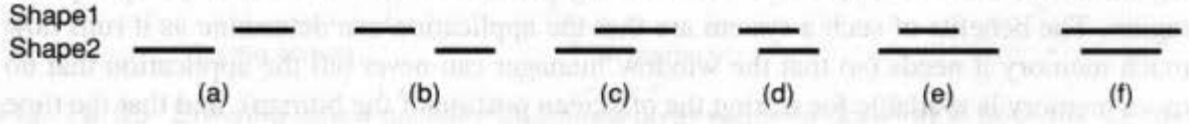


Fig. 19.68 The six ways that two intervals can overlap.

analytic clipping to each rectangle of the shape (Gosling suggests this for lines [GOSL89]). Second, shapes may become fragmented after many operations (taking $(A - B) \cup B$ can do this), and it may be worthwhile to condense a shape after several operations. Third, in window managers, it may be worthwhile to write a special-purpose intersect-and-fill routine that computes the shape intersection between a primitive and a window, optionally applies a pattern to the result, and draws the result all in one operation. Thus, instead of the output shape data structure being generated, the spans generated by the intersection routine are passed directly to a drawing routine.

19.8 MANAGING WINDOWS WITH bitBlt

In a seminal paper, Pike [PIKE83] described using the bitBlt procedure to manage windows (which he called "layers") on a bitmapped screen. The critical features were that the overlapping and refreshing of windows were isolated from the application program generating the window contents, that the nonscreen memory used was not too great (about one screen's worth of offscreen memory), and that the windows could be drawn in even when partially or completely obscured. Although overlapping windows had already been introduced in the Smalltalk world [BYTE85], this paper was the first to show how to implement them on an arbitrary machine supporting bitBlt. The ideas of this paper are now seen in the X Windows System, the Lisa operating system, the Macintosh operating system, and many other windowing systems.

Offscreen management of obscured windows is still done in some systems, as we discussed in Chapter 10 (it is called "backing store" in the X Windows System), but it has a substantial cost: As the number of windows increases, the amount of backing store required increases as well. The results when the machine runs out of physical memory must be considered. If the writer of applications for the window system knows that the window system provides a "virtual window" in which to write, that the application may write to it even when it is obscured, and that the window system takes care of refreshing the screen when the window is moved to the top, then the window system must provide this feature regardless of what else appears on the screen. Since an application's window may be completely covered at some point, the window system must allocate enough memory to provide backing store for the whole window at the time the window is created. Doing this for every one of several applications running simultaneously may become prohibitively expensive.

For this reason, some window systems (e.g., the X Windows System) provide backing store as a option but also allow windows that are not backed up, which must use a different regeneration strategy, as we discussed in Chapter 10. When a window is exposed after being partially or completely hidden, the window manager tells the application which parts of the window need to be refreshed, and the application may regenerate these portions in any manner it chooses. Typically, it redraws any primitive that intersects the newly exposed region. The benefits of such a system are that the application can determine as it runs how much memory it needs (so that the window manager can never tell the application that no more memory is available for storing the offscreen portion of the bitmap), and that the time spent in restoring the offscreen portions can be eliminated in cases where the restoration is unnecessary. The costs are that each application must be aware of the status of the window

into which it is drawing, and that application programmers cannot pretend they have completely exposed windows.

Assuming, however, that we wish to have backing store for each window, we still must implement, in an efficient manner, the various window operations described in Chapter 10. These include window creation, deletion, exposure, and hiding, as well as drawing into windows. The remainder of this section is a description of Pike's method for doing these operations.

The data structure representing a window consists of a rectangle and an array of words (i.e., the bitmap record of Section 19.6), together with three additional fields: two window pointers, to the windows in front of and in back of the window; and one pointer to an obscured list, which is a linked list of bitmaps representing obscured portions of a window. The pointers to the windows in front and in back help in making windows visible or hiding them, since the list helps to determine what must be exposed or hidden. The ordering of windows is only partial, so the choices of front and back pointers are not completely determined. In any case where the choice is not obvious, an arbitrary one is made.

Hiding a window entails finding all windows behind it and placing them in front of it, in order. Doing this may require merely clearing space on the screen and calling the application that owns each window to do damage repair, or may involve more work if the windows actually maintain information in their obscured list. For example, if the refresh task has been left to the windows' software, then hiding involves bitBlting the obscured portion of the newly exposed window onto the screen, while bitBlting the newly obscured portion of the current window into its obscured list. Fortunately, the amount by which window *A* can cover window *B* is exactly the same as the amount by which window *B* can cover window *A*, so no new memory needs to be allocated to execute this change of places. Exposing a window is similar, and moving a window involves revealing or hiding new portions and doing bitBlts to move the visible portions of the window. Of course, the assumption that there is a piece of memory that is just the right size to represent the amount by which *B* covers *A* or vice versa requires that the obscured lists be very finely divided: Each bounding rectangle for an obscured region must lie entirely within any window that it intersects.

Another task to consider is drawing into a window. In Fig. 19.69, there are two overlapping windows. In drawing an item in the lower window (shown as a shaded region),

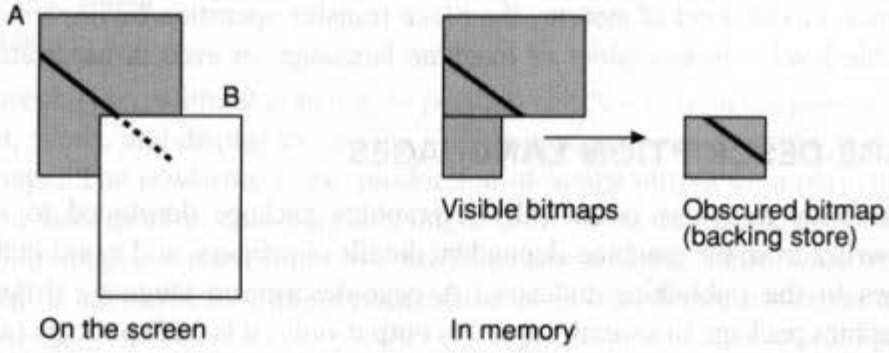


Fig. 19.69 Drawing into a partially obscured layer requires drawing in both the visible portion and in some obscured portions. We split the item to be drawn into two pieces, and draw each in the appropriate bitmap.

part of the item is drawn in the visible portion and part is drawn in an obscured rectangle. This is a specific case of a general task: Take some operation and do it in each of the bitmaps representing the target region of the operation. Such an operation might be to clear a rectangle or to draw a line, for example. This operation can be condensed into a single recursive procedure that applies the operation to the intersection of the target region and the window, and then calls itself on each item in the window's obscured list.

Note that the design of the window data structure includes a choice regarding rectangles: each rectangle is defined to contain its left and bottom edges and its lower-left corner, just as it did in Chapter 3. Thus, no two abutting rectangles share any points. This choice vastly simplifies many operations, since it becomes possible to satisfy the rule that each rectangle in any obscured list lies entirely within any window it intersects. If rectangles contained both their edges, satsifying the condition would be impossible.

The original "layers" model for window management has been greatly extended. Several features have been added, including color. When color is added, copying images to and from the screen becomes more complex. Each pixel is represented by a collection of bits, rather than by a single bit, and transferring all the bits for the pixel may take a long time. Suppose, for example, that the display uses 4 bits for each of red, green, and blue. We could imagine this as simply 12 planes' worth of bits needing to be transferred to the screen, and we could apply the bitBlt operation to each. This strategy is called the *plane-serial* approach, since the planes are processed one at a time. The plane-serial approach has the advantage of being easy to implement as an extension of the bitmap version of the window-management software, but has two serious drawbacks: It is (in this case) 12 times as slow, which may be disastrous for performance; and, while the various planes are being transferred, the window may look very peculiar. For example, when all the red planes have been handled but the green and blue have not, the window looks like a blue-green version of the old material with a red overlay of the new material. This effect is extremely distracting if it persists for long enough to be detected. In a color-table system, the results would be even more exotic: During the transfer of planes, the color indices would become permuted wildly, which would result in an extremely distracting appearance. The alternative is the *plane-parallel* approach, in which all the planes of the pixmap are copied at the same time (perhaps by special-purpose hardware). Since the organization of display memory and of main memory can be somewhat different (although the Pike paper assumes they are not), it is essential in doing block transfers to use the most efficient method for the given source and target. This requirement in turn demands that, in any practical implementation of this kind of system, the block transfer operation be implemented at the lowest possible level—in assembler or machine language, or even in hardware.

19.9 PAGE-DESCRIPTION LANGUAGES

A *page-description language* is basically a graphics package developed to insulate the application writer from the machine-dependent details of printers, and to aid in the layout of printed pages in the publishing industry. A page-description language differs from an ordinary graphics package in several ways: it is output-only, it is a 2D package (although 3D extensions to some languages are being developed), it has extensive support for curves and text, and it supports sampled images as first-class primitives. More important, it is a

language instead of a subroutine package. Thus, an interpreter for the language can be resident in a printer; as a result, short programs, instead of huge volumes of pixel data, are sent to the printer. Furthermore, page-description languages can be used more readily than can subroutine packages as an interchange format. A sequence of subroutine calls can be made in many different languages; transferring the sequence of calls to another installation may require translation to a new language. A program in a page-description language can be transferred to any installation that supports the language. Thus, a page-description language supersedes the notion of metafiles described in Section 7.11.3.

The best-known page-description language is POSTSCRIPT. The original intent of this language was to describe the appearance of a bitmap page (to be produced typically on a very high-resolution printer). POSTSCRIPT is now being used as a screen-description language as well. Page-description languages are particularly well suited to this task, especially in client–server window managers, where downloading a POSTSCRIPT program to the server can reduce network traffic and overhead on the client. As described in Chapter 10, invoking a dialogue box in a POSTSCRIPT-based window manager can be done by downloading a POSTSCRIPT procedure that displays the dialogue box and then invoking the procedure. Subsequent invocations of the dialogue box are made by simply invoking the procedure again.

There is also interest in generating a national or international standard page-description language derived from POSTSCRIPT and Interpress [ISO]. Such a language might provide a standard format for both the publishing industry and the computing industry.

In this section, we describe aspects of POSTSCRIPT, to give you a sense of how page-description languages work. The *imaging model* of a page-description language is the definition of the abstract behaviour of the language on an ideal 2D plane. In POSTSCRIPT, the imaging model is based on the notion of painting with opaque paint on a plane. The paint is applied by a pen or brush of a user-specified width, and many POSTSCRIPT operators control the position of this pen. This imaging model is actually implemented by having a large raster memory that reflects the contents of a page; the contents of this raster memory are eventually transferred to the output page by a printer.

The POSTSCRIPT language has effectively three components: the syntax, the semantics, and the rendering. The difference between the semantics and the rendering is a subtle one. The POSTSCRIPT program

```
10.5  11.3  moveto
40    53.6  lineto
showpage
```

means "move the pen, without drawing, to position (10.5, 11.3) on the page, draw a line to position (40, 53.6), and display the results." This sentence is an example of the semantics of the language. The *rendering* is the production of actual output on a particular graphics device. On a laser printer, this program might draw 5000 tiny black dots on a piece of paper; on a bitmapped screen, it might draw 87 pixels in black. Thus, what we are calling the rendering of a POSTSCRIPT program is at the level of a device driver for a bitmapped device.

POSTSCRIPT syntax is fairly simple. It is a postfix interpreter in which operands are pushed onto a stack and then are processed by operators that use some number of operands

from the stack (popping them off the stack) and place some number of results on the stack. Thus, in the preceding example, the operands 10.5 and 11.3 were pushed on the stack, and the **moveto** operator popped them from the stack and used them. The data types supported include numbers, arrays, strings, and associative tables (also known as dictionaries). (More precisely, there is only one data class—the *object*; each object has a type, some attributes, and a value. The type may be "integer," "real," "operator," etc.) The associative tables are used to store the definitions of various objects, including operator definitions. The flow-of-control constructs include conditionals, looping, and procedures. Thus, a typical application producing POSTSCRIPT output may either (1) produce a long string of standard POSTSCRIPT calls, or (2) define a collection of procedures more closely related to its own needs (called a *prologue*), then produce a collection of calls to these procedures.

POSTSCRIPT also includes the notion of *contexts* that can be saved on a stack, so that the state of the POSTSCRIPT world can be saved before an operation is performed, and restored afterward. Applications that define their own procedures do not need to worry about naming conflicts if they agree to restore the POSTSCRIPT world to its initial state.

The semantics of POSTSCRIPT are more complex. The fundamental entities are graphical entities and operators acting on them. All graphical entities are considered the same, so a line, a circular arc, a cubic curve, a blob, a string of text, and a sampled image can all be translated, rotated, or scaled by identical operators. Various operators are used to support multiple coordinate systems, so objects can be created in one coordinate system and then transformed by arbitrary affine transformations into any other coordinate system. A few operators can detect the environment in which POSTSCRIPT is running, which is important for making device-independent images. Thus, we can write a POSTSCRIPT program to produce a 1- by 1-inch square, regardless of the resolution of the device being used (unless of course its pixels are 1.5 inches wide!), by enquiring the environment to determine the number of pixels per inch on the current device.

POSTSCRIPT operators fall into six categories [ADOB85b]:

1. *Graphics-state operators*. These operators affect a collection of objects defining certain current attributes of the POSTSCRIPT world, such as the current line width, or the current clipping region.
2. *Coordinate-system operators and transformations*. These operators are used to alter the coordinate systems in which further objects are to be defined. In particular, they alter the mapping from coordinates within POSTSCRIPT to the coordinates of the output device.
3. *Path-construction operators*. These operators are used to define and update another graphics-state entity called the *current path*. They can be used to begin a path, to add collections of lines or arcs to the path, or to close the path (i.e., to join its beginning to its end with a straight-line segment). The current path is an abstract entity—it is not rendered on the page unless some painting operator is invoked.
4. *Painting operators*. These "rendering" operators generate data in a raster memory that eventually determine which dots appear on the printed page. All painting operators refer to the current path. If we imagine the path-construction operations as defining a mask, then the painting operators can be thought of as placing paint dots on the raster memory at each place allowed by the mask.