3
Basic Raster Graphics Algorithms for Drawing 2D Primitives

A raster graphics package approximates mathematical ("ideal") primitives, described in terms of vertices on a Cartesian grid, by sets of pixels of the appropriate intensity of gray or color. These pixels are stored as a bitmap or pixmap in CPU memory or in a frame buffer. In the previous chapter, we studied the features of SRGP, a typical raster graphics package, from an *application programmer's* point of view. The purpose of this chapter is to look at SRGP from a *package implementor's* point of view—that is, in terms of the fundamental algorithms for scan converting primitives to pixels, subject to their attributes, and for clipping them against an upright clip rectangle. Examples of scan-converted and clipped primitives are shown in Fig. 3.1.

More advanced algorithms that handle features not supported in SRGP are used in more sophisticated and complex packages; such algorithms are treated in Chapter 19. The algorithms in this chapter are discussed in terms of the 2D integer Cartesian grid, but most of the scan-conversion algorithms can be extended to floating point, and the clipping algorithms can be extended both to floating point and to 3D. The final section introduces the concept of antialiasing—that is, minimizing jaggies by making use of a system's ability to vary a pixel's intensity.

3.1 OVERVIEW

3.1.1 Implications of Display-System Architecture

The fundamental conceptual model of Section 1.7 presents a graphics package as the system that mediates between the application program (and its application data structure/model) and the display hardware. The package gives the application program a device-

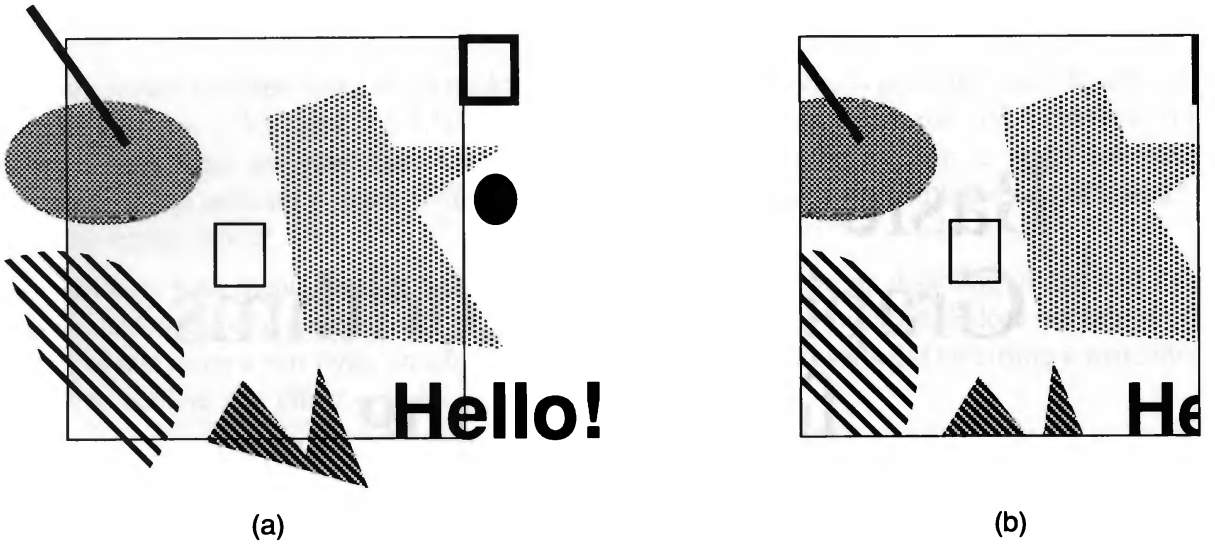

Fig. 3.1 Clipping SRGP primitives to a rectangular clip region. (a) Primitives and clipping rectangle. (b) Clipped results.

independent interface to the hardware, as shown in Fig. 3.2, where SRGP's procedures are partitioned into those forming an output pipeline and those forming an input pipeline.

In the *output pipeline*, the application program takes descriptions of objects in terms of primitives and attributes stored in or derived from an application model or data structure, and specifies them to the graphics package, which in turn clips and scan converts them to the pixels seen on the screen. The package's primitive-generation procedures specify *what* is to be generated, the attribute procedures specify *how* primitives are to be generated, the SRGP_copyPixel procedure specifies *how* images are to be modified, and the canvas-control procedures specify *where* the images are to be generated. In the *input pipeline*, a user interaction at the display end is converted to measure values returned by the package's sampling or event-driven input procedures to the application program; it typically uses those

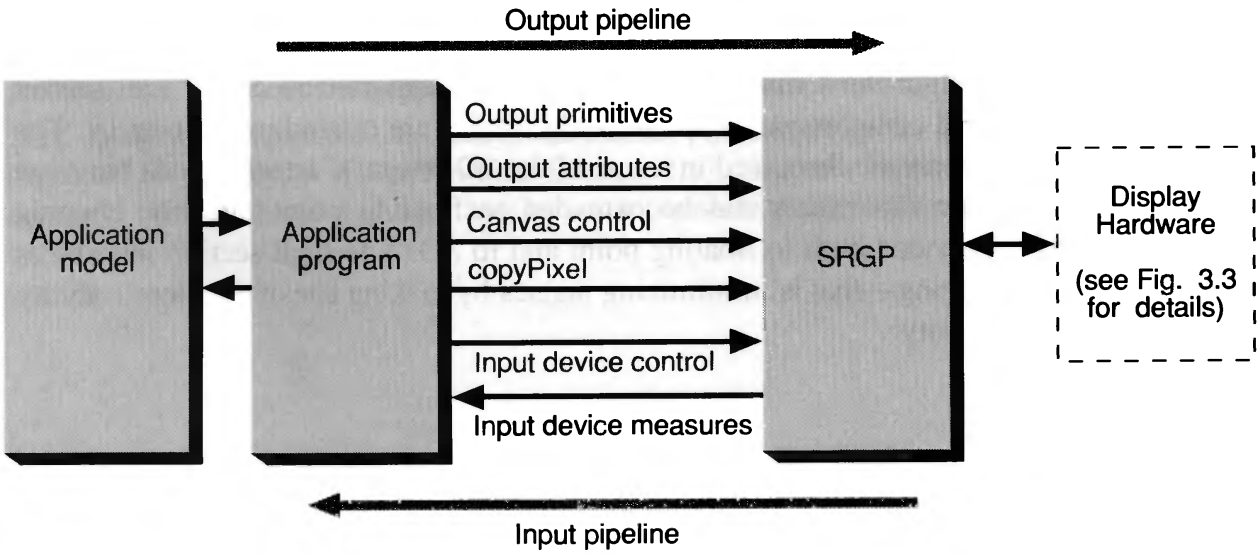

Fig. 3.2 SRGP as intermediary between the application program and the graphics system, providing output and input pipelines.

values to modify the model or the image on the screen. Procedures relating to input include those to initialize and control input devices and those to obtain the latter's measures during interaction. We do not cover either SRGP's canvas management or its input handling in this book, since these topics have little to do with raster graphics and are primarily data-structure and low-level systems-software issues, respectively.

An SRGP implementation must communicate with a potentially wide variety of display devices. Some display systems are attached as peripherals with their own internal frame buffers and display controllers. These display controllers are processors specialized to interpret and execute drawing commands that generate pixels into the frame buffer. Other, simpler systems are refreshed directly from the memory used by the CPU. Output-only subsets of the package may drive raster hardcopy devices. These various types of hardware architectures are discussed in more detail in Chapters 4 and 18. In any display-system architecture, the CPU must be able to read and write individual pixels in the frame buffer. It is also convenient to be able to move rectangular blocks of pixels to and from the frame buffer to implement the copyPixel (bitBlt) type of operation. This facility is used not for generating primitives directly but to make portions of offscreen bitmaps or pixmaps visible and to save and restore pieces of the screen for window management, menu handling, scrolling, and so on.

Whereas all implementations for systems that refresh from CPU memory are essentially identical because all the work is done in software, implementations for display controller and hardcopy systems vary considerably, depending on what the respective hardware devices can do by themselves and what remains for the software to do. Naturally, in any architecture, software scan conversion must be used to generate both primitives and attributes not directly supported in hardware. Let's look briefly at the range of architectures and implementations.

Displays with frame buffers and display controllers. SRGP has the least amount of work to do if it drives a display controller that does its own scan conversion and handles all of SRGP's primitives and attributes directly. In this case, SRGP needs only to convert its internal representation of primitives, attributes, and write modes to the formats accepted by the display peripheral that actually draws the primitives (Fig. 3.3 a).

The display-controller architecture is most powerful when memory mapping allows the CPU to access the frame buffer directly and the display controller to access the CPU's memory. The CPU can then read and write individual pixels and copyPixel blocks of pixels with normal CPU instructions, and the display controller can scan convert into offscreen canvases and also use its copyPixel instruction to move pixels between the two memories or within its own frame buffer. When the CPU and the display controller can run asynchronously, there must be synchronization to avoid memory conflicts. Often, the display controller is controlled by the CPU as a coprocessor. If the display peripheral's display controller can only scan convert into its own frame buffer and cannot write pixels into CPU memory, we need a way to generate primitives in an offscreen canvas. The package then uses the display controller for scan conversion into the screen canvas but must do its own software scan conversion for offscreen canvases. The package can, of course, copyPixel images scan converted by the hardware from the frame buffer to offscreen canvases.

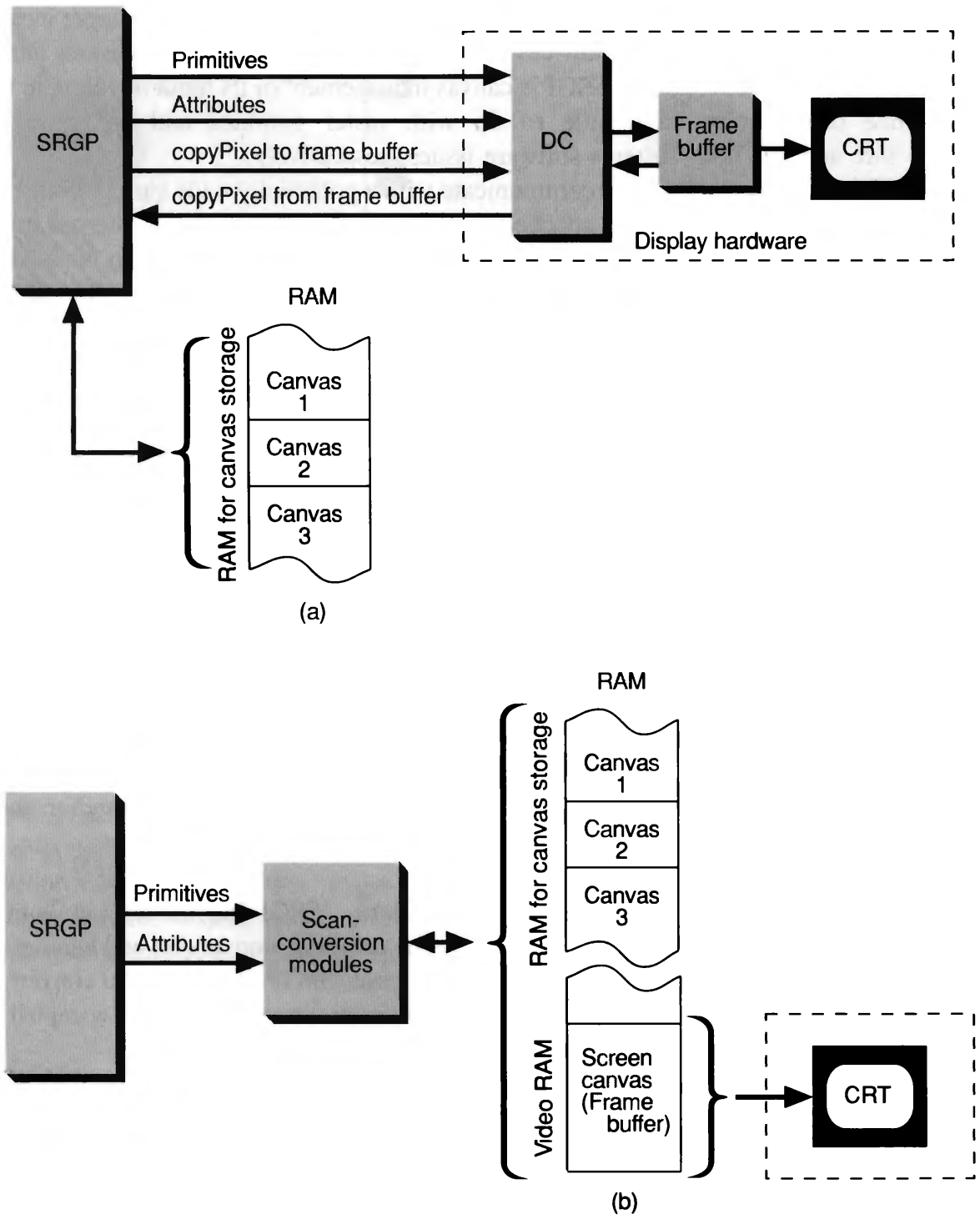

Fig. 3.3 SRGP driving two types of display systems. (a) Display peripheral with display controller and frame buffer. (b) No display controller, memory-shared frame buffer.

Displays with frame buffers only. For displays without a display controller, SRGP does its own scan conversion into both offscreen canvases and the frame buffer. A typical organization for such an SRGP implementation that drives a shared-memory frame buffer is shown in Fig. 3.3 (b). Note that we show only the parts of memory that constitute the frame

buffer and store the canvases managed by SRGP; the rest of the memory is occupied by all the usual software and data, including, of course, SRGP itself.

Hardcopy devices. As explained in Chapter 4, hardcopy devices range in their capabilities along the same spectrum as display systems. The simplest devices accept only one scan line at a time and rely on the software to provide that scan line exactly when it is to be imaged on film or on paper. For such simple hardware, SRGP must generate a complete bitmap or pixmap and scan it out one line at a time to the output device. Slightly smarter devices can accept an entire frame (page) at a time. Yet more powerful equipment has built-in scan-conversion hardware, often called raster image processors (RIPs). At the high end of the scale, PostScript printers have internal "engines" that read PostScript programs describing pages in a device-independent fashion; they interpret such programs to produce the primitives and attributes that are then scan converted. The fundamental clipping and scan-conversion algorithms are essentially independent of the raster device's output technology; therefore, we need not address hardcopy devices further in this chapter.

3.1.2 The Output Pipeline in Software

Here we examine the output pipeline driving simple frame-buffer displays only in order to address the problems of software clipping and scan conversion. The various algorithms introduced are discussed at a general, machine-independent level, so they apply to both software and hardware (or microcode) implementations.

As each output primitive is encountered by SRGP, the package *scan converts* the primitive: Pixels are written in the current canvas according to their applicable attributes and current write mode. The primitive is also *clipped* to the clip rectangle; that is, pixels belonging to the primitive that are outside the clip region are not displayed. There are several ways of doing clipping. The obvious technique is to clip a primitive prior to scan conversion by computing its analytical intersections with the clip-rectangle boundaries; these intersection points are then used to define new vertices for the clipped version of the primitive. The advantage of clipping before scan converting is, of course, that the scan converter must deal with only the clipped version of the primitive, not with the original (possibly much larger) one. This technique is used most often for clipping lines, rectangles, and polygons, for which clipping algorithms are fairly simple and efficient.

The simplest, brute-force clipping technique, called *scissoring*, is to scan convert the entire primitive but to write only the visible pixels in the clip-rectangle region of the canvas. In principle, this is done by checking each pixel's coordinates against the (x, y) bounds of the rectangle before writing that pixel. In practice, there are shortcuts that obviate having to check adjacent pixels on a scan line, as we shall see later. This type of clipping is thus accomplished on the fly; if the bounds check can be done quickly (e.g., by a tight inner loop running completely in microcode or in an instruction cache), this approach may actually be faster than first clipping the primitive and then scan converting the resulting, clipped portions. It also generalizes to arbitrary clip regions.

A third technique is to generate the entire collection of primitives into a temporary canvas and then to copyPixel only the contents of the clip rectangle to the destination canvas. This approach is wasteful of both space and time, but is easy to implement and is often used for text. Data structures for minimizing this overhead are discussed in Chapter 19.

Raster displays invoke clipping and scan-conversion algorithms each time an image is created or modified. Hence, these algorithms not only must create visually satisfactory images, but also must execute as rapidly as possible. As discussed in detail in later sections, scan-conversion algorithms use *incremental methods* to minimize the number of calculations (especially multiplies and divides) performed during each iteration; further, these calculations employ integer rather than floating-point arithmetic. As shown in Chapter 18, speed can be increased even further by using multiple parallel processors to scan convert simultaneously entire output primitives or pieces of them.

3.2 SCAN CONVERTING LINES

A scan-conversion algorithm for lines computes the coordinates of the pixels that lie on or near an ideal, infinitely thin straight line imposed on a 2D raster grid. In principle, we would like the sequence of pixels to lie as close to the ideal line as possible and to be as straight as possible. Consider a 1-pixel-thick approximation to an ideal line; what properties should it have? For lines with slopes between -1 and 1 inclusive, exactly 1 pixel should be illuminated in each column; for lines with slopes outside this range, exactly 1 pixel should be illuminated in each row. All lines should be drawn with constant brightness, independent of length and orientation, and as rapidly as possible. There should also be provisions for drawing lines that are more than 1 pixel wide, centered on the ideal line, that are affected by line-style and pen-style attributes, and that create other effects needed for high-quality illustrations. For example, the shape of the endpoint regions should be under programmer control to allow beveled, rounded, and mitered corners. We would even like to be able to minimize the jaggies due to the discrete approximation of the ideal line by using antialiasing techniques exploiting the ability to set the intensity of individual pixels on n-bits-per-pixel displays.

For now, we consider only "optimal," 1-pixel-thick lines that have exactly 1 bilevel pixel in each column (or row for steep lines). Later in the chapter, we consider thick primitives and deal with styles.

To visualize the geometry, we recall that SRGP represents a pixel as a circular dot centered at that pixel's (x, y) location on the integer grid. This representation is a convenient approximation to the more or less circular cross-section of the CRT's electron beam, but the exact spacing between the beam spots on an actual display can vary greatly among systems. In some systems, adjacent spots overlap; in others, there may be space between adjacent vertical pixels; in most systems, the spacing is tighter in the horizontal than in the vertical direction. Another variation in coordinate-system representation arises in systems, such as the Macintosh, that treat pixels as being centered in the rectangular box between adjacent grid lines instead of on the grid lines themselves. In this scheme, rectangles are defined to be all pixels interior to the mathematical rectangle defined by two corner points. This definition allows zero-width (null) canvases: The rectangle from (x, y) to (x, y) contains no pixels, unlike the SRGP canvas, which has a single pixel at that point. For now, we continue to represent pixels as disjoint circles centered on a uniform grid, although we shall make some minor changes when we discuss antialiasing.

Figure 3.4 shows a highly magnified view of a 1-pixel-thick line and of the ideal line that it approximates. The intensified pixels are shown as filled circles and the nonintensified

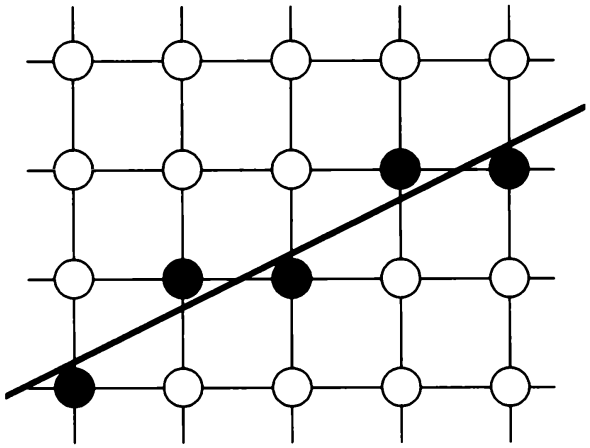

Fig. 3.4 A scan-converted line showing intensified pixels as black circles.

pixels are shown as unfilled circles. On an actual screen, the diameter of the roughly circular pixel is larger than the interpixel spacing, so our symbolic representation exaggerates the discreteness of the pixels.

Since SRGP primitives are defined on an integer grid, the endpoints of a line have integer coordinates. In fact, if we first clip the line to the clip rectangle, a line intersecting a clip edge may actually have an endpoint with a noninteger coordinate value. The same is true when we use a floating-point raster graphics package. (We discuss these noninteger intersections in Section 3.2.3.) Assume that our line has slope $|m| \leq 1$; lines at other slopes can be handled by suitable changes in the development that follows. Also, the most common lines—those that are horizontal, are vertical, or have a slope of ± 1—can be handled as trivial special cases because these lines pass through only pixel centers (see Exercise 3.1).

3.2.1 The Basic Incremental Algorithm

The simplest strategy for scan conversion of lines is to compute the slope m as $\Delta y/\Delta x$, to increment x by 1 starting with the leftmost point, to calculate $y_i = mx_i + B$ for each x_i, and to intensify the pixel at $(x_i, \text{Round}(y_i))$, where $\text{Round}(y_i) = \text{Floor}(0.5 + y_i)$. This computation selects the closest pixel—that is, the pixel whose distance to the true line is smallest.[1] This brute-force strategy is inefficient, however, because each iteration requires a floating-point (or binary fraction) multiply, addition, and invocation of Floor. We can eliminate the multiplication by noting that

$$y_{i+1} = mx_{i+1} + B = m(x_i + \Delta x) + B = y_i + m\Delta x,$$

and, if $\Delta x = 1$, then $y_{i+1} = y_i + m$.

Thus, a unit change in x changes y by m, which is the slope of the line. For all points (x_i, y_i) on the line, we know that, if $x_{i+1} = x_i + 1$, then $y_{i+1} = y_i + m$; that is, the values of x and y are defined in terms of their previous values (see Fig. 3.5). This is what defines an

[1] In Chapter 19, we discuss various measures of closeness for lines and general curves (also called *error measures*).

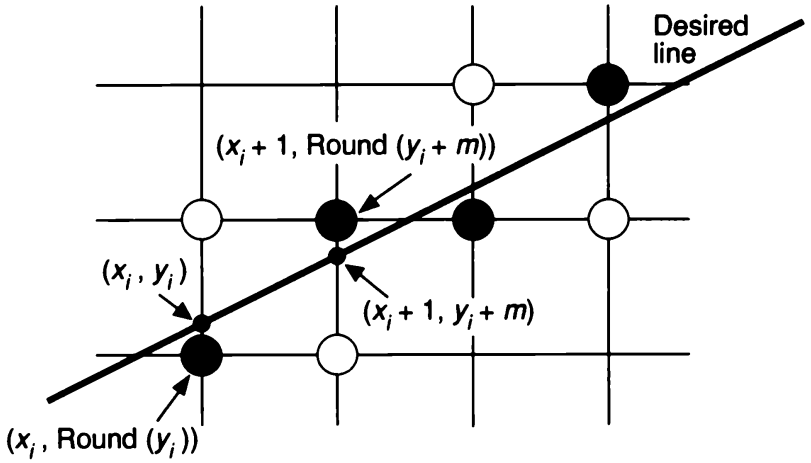

Fig. 3.5 Incremental calculation of (x_i, y_i).

incremental algorithm: At each step, we make incremental calculations based on the preceding step.

We initialize the incremental calculation with (x_0, y_0), the integer coordinates of an endpoint. Note that this incremental technique avoids the need to deal with the y intercept, B, explicitly. If $|m| > 1$, a step in x creates a step in y that is greater than 1. Thus, we must reverse the roles of x and y by assigning a unit step to y and incrementing x by $\Delta x = \Delta y/m = 1/m$. Line, the procedure in Fig. 3.6, implements this technique. The start point must be the left endpoint. Also, it is limited to the case $-1 \leq m \leq 1$, but other slopes may be accommodated by symmetry. The checking for the special cases of horizontal, vertical, or diagonal lines is omitted.

WritePixel, used by Line, is a low-level procedure provided by the device-level software; it places a value into a canvas for a pixel whose coordinates are given as the first two arguments.[2] We assume here that we scan convert only in replace mode; for SRGP's other write modes, we must use a low-level ReadPixel procedure to read the pixel at the destination location, logically combine that pixel with the source pixel, and then write the result into the destination pixel with WritePixel.

This algorithm is often referred to as a *digital differential analyzer (DDA)* algorithm. The DDA is a mechanical device that solves differential equations by numerical methods: It traces out successive (x, y) values by simultaneously incrementing x and y by small steps proportional to the first derivative of x and y. In our case, the x increment is 1, and the y increment is $dy/dx = m$. Since real variables have limited precision, summing an inexact m repetitively introduces cumulative error buildup and eventually a drift away from a true $\text{Round}(y_i)$; for most (short) lines, this will not present a problem.

3.2.2 Midpoint Line Algorithm

The drawbacks of procedure Line are that rounding y to an integer takes time, and that the variables y and m must be real or fractional binary because the slope is a fraction. Bresenham developed a classic algorithm [BRES65] that is attractive because it uses only

[2]If such a low-level procedure is not available, the SRGP_pointCoord procedure may be used, as described in the SRGP reference manual.

```
void Line (                                  /* Assumes -1 ≤ m ≤ 1, x0 < x1 */
      int x0, int y0,                        /* Left endpoint */
      int x1, int y1,                        /* Right endpoint */
      int value)                             /* Value to place in line's pixels */
{
   int x;                                    /* x runs from x0 to x1 in unit increments. */

   double dy = y1 - y0;
   double dx = x1 - x0;
   double m = dy / dx;
   double y = y0;
   for (x = x0; x <= x1; x++) {
      WritePixel (x, Round (y), value);      /* Set pixel to value */
      y += m;                                /* Step y by slope m */
   }
} /* Line */
```

Fig. 3.6 The incremental line scan-conversion algorithm.

integer arithmetic, thus avoiding the Round function, and allows the calculation for (x_{i+1}, y_{i+1}) to be performed incrementally—that is, by using the calculation already done at (x_i, y_i). A floating-point version of this algorithm can be applied to lines with arbitrary real-valued endpoint coordinates. Furthermore, Bresenham's incremental technique may be applied to the integer computation of circles as well, although it does not generalize easily to arbitrary conics. We therefore use a slightly different formulation, the *midpoint technique*, first published by Pitteway [PITT67] and adapted by Van Aken [VANA84] and other researchers. For lines and integer circles, the midpoint formulation, as Van Aken shows [VANA85], reduces to the Bresenham formulation and therefore generates the same pixels. Bresenham showed that his line and integer circle algorithms provide the best-fit approximations to true lines and circles by minimizing the error (distance) to the true primitive [BRES77]. Kappel discusses the effects of various error criteria in [KAPP85].

We assume that the line's slope is between 0 and 1. Other slopes can be handled by suitable reflections about the principal axes. We call the lower-left endpoint (x_0, y_0) and the upper-right endpoint (x_1, y_1).

Consider the line in Fig. 3.7, where the previously selected pixel appears as a black circle and the two pixels from which to choose at the next stage are shown as unfilled circles. Assume that we have just selected the pixel P at (x_P, y_P) and now must choose between the pixel one increment to the right (called the east pixel, E) or the pixel one increment to the right and one increment up (called the northeast pixel, NE). Let Q be the intersection point of the line being scan-converted with the grid line $x = x_P + 1$. In Bresenham's formulation, the difference between the vertical distances from E and NE to Q is computed, and the sign of the difference is used to select the pixel whose distance from Q is smaller as the best approximation to the line. In the midpoint formulation, we observe on which side of the line the midpoint M lies. It is easy to see that, if the midpoint lies above the line, pixel E is closer to the line; if the midpoint lies below the line, pixel NE is closer to the line. The line may pass between E and NE, or both pixels may lie on one side, but in any

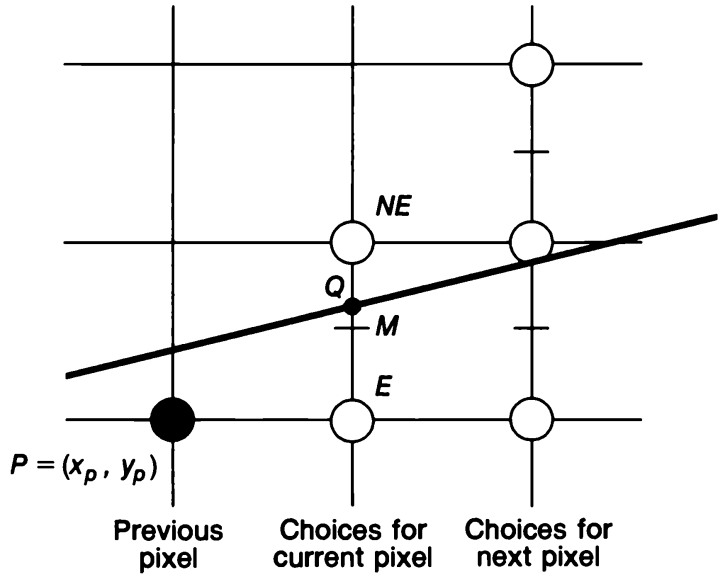

Fig. 3.7 The pixel grid for the midpoint line algorithm, showing the midpoint *M*, and the *E* and *NE* pixels to choose between.

case, the midpoint test chooses the closest pixel. Also, the error—that is, the vertical distance between the chosen pixel and the actual line—is always $\leq 1/2$.

The algorithm chooses *NE* as the next pixel for the line shown in Fig. 3.7. Now all we need is a way to calculate on which side of the line the midpoint lies. Let's represent the line by an implicit function[3] with coefficients a, b, and c: $F(x, y) = ax + by + c = 0$. (The b coefficient of y is unrelated to the y intercept B in the slope-intercept form.) If $dy = y_1 - y_0$, and $dx = x_1 - x_0$, the slope-intercept form can be written as

$$y = \frac{dy}{dx}x + B\ ;$$

therefore,

$$F(x, y) = dy \cdot x - dx \cdot y + B \cdot dx = 0.$$

Here $a = dy$, $b = -dx$, and $c = B \cdot dx$ in the implicit form.[4]

It can easily be verified that $F(x, y)$ is zero on the line, positive for points below the line, and negative for points above the line. To apply the midpoint criterion, we need only to compute $F(M) = F(x_P + 1, y_P + \frac{1}{2})$ and to test its sign. Because our decision is based on the value of the function at $(x_P + 1, y_P + \frac{1}{2})$, we define a *decision variable* $d = F(x_P + 1, y_P + \frac{1}{2})$. By definition, $d = a(x_P + 1) + b(y_P + \frac{1}{2}) + c$. If $d > 0$, we choose pixel *NE*; if $d < 0$, we choose E; and if $d = 0$, we can choose either, so we pick E.

Next, we ask what happens to the location of M and therefore to the value of d for the next grid line; both depend, of course, on whether we chose E or *NE*. If E is chosen, M is

[3]This functional form extends nicely to the implicit formulation of both circles and ellipses.

[4]It is important for the proper functioning of the midpoint algorithm to choose a to be positive; we meet this criterion if dy is positive, since $y_1 > y_0$.

incremented by one step in the x direction. Then,

$$d_{new} = F(x_P + 2, y_P + \tfrac{1}{2}) = a(x_P + 2) + b(y_P + \tfrac{1}{2}) + c,$$

but

$$d_{old} = a(x_P + 1) + b(y_P + \tfrac{1}{2}) + c.$$

Subtracting d_{old} from d_{new} to get the incremental difference, we write $d_{new} = d_{old} + a$.

We call the increment to add after E is chosen Δ_E; $\Delta_E = a = dy$. In other words, we can derive the value of the decision variable at the next step incrementally from the value at the current step without having to compute $F(M)$ directly, by merely adding Δ_E.

If NE is chosen, M is incremented by one step each in both the x and y directions. Then,

$$d_{new} = F(x_P + 2, y_P + \tfrac{3}{2}) = a(x_P + 2) + b(y_P + \tfrac{3}{2}) + c.$$

Subtracting d_{old} from d_{new} to get the incremental difference, we write

$$d_{new} = d_{old} + a + b.$$

We call the increment to add to d after NE is chosen Δ_{NE}; $\Delta_{NE} = a + b = dy - dx$.

Let's summarize the incremental midpoint technique. At each step, the algorithm chooses between 2 pixels based on the sign of the decision variable calculated in the previous iteration; then, it updates the decision variable by adding either Δ_E or Δ_{NE} to the old value, depending on the choice of pixel.

Since the first pixel is simply the first endpoint (x_0, y_0), we can directly calculate the initial value of d for choosing between E and NE. The first midpoint is at $(x_0 + 1, y_0 + \frac{1}{2})$, and

$$\begin{aligned} F(x_0 + 1, y_0 + \tfrac{1}{2}) &= a(x_0 + 1) + b(y_0 + \tfrac{1}{2}) + c \\ &= ax_0 + by_0 + c + a + b/2 \\ &= F(x_0, y_0) + a + b/2. \end{aligned}$$

But (x_0, y_0) is a point on the line and $F(x_0, y_0)$ is therefore 0; hence, d_{start} is just $a + b/2 = dy - dx/2$. Using d_{start}, we choose the second pixel, and so on. To eliminate the fraction in d_{start}, we redefine our original F by multiplying it by 2; $F(x, y) = 2(ax + by + c)$. This multiplies each constant and the decision variable by 2, but does not affect the sign of the decision variable, which is all that matters for the midpoint test.

The arithmetic needed to evaluate d_{new} for any step is simple addition. No time-consuming multiplication is involved. Further, the inner loop is quite simple, as seen in the midpoint algorithm of Fig. 3.8. The first statement in the loop, the test of d, determines the choice of pixel, but we actually increment x and y to that pixel location after updating the decision variable (for compatibility with the circle and ellipse algorithms). Note that this version of the algorithm works for only those lines with slope between 0 and 1; generalizing the algorithm is left as Exercise 3.2. In [SPRO82], Sproull gives an elegant derivation of Bresenham's formulation of this algorithm as a series of program transformations from the original brute-force algorithm. No equivalent of that derivation for circles or ellipses has yet appeared, but the midpoint technique does generalize, as we shall see.

```
void MidpointLine (int x0, int y0, int x1, int y1, int value)
{
    int dx = x1 - x0;
    int dy = y1 - y0;
    int d = 2 * dy - dx;              /* Initial value of d */
    int incrE = 2 * dy;               /* Increment used for move to E */
    int incrNE = 2 * (dy - dx);       /* Increment used for move to NE */
    int x = x0;
    int y = y0;
    WritePixel (x, y, value);         /* The start pixel */

    while (x < x1) {
        if (d <= 0) {                 /* Choose E */
            d += incrE;
            x++;
        } else {                      /* Choose NE */
            d += incrNE;
            x++;
            y++;
        }
        WritePixel (x, y, value);     /* The selected pixel closest to the line */
    } /* while */

} /* MidpointLine */
```

Fig. 3.8 The midpoint line scan-conversion algorithm.

For a line from point (5, 8) to point (9, 11), the successive values of d are 2, 0, 6, and 4, resulting in the selection of *NE*, *E*, *NE*, and then *NE*, respectively, as shown in Fig. 3.9. The line appears abnormally jagged because of the enlarged scale of the drawing and the artificially large interpixel spacing used to make the geometry of the algorithm clear. For the same reason, the drawings in the following sections also make the primitives appear blockier than they look on an actual screen.

3.2.3 Additional Issues

Endpoint order. Among the complications to consider is that we must ensure that a line from P_0 to P_1 contains the same set of pixels as the line from P_1 to P_0, so that the appearance of the line is independent of the order of specification of the endpoints. The only place where the choice of pixel is dependent on the direction of the line is where the line passes exactly through the midpoint and the decision variable is zero; going left to right, we chose to pick E for this case. By symmetry, while going from right to left, we would also expect to choose W for $d = 0$, but that would choose a pixel one unit up in y relative to the one chosen for the left-to-right scan. We therefore need to choose SW when $d = 0$ for right-to-left scanning. Similar adjustments need to be made for lines at other slopes.

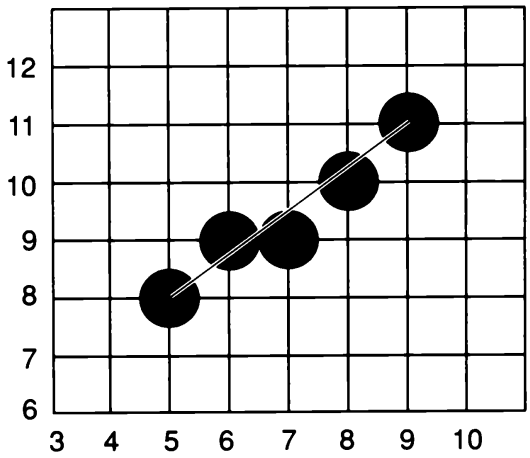

Fig. 3.9 The midpoint line from point (5, 8) to point (9, 11).

The alternative solution of switching a given line's endpoints as needed so that scan conversion always proceeds in the same direction does not work when we use line styles. The line style always "anchors" the specified write mask at the start point, which would be the bottom-left point, independent of line direction. That does not necessarily produce the desired visual effect. In particular, for a dot-dash line pattern of, say, 111100, we would like to have the pattern start at whichever start point is specified, not automatically at the bottom-left point. Also, if the algorithm always put endpoints in a canonical order, the pattern might go left to right for one segment and right to left for the adjoining segment, as a function of the second line's slope; this would create an unexpected discontinuity at the shared vertex, where the pattern should follow seamlessly from one line segment to the next.

Starting at the edge of a clip rectangle. Another issue is that we must modify our algorithm to accept a line that has been analytically clipped by one of the algorithms in Section 3.12. Fig. 3.10(a) shows a line being clipped at the left edge, $x = x_{min}$, of the clip rectangle. The intersection point of the line with the edge has an integer x coordinate but a real y coordinate. The pixel at the left edge, $(x_{min}, \text{Round}(mx_{min} + B))$, is the same pixel that would be drawn at this x value for the unclipped line by the incremental algorithm.[5] Given this initial pixel value, we must next initialize the decision variable at the midpoint between the E and NE positions in the next column over. It is important to realize that this strategy produces the correct sequence of pixels, while clipping the line at the x_{min} boundary and then scan converting the clipped line from $(x_{min}, \text{Round}(mx_{min} + B))$ to (x_1, y_1) using the integer midpoint line algorithm would not—that clipped line has a different slope!

The situation is more complicated if the line intersects a horizontal rather than a vertical edge, as shown in Fig. 3.10 (b). For the type of shallow line shown, there will be multiple pixels lying on the scan line $y = y_{min}$ that correspond to the bottom edge of the clip region. We want to count each of these as inside the clip region, but simply computing the analytical intersection of the line with the $y = y_{min}$ scan line and then rounding the x value of the intersection point would produce pixel A, not the leftmost point of the span of pixels shown, pixel B. From the figure, it is clear that the leftmost pixel of the span, B, is the one

[5]When $mx_{min} + B$ lies exactly halfway between horizontal grid lines, we actually must round down. This is a consequence of choosing pixel E when $d = 0$.

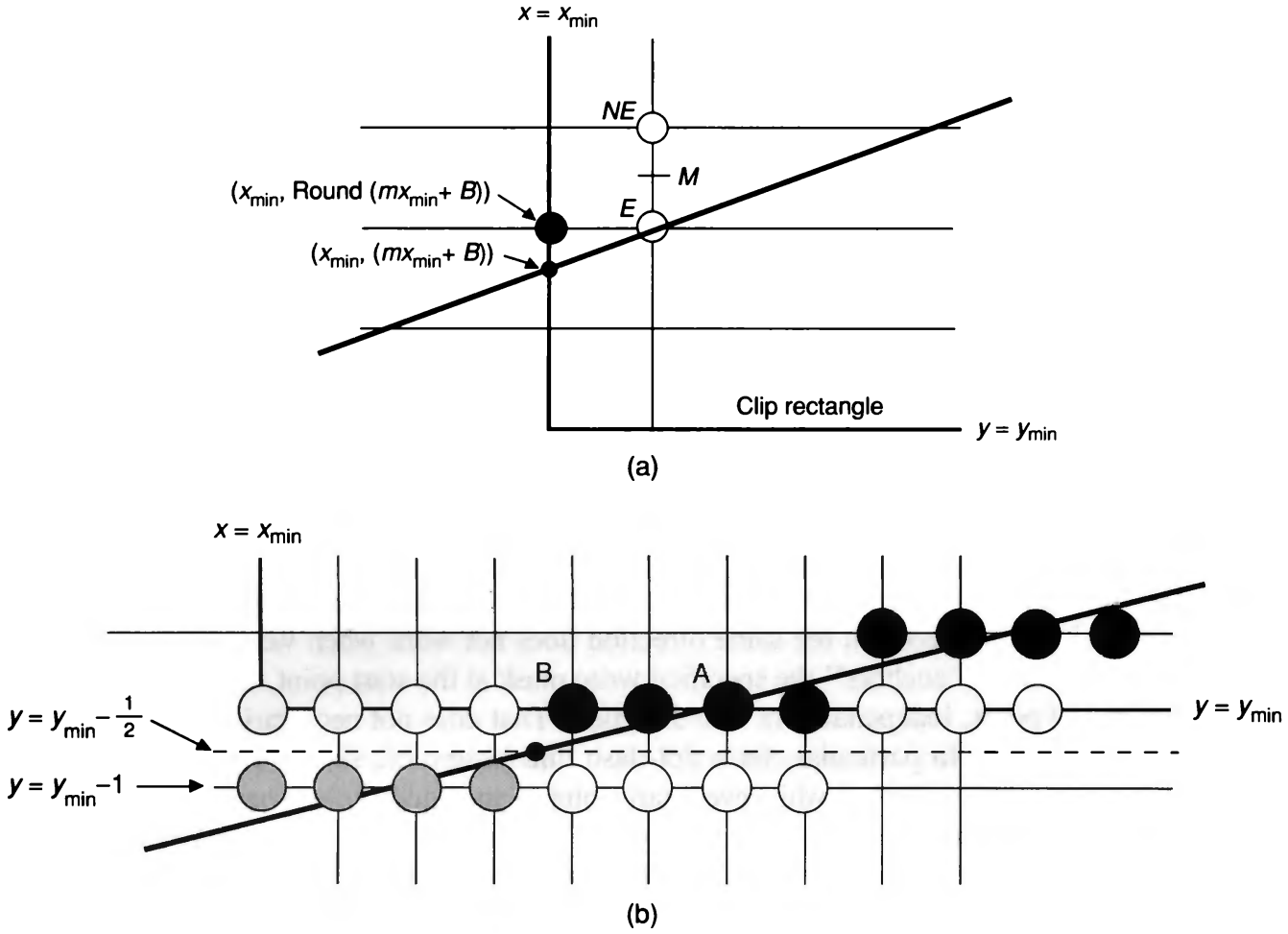

Fig. 3.10 Starting the line at a clip boundary. (a) Intersection with a vertical edge. (b) Intersection with a horizontal edge (gray pixels are on the line but are outside the clip rectangle).

that lies just above and to the right of the place on the grid where the line first crosses above the midpoint $y = y_{min} - \frac{1}{2}$. Therefore, we simply find the intersection of the line with the horizontal line $y = y_{min} - \frac{1}{2}$, and round up the x value; the first pixel, B, is then the one at $(\text{Round}(x_{y_{min}-\frac{1}{2}}), y_{min})$.

Finally, the incremental midpoint algorithm works even if endpoints are specified in a floating-point raster graphics package; the only difference is that the increments are now reals, and the arithmetic is done with reals.

Varying the intensity of a line as a function of slope. Consider the two scan converted lines in Fig. 3.11. Line B, the diagonal line, has a slope of 1 and hence is $\sqrt{2}$ times as long as A, the horizontal line. Yet the same number of pixels (10) is drawn to represent each line. If the intensity of each pixel is I, then the intensity per unit length of line A is I, whereas for line B it is only $I/\sqrt{2}$; this discrepancy is easily detected by the viewer. On a bilevel display, there is no cure for this problem, but on an n-bits-per-pixel system we can compensate by setting the intensity to be a function of the line's slope. Antialiasing, discussed in Section 3.17, achieves an even better result by treating the line as a thin rectangle and computing

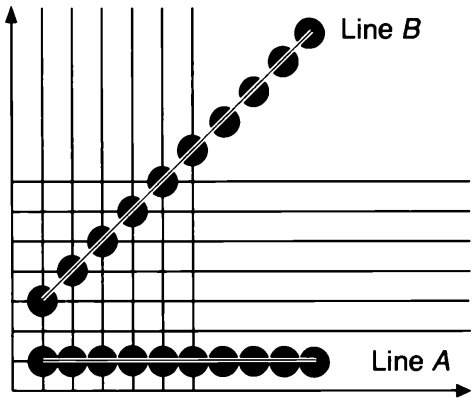

Fig. 3.11 Varying intensity of raster lines as a function of slope.

appropriate intensities for the multiple pixels in each column that lie in or near the rectangle.

Treating the line as a rectangle is also a way to create thick lines. In Section 3.9, we show how to modify the basic scan-conversion algorithms to deal with thick primitives and with primitives whose appearance is affected by line-style and pen-style attributes. Chapter 19 treats several other enhancements of the fundamental algorithms, such as handling endpoint shapes and creating joins between lines with multiple-pixel width.

Outline primitives composed of lines. Knowing how to scan convert lines, how do we scan convert primitives made from lines? Polylines can be scan-converted one line segment at a time. Scan converting rectangles and polygons as area-defining primitives could be done a line segment at a time but that would result in some pixels being drawn that lie outside a primitive's area—see Sections 3.5 and 3.6 for special algorithms to handle this problem. Care must be taken to draw shared vertices of polylines only once, since drawing a vertex twice causes it to change color or to be set to background when writing in **xor** mode to a screen, or to be written at double intensity on a film recorder. In fact, other pixels may be shared by two line segments that lie close together or cross as well. See Section 19.7 and Exercise 3.8 for a discussion of this, and of the difference between a polyline and a sequence of connected line segments.

3.3 SCAN CONVERTING CIRCLES

Although SRGP does not offer a circle primitive, the implementation will benefit from treating the circular ellipse arc as a special case because of its eight-fold symmetry, both for clipping and for scan conversion. The equation of a circle centered at the origin is $x^2 + y^2 = R^2$. Circles not centered at the origin may be translated to the origin by integer amounts and then scan converted, with pixels written with the appropriate offset. There are several easy but inefficient ways to scan convert a circle. Solving for y in the implicit circle equation, we get the explicit $y = f(x)$ as

$$y = \pm\sqrt{R^2 - x^2}.$$

To draw a quarter circle (the other quarters are drawn by symmetry), we can increment x from 0 to R in unit steps, solving for $+y$ at each step. This approach works, but it is

inefficient because of the multiply and square-root operations. Furthermore, the circle will have large gaps for values of x close to R, because the slope of the circle becomes infinite there (see Fig. 3.12). A similarly inefficient method, which does, however, avoid the large gaps, is to plot ($R \cos\theta$, $R \sin\theta$) by stepping θ from 0° to 90°.

3.3.1 Eight-Way Symmetry

We can improve the drawing process of the previous section by taking greater advantage of the symmetry in a circle. Consider first a circle centered at the origin. If the point (x, y) is on the circle, then we can trivially compute seven other points on the circle, as shown in Fig. 3.13. Therefore, we need to compute only one 45° segment to determine the circle completely. For a circle centered at the origin, the eight symmetrical points can be displayed with procedure CirclePoints (the procedure is easily generalized to the case of circles with arbitrary origins):

```
void CirclePoints (int x, int y, int value)
{
    WritePixel (x, y, value);
    WritePixel (y, x, value);
    WritePixel (y, −x, value);
    WritePixel (x, −y, value);
    WritePixel (−x, −y, value);
    WritePixel (−y, −x, value);
    WritePixel (−y, x, value);
    WritePixel (−x, y, value);
} /* CirclePoints */
```

We do not want to call CirclePoints when $x = y$, because each of four pixels would be set twice; the code is easily modified to handle that boundary condition.

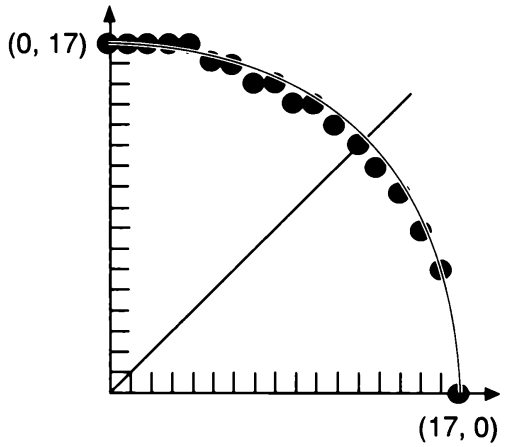

Fig. 3.12 A quarter circle generated with unit steps in x, and with y calculated and then rounded. Unique values of y for each x produce gaps.

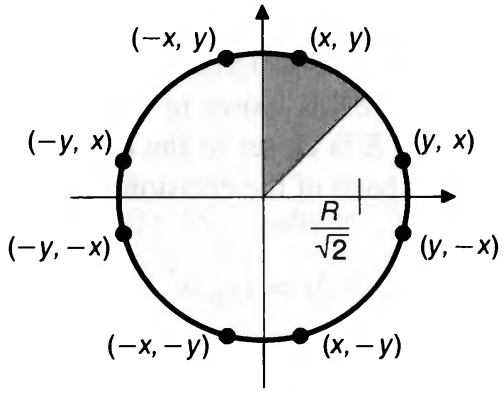

Fig. 3.13 Eight symmetrical points on a circle.

3.3.2 Midpoint Circle Algorithm

Bresenham [BRES77] developed an incremental circle generator that is more efficient than the methods we have discussed. Conceived for use with pen plotters, the algorithm generates all points on a circle centered at the origin by incrementing all the way around the circle. We derive a similar algorithm, again using the midpoint criterion, which, for the case of integer center point and radius, generates the same, optimal set of pixels. Furthermore, the resulting code is essentially the same as that specified in patent 4,371,933 [BRES83].

We consider only 45° of a circle, the second octant from $x = 0$ to $x = y = R/\sqrt{2}$, and use the CirclePoints procedure to display points on the entire circle. As with the midpoint line algorithm, the strategy is to select which of 2 pixels is closer to the circle by evaluating a function at the midpoint between the 2 pixels. In the second octant, if pixel P at (x_P, y_P) has been previously chosen as closest to the circle, the choice of the next pixel is between pixel E and SE (see Fig. 3.14).

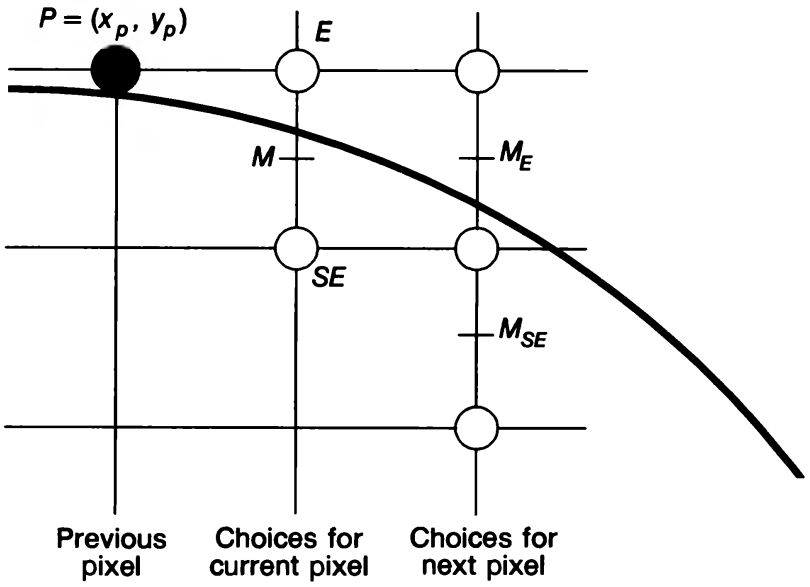

Fig. 3.14 The pixel grid for the midpoint circle algorithm showing M and the pixels E and SE to choose between.

Let $F(x, y) = x^2 + y^2 - R^2$; this function is 0 on the circle, positive outside the circle, and negative inside the circle. It can be shown that if the midpoint between the pixels E and SE is outside the circle, then pixel SE is closer to the circle. On the other hand, if the midpoint is inside the circle, pixel E is closer to the circle.

As for lines, we choose on the basis of the decision variable d, which is the value of the function at the midpoint,

$$d_{\text{old}} = F(x_P + 1, y_P - \tfrac{1}{2}) = (x_P + 1)^2 + (y_P - \tfrac{1}{2})^2 - R^2.$$

If $d_{\text{old}} < 0$, E is chosen, and the next midpoint will be one increment over in x. Then,

$$d_{\text{new}} = F(x_P + 2, y_P - \tfrac{1}{2}) = (x_P + 2)^2 + (y_P - \tfrac{1}{2})^2 - R^2,$$

and $d_{\text{new}} = d_{\text{old}} + (2x_P + 3)$; therefore, the increment $\Delta_E = 2x_P + 3$.

If $d_{\text{old}} \geq 0$, SE is chosen,[6] and the next midpoint will be one increment over in x and one increment down in y. Then

$$d_{\text{new}} = F(x_P + 2, y_P - \tfrac{3}{2}) = (x_P + 2)^2 + (y_P - \tfrac{3}{2})^2 - R^2.$$

Since $d_{\text{new}} = d_{\text{old}} + (2x_P - 2y_P + 5)$, the increment $\Delta_{SE} = 2x_P - 2y_P + 5$.

Recall that, in the linear case, Δ_E and Δ_{NE} were constants; in the quadratic case, however, Δ_E and Δ_{SE} vary at each step and are functions of the particular values of x_P and y_P at the pixel chosen in the previous iteration. Because these functions are expressed in terms of (x_P, y_P), we call P the *point of evaluation*. The Δ functions can be evaluated directly at each step by plugging in the values of x and y for the pixel chosen in the previous iteration. This direct evaluation is not expensive computationally, since the functions are only linear.

In summary, we do the same two steps at each iteration of the algorithm as we did for the line: (1) choose the pixel based on the sign of the variable d computed during the previous iteration, and (2) update the decision variable d with the Δ that corresponds to the choice of pixel. The only difference from the line algorithm is that, in updating d, we evaluate a linear function of the point of evaluation.

All that remains now is to compute the initial condition. By limiting the algorithm to integer radii in the second octant, we know that the starting pixel lies on the circle at $(0, R)$. The next midpoint lies at $(1, R - \frac{1}{2})$, therefore, and $F(1, R - \frac{1}{2}) = 1 + (R^2 - R + \frac{1}{4}) - R^2 = \frac{5}{4} - R$. Now we can implement the algorithm directly, as in Fig. 3.15. Notice how similar in structure this algorithm is to the line algorithm.

The problem with this version is that we are forced to do real arithmetic because of the fractional initialization of d. Although the procedure can be easily modified to handle circles that are not located on integer centers or do not have integer radii, we would like a more efficient, purely integer version. We thus do a simple program transformation to eliminate fractions.

First, we define a new decision variable, h, by $h = d - \frac{1}{4}$, and we substitute $h + \frac{1}{4}$ for d in the code. Now, the intialization is $h = 1 - R$, and the comparison $d < 0$ becomes $h < -\frac{1}{4}$.

---

[6]Choosing SE when $d = 0$ differs from our choice in the line algorithm and is arbitrary. The reader may wish to simulate the algorithm by hand to see that, for R = 17, 1 pixel is changed by this choice.

```
void MidpointCircle (int radius, int value)
/* Assumes center of circle is at origin */
{
    int x = 0;
    int y = radius;
    double d = 5.0 / 4.0 - radius;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0)          /* Select E */
            d += 2.0 * x + 3.0;
        else {              /* Select SE */
            d += 2.0 * (x - y) + 5.0;
            y--;
        }
        x++;
        CirclePoints (x, y, value);
    } /* while */
} /* MidpointCircle */
```

Fig. 3.15 The midpoint circle scan-conversion algorithm.

However, since h starts out with an integer value and is incremented by integer values (Δ_E and Δ_{SE}), we can change the comparison to just $h < 0$. We now have an integer algorithm in terms of h; for consistency with the line algorithm, we will substitute d for h throughout. The final, fully integer algorithm is shown in Fig. 3.16.

Figure 3.17 shows the second octant of a circle of radius 17 generated with the algorithm, and the first octant generated by symmetry (compare the results to Fig. 3.12).

Second-order differences. We can improve the performance of the midpoint circle algorithm by using the incremental computation technique even more extensively. We noted that the Δ functions are linear equations, and we computed them directly. Any polynomial can be computed incrementally, however, as we did with the decision variables for both the line and the circle. In effect, we are calculating *first-* and *second-order partial differences*, a useful technique that we encounter again in Chapters 11 and 19. The strategy is to evaluate the function directly at two adjacent points, to calculate the difference (which, for polynomials, is always a polynomial of lower degree), and to apply that difference in each iteration.

If we choose E in the current iteration, the point of evaluation moves from (x_P, y_P) to $(x_P + 1, y_P)$. As we saw, the first-order difference is $\Delta_{E_{old}}$ at $(x_P, y_P) = 2x_P + 3$. Therefore,

$$\Delta_{E_{new}} \text{ at } (x_P + 1, y_P) = 2(x_P + 1) + 3,$$

and the second-order difference is $\Delta_{E_{new}} - \Delta_{E_{old}} = 2$.

```
void MidpointCircle (int radius, int value)
/* Assumes center of circle is at origin. Integer arithmetic only */
{
    int x = 0;
    int y = radius;
    int d = 1 - radius;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0)          /* Select E */
            d += 2 * x + 3;
        else {              /* Select SE */
            d += 2 * (x - y) + 5;
            y--;
        }
        x++;
        CirclePoints (x, y, value);
    } /* while */
} /* MidpointCircle */
```

Fig. 3.16 The integer midpoint circle scan-conversion algorithm.

Similarly, $\Delta_{SE_{old}}$ at $(x_P, y_P) = 2x_P - 2y_P + 5$. Therefore,

$$\Delta_{SE_{new}} \text{ at } (x_P + 1, y_P) = 2(x_P + 1) - 2y_P + 5,$$

and the second-order difference is $\Delta_{SE_{new}} - \Delta_{SE_{old}} = 2$.

If we choose SE in the current iteration, the point of evaluation moves from (x_P, y_P) to $(x_P + 1, y_P - 1)$. Therefore,

$$\Delta_{E_{new}} \text{ at } (x_P + 1, y_P - 1) = 2(x_P + 1) + 3,$$

and the second-order difference is $\Delta_{E_{new}} - \Delta_{E_{old}} = 2$. Also,

$$\Delta_{SE_{new}} \text{ at } (x_P + 1, y_P - 1) = 2(x_P + 1) - 2(y_P - 1) + 5,$$

and the second-order difference is $\Delta_{SE_{new}} - \Delta_{SE_{old}} = 4$.

The revised algorithm then consists of the following steps: (1) choose the pixel based on the sign of the variable d computed during the previous iteration; (2) update the decision variable d with either Δ_E or Δ_{SE}, using the value of the corresponding Δ computed during the previous iteration; (3) update the Δs to take into account the move to the new pixel, using the constant differences computed previously; and (4) do the move. Δ_E and Δ_{SE} are initialized using the start pixel $(0, R)$. The revised procedure using this technique is shown in Fig. 3.18.

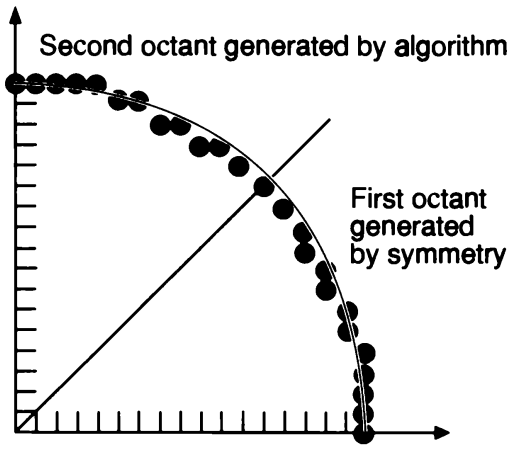

Fig. 3.17 Second octant of circle generated with midpoint algorithm, and first octant generated by symmetry.

```
void MidpointCircle (int radius, int value)
/* This procedure uses second-order partial differences to compute increments */
/* in the decision variable. Assumes center of circle is at origin */
{
    int x = 0;
    int y = radius;
    int d = 1 - radius;
    int deltaE = 3;
    int deltaSE = -2 * radius + 5;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0) {                /* Select E */
            d += deltaE;
            deltaE += 2;
            deltaSE += 2;
        } else {
            d += deltaSE;           /* Select SE */
            deltaE += 2;
            deltaSE += 4;
            y--;
        }
        x++;
        CirclePoints (x, y, value);
    }  /* while */
}  /* MidpointCircle */
```

Fig. 3.18 Midpoint circle scan-conversion algorithm using second-order differences.

3.4 SCAN CONVERTING ELLIPSES

Consider the standard ellipse of Fig. 3.19, centered at (0, 0). It is described by the equation

$$F(x, y) = b^2x^2 + a^2y^2 - a^2b^2 = 0,$$

where $2a$ is the length of the major axis along the x axis, and $2b$ is the length of the minor axis along the y axis. The midpoint technique discussed for lines and circles can also be applied to the more general conics. In this chapter, we consider the standard ellipse that is supported by SRGP; in Chapter 19, we deal with ellipses at any angle. Again, to simplify the algorithm, we draw only the arc of the ellipse that lies in the first quadrant, since the other three quadrants can be drawn by symmetry. Note also that standard ellipses centered at integer points other than the origin can be drawn using a simple translation. The algorithm presented here is based on Da Silva's algorithm, which combines the techniques used by Pitteway [PITT67], Van Aken [VANA84] and Kappel [KAPP85] with the use of partial differences [DASI89].

We first divide the quadrant into two regions; the boundary between the two regions is the point at which the curve has a slope of -1 (see Fig. 3.20).

Determining this point is more complex than it was for circles, however. The vector that is perpendicular to the tangent to the curve at point P is called the *gradient*, defined as

$$\text{grad } F(x, y) = \partial F/\partial x\ \mathbf{i} + \partial F/\partial y\ \mathbf{j} = 2b^2x\ \mathbf{i} + 2a^2y\ \mathbf{j}.$$

The boundary between the two regions is the point at which the slope of the curve is -1, and that point occurs when the gradient vector has a slope of 1—that is, when the **i** and **j** components of the gradient are of equal magnitude. The **j** component of the gradient is larger than the **i** component in region 1, and vice versa in region 2. Thus, if at the next midpoint, $a^2(y_P - \frac{1}{2}) \leq b^2(x_P + 1)$, we switch from region 1 to region 2.

As with any midpoint algorithm, we evaluate the function at the midpoint between two pixels and use the sign to determine whether the midpoint lies inside or outside the ellipse and, hence, which pixel lies closer to the ellipse. Therefore, in region 1, if the current pixel is located at (x_P, y_P), then the decision variable for region 1, d_1, is $F(x, y)$ evaluated at $(x_P + 1, y_P - \frac{1}{2})$, the midpoint between E and SE. We now repeat the process we used for deriving the

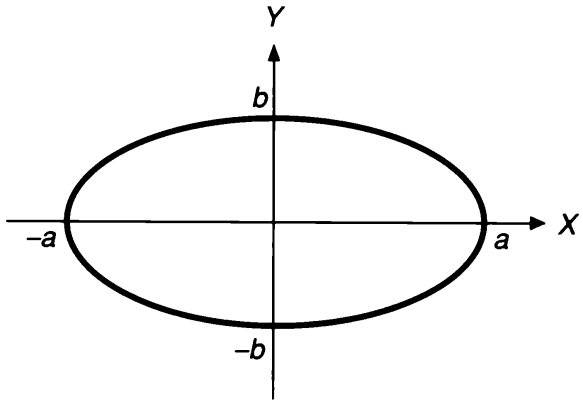

Fig. 3.19 Standard ellipse centered at the origin.

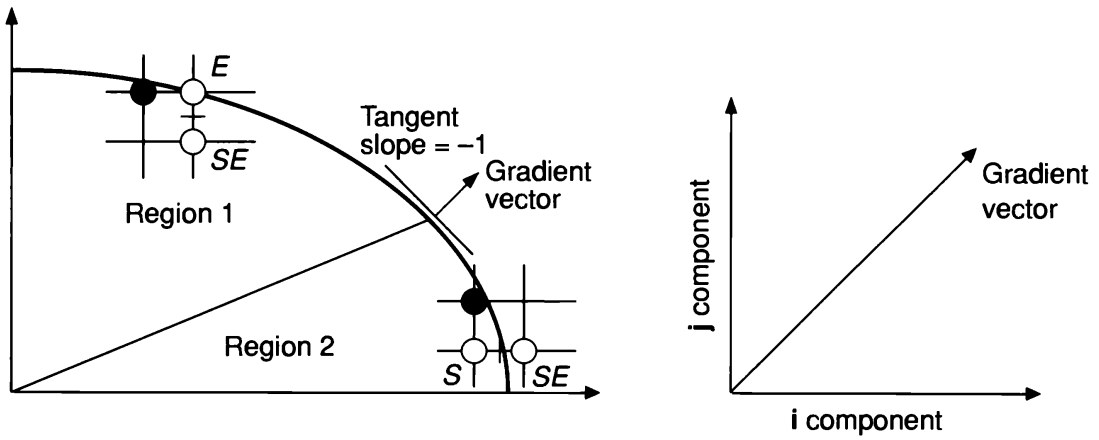

Fig. 3.20 Two regions of the ellipse defined by the 45° tangent.

two Δs for the circle. For a move to E, the next midpoint is one increment over in x. Then,

$$d_{\text{old}} = F(x_P + 1, y_P - \tfrac{1}{2}) = b^2(x_P + 1)^2 + a^2(y_P - \tfrac{1}{2})^2 - a^2b^2,$$

$$d_{\text{new}} = F(x_P + 2, y_P - \tfrac{1}{2}) = b^2(x_P + 2)^2 + a^2(y_P - \tfrac{1}{2})^2 - a^2b^2.$$

Since $d_{\text{new}} = d_{\text{old}} + b^2(2x_P + 3)$, the increment $\Delta_E = b^2(2x_P + 3)$.

For a move to SE, the next midpoint is one increment over in x and one increment down in y. Then,

$$d_{\text{new}} = F(x_P + 2, y_P - \tfrac{3}{2}) = b^2(x_P + 2)^2 + a^2(y_P - \tfrac{3}{2})^2 - a^2b^2.$$

Since $d_{\text{new}} = d_{\text{old}} + b^2(2x_P + 3) + a^2(-2y_P + 2)$, the increment $\Delta_{SE} = b^2(2x_P + 3) + a^2(-2y_P + 2)$.

In region 2, if the current pixel is at (x_P, y_P), the decision variable d_2 is $F(x_P + \frac{1}{2}, y_P - 1)$, the midpoint between S and SE. Computations similar to those given for region 1 may be done for region 2.

We must also compute the initial condition. Assuming integer values a and b, the ellipse starts at $(0, b)$, and the first midpoint to be calculated is at $(1, b - \frac{1}{2})$. Then,

$$F(1, b - \tfrac{1}{2}) = b^2 + a^2(b - \tfrac{1}{2})^2 - a^2b^2 = b^2 + a^2(-b + \tfrac{1}{4}).$$

At every iteration in region 1, we must not only test the decision variable d_1 and update the Δ functions, but also see whether we should switch regions by evaluating the gradient at the midpoint between E and SE. When the midpoint crosses over into region 2, we change our choice of the 2 pixels to compare from E and SE to SE and S. At the same time, we have to initialize the decision variable d_2 for region 2 to the midpoint between SE and S. That is, if the last pixel chosen in region 1 is located at (x_P, y_P), then the decision variable d_2 is initialized at $(x_P + \frac{1}{2}, y_P - 1)$. We stop drawing pixels in region 2 when the y value of the pixel is equal to 0.

As with the circle algorithm, we can either calculate the Δ functions directly in each iteration of the loop or compute them with differences. Da Silva shows that computation of second-order partials done for the Δs can, in fact, be used for the gradient as well [DASI89]. He also treats general ellipses that have been rotated and the many tricky

boundary conditions for very thin ellipses. The pseudocode algorithm of Fig. 3.21 uses the simpler direct evaluation rather than the more efficient formulation using second-order differences; it also skips various tests (see Exercise 3.9). In the case of integer a and b, we can eliminate the fractions via program transformations and use only integer arithmetic.

```
void MidpointEllipse (int a, int b, int value)
/* Assumes center of ellipse is at the origin. Note that overflow may occur */
/* for 16-bit integers because of the squares. */
{
    double d2;

    int x = 0;
    int y = b;
    double d1 = b^2 - (a^2 b) + (0.25 a^2);
    EllipsePoints (x, y, value);              /* The 4-way symmetrical WritePixel */

    /* Test gradient if still in region 1 */
    while ( a^2 (y - 0.5) > b^2 (x + 1) ) {   /* Region 1 */
        if (d1 < 0)                           /* Select E */
            d1 += b^2 (2x + 3);
        else {                                /* Select SE */
            d1 += b^2 (2x + 3) + a^2 (-2y + 2);
            y--;
        }
        x++;
        EllipsePoints (x, y, value);
    } /* Region 1 */

    d2 = b^2 (x + 0.5)^2 + a^2 (y - 1)^2 - a^2 b^2;
    while (y > 0) {                           /* Region 2 */
        if (d2 < 0) {                         /* Select SE */
            d2 += b^2 (2x + 2) + a^2 (-2y + 3);
            x++;
        } else
            d2 += a^2 (-2y + 3);              /* Select S */
        y--;
        EllipsePoints (x, y, value);
    } /* Region 2 */
} /* MidpointEllipse */
```

Fig. 3.21 Pseudocode for midpoint ellipse scan-conversion algorithm.

Now that we have seen how to scan convert lines 1 pixel thick as well as unfilled primitives, we turn our attention to modifications of these algorithms that fill area-defining primitives with a solid color or a pattern, or that draw unfilled primitives with a combination of the line-width and pen-style attributes.

3.5 FILLING RECTANGLES

The task of filling primitives can be broken into two parts: the decision of which pixels to fill (this depends on the shape of the primitive, as modified by clipping), and the easier decision of with what value to fill them. We first discuss filling unclipped primitives with a solid color; we deal with pattern filling in Section 3.8. In general, determining which pixels to fill consists of taking successive scan lines that intersect the primitive and filling in *spans* of adjacent pixels that lie inside the primitive from left to right.

To fill a rectangle with a solid color, we set each pixel lying on a scan line running from the left edge to the right edge to the same pixel value; i.e., fill each span from x_{min} to x_{max}. Spans exploit a primitive's *spatial coherence*: the fact that primitives often do not change from pixel to pixel within a span or from scan line to scan line. We exploit coherence in general by looking for only those pixels at which changes occur. For a solidly shaded primitive, all pixels on a span are set to the same value, which provides *span coherence*. The solidly shaded rectangle also exhibits strong *scan-line coherence* in that consecutive scan lines that intersect the rectangle are identical; later, we also use *edge coherence* for the edges of general polygons. We take advantage of various types of coherence not only for scan converting 2D primitives, but also for rendering 3D primitives, as discussed in Section 15.2.

Being able to treat multiple pixels in a span identically is especially important because we should write the frame buffer one word at a time to minimize the number of time-consuming memory accesses. For a bilevel display, we thus write 16 or 32 pixels at a time; if spans are not word-aligned, the algorithm must do suitable masking of words containing fewer than the full set of pixels. The need for writing memory efficiently is entirely similar for implementing copyPixel, as briefly discussed in Section 3.16. In our code, we concentrate on defining spans and ignore the issue of writing memory efficiently; see Chapters 4 and 19 and Exercise 3.13.

Rectangle scan conversion is thus simply a nested **for** loop:

```
for (y from ymin to ymax of the rectangle)      /* By scan line */
    for (x from xmin to xmax )                  /* By pixel in span */
        WritePixel (x, y, value);
```

An interesting problem arises in this straightforward solution, similar to the problem of scan converting a polyline with line segments that share pixels. Consider two rectangles that share a common edge. If we scan convert each rectangle in turn, we will write the pixels on the shared edge twice, which is undesirable, as noted earlier. This problem is a manifestation of a larger problem of area-defining primitives, that of defining which pixels belong to a primitive and which pixels do not. Clearly, those pixels that lie in the mathematical interior of an area-defining primitive belong to that primitive. But what about those pixels on the boundary? If we were looking at a single rectangle (or just thinking

about the problem in a mathematical way), a straightforward answer would be to include the pixels on the boundary, but since we want to avoid the problem of scan converting shared edges twice, we must define some rule that assigns boundary pixels uniquely.

A simple rule is to say that a boundary pixel—that is, a pixel lying on an edge—is not considered part of the primitive if the halfplane defined by that edge and containing the primitive lies below or to the left of the edge. Thus, pixels on left and bottom edges will be drawn, but pixels that lie on top and right edges will not be drawn. A shared vertical edge therefore "belongs" to the rightmost of the two sharing rectangles. In effect, spans within a rectangle represent an interval that is closed on the left end and open on the right end.

A number of points must be made about this rule. First, it applies to arbitrary polygons as well as to rectangles. Second, the bottom-left vertex of a rectangle still would be drawn twice—we need another rule to deal with that special case, as discussed in the next section. Third, we may apply the rule also to unfilled rectangles and polygons. Fourth, the rule causes each span to be missing its rightmost pixel, and each rectangle to be missing its topmost row. These problems illustrate that there is no "perfect" solution to the problem of not writing pixels on (potentially) shared edges twice, but implementors generally consider that it is better (visually less distracting) to have missing pixels at the right and top edge than it is to have pixels that disappear or are set to unexpected colors in **xor** mode.

3.6 FILLING POLYGONS

The general polygon scan-conversion algorithm described next handles both convex and concave polygons, even those that are self-intersecting or have interior holes. It operates by computing spans that lie between left and right edges of the polygon. The span extrema are calculated by an incremental algorithm that computes a scan line/edge intersection from the intersection with the previous scan line. Figure 3.22, which illustrates the basic polygon scan-conversion process, shows a polygon and one scan line passing through it. The intersections of scan line 8 with edges *FA* and *CD* lie on integer coordinates, whereas those for *EF* and *DE* do not; the intersections are marked in the figure by vertical tick marks labeled *a* through *d*.

We must determine which pixels on each scan line are within the polygon, and we must set the corresponding pixels (in this case, spans from $x = 2$ through 4 and 9 through 13) to

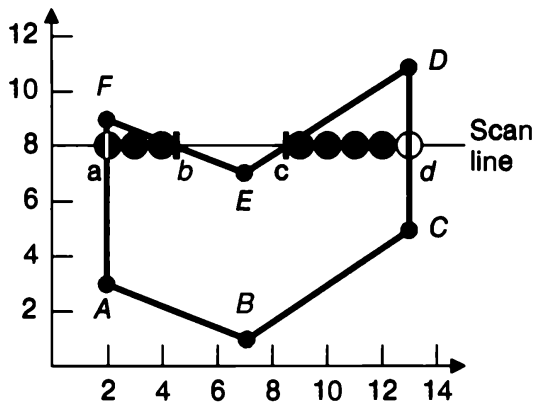

Fig. 3.22 Polygon and scan line 8.

their appropriate values. By repeating this process for each scan line that intersects the polygon, we scan convert the entire polygon, as shown for another polygon in Fig. 3.23.

Figure 3.23(a) shows the pixels defining the extrema of spans in black and the interior pixels on the span in gray. A straightforward way of deriving the extrema is to use the midpoint line scan-conversion algorithm on each edge and to keep a table of span extrema for each scan line, updating an entry if a new pixel is produced for an edge that extends the span. Note that this strategy produces some extrema pixels that lie outside the polygon; they were chosen by the scan-conversion algorithm because they lie closest to an edge, without regard to the side of the edge on which they lie—the line algorithm has no notions of interior and exterior. We do not want to draw such pixels on the outside of a shared edge, however, because they would intrude into the regions of neighboring polygons, and this would look odd if these polygons had different colors. It is obviously preferable to draw only those pixels that are strictly interior to the region, even when an exterior pixel would be closer to the edge. We must therefore adjust the scan-conversion algorithm accordingly; compare Fig. 3.23 (a) with Fig. 3.23 (b), and note that a number of pixels outside the ideal primitive are not drawn in part (b).

With this technique, a polygon does not intrude (even by a single pixel) into the regions defined by other primitives. We can apply the same technique to unfilled polygons for consistency or can choose to scan convert rectangles and polygons a line segment at a time, in which case unfilled and filled polygons do not contain the same boundary pixels!

As with the original midpoint algorithm, we use an incremental algorithm to calculate the span extrema on one scan line from those at the previous scan line without having to compute the intersections of a scan line with each polygon edge analytically. In scan line 8 of Fig. 3.22, for instance, there are two spans of pixels within the polygon. The spans can be filled in by a three-step process:

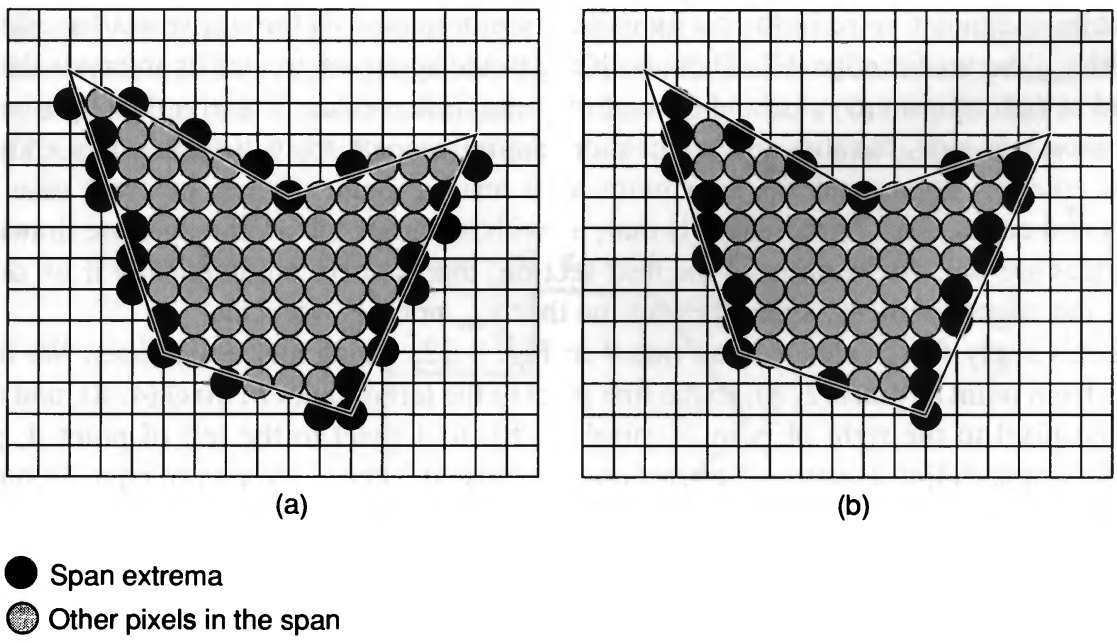

Fig. 3.23 Spans for a polygon. Extrema shown in black, interior pixels in gray. (a) Extrema computed by midpoint algorithm. (b) Extrema interior to polygon.

1. Find the intersections of the scan line with all edges of the polygon.
2. Sort the intersections by increasing x coordinate.
3. Fill in all pixels between pairs of intersections that lie interior to the polygon, using the odd-parity rule to determine that a point is inside a region: Parity is initially even, and each intersection encountered thus inverts the parity bit—draw when parity is odd, do not draw when it is even.

The first two steps of the process, finding intersections and sorting them, are treated in the next section. Let's look now at the span-filling strategy. In Fig. 3.22, the sorted list of x coordinates is (2, 4.5, 8.5, 13). Step 3 requires four elaborations:

3.1 Given an intersection with an arbitrary, fractional x value, how do we determine which pixel on either side of that intersection is interior?

3.2 How do we deal with the special case of intersections at integer pixel coordinates?

3.3 How do we deal with the special case in 3.2 for shared vertices?

3.4 How do we deal with the special case in 3.2 in which the vertices define a horizontal edge?

To handle case 3.1, we say that, if we are approaching a fractional intersection to the right and are inside the polygon, we round down the x coordinate of the intersection to define the interior pixel; if we are outside the polygon, we round up to be inside. We handle case 3.2 by applying the criterion we used to avoid conflicts at shared edges of rectangles: If the leftmost pixel in a span has integer x coordinate, we define it to be interior; if the rightmost pixel has integer x coordinate, we define it to be exterior. For case 3.3, we count the y_{min} vertex of an edge in the parity calculation but not the y_{max} vertex; therefore, a y_{max} vertex is drawn only if it is the y_{min} vertex for the adjacent edge. Vertex A in Fig. 3.22, for example, is counted once in the parity calculation because it is the y_{min} vertex for edge FA but the y_{max} vertex for edge AB. Thus, both edges and spans are treated as intervals that are closed at their minimum value and open at their maximum value. Clearly, the opposite rule would work as well, but this rule seems more natural since it treats the minimum endpoint as an entering point, and the maximum as a leaving point. When we treat case 3.4, horizontal edges, the desired effect is that, as with rectangles, bottom edges are drawn but top edges are not. As we show in the next section, this happens automatically if we do not count the edges' vertices, since they are neither y_{min} nor y_{max} vertices.

Let's apply these rules to scan line 8 in Fig. 3.22, which hits no vertices. We fill in pixels from point a, pixel (2, 8), to the first pixel to the left of point b, pixel (4, 8), and from the first pixel to the right of point c, pixel (9, 8), to 1 pixel to the left of point d, pixel (12, 8). For scan line 3, vertex A counts once because it is the y_{min} vertex of edge FA but the y_{max} vertex of edge AB; this causes odd parity, so we draw the span from there to 1 pixel to the left of the intersection with edge CB, where the parity is set to even and the span is terminated. Scan line 1 hits only vertex B; edges AB and BC both have their y_{min} vertices at B, which is therefore counted twice and leaves the parity even. This vertex acts as a null span—enter at the vertex, draw the pixel, exit at the vertex. Although such local minima

draw a single pixel, no pixel is drawn at a local maximum, such as the intersection of scan line 9 with the vertex F, shared by edges FA and EF. Both vertices are y_{max} vertices and therefore do not affect the parity, which stays even.

3.6.1 Horizontal Edges

We deal properly with horizontal edges by not counting their vertices, as we can see by examining various cases in Fig. 3.24. Consider bottom edge AB. Vertex A is a y_{min} vertex for edge JA, and AB does not contribute. Therefore, the parity is odd and the span AB is drawn. Vertical edge BC has its y_{min} at B, but again AB does not contribute. The parity becomes even and the span is terminated. At vertex J, edge IJ has a y_{min} vertex but edge JA does not, so the parity becomes odd and the span is drawn to edge BC. The span that starts at edge IJ and hits C sees no change at C because C is a y_{max} vertex for BC, so the span continues along bottom edge CD; at D, however, edge DE has a y_{min} vertex, so the parity is reset to even and the span ends. At I, edge IJ has its y_{max} vertex and edge HI also does not contribute, so parity stays even and the top edge IH is not drawn. At H, however, edge GH has a y_{min} vertex, the parity becomes odd, and the span is drawn from H to the pixel to the left of the intersection with edge EF. Finally, there is no y_{min} vertex at G, nor is there one at F, so top edge FG is not drawn.

The algorithm above deals with shared vertices in a polygon, with edges shared by two adjacent polygons, and with horizontal edges. It allows self-intersecting polygons. As noted, it does not work perfectly in that it omits pixels. Worse, it cannot totally avoid writing shared pixels multiple times without keeping a history: Consider edges shared by more than two polygons or a y_{min} vertex shared by two otherwise disjoint triangles (see Exercise 3.14).

3.6.2 Slivers

There is another problem with our scan-conversion algorithm that is not resolved as satisfactorily as is that of horizontal edges: polygons with edges that lie sufficiently close together create a *sliver*—a polygonal area so thin that its interior does not contain a distinct span for each scan line. Consider, for example, the triangle from (0, 0) to (3, 12) to (5, 12) to (0, 0), shown in Fig. 3.25. Because of the rule that only pixels that lie interior or on a left

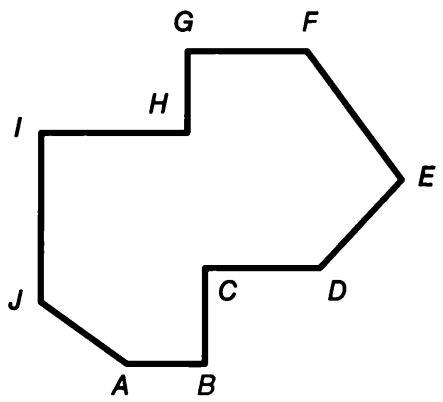

Fig. 3.24 Horizontal edges in a polygon.

or bottom edge are drawn, there will be many scan lines with only a single pixel or no pixels. The problem of having "missing" pixels is yet another example of the *aliasing* problem; that is, of representing a continuous signal with a discrete approximation. If we had multiple bits per pixel, we could use antialiasing techniques, as introduced for lines in Section 3.17 and for polygons in Section 19.3. Antialiasing would involve softening our rule "draw only pixels that lie interior or on a left or bottom edge" to allow boundary pixels and even exterior pixels to take on intensity values that vary as a function of distance between a pixel's center and the primitive; multiple primitives can then contribute to a pixel's value.

3.6.3 Edge Coherence and the Scan-Line Algorithm

Step 1 in our procedure—calculating intersections—must be done cleverly lest it be slow. In particular, we must avoid the brute-force technique of testing each polygon edge for intersection with each new scan line. Very often, only a few of the edges are of interest for a given scan line. Furthermore, we note that many edges intersected by scan line i are also intersected by scan line $i + 1$. This *edge coherence* occurs along an edge for as many scan lines as intersect the edge. As we move from one scan line to the next, we can compute the new x intersection of the edge on the basis of the old x intersection, just as we computed the next pixel from the current pixel in midpoint line scan conversion, by using

$$x_{i+1} = x_i + 1/m,$$

where m is the slope of the edge. In the midpoint algorithm for scan converting lines, we avoided fractional arithmetic by computing an integer decision variable and checking only its sign to choose the pixel closest to the mathematical line; here, we would like to use integer arithmetic to do the required rounding for computing the closest interior pixel.

Consider lines with a slope greater than +1 that are left edges; right edges and other slopes are handled by similar, though somewhat trickier, arguments, and vertical edges are special cases. (Horizontal edges are handled implicitly by the span rules, as we saw.) At the (x_{min}, y_{min}) endpoint, we need to draw a pixel. As y is incremented, the x coordinate of the point on the ideal line will increase by $1/m$, where $m = (y_{max} - y_{min})/(x_{max} - x_{min})$ is the

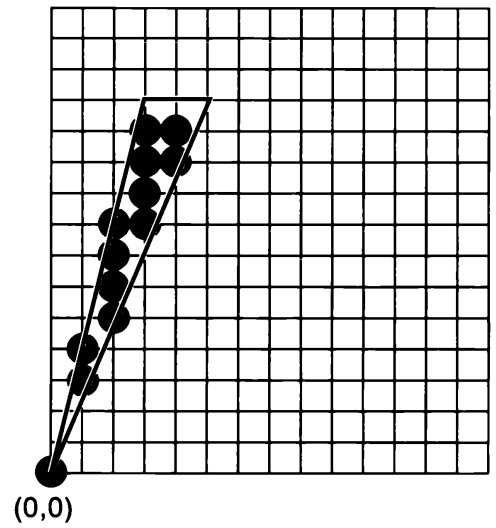

Fig. 3.25 Scan converting slivers of polygons.

slope of the line. This increase will result in x having an integer and a fractional part, which can be expressed as a fraction with a denominator of $y_{max} - y_{min}$. As we iterate this process, the fractional part will overflow and the integer part will have to be incremented. For example, if the slope is $\frac{5}{2}$, and x_{min} is 3, then the sequence of x values will be 3, $3\frac{2}{5}$, $3\frac{4}{5}$, $3\frac{6}{5} = 4\frac{1}{5}$, and so on. When the fractional part of x is zero, we can draw the pixel (x, y) that lies on the line, but when the fractional part of x is nonzero, we need to round up in order to get a pixel that lies strictly inside the line. When the fractional part of x becomes greater than 1, we increment x and subtract 1 from the fractional part; we must also move 1 pixel to the right. If we increment to lie exactly on a pixel, we draw that pixel but must decrement the fraction by 1 to have it be less than 1.

We can avoid the use of fractions by keeping track only of the numerator of the fraction and observing that the fractional part is greater than 1 when the numerator is greater than the denominator. We implement this technique in the algorithm of Fig. 3.26, using the variable *increment* to keep track of successive additions of the numerator until it "overflows" past the denominator, when the numerator is decremented by the denominator and x is incremented.

We now develop a *scan-line algorithm* that takes advantage of this edge coherence and, for each scan line, keeps track of the set of edges it intersects and the intersection points in a data structure called the *active-edge table* (AET). The edges in the AET are sorted on their x intersection values so that we can fill the spans defined by pairs of (suitably rounded) intersection values—that is, the span extrema. As we move to the next scan line at $y + 1$, the AET is updated. First, edges currently in the AET but not intersected by this next scan line (i.e., those whose $y_{max} = y$) are deleted. Second, any new edges intersected by this next scan line (i.e., those edges whose $y_{min} = y + 1$) are added to the AET. Finally, new x intersections are calculated, using the preceding incremental edge algorithm, for edges that were in the AET but are not yet completed.

```
void LeftEdgeScan (int xmin, int ymin, int xmax, int ymax, int value)
{
    int y;

    int x = xmin;
    int numerator = xmax - xmin;
    int denominator = ymax - ymin;
    int increment = denominator;

    for (y = ymin; y <= ymax; y++) {
        WritePixel (x, y, value);
        increment += numerator;
        if (increment > denominator) {
            /* Overflow, so round up to next pixel and decrement the increment. */
            x++;
            increment -= denominator;
        }
    }
} /* LeftEdgeScan */
```

Figure 3.26 Scan converting left edge of a polygon.

To make the addition of edges to the AET efficient, we initially create a global *edge table* (ET) containing all edges sorted by their smaller y coordinate. The ET is typically built by using a bucket sort with as many buckets as there are scan lines. Within each bucket, edges are kept in order of increasing x coordinate of the lower endpoint. Each entry in the ET contains the y_{max} coordinate of the edge, the x coordinate of the bottom endpoint (x_{min}), and the x increment used in stepping from one scan line to the next, $1/m$. Figure 3.27 shows how the six edges from the polygon of Fig. 3.22 would be sorted, and Fig. 3.28 shows the AET at scan lines 9 and 10 for that polygon. (In an actual implementation, we would probably add a flag indicating left or right edge.)

Once the ET has been formed, the processing steps for the scan-line algorithm are as follows:

1. Set y to the smallest y coordinate that has an entry in the ET; i.e., y for the first nonempty bucket
2. Initialize the AET to be empty
3. Repeat until the AET and ET are empty:
   - 3.1 Move from ET bucket y to the AET those edges whose $y_{min} = y$ (entering edges).
   - 3.2 Remove from the AET those entries for which $y = y_{max}$ (edges not involved in the next scan line), then sort the AET on x (made easier because ET is presorted).
   - 3.3 Fill in desired pixel values on scan line y by using pairs of x coordinates from the AET
   - 3.4 Increment y by 1 (to the coordinate of the next scan line)
   - 3.5 For each nonvertical edge remaining in the AET, update x for the new y

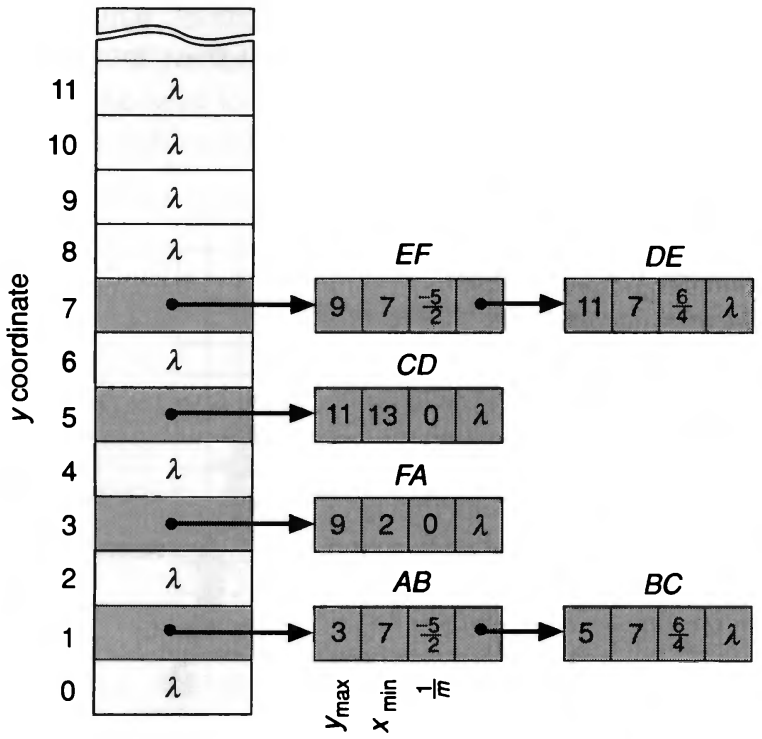

Fig. 3.27 Bucket-sorted edge table for polygon of Fig. 3.22.

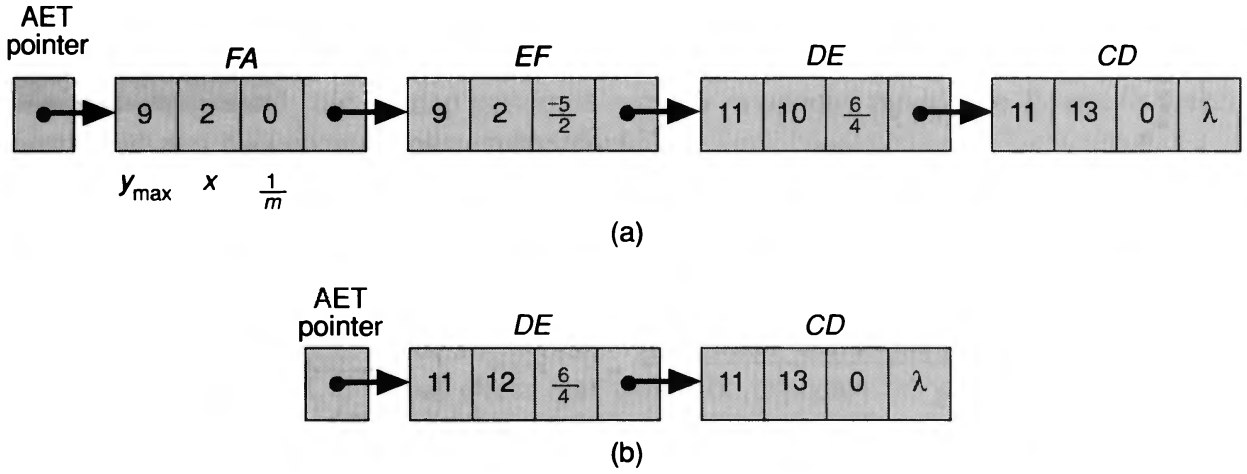

Fig. 3.28 Active-edge table for polygon of Fig. 3.22. (a) Scan line 9. (b) Scan line 10. (Note *DE*'s *x* coordinate in (b) has been rounded up for that left edge.)

This algorithm uses both edge coherence to calculate x intersections and scan-line coherence (along with sorting) to calculate spans. Since the sorting works on a small number of edges and since the resorting of step 3.1 is applied to a mostly or completely sorted list, either insertion sort or a simple bubble sort that is $O(N)$ in this case may be used. In Chapters 15 and 16, we see how to extend this algorithm to handle multiple polygons during visible-surface determination, including the case of handling polygons that are transparent; in Chapter 17, we see how to blend polygons that overlap at a pixel.

For purposes of scan conversion, triangles and trapezoids can be treated as special cases of polygons, since they have only two edges for any scan line (given that horizontal edges are not scan-converted explicitly). Indeed, since an arbitrary polygon can be decomposed into a mesh of triangles sharing vertices and edges (see Exercise 3.17), we could scan convert general polygons by first decomposing them into triangle meshes, and then scan converting the component triangles. Such triangulation is a classic problem in computational geometry [PREP85] and is easy to do for convex polygons; doing it efficiently for nonconvex polygons is difficult.

Note that the calculation of spans is cumulative. That is, when the current iteration of the scan-conversion algorithm in Step 3.5 generates multiple pixels falling on the same scan line, the span extrema must be updated appropriately. (Dealing with span calculations for edges that cross and for slivers takes a bit of special casing.) We can either compute all spans in one pass, then fill the spans in a second pass, or compute a span and fill it when completed. Another benefit of using spans is that clipping can be done at the same time as span arithmetic: The spans may be individually clipped at the left and right coordinates of the clip rectangle. Note that, in Section 15.10.3 we use a slightly different version of span arithmetic to combine 3D solid objects that are rendered using "raytracing."

3.7 FILLING ELLIPSE ARCS

The same general strategy of calculating spans for each scan line can be used for circles and ellipses as well. We accumulate span extrema for each iteration of the algorithm, rounding each extremum to ensure that the pixel is inside the region. As with scan converting the

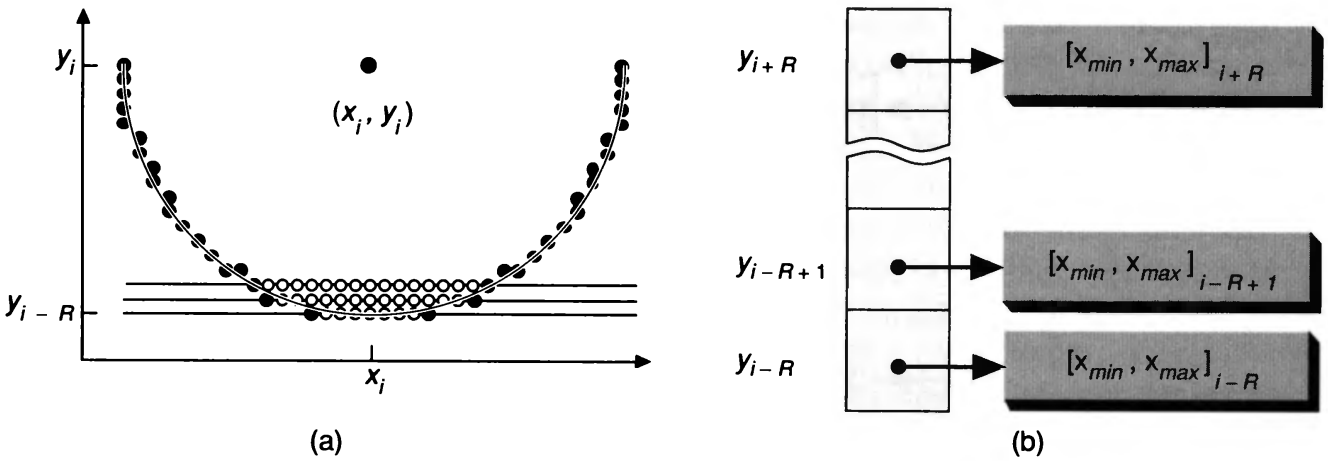

Fig. 3.29 Filling a circle with spans. (a) Three spans. (b) Span table. Each span is stored with its extrema.

unfilled primitive, we can take advantage of symmetry to scan convert only one arc, being careful of region changes, especially for ellipses. Each iteration generates either a new pixel on the same scan line, thereby potentially adjusting the span extrema, or a pixel on the next scan line, starting the next span (see Fig. 3.29). To determine whether a pixel P lies inside the region, we simply check the sign of the function $F(P)$ and choose the next pixel over if the sign is positive. Clearly, we do not want to evaluate the function directly, any more than we wanted to evaluate the function $F(M)$ at the midpoint directly; we can derive the value of the function from that of the decision variable.

Since we know that a scan line crosses the boundary only twice, we do not need any equivalent of an edge table, and we maintain only a current span. As we did for polygons, we can either make a list of such spans for each scan line intersecting the primitive and then fill them after they have all been computed, or we can fill each one as it is completed—for example, as soon as the y value is incremented.

The special case of filled wedges should be mentioned. The first problem is to calculate the intersection of the rays that define the starting and ending angles with the boundary, and to use those to set the starting and ending values of the decision variable. For circles we can do this calculation by converting from rectangular degrees to circular degrees, then using (Round($R \cos\theta$), Round($R \sin\theta$)) in the midpoint formula. The perimeter of the region to scan convert, then, consists of the two rays and the boundary arc between them. Depending on the angles, a scan line may start or end on either a ray or the boundary arc, and the corresponding incremental algorithms, modified to select only interior pixels, must be applied (see Exercise 3.19).

3.8 PATTERN FILLING

In the previous sections, we filled the interiors of SRGP's area-defining primitives with a solid color by passing the color in the *value* field of the WritePixel procedure. Here, we consider filling with a pattern, which we do by adding extra control to the part of the scan-conversion algorithm that actually writes each pixel. For pixmap patterns, this control causes the color value to be picked up from the appropriate position in the pixmap pattern,

as shown next. To write bitmap patterns transparently, we do a WritePixel with foreground color at a pixel for a 1 in the pattern, and we inhibit the WritePixel for a 0, as with line style. If, on the other hand, the bitmap pattern is applied in opaque mode, the 1s and 0s select foreground and background color, respectively.

The main issue for filling with patterns is the relation of the area of the pattern to that of the primitive. In other words, we need to decide where the pattern is "anchored" so that we know which pixel in the pattern corresponds to the current pixel of the primitive.

The first technique is to anchor the pattern at a vertex of a polygon by placing the leftmost pixel in the pattern's first row there. This choice allows the pattern to move when the primitive is moved, a visual effect that would be expected for patterns with a strong geometric organization, such as the cross-hatches often used in drafting applications. But there is no distinguished point on a polygon that is obviously right for such a relative anchor, and no distinguished points at all on smoothly varying primitives such as circles and ellipses. Therefore, the programmer must specify the anchor point as a point on or within the primitive. In some systems, the anchor point may even be applied to a group of primitives.

The second technique, used in SRGP, is to consider the entire screen as being tiled with the pattern and to think of the primitive as consisting of an outline or filled area of transparent bits that let the pattern show through. The standard position for such an absolute anchor is the screen origin. The pixels of the primitive are then treated as 1s that are **and**ed with the pattern. A side effect of this technique is that the pattern does not "stick to" the primitive if the primitive is moved slightly. Instead, the primitive moves as though it were a cutout on a fixed, patterned background, and thus its appearance may change as it is moved; for regular patterns without a strong geometric orientation, users may not even be aware of this effect. In addition to being computationally efficient, absolute anchoring allows primitives to overlap and abut seamlessly.

To apply the pattern to the primitive, we index it with the current pixel's (x, y) coordinates. Since patterns are defined as small M by N bitmaps or pixmaps, we use modular arithmetic to make the pattern repeat. The *pattern*[0, 0] pixel is considered coincident with the screen origin,[7] and we can write, for example, a bitmap pattern in transparent mode with the statement

```
if (pattern[x % M][y % N])
    WritePixel (x, y, value);
```

If we are filling an entire span in **replace** write mode, we can copy a whole row of the pattern at once, assuming a low-level version of a copyPixel facility is available to write multiple pixels. Let's say, for example, that the pattern is an 8 by 8 matrix. It thus repeats for every span of 8 pixels. If the leftmost point of a span is byte-aligned—that is, if the x value of the first pixel mod 8 is 0—then the entire first row of the pattern can be written out with a copyPixel of a 1 by 8 array; this procedure is repeated as many times as is necessary to fill the span. If either end of the span is not byte-aligned, the pixels not in the span must be masked out. Implementors spend much time making special cases of raster algorithms particularly efficient; for example, they test up-front to eliminate inner loops, and they write

[7] In window systems, the pattern is often anchored at the origin of the window coordinate system.

hand-tuned assembly-language code for inner loops that takes advantage of special hardware features such as instruction caches or particularly efficient loop instructions. This type of optimization is discussed in Chapter 19.

3.8.1 Pattern Filling Without Repeated Scan Conversion

So far, we have discussed filling in the context of scan conversion. Another technique is to scan convert a primitive first into a rectangular work area, and then to write each pixel from that bitmap to the appropriate place in the canvas. This so-called *rectangle write* to the canvas is simply a nested **for** loop in which a 1 writes the current color and a 0 writes nothing (for transparency) or writes the background color (for opacity). This two-step process is twice as much work as filling during scan conversion, and therefore is not worthwhile for primitives that are encountered and scan-converted only once. It pays off, however, for primitives that would otherwise be scan-converted repeatedly. This is the case for characters in a given font and size, which can be scan-converted ahead of time from their outlines. For characters defined only as bitmap fonts, or for other objects, such as icons and application symbols, that are painted or scanned in as bitmap images, scan conversion is not used in any case, and the rectangle write of their bitmaps is the only applicable technique. The advantage of a pre–scan-converted bitmap lies in the fact that it is clearly faster to write each pixel in a rectangular region, without having to do any clipping or span arithmetic, than to scan convert the primitive each time from scratch while doing such clipping.[8]

But since we have to write a rectangular bitmap into the canvas, why not just copyPixel the bitmap directly, rather than writing 1 pixel at a time? For bilevel displays, writing current color 1, copyPixel works fine: For transparent mode, we use **or** write mode; for opaque mode, we use **replace** write mode. For multilevel displays, we cannot write the bitmap directly with a single bit per pixel, but must convert each bit to a full n-bit color value that is then written.

Some systems have a more powerful copyPixel that can make copying subject to one or more source-read or destination-write masks. We can make good use of such a facility for transparent mode (used for characters in SRGP) if we can specify the bitmap as a destination-write mask and the source as an array of constant (current) color. Then, pixels are written in the current color only where the bitmap write mask has 1s; the bitmap write mask acts as an arbitrary clip region. In a sense, the explicit nested **for** loop for implementing the rectangle write on n-bits-per-pixel systems simulates this more powerful "copyPixel with write mask" facility.

Now consider another variation. We wish to draw a filled letter, or some other shape, not with a solid interior but with a patterned one. For example, we would like to create a thick letter "P" with a 50 percent gray stipple pattern (graying out the character), or a house icon with a two-tone brick-and-mortar pattern. How can we write such an object in opaque mode without having to scan convert it each time? The problem is that "holes" interior to the region where there are 0s in the bitmap should be written in background color, whereas holes outside the region (such as the cavity in the "P") must still be written

[8]There are added complications in the case of antialiasing, as discussed in Chapter 19.

transparently so as not to affect the image underneath. In other words, we want 0s in the shape's interior to signify background color, and 0s in its exterior, including any cavities, to belong to a write mask used to protect pixels outside the shape. If we scan convert on the fly, the problem that the 0s mean different things in different regions of the bitmap does not arise, because we never look at pixels outside the shape's boundary.

We use a four-step process to avoid repeated scan conversion, as shown in the mountain scene of Fig. 3.30. Using the outline of our icon (b), the first step is to create a "solid" bitmap to be used as a write mask/clipping region, with pixels interior to the object set to 1s, and those exterior set to 0s; this is depicted in (c), where white represents background pixels (0s) and black represents 1s. This scan conversion is done only once. As the second step, any time a patterned copy of the object is needed, we write the solid bitmap

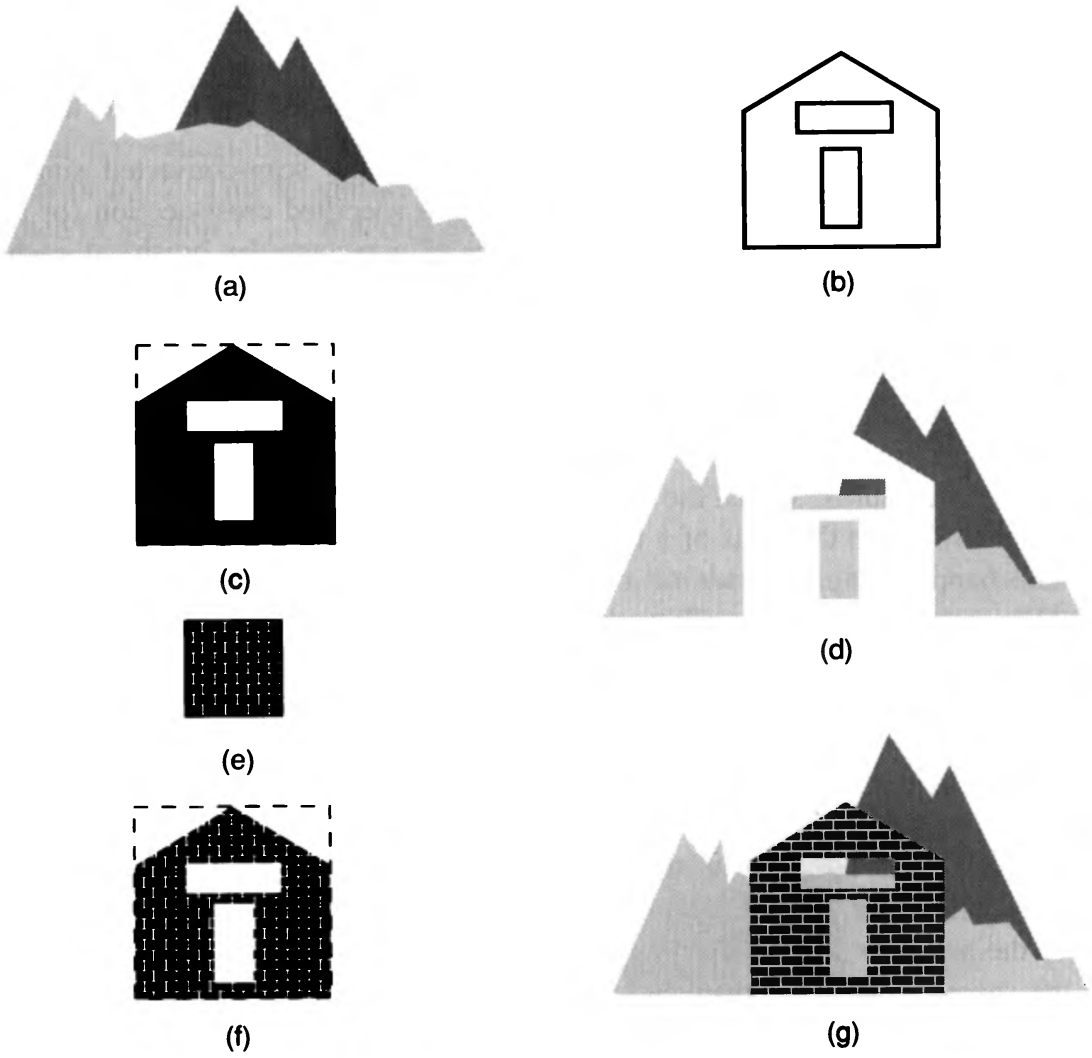

Fig. 3.30 Writing a patterned object in opaque mode with two transparent writes. (a) Mountain scene. (b) Outline of house icon. (c) Bitmap for solid version of house icon. (d) Clearing the scene by writing background. (e) Brick pattern. (f) Brick pattern applied to house icon. (g) Writing the screen transparently with patterned house icon.

transparently in background color to the canvas. This clears to background color a region of the shape of the object, as shown in (d), where the house-shaped region is set to white background within the existing mountain image. The third step is to create a patterned version of the object's solid bitmap by doing a copyPixel of a pattern rectangle (e) to the solid bitmap, using **and** mode. This turns some pixels internal to the object's shape from 1s to 0s (f), and can be seen as clipping out a piece of the arbitrarily large pattern in the shape of the object. Finally, we again write this new bitmap transparently to the same place in the canvas, but this time in the current, foreground color, as shown in (g). As in the first write to the canvas, all pixels outside the object's region are 0s, to protect pixels outside the region, whereas 0s inside the region do not affect the previously written (white) background; only where there are 1s is the (black) foreground written. To write the house with a solid red-brick pattern with gray mortar, we would write the solid bitmap in gray and the patterned bitmap in red; the pattern would have 1s everywhere except for small bands of 0s representing the mortar. In effect, we have reduced the rectangular write procedure that had to write two colors subject to a write mask to two write procedures that write transparently or copyPixel with a write mask.

3.9 THICK PRIMITIVES

Conceptually, we produce thick primitives by tracing the scan-converted single-pixel outline primitive. We place the center of a brush of a specified cross-section (or another distinguished point, such as the upper-left corner of a rectangular brush) at each pixel chosen by the scan-conversion algorithm. A single-pixel-wide line can be conceived as being drawn with a brush the size of a single pixel. However, this simple description masks a number of tricky questions. First, what shape is the brush? Typical implementations use circular and rectangular brushes. Second, what is the orientation of a noncircular brush? Does the rectangular pen always stay upright, so that the brush has constant width, or does it turn as the primitive turns, so that the vertical axis of the brush is aligned with the tangent to the primitive? What do the ends of a thick line look like, both ideally and on the integer grid? What happens at the vertex of a thick polygon? How do line style and pen style interact? We shall answer the simpler questions in this section, and the others in Chapter 19.

There are four basic methods for drawing thick primitives, illustrated in Figs. 3.31 through 3.36. We show the ideal primitives for these lines in black-on-white outline; the pixels generated to define the 1-pixel-thick scan-converted primitive in black; and the pixels added to form the thick primitive in gray. The reduced-scale versions show what the thick primitive actually looks like at still rather low resolution, with all pixels set to black. The first method is a crude approximation that uses more than 1 pixel for each column (or row) during scan conversion. The second traces the pen's cross-section along the single-pixel outline of the primitive. The third draws two copies of a primitive a thickness t apart and fills in the spans between these inner and outer boundaries. The fourth approximates all primitives by polylines and then uses a thick line for each polyline segment.

Let's look briefly at each of these methods and consider its advantages and disadvantages. All the methods produce effects that are satisfactory for many, if not most, purposes, at least for viewing on the screen. For printing, the higher resolution should be used to good advantage, especially since the speed of an algorithm for printing is not as

critical as for online primitive generation. We can then use more complex algorithms to produce better-looking results. A package may even use different techniques for different primitives. For example, QuickDraw traces an upright rectangular pen for lines, but fills spans between confocal ellipse boundaries.

3.9.1 Replicating Pixels

A quick extension to the scan-conversion inner loop to write multiple pixels at each computed pixel works reasonably well for lines; here, pixels are duplicated in columns for lines with $-1 <$ slope < 1 and in rows for all other lines. The effect, however, is that the line ends are always vertical or horizontal, which is not pleasing for rather thick lines, as Fig. 3.31 shows.

The pixel-replication algorithm also produces noticeable gaps in places where line segments meet at an angle, and misses pixels where there is a shift from horizontal to vertical replication as a function of the slope. This latter anomaly shows up as abnormal thinness in ellipse arcs at the boundaries between octants, as in Fig. 3.32.

Furthermore, lines that are horizontal and vertical have a different thickness from lines at an angle, where the *thickness* of the primitive is defined as the distance between the primitive's boundaries perpendicular to its tangent. Thus, if the thickness parameter is t, a horizontal or vertical line has thickness t, whereas one drawn at 45° has an average thickness of $t/\sqrt{2}$. This is another result of having fewer pixels in the line at an angle, as first noted in Section 3.2.3; it decreases the brightness contrast with horizontal and vertical lines of the same thickness. Still another problem with pixel replication is the generic problem of even-numbered widths: We cannot center the duplicated column or row about the selected pixel, so we must choose a side of the primitive to have an "extra" pixel. Altogether, pixel replication is an efficient but crude approximation that works best for primitives that are not very thick.

3.9.2 The Moving Pen

Choosing a rectangular pen whose center or corner travels along the single-pixel outline of the primitive works reasonably well for lines; it produces the line shown in Fig. 3.33. Notice that this line is similar to that produced by pixel replication but is thicker at the endpoints. As with pixel replication, because the pen stays vertically aligned, the perceived thickness of the primitive varies as a function of the primitive's angle, but in the opposite

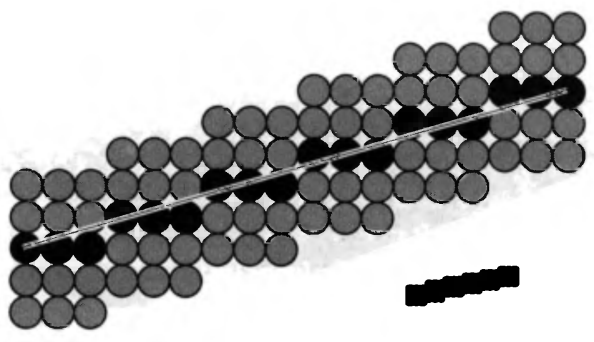

Fig. 3.31 Thick line drawn by column replication.

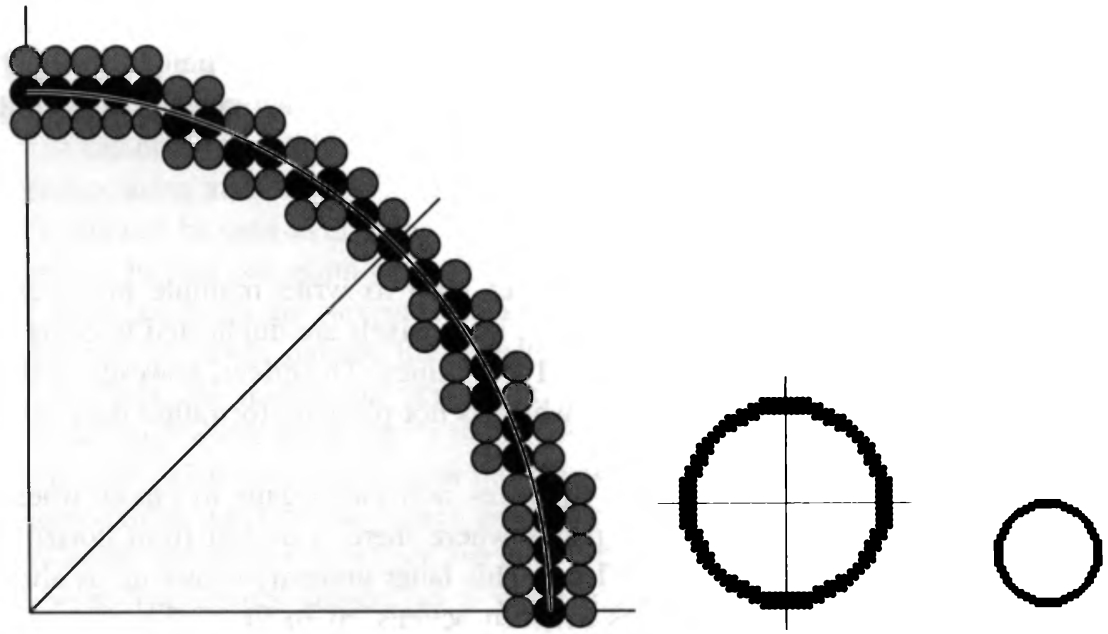

Fig. 3.32 Thick circle drawn by column replication.

way: The width is thinnest for horizontal segments and thickest for segments with slope of ± 1. An ellipse arc, for example, varies in thickness along its entire trajectory, being of the specified thickness when the tangent is nearly horizontal or vertical, and thickened by a factor of $\sqrt{2}$ around $\pm 45°$ (see Fig. 3.34). This problem would be eliminated if the square turned to follow the path, but it is much better to use a circular cross-section so that the thickness is angle-independent.

Now let's look at how to implement the moving-pen algorithm for the simple case of an upright rectangular or circular cross-section. The easiest solution is to copyPixel the required solid or patterned cross-section (also called *footprint*) so that its center or corner is at the chosen pixel; for a circular footprint and a pattern drawn in opaque mode, we must in addition mask off the bits outside the circular region, which is not an easy task unless our low-level copyPixel has a write mask for the destination region. The brute-force copyPixel solution writes pixels more than once, since the pen's footprints overlap at adjacent pixels. A better technique that also handles the circular-cross-section problem is to use the spans of the footprint to compute spans for successive footprints at adjacent pixels. As in filling

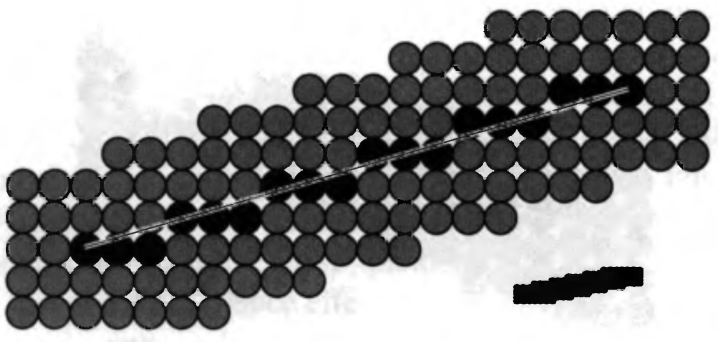

Fig. 3.33 Thick line drawn by tracing a rectangular pen.

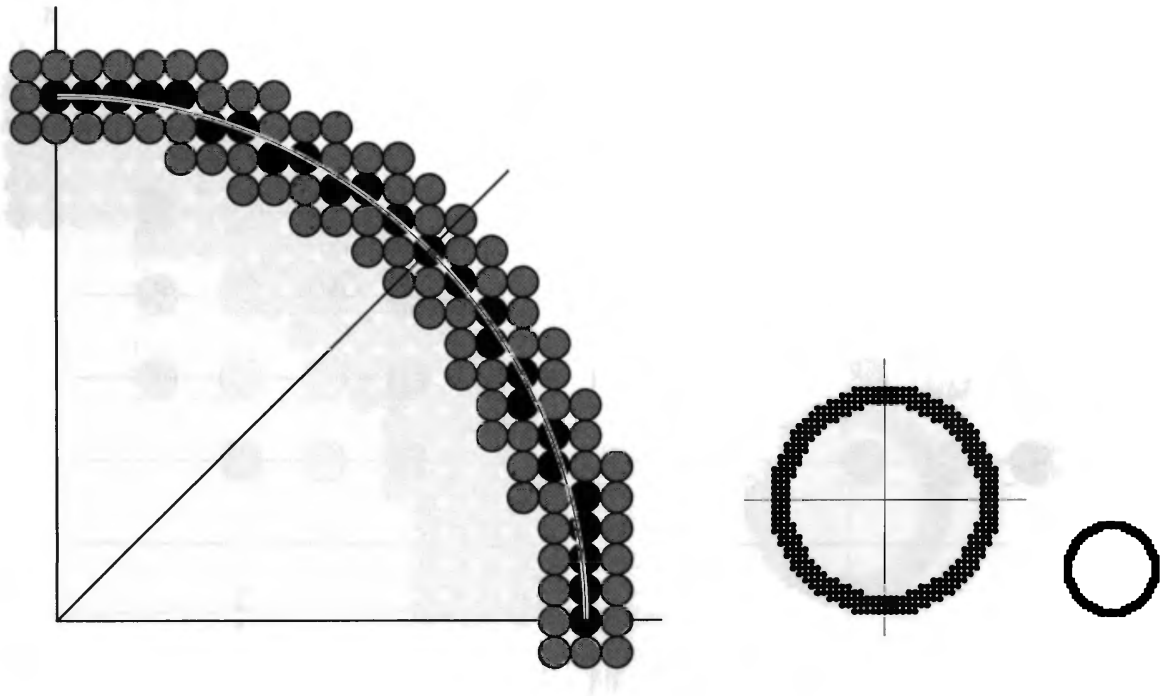

Fig. 3.34 Thick circle drawn by tracing a rectangular pen.

area-defining primitives, such combining of spans on a raster line is merely a union or merge of line segments, entailing keeping track of the minimum and maximum x of the accumulated span for each raster line. Figure 3.35 shows a sequence of two positions of the rectangular footprint and a portion of the temporary data structure that stores span extremes for each scan line. Each scan-line bucket may contain a list of spans when a thick polygon or ellipse arc is intersected more than once on a scan line, much like the active-edge table for polygons.

3.9.3 Filling Areas Between Boundaries

The third method for displaying a thick primitive is to construct the primitive's inner and outer boundary at a distance $t/2$ on either side of the ideal (single-pixel) primitive trajectory. Alternatively, for area-defining primitives, we can leave the original boundary as the outer boundary, then draw the inner boundary inward. This filling technique has the advantage of handling both odd and even thicknesses, and of not increasing the extent of a primitive when the primitive is thickened. The disadvantage of this technique, however, is that an area-defining primitive effectively "shrinks" a bit, and that its "center line," the original 1-pixel outline, appears to shift.

A thick line is drawn as a rectangle with thickness t and length of the original line. Thus, the rectangle's thickness is independent of the line's angle, and the rectangle's edges are perpendicular to the line. In general, the rectangle is rotated and its vertices do not lie on the integer grid; thus, they must be rounded to the nearest pixel, and the resulting rectangle must then be scan-converted as a polygon.

To create thick circles, we scan convert two circles, the outer one of radius $R + t/2$, the inner one of radius $R - t/2$, and fill in the single or double spans between them, as shown in Fig. 3.36.

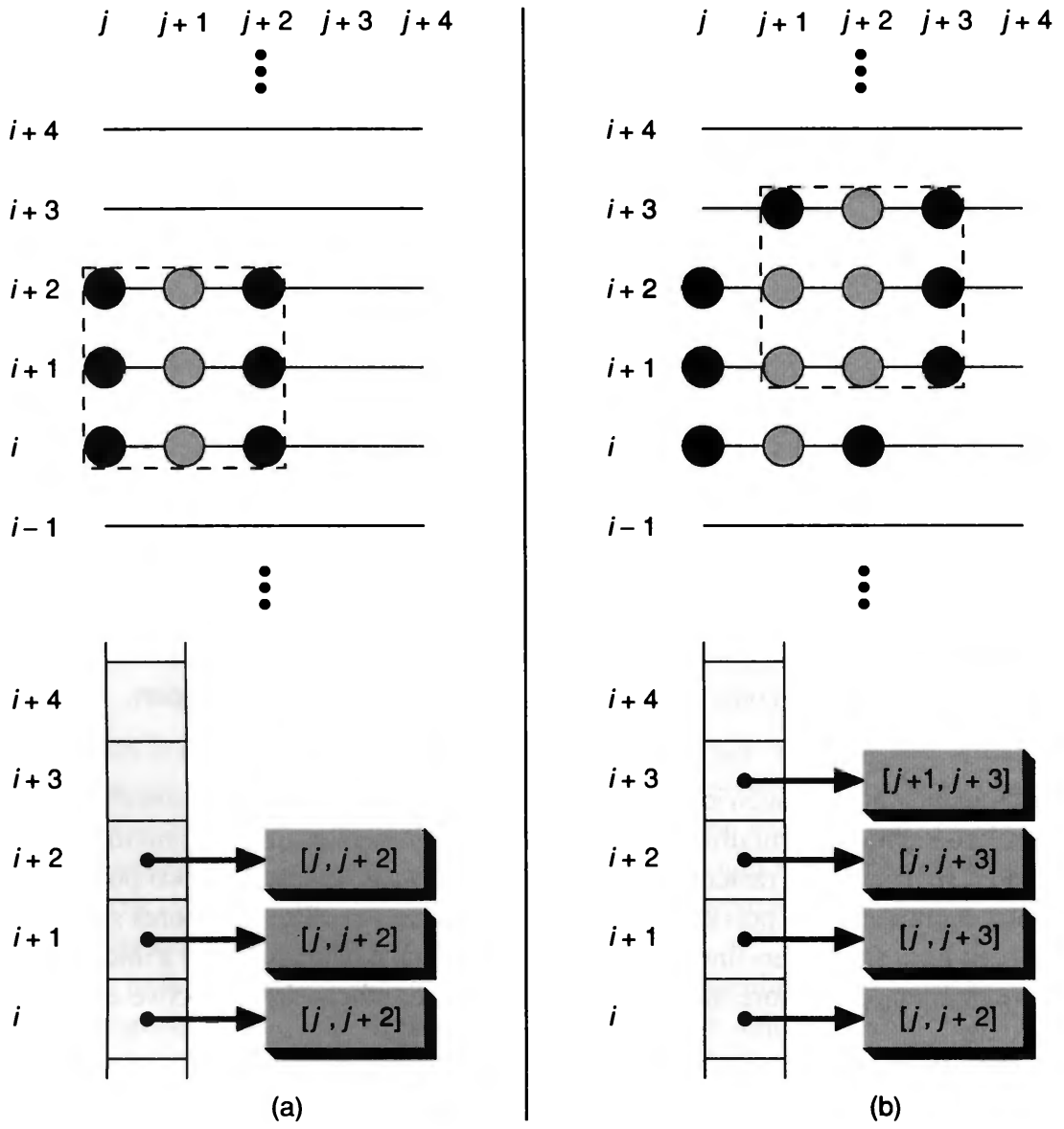

Fig. 3.35 Recording spans of the rectangular pen: (a) footprint at $x = j + 1$; (b) $x = j + 2$.

For ellipses, the situation is not nearly so simple. It is a classic result in differential geometry that the curves formed by moving a distance $t/2$ perpendicular to an ellipse are not confocal ellipses, but are described by eighth-order equations [SALM96].[9] These functions are computationally expensive to scan convert; therefore, as usual, we approximate. We scan convert two confocal ellipses, the inner with semidiameters $a - t/2$ and $b - t/2$, the outer with semidiameters $a + t/2$ and $b + t/2$. Again, we calculate spans and fill them in, either after all span arithmetic is done, or on the fly. The standard problems of thin ellipses (treated in Chapter 19) pertain. Also, the problem of generating the inner boundary, noted here for ellipses, also can occur for other primitives supported in raster graphics packages.

[9]The eighth-order curves so generated may have self-intersections or cusps, as may be seen by constructing the normal lines by hand.

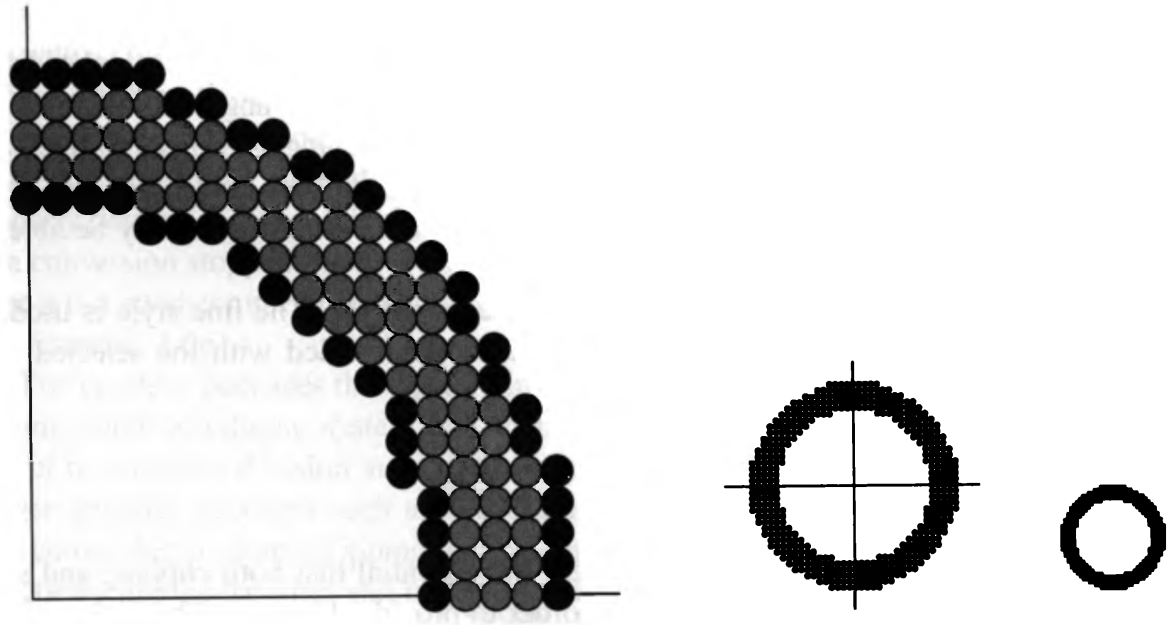

Fig. 3.36 Thick circle drawn by filling between concentric circles.

3.9.4 Approximation by Thick Polylines

We can do piecewise-linear approximation of any primitive by computing points on the boundary (with floating-point coordinates), then connecting these points with line segments to form a polyline. The advantage of this approach is that the algorithms for both line clipping and line scan conversion (for thin primitives), and for polygon clipping and polygon scan conversion (for thick primitives), are efficient. Naturally, the segments must be quite short in places where the primitive changes direction rapidly. Ellipse arcs can be represented as ratios of parametric polynomials, which lend themselves readily to such piecewise-linear approximation (see Chapter 11). The individual line segments are then drawn as rectangles with the specified thickness. To make the thick approximation look nice, however, we must solve the problem of making thick lines join smoothly, as discussed in Chapter 19.

3.10 LINE STYLE AND PEN STYLE

SRGP's line-style atribute can affect any outline primitive. In general, we must use conditional logic to test whether or not to write a pixel, writing only for 1s. We store the pattern write mask as a string of 16 **booleans** (e.g., a 16-bit **integer**); it should therefore repeat every 16 pixels. We modify the unconditional WritePixel statement in the line scan-conversion algorithm to handle this; for example,

```
if (bitstring[i % 16])
    WritePixel (x, y, value);
```

where the index i is a new variable incremented in the inner loop for this purpose. There is a drawback to this technique, however. Since each bit in the mask corresponds to an iteration of the loop, and not to a unit distance along the line, the length of dashes varies with the

angle of the line; a dash at an angle is longer than is a horizontal or vertical dash. For engineering drawings, this variation is unacceptable, and the dashes must be calculated and scan-converted as individual line segments of length invariant with angle. Thick lines are created as sequences of alternating solid and transparent rectangles whose vertices are calculated exactly as a function of the line style selected. The rectangles are then scan-converted individually; for horizontal and vertical lines, the program may be able to copyPixel the rectangle.

Line style and pen style interact in thick outline primitives. The line style is used to calculate the rectangle for each dash, and each rectangle is filled with the selected pen pattern (Fig. 3.37).

3.11 CLIPPING IN A RASTER WORLD

As we noted in the introduction to this chapter, it is essential that both clipping and scan conversion be done as rapidly as possible, in order to provide the user with quick updates resulting from changes to the application model. Clipping can be done analytically, on the fly during scan conversion, or as part of a copyPixel with the desired clip rectangle from a canvas storing unclipped primitives to the destination canvas. This third technique would be useful in situations where a large canvas can be generated ahead of time, and the user can then examine pieces of it for a significant period of time by panning the clip rectangle, without updating the contents of the canvas.

Combining clipping and scan conversion, sometimes called *scissoring*, is easy to do for filled or thick primitives as part of span arithmetic: Only the extrema need to be clipped, and no interior pixels need be examined. Scissoring shows yet another advantage of span coherence. Also, if an outline primitive is not much larger than the clip rectangle, not many pixels, relatively speaking, will fall outside the clip region. For such a case, it may well be faster to generate each pixel and to clip it (i.e., to write it conditionally) than to do analytical clipping beforehand. In particular, although the bounds test is in the inner loop, the expensive memory write is avoided for exterior pixels, and both the incremental computation and the testing may run entirely in a fast memory, such as a CPU instruction cache or a display controller's microcode memory.

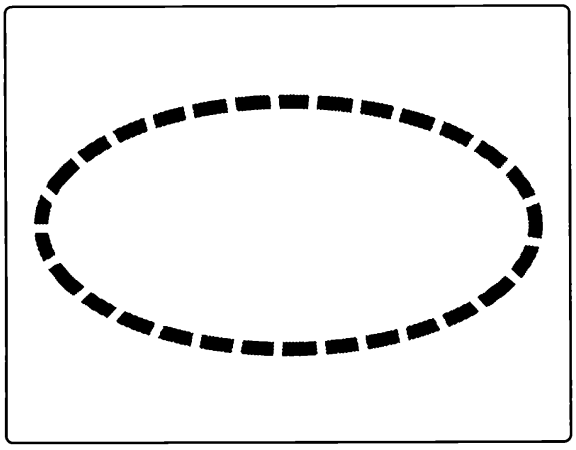

Fig. 3.37 Combining pen pattern and line style.

Other tricks may be useful. For example, one may "home in" on the intersection of a line with a clip edge by doing the standard midpoint scan-conversion algorithm on every *i*th pixel and testing the chosen pixel against the rectangle bounds until the first pixel that lies inside the region is encountered. Then the algorithm has to back up, find the first pixel inside, and to do the normal scan conversion thereafter. The last interior pixel could be similarly determined, or each pixel could be tested as part of the scan-conversion loop and scan conversion stopped the first time the test failed. Testing every eighth pixel works well, since it is a good compromise between having too many tests and too many pixels to back up (see Exercise 3.26).

For graphics packages that operate in floating point, it is best to clip analytically in the floating-point coordinate system and then to scan convert the clipped primitives, being careful to initialize decision variables correctly, as we did for lines in Section 3.2.3. For integer graphics packages such as SRGP, there is a choice between preclipping and then scan converting or doing clipping during scan conversion. Since it is relatively easy to do analytical clipping for lines and polygons, clipping of those primitives is often done before scan conversion, while it is faster to clip other primitives during scan conversion. Also, it is quite common for a floating-point graphics package to do analytical clipping in its coordinate system and then to call lower-level scan-conversion software that actually generates the clipped primitives; this integer graphics software could then do an additional raster clip to rectangular (or even arbitrary) window boundaries. Because analytic clipping of primitives is both useful for integer graphics packages and essential for 2D and 3D floating-point graphics packages, we discuss the basic analytical clipping algorithms in this chapter.

3.12 CLIPPING LINES

This section treats analytical clipping of lines against rectangles;[10] algorithms for clipping other primitives are handled in subsequent sections. Although there are specialized algorithms for rectangle and polygon clipping, it is important to note that SRGP primitives built out of lines (i.e., polylines, unfilled rectangles, and polygons) can be clipped by repeated application of the line clipper. Furthermore, circles and ellipses may be piecewise-linearly approximated with a sequence of very short lines, so that boundaries can be treated as a single polyline or polygon for both clipping and scan conversion. Conics are represented in some systems as ratios of parametric polynomials (see Chapter 11), a representation that also lends itself readily to an incremental, piecewise linear approximation suitable for a line-clipping algorithm. Clipping a rectangle against a rectangle results in at most a single rectangle. Clipping a convex polygon against a rectangle results in at most a single convex polygon, but clipping a concave polygon may produce more than one concave polygon. Clipping a circle or ellipse against a rectangle results in as many as four arcs.

Lines intersecting a rectangular clip region (or any convex polygon) are always clipped to a single line segment; lines lying on the clip rectangle's border are considered inside and hence are displayed. Figure 3.38 shows several examples of clipped lines.

---

[10]This chapter does not cover clipping primitives to multiple rectangles (as when windows overlap in a windowing system) or to nonrectangular regions; the latter topic is discussed briefly in Section 19.7.

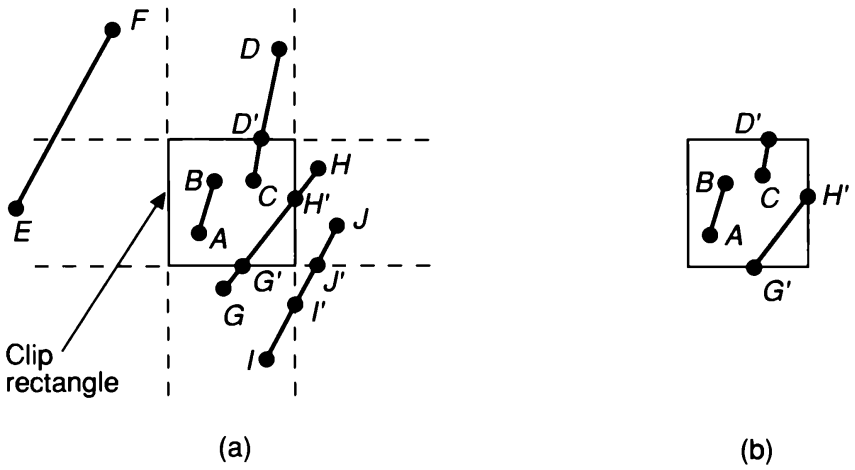

Fig. 3.38 Cases for clipping lines.

3.12.1 Clipping Endpoints

Before we discuss clipping lines, let's look at the simpler problem of clipping individual points. If the x coordinate boundaries of the clip rectangle are at x_{min} and x_{max}, and the y coordinate boundaries are at y_{min} and y_{max}, then four inequalities must be satisfied for a point at (x, y) to be inside the clip rectangle:

$$x_{min} \leq x \leq x_{max}, \; y_{min} \leq y \leq y_{max}.$$

If any of the four inequalities does not hold, the point is outside the clip rectangle.

3.12.2 Clipping Lines by Solving Simultaneous Equations

To clip a line, we need to consider only its endpoints, not its infinitely many interior points. If both endpoints of a line lie inside the clip rectangle (e.g., *AB* in Fig. 3.38), the entire line lies inside the clip rectangle and can be *trivially accepted*. If one endpoint lies inside and one outside (e.g., *CD* in the figure), the line intersects the clip rectangle and we must compute the intersection point. If both endpoints are outside the clip rectangle, the line may (or may not) intersect with the clip rectangle (*EF*, *GH*, and *IJ* in the figure), and we need to perform further calculations to determine whether there are any intersections, and if there are, where they occur.

The brute-force approach to clipping a line that cannot be trivially accepted is to intersect that line with each of the four clip-rectangle edges to see whether any intersection points lie on those edges; if so, the line cuts the clip rectangle and is partially inside. For each line and clip-rectangle edge, we therefore take the two mathematically infinite lines that contain them and intersect them. Next, we test whether this intersection point is "interior"—that is, whether it lies within both the clip rectangle edge and the line; if so, there is an intersection with the clip rectangle. In Fig. 3.38, intersection points G' and H' are interior, but I' and J' are not.

When we use this approach, we must solve two simultaneous equations using multiplication and division for each <edge, line> pair. Although the slope-intercept

formula for lines learned in analytic geometry could be used, it describes infinite lines, whereas in graphics and clipping we deal with finite lines (called *line segments* in mathematics). In addition, the slope-intercept formula does not deal with vertical lines—a serious problem, given our upright clip rectangle. A parametric formulation for line segments solves both problems:

$$x = x_0 + t(x_1 - x_0), \; y = y_0 + t(y_1 - y_0).$$

These equations describe (x, y) on the directed line segment from (x_0, y_0) to (x_1, y_1) for the parameter t in the range [0, 1], as simple substitution for t confirms. Two sets of simultaneous equations of this parametric form can be solved for parameters t_{edge} for the edge and t_{line} for the line segment. The values of t_{edge} and t_{line} can then be checked to see whether both lie in [0, 1]; if they do, the intersection point lies within both segments and is a true clip-rectangle intersection. Furthermore, the special case of a line parallel to a clip-rectangle edge must also be tested before the simultaneous equations can be solved. Altogether, the brute-force approach involves considerable calculation and testing; it is thus inefficient.

3.12.3 The Cohen–Sutherland Line-Clipping Algorithm

The more efficient Cohen–Sutherland algorithm performs initial tests on a line to determine whether intersection calculations can be avoided. First, endpoint pairs are checked for trivial acceptance. If the line cannot be trivially accepted, *region checks* are done. For instance, two simple comparisons on x show that both endpoints of line *EF* in Fig. 3.38 have an x coordinate less than x_{min} and thus lie in the region to the left of the clip rectangle (i.e., in the outside halfplane defined by the left edge); therefore, line segment *EF* can be *trivially rejected* and needs to be neither clipped nor displayed. Similarly, we can trivially reject lines with both endpoints in regions to the right of x_{max}, below y_{min}, and above y_{max}.

If the line segment can be neither trivially accepted nor rejected, it is divided into two segments at a clip edge, so that one segment can be trivially rejected. Thus, a segment is iteratively clipped by testing for trivial acceptance or rejection, and is then subdivided if neither test is successful, until what remains is completely inside the clip rectangle or can be trivially rejected. The algorithm is particularly efficient for two common cases. In the first case of a large clip rectangle enclosing all or most of the display area, most primitives can be trivially accepted. In the second case of a small clip rectangle, almost all primitives can be trivially rejected. This latter case arises in a standard method of doing pick correlation in which a small rectangle surrounding the cursor, called the *pick window*, is used to clip primitives to determine which primitives lie within a small (rectangular) neighborhood of the cursor's *pick point* (see Section 7.12.2).

To perform trivial accept and reject tests, we extend the edges of the clip rectangle to divide the plane of the clip rectangle into nine regions (see Fig. 3.39). Each region is assigned a 4-bit code, determined by where the region lies with respect to the outside halfplanes of the clip-rectangle edges. Each bit in the outcode is set to either 1 (true) or 0 (false); the 4 bits in the code correspond to the following conditions:

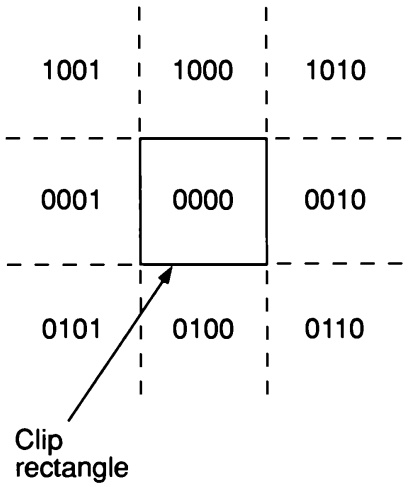

Fig. 3.39 Region outcodes.

First bit	outside halfplane of top edge, above top edge	$y > y_{max}$
Second bit	outside halfplane of bottom edge, below bottom edge	$y < y_{min}$
Third bit	outside halfplane of right edge, to the right of right edge	$x > x_{max}$
Fourth bit	outside halfplane of left edge, to the left of left edge	$x < x_{min}$

Since the region lying above and to the left of the clip rectangle, for example, lies in the outside halfplane of the top and left edges, it is assigned a code of 1001. A particularly efficient way to calculate the outcode derives from the observation that bit 1 is the sign bit of $(y_{max} - y)$; bit 2 is that of $(y - y_{min})$; bit 3 is that of $(x_{max} - x)$; and bit 4 is that of $(x - x_{min})$. Each endpoint of the line segment is then assigned the code of the region in which it lies. We can now use these endpoint codes to determine whether the line segment lies completely inside the clip rectangle or in the outside halfplane of an edge. If both 4-bit codes of the endpoints are zero, then the line lies completely inside the clip rectangle. However, if both endpoints lie in the outside halfplane of a particular edge, as for *EF* in Fig. 3.38, the codes for both endpoints each have the bit set showing that the point lies in the outside halfplane of that edge. For *EF*, the outcodes are 0001 and 1001, respectively, showing with the fourth bit that the line segment lies in the outside halfplane of the left edge. Therefore, if the logical **and** of the codes of the endpoints is not zero, the line can be trivially rejected.

If a line cannot be trivially accepted or rejected, we must subdivide it into two segments such that one or both segments can be discarded. We accomplish this subdivision by using an edge that the line crosses to cut the line into two segments: The section lying in the outside halfplane of the edge is thrown away. We can choose any order in which to test edges, but we must, of course, use the same order each time in the algorithm; we shall use the top-to-bottom, right-to-left order of the outcode. A key property of the outcode is that bits that are set in a nonzero outcode correspond to edges crossed: If one endpoint lies in the outside halfplane of an edge and the line segment fails the trivial-rejection tests, then the other point must lie on the inside halfplane of that edge and the line segment must cross it. Thus, the algorithm always chooses a point that lies outside and then uses an outcode bit that is set to determine a clip edge; the edge chosen is the first in the top-to-bottom, right-to-left order—that is, it is the leftmost bit that is set in the outcode.

The algorithm works as follows. We compute the outcodes of both endpoints and check

for trivial acceptance and rejection. If neither test is succesful, we find an endpoint that lies outside (at least one will), and then test the outcode to find the edge that is crossed and to determine the corresponding intersection point. We can then clip off the line segment from the outside endpoint to the intersection point by replacing the outside endpoint with the intersection point, and compute the outcode of this new endpoint to prepare for the next iteration.

For example, consider the line segment *AD* in Fig. 3.40. Point *A* has outcode 0000 and point *D* has outcode 1001. The line can be neither trivially accepted or rejected. Therefore, the algorithm chooses *D* as the outside point, whose outcode shows that the line crosses the top edge and the left edge. By our testing order, we first use the top edge to clip *AD* to *AB*, and we compute *B*'s outcode as 0000. In the next iteration, we apply the trivial acceptance/rejection tests to *AB*, and it is trivially accepted and displayed.

Line *EI* requires multiple iterations. The first endpoint, *E*, has an outcode of 0100, so the algorithm chooses it as the outside point and tests the outcode to find that the first edge against which the line is cut is the bottom edge, where *EI* is clipped to *FI*. In the second iteration, *FI* cannot be trivially accepted or rejected. The outcode of the first endpoint, *F*, is 0000, so the algorithm chooses the outside point *I* that has outcode 1010. The first edge clipped against is therefore the top edge, yielding *FH*. *H*'s outcode is determined to be 0010, so the third iteration results in a clip against the right edge to *FG*. This is trivially accepted in the fourth and final iteration and displayed. A different sequence of clips would have resulted if we had picked *I* as the initial point: On the basis of its outcode, we would have clipped against the top edge first, then the right edge, and finally the bottom edge.

In the code of Fig. 3.41, we use constant integers and bitwise arithmetic to represent the outcodes, because this representation is more natural than an array with an entry for each outcode. We use an internal procedure to calculate the outcode for modularity; to improve performance, we would, of course, put this code in line.

We can improve the efficiency of the algorithm slightly by not recalculating slopes (see Exercise 3.28). Even with this improvement, however, the algorithm is not the most efficient one. Because testing and clipping are done in a fixed order, the algorithm will sometimes perform needless clipping. Such clipping occurs when the intersection with a rectangle edge is an "external intersection"; that is, when it does not lie on the clip-rectangle boundary (e.g., point *H* on line *EI* in Fig. 3.40). The Nicholl, Lee, and

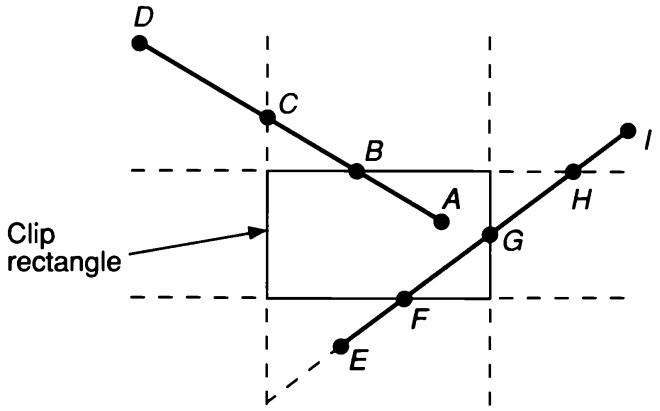

Fig. 3.40 Illustration of Cohen–Sutherland line clipping.

```
typedef unsigned int outcode;
enum {TOP = 0x1, BOTTOM = 0x2, RIGHT = 0x4, LEFT = 0x8};

void CohenSutherlandLineClipAndDraw (
    double x0, double y0, double x1, double y1, double xmin, double xmax,
    double ymin, double ymax, int value)
/* Cohen-Sutherland clipping algorithm for line P0 = (x0, y0) to P1 = (x1, y1) and */
/* clip rectangle with diagonal from (xmin, ymin) to (xmax, ymax) */
{
    /* Outcodes for P0, P1, and whatever point lies outside the clip rectangle */
    outcode outcode0, outcode1, outcodeOut;
    boolean accept = FALSE, done = FALSE;
    outcode0 = CompOutCode (x0, y0, xmin, xmax, ymin, ymax);
    outcode1 = CompOutCode (x1, y1, xmin, xmax, ymin, ymax);
    do {
        if (!(outcode0 | outcode1)) {              /* Trivial accept and exit */
            accept = TRUE; done = TRUE;
        } else if (outcode0 & outcode1)            /* Logical and is true, so trivial reject and exit */
            done = TRUE;
        else {
            /* Failed both tests, so calculate the line segment to clip: */
            /* from an outside point to an intersection with clip edge. */
            double x, y;
            /* At least one endpoint is outside the clip rectangle; pick it. */
            outcodeOut = outcode0 ? outcode0 : outcode1;
            /* Now find intersection point; */
            /* use formulas y = y0 + slope * (x - x0), x = x0 + (1/slope) * (y - y0). */
            if (outcodeOut & TOP) {                /* Divide line at top of clip rect */
                x = x0 + (x1 - x0) * (ymax - y0) / (y1 - y0);
                y = ymax;
            } else if (outcodeOut & BOTTOM) {      /* Divide line at bottom edge of clip rect */
                x = x0 + (x1 - x0) * (ymin - y0) / (y1 - y0);
                y = ymin;
            } else if (outcodeOut & RIGHT) {       /* Divide line at right edge of clip rect */
                y = y0 + (y1 - y0) * (xmax - x0) / (x1 - x0);
                x = xmax;
            } else {                               /* Divide line at left edge of clip rect */
                y = y0 + (y1 - y0) * (xmin - x0) / (x1 - x0);
                x = xmin;
            }
            /* Now we move outside point to intersection point to clip, */
            /* and get ready for next pass. */
            if (outcodeOut == outcode0) {
                x0 = x; y0 = y; outcode0 = CompOutCode (x0, y0, xmin, xmax, ymin, ymax);
            } else {
                x1 = x; y1 = y; outcode1 = CompOutCode (x1, y1, xmin, xmax, ymin, ymax);
            }
        }  /* Subdivide */
    } while (done == FALSE);
```

Fig. 3.41 *(Cont.).*

```
    if (accept)
        MidpointLineReal (x0, y0, x1, y1, value);    /* Version for double coordinates */
} /* CohenSutherlandLineClipAndDraw */

outcode CompOutCode (
    double x, double y, double xmin, double xmax, double ymin, double ymax);
{
    outcode code = 0;
    if (y > ymax)
        code |= TOP;
    else if (y < ymin)
        code |= BOTTOM;
    if (x > xmax)
        code |= RIGHT;
    else if (x < xmin)
        code |= LEFT;
    return code;
} /* CompOutCode */
```

Fig. 3.41 Cohen–Sutherland line-clipping algorithm.

Nicholl [NICH87] algorithm, by contrast, avoids calculating external intersections by subdividing the plane into many more regions; it is discussed in Chapter 19. An advantage of the much simpler Cohen–Sutherland algorithm is that its extension to a 3D orthographic view volume is straightforward, as seen in Section 6.5.3

3.12.4 A Parametric Line-Clipping Algorithm

The Cohen–Sutherland algorithm is probably still the most commonly used line-clipping algorithm because it has been around longest and has been published widely. In 1978, Cyrus and Beck published an algorithm that takes a fundamentally different and generally more efficient approach to line clipping [CYRU78]. The Cyrus–Beck technique can be used to clip a 2D line against a rectangle or an arbitrary convex polygon in the plane, or a 3D line against an arbitrary convex polyhedron in 3D space. Liang and Barsky later independently developed a more efficient parametric line-clipping algorithm that is especially fast in the special cases of upright 2D and 3D clip regions [LIAN84]. In addition to taking advantage of these simple clip boundaries, they introduced more efficient trivial rejection tests that work for general clip regions. Here we follow the original Cyrus–Beck development to introduce parametric clipping. Since we are concerned only with upright clip rectangles, however, we reduce the Cyrus–Beck formulation to the more efficient Liang–Barsky case at the end of the development.

Recall that the Cohen–Sutherland algorithm, for lines that cannot be trivially accepted or rejected, calculates the (x, y) intersection of a line segment with a clip edge by substituting the known value of x or y for the vertical or horizontal clip edge, respectively. The parametric line algorithm, however, finds the value of the parameter t in the parametric

representation of the line segment for the point at which that segment intersects the infinite line on which the clip edge lies. Because all clip edges are in general intersected by the line, four values of t are calculated. A series of simple comparisons is used to determine which (if any) of the four values of t correspond to actual intersections. Only then are the (x, y) values for one or two actual intersections calculated. In general, this approach saves time over the Cohen–Sutherland intersection-calculation algorithm because it avoids the repetitive looping needed to clip to multiple clip-rectangle edges. Also, calculations in 1D parameter space are simpler than those in 3D coordinate space. Liang and Barsky improve on Cyrus–Beck by examining each t-value as it is generated, to reject some line segments before all four t-values have been computed.

The Cyrus–Beck algorithm is based on the following formulation of the intersection between two lines. Figure 3.42 shows a single edge E_i of the clip rectangle and that edge's outward normal N_i (i.e., outward to the clip rectangle[11]), as well as the line segment from P_0 to P_1 that must be clipped to the edge. Either the edge or the line segment may have to be extended to find the intersection point.

As before, this line is represented parametrically as

$$P(t) = P_0 + (P_1 - P_0)t,$$

where $t = 0$ at P_0 and $t = 1$ at P_1. Now, pick an arbitrary point P_{E_i} on edge E_i and consider the three vectors $P(t) - P_{E_i}$ from P_{E_i} to three designated points on the line from P_0 to P_1: the intersection point to be determined, an endpoint of the line on the inside halfplane of the edge, and an endpoint on the line in the outside halfplane of the edge. We can distinguish in which region a point lies by looking at the value of the dot product $N_i \cdot [P(t) - P_{E_i}]$. This value is negative for a point in the inside halfplane, zero for a point on the line containing the edge, and positive for a point that lies in the outside halfplane. The definitions of inside and outside halfplanes of an edge correspond to a counterclockwise enumeration of the edges of the clip region, a convention we shall use throughout this book. Now we can solve for the value of t at the intersection of P_0P_1 with the edge:

$$N_i \cdot [P(t) - P_{E_i}] = 0.$$

First, substitute for $P(t)$:

$$N_i \cdot [P_0 + (P_1 - P_0)t - P_{E_i}] = 0.$$

Next, group terms and distribute the dot product:

$$N_i \cdot [P_0 - P_{E_i}] + N_i \cdot [P_1 - P_0]t = 0.$$

Let $D = (P_1 - P_0)$ be the vector from P_0 to P_1, and solve for t:

$$t = \frac{N_i \cdot [P_0 - P_{E_i}]}{-N_i \cdot D}. \tag{3.1}$$

Note that this gives a valid value of t only if the denominator of the expression is nonzero.

[11]Cyrus and Beck use inward normals, but we prefer to use outward normals for consistency with plane normals in 3D, which are outward. Our formulation therefore differs only in the testing of a sign.

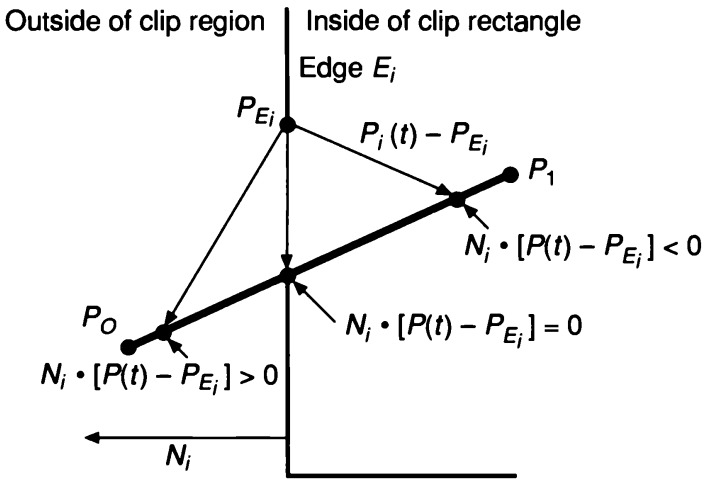

Fig. 3.42 Dot products for three points outside, inside, and on the boundary of the clip region.

For this to be true, the algorithm checks that

$N_i \neq 0$ (that is, the normal should not be 0; this could occur only as a mistake),

$D \neq 0$ (that is, $P_1 \neq P_0$),

$N_i \cdot D \neq 0$ (that is, the edge E_i and the line from P_0 to P_1 are not parallel. If they were parallel, there can be no single intersection for this edge, so the algorithm moves on to the next case.).

Equation (3.1) can be used to find the intersections between P_0P_1 and each edge of the clip rectangle. We do this calculation by determining the normal and an arbitrary P_{E_i}—say, an endpoint of the edge—for each clip edge, then using these values for all line segments. Given the four values of t for a line segment, the next step is to determine which (if any) of the values correspond to internal intersections of the line segment with edges of the clip rectangle. As a first step, any value of t outside the interval [0, 1] can be discarded, since it lies outside P_0P_1. Next, we need to determine whether the intersection lies on the clip boundary.

We could try simply sorting the remaining values of t, choosing the intermediate values of t for intersection points, as suggested in Fig. 3.43 for the case of line 1. But how do we distinguish this case from that of line 2, in which no portion of the line segment lies in the clip rectangle and the intermediate values of t correspond to points not on the clip boundary? Also, which of the four intersections of line 3 are the ones on the clip boundary?

The intersections in Fig. 3.43 are characterized as "potentially entering" (PE) or "potentially leaving" (PL) the clip rectangle, as follows: If moving from P_0 to P_1 causes us to cross a particular edge to enter the edge's inside halfplane, the intersection is PE; if it causes us to leave the edge's inside halfplane, it is PL. Notice that, with this distinction, two interior intersection points of a line intersecting the clip rectangle have opposing labels.

Formally, intersections can be classified as PE or PL on the basis of the angle between P_0P_1 and N_i: If the angle is less than 90°, the intersection is PL; if it is greater than 90°, it is PE. This information is contained in the sign of the dot product of N_i and P_0P_1:

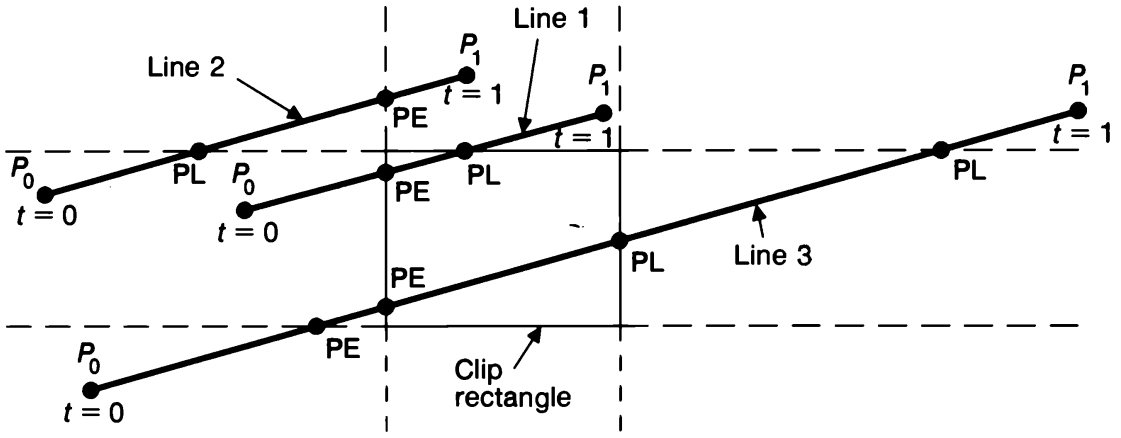

Fig. 3.43 Lines lying diagonal to the clip rectangle.

$$N_i \cdot D < 0 \Rightarrow \text{PE (angle greater than 90)},$$
$$N_i \cdot D > 0 \Rightarrow \text{PL (angle less than 90)}.$$

Notice that $N_i \cdot D$ is merely the denominator of Eq. (3.1), which means that, in the process of calculating t, the intersection can be trivially categorized.

With this categorization, line 3 in Fig. 3.43 suggests the final step in the process. We must choose a (PE, PL) pair that defines the clipped line. The portion of the infinite line through P_0P_1 that is within the clipping region is bounded by the PE intersection with the largest t value, which we call t_E, and the PL intersection with the smallest t value, t_L. The intersecting line segment is then defined by the range (t_E, t_L). But because we are interested in intersecting P_0P_1, not the infinite line, the definition of the range must be further modified so that $t = 0$ is a lower bound for t_E and $t = 1$ is an upper bound for t_L. What if $t_E > t_L$? This is exactly the case for line 2. It means that no portion of P_0P_1 is within the clip rectangle, and the entire line is rejected. Values of t_E and t_L that correspond to actual intersections are used to calculate the corresponding x and y coordinates.

The completed algorithm for upright clip rectangles is pseudocoded in Fig. 3.44. Table 3.1 shows for each edge the values of N_i, a canonical point on the edge, P_{E_i}, the vector $P_0 - P_{E_i}$ and the parameter t. Interestingly enough, because one coordinate of each normal is 0, we do not need to pin down the corresponding coordinate of P_{E_i} (denoted by an indeterminate x or y). Indeed, because the clip edges are horizontal and vertical, many simplifications apply that have natural interpretations. Thus we see from the table that the numerator, the dot product $N_i \cdot (P_0 - P_{E_i})$ determining whether the endpoint P_0 lies inside or outside a specified edge, reduces to the directed horizontal or vertical distance from the point to the edge. This is exactly the same quantity computed for the corresponding component of the Cohen-Sutherland outcode. The denominator dot product $N_i \cdot D$, which determines whether the intersection is potentially entering or leaving, reduces to $\pm dx$ or dy: if dx is positive, the line moves from left to right and is PE for the left edge, PL for the right edge, and so on. Finally, the parameter t, the ratio of numerator and denominator, reduces to the distance to an edge divided by dx or dy, exactly the constant of proportionality we could calculate directly from the parametric line formulation. Note that it is important to preserve the signs of the numerator and denominator instead of cancelling minus signs, because the numerator and denominator as signed distances carry information that is used in the algorithm.

```
precalculate N_i and select a P_Ei for each edge;

for (each line segment to be clipped) {
    if (P1 == P0)
        line is degenerate so clip as a point;
    else {
        t_E = 0; t_L = 1;
        for (each candidate intersection with a clip edge) {
            if (N_i • D != 0) {      /* Ignore edges parallel to line for now */
                calculate t;
                use sign of N_i • D to categorize as PE or PL;
                if (PE) t_E = max (t_E, t);
                if (PL) t_L = min (t_L, t);
            }
        }
        if (t_E > t_L)
            return NULL;
        else
            return P(t_E) and P(t_L) as true clip intersections;
    }
}
```

Fig. 3.44 Pseudocode for Cyrus–Beck parametric line clipping algorithm.

The complete version of the code, adapted from [LIAN84] is shown in Fig. 3.45. The procedure calls an internal function, CLIPt(), that uses the sign of the denominator to determine whether the line segment-edge intersection is potentially entering (PE) or leaving (PL), computes the parameter value of the intersection and checks to see if trivial rejection can be done because the new value of t_E or t_L would cross the old value of t_L or t_E, respectively. It also signals trivial rejection if the line is parallel to the edge and on the

TABLE 3.1 CALCULATIONS FOR PARAMETRIC LINE CLIPPING ALGORITHM*

Clip edge$_i$	Normal N_i	P_{E_i}	$P_0 - P_{E_i}$	$t = \frac{N_i \cdot (P_0 - P_{E_i})}{-N_i \cdot D}$
left: $x = x_{min}$	$(-1, 0)$	(x_{min}, y)	$(x_0 - x_{min}, y_0 - y)$	$\frac{-(x_0 - x_{min})}{(x_1 - x_0)}$
right: $x = x_{max}$	$(1, 0)$	(x_{max}, y)	$(x_0 - x_{max}, y_0 - y)$	$\frac{(x_0 - x_{max})}{-(x_1 - x_0)}$
bottom: $y = y_{min}$	$(0, -1)$	(x, y_{min})	$(x_0 - x, y_0 - y_{min})$	$\frac{-(y_0 - y_{min})}{(y_1 - y_0)}$
top: $y = y_{max}$	$(0, 1)$	(x, y_{max})	$(x_0 - x, y_0 - y_{max})$	$\frac{(y_0 - y_{max})}{-(y_1 - y_0)}$

*The exact coordinates of the point P_{E_i} on each edge are irrelevant to the computation, so they have been denoted by variables x and y. For a point on the right edge, $x=x_{min}$ as indicated in the first row, third entry.

```
void Clip2D (double *x0, double *y0, double *x1, double *y1, boolean *visible)
/* Clip 2D line segment with endpoints (x0, y0) and (x1, y1), against upright */
/* clip rectangle with corners at (xmin, ymin) and (xmax, ymax); these are */
/* globals or could be passed as parameters also. The flag visible is set TRUE. */
/* if a clipped segment is returned in endpoint parameters. If the line */
/* is rejected, the endpoints are not changed and visible is set to FALSE. */
{
    double dx = *x1 - *x0;
    double dy = *y1 - *y0;
    /* Output is generated only if line is inside all four edges. */
    *visible = FALSE;
    /* First test for degenerate line and clip the point; ClipPoint returns */
    /* TRUE if the point lies inside the clip rectangle. */
    if (dx == 0 && dy == 0 && ClipPoint (*x0, *y0))
        *visible = TRUE;
    else {
        double tE = 0.0;
        double tL = 1.0;
        if (CLIPt (dx, xmin - *x0, &tE, &tL))                    /* Inside wrt left edge */
            if (CLIPt (-dx, *x0 - xmax, &tE, &tL))               /* Inside wrt right edge */
                if (CLIPt (dy, ymin - *y0, &tE, &tL))            /* Inside wrt bottom edge */
                    if (CLIPt (-dy, *y0 - ymax, &tE, &tL)) {     /* Inside wrt top edge */
                        *visible = TRUE;
                        /* Compute PL intersection, if tL has moved */
                        if (tL < 1) {
                            *x1 = *x0 + tL * dx;
                            *y1 = *y0 + tL * dy;
                        }
                        /* Compute PE intersection, if tE has moved */
                        if (tE > 0) {
                            *x0 += tE * dx;
                            *y0 += tE * dy;
                        }
                    }
    }
} /* Clip2D */

boolean CLIPt (double denom, double num, double *tE, double *tL)
/* This function computes a new value of tE or tL for an interior intersection */
/* of a line segment and an edge. Parameter denom is -(N_i • D), which reduces to */
/* ± Δx, Δy for upright rectangles (as shown in Table 3.1); its sign */
/* determines whether the intersection is PE or PL. Parameter num is N_i • (P_0 - P_Ei) */
/* for a particular edge/line combination, which reduces to directed horizontal */
/* and vertical distances from P_0 to an edge; its sign determines visibility */
/* of P_0 and is used to trivially reject horizontal or vertical lines. If the */
/* line segment can be trivially rejected, FALSE is returned; if it cannot be, */
/* TRUE is returned and the value of tE or tL is adjusted, if needed, for the */
/* portion of the segment that is inside the edge. */
```

Fig. 3.45 *(Cont.).*

```
{
    double t;

    if (denom > 0) {            /* PE intersection */
        t = num / denom;        /* Value of t at the intersection */
        if (t > tL)             /* tE and tL crossover */
            return FALSE;       /* so prepare to reject line */
        else if (t > tE)        /* A new tE has been found */
            tE = t;
    } else if (denom < 0) {     /* PL intersection */
        t = num / denom;        /* Value of t at the intersection */
        if (t < tE)             /* tE and tL crossover */
            return FALSE;       /* so prepare to reject line */
        else                    /* A new tL has been found */
            tL = t;
    } else if (num > 0)         /* Line on outside of edge */
        return FALSE;
    return TRUE;
}  /* CLIPt */
```

Fig. 3.45 Code for Liang–Barsky parametric line-clipping algorithm.

outside; i.e., would be invisible. The main procedure then does the actual clipping by moving the endpoints to the most recent values of t_E and t_L computed, but only if there is a line segment inside all four edges. This condition is tested for by a four-deep nested **if** that checks the flags returned by the function signifying whether or not the line segment was rejected.

In summary, the Cohen–Sutherland algorithm is efficient when outcode testing can be done cheaply (for example, by doing bitwise operations in assembly language) and trivial acceptance or rejection is applicable to the majority of line segments. Parametric line clipping wins when many line segments need to be clipped, since the actual calculation of the coordinates of the intersection points is postponed until needed, and testing can be done on parameter values. This parameter calculation is done even for endpoints that would have been trivially accepted in the Cohen–Sutherland strategy, however. The Liang–Barsky algorithm is more efficient than the Cyrus–Beck version because of additional trivial rejection testing that can avoid calculation of all four parameter values for lines that do not intersect the clip rectangle. For lines that cannot be trivially rejected by Cohen–Sutherland because they do not lie in an invisible halfplane, the rejection tests of Liang–Barsky are clearly preferable to the repeated clipping required by Cohen–Sutherland. The Nicholl et

al. algorithm of Section 19.1.1 is generally preferable to either Cohen–Sutherland or Liang–Barsky but does not generalize to 3D, as does parametric clipping. Speed-ups to Cohen–Sutherland are discussed in [DUVA90]. Exercise 3.29 concerns instruction counting for the two algorithms covered here, as a means of contrasting their efficiency under various conditions.

3.13 CLIPPING CIRCLES AND ELLIPSES

To clip a circle against a rectangle, we can first do a trivial accept/reject test by intersecting the circle's extent (a square of the size of the circle's diameter) with the clip rectangle, using the algorithm in the next section for polygon clipping. If the circle intersects the rectangle, we divide it into quadrants and do the trivial accept/reject test for each. These tests may lead in turn to tests for octants. We can then compute the intersection of the circle and the edge analytically by solving their equations simultaneously, and then scan convert the resulting arcs using the appropriately initialized algorithm with the calculated (and suitably rounded) starting and ending points. If scan conversion is fast, or if the circle is not too large, it is probably more efficient to scissor on a pixel-by-pixel basis, testing each boundary pixel against the rectangle bounds before it is written. An extent check would certainly be useful in any case. If the circle is filled, spans of adjacent interior pixels on each scan line can be filled without bounds checking by clipping each span and then filling its interior pixels, as discussed in Section 3.7.

To clip ellipses, we use extent testing at least down to the quadrant level, as with circles. We can then either compute the intersections of ellipse and rectangle analytically and use those (suitably rounded) endpoints in the appropriately initialized scan-conversion algorithm given in the next section, or clip as we scan convert.

3.14 CLIPPING POLYGONS

An algorithm that clips a polygon must deal with many different cases, as shown in Fig. 3.46. The case in part (a) is particularly noteworthy in that the concave polygon is clipped into two separate polygons. All in all, the task of clipping seems rather complex. Each edge of the polygon must be tested against each edge of the clip rectangle; new edges must be added, and existing edges must be discarded, retained, or divided. Multiple polygons may result from clipping a single polygon. We need an organized way to deal with all these cases.

3.14.1 The Sutherland–Hodgman Polygon-Clipping Algorithm

Sutherland and Hodgman's polygon-clipping algorithm [SUTH74b] uses a divide-and-conquer strategy: It solves a series of simple and identical problems that, when combined, solve the overall problem. The simple problem is to clip a polygon against a single infinite clip edge. Four clip edges, each defining one boundary of the clip rectangle (see Fig. 3.47), successively clip a polygon against a clip rectangle.

Note the difference between this strategy for a polygon and the Cohen–Sutherland algorithm for clipping a line: The polygon clipper clips against four edges in succession, whereas the line clipper tests the outcode to see which edge is crossed, and clips only when

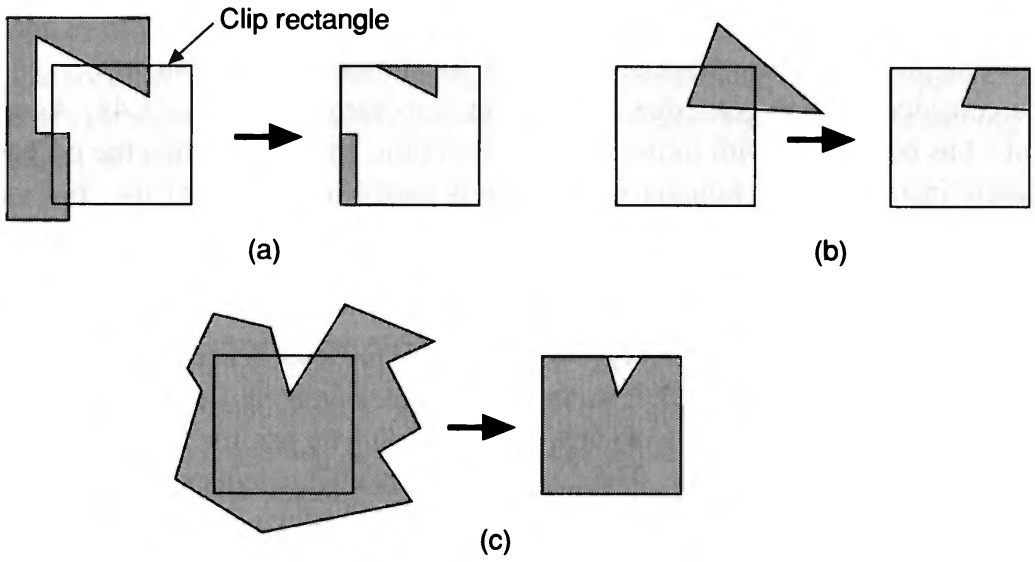

Fig. 3.46 Examples of polygon clipping. (a) Multiple components. (b) Simple convex case. (c) Concave case with many exterior edges.

necessary. The actual Sutherland–Hodgman algorithm is in fact more general: A polygon (convex or concave) can be clipped against any convex clipping polygon; in 3D, polygons can be clipped against convex polyhedral volumes defined by planes. The algorithm accepts a series of polygon vertices $v_1, v_2, \ldots, v_n$. In 2D, the vertices define polygon edges from v_i to v_{i+1} and from v_n to v_1. The algorithm clips against a single, infinite clip edge and outputs another series of vertices defining the clipped polygon. In a second pass, the partially clipped polygon is then clipped against the second clip edge, and so on.

The algorithm moves around the polygon from v_n to v_1 and then on back to v_n, at each step examining the relationship between successive vertices and the clip edge. At each step,

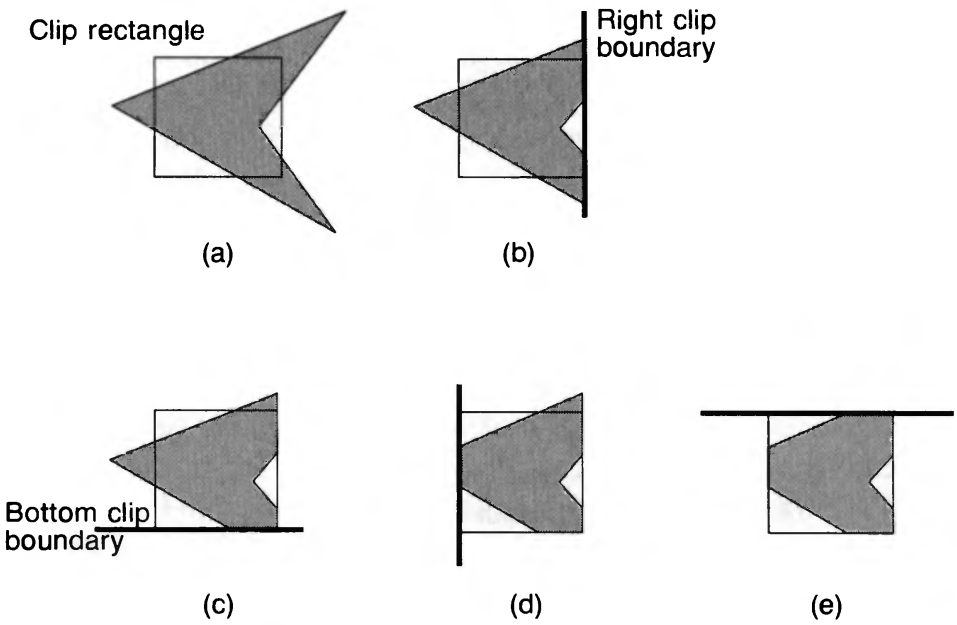

Fig. 3.47 Polygon clipping, edge by edge. (a) Before clipping. (b) Clip on right. (c) Clip on bottom. (d) Clip on left. (e) Clip on top; polygon is fully clipped.

zero, one, or two vertices are added to the output list of vertices that defines the clipped polygon. Four possible cases must be analyzed, as shown in Fig. 3.48.

Let's consider the polygon edge from vertex s to vertex p in Fig. 3.48. Assume that start point s has been dealt with in the previous iteration. In case 1, when the polygon edge is completely inside the clip boundary, vertex p is added to the output list. In case 2, the intersection point i is output as a vertex because the edge intersects the boundary. In case 3, both vertices are outside the boundary, so there is no output. In case 4, the intersection point i and p are both added to the output list.

Function SutherlandHodgmanPolygonClip( ) in Fig. 3.49 accepts an array *inVertexArray* of vertices and creates another array *outVertexArray* of vertices. To keep the code simple, we show no error checking on array bounds, and we use the function Output( ) to place a vertex into *outVertexArray*. The function Intersect( ) calculates the intersection of the polygon edge from vertex s to vertex p with *clip Boundary*, which is defined by two vertices on the clip polygon's boundary. The function Inside( ) returns **true** if the vertex is on the inside of the clip boundary, where "inside" is defined as "to the left of the clip boundary when one looks from the first vertex to the second vertex of the clip boundary." This sense corresponds to a counterclockwise enumeration of edges. To calculate whether a point lies outside a clip boundary, we can test the sign of the dot product of the normal to the clip boundary and the polygon edge, as described in Section 3.12.4. (For the simple case of an upright clip rectangle, we need only test the sign of the horizontal or vertical distance to its boundary.)

Sutherland and Hodgman show how to structure the algorithm so that it is reentrant [SUTH74b]. As soon as a vertex is output, the clipper calls itself with that vertex. Clipping is performed against the next clip boundary, so that no intermediate storage is necessary for the partially clipped polygon: In essence, the polygon is passed through a "pipeline" of clippers. Each step can be implemented as special-purpose hardware with no intervening buffer space. This property (and its generality) makes the algorithm suitable for today's hardware implementations. In the algorithm as it stands, however, new edges may be introduced on the border of the clip rectangle. Consider Fig. 3.46 (a)—a new edge is introduced by connecting the left top of the triangle and the left top of the rectangle. A

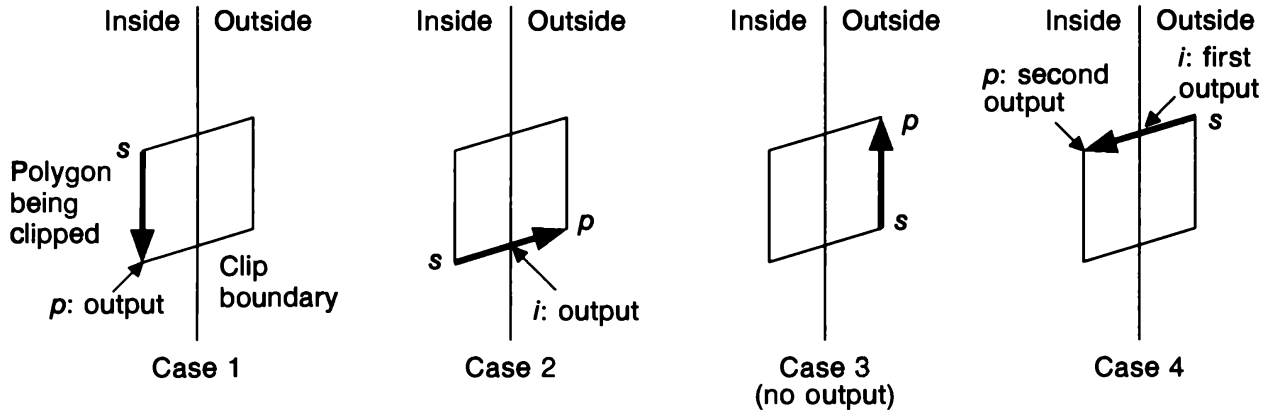

Fig. 3.48 Four cases of polygon clipping.

postprocessing phase can eliminate these edges, as discussed in Chapter 19. A polygon-clipping algorithm based on the parametric-line representation for clipping to upright rectangular clip regions is discussed, along with the Weiler algorithm for clipping polygons to polygons, in Section 19.1.

3.15 GENERATING CHARACTERS

3.15.1 Defining and Clipping Characters

There are two basic techniques for defining characters. The most general but most computationally expensive way is to define each character as a curved or polygonal outline and to scan convert it as needed. We first discuss the other, simpler way, in which each character in a given font is specified as a small rectangular bitmap. Generating a character then entails simply using a copyPixel to copy the character's image from an offscreen canvas, called a *font cache*, into the frame buffer at the desired position.

The font cache may actually be in the frame buffer, as follows. In most graphics systems in which the display is refreshed from a private frame buffer, that memory is larger than is strictly required for storing the displayed image. For example, the pixels for a rectangular screen may be stored in a square memory, leaving a rectangular strip of "invisible"screen memory. Alternatively, there may be enough memory for two screens, one of which is being refreshed and one of which is being drawn in, to double-buffer the image. The font cache for the currently displayed font(s) is frequently stored in such invisible screen memory because the display controller's copyPixel works fastest within local image memory. A related use for such invisible memory is for saving screen areas temporarily obscured by popped-up images, such as windows, menus, and forms.

The bitmaps for the font cache are usually created by scanning in enlarged pictures of characters from typesetting fonts in various sizes; a typeface designer can then use a paint program to touch up individual pixels in each character's bitmap as necessary. Alternatively, the type designer may use a paint program to create, from scratch, fonts that are especially designed for screens and low-resolution printers. Since small bitmaps do not scale well, more than one bitmap must be defined for a given character in a given font just to provide various standard sizes. Furthermore, each type face requires its own set of bitmaps. Therefore, a distinct font cache is needed for each font loaded by the application.

Bitmap characters are clipped automatically by SRGP as part of its implementation of copyPixel. Each character is clipped to the destination rectangle on a pixel-by-pixel basis, a technique that lets us clip a character at any row or column of its bitmap. For systems with slow copyPixel operations, a much faster, although cruder, method is to clip the character or even the entire string on an all-or-nothing basis by doing a trivial accept of the character or string extent. Only if the extent is trivially accepted is the copyPixel applied to the character or string. For systems with a fast copyPixel, it is still useful to do trivial accept/reject testing of the string extent as a precursor to clipping individual characters during the copyPixel operation.

SRGP's simple bitmap font-cache technique stores the characters side by side in a canvas that is quite wide but is only as tall as the tallest character; Fig. 3.50 shows a portion

```
typedef point vertex;                          /* point holds double x, y */
typedef vertex edge[2];
typedef vertex vertexArray[MAX];               /* MAX is a declared constant */

static void Output (vertex, int *, vertexArray);
static boolean Inside (vertex, edge);
static vertex Intersect (vertex, vertex, edge);

void SutherlandHodgmanPolygonClip (
        vertexArray inVertexArray,             /* Input vertex array */
        vertexArray outVertexArray,            /* Output vertex array */
        int inLength,                          /* No. of entries in inVertexArray */
        int *outLength,                        /* No. of entries in outVertexArray */
        edge clipBoundary)                     /* Edge of clip polygon */
{
    vertex s, p,                               /* Start, end pt. of current polygon edge */
    i;                                         /* Intersection pt. with a clip boundary */
    int j;                                     /* Vertex loop counter */

    *outLength = 0;   /* Start with the last vertex in inVertexArray */
    s = inVertexArray[inLength - 1];
    for (j = 0; j < inLength; j++) {
        p = inVertexArray[j];   /* Now s and p correspond to the vertices in Fig. 3.48 */
        if (Inside (p, clipBoundary)) {            /* Cases 1 and 4 */
            if (Inside (s, clipBoundary))          /* Case 1 */
                Output (p, outLength, outVertexArray);
            else {                                 /* Case 4 */
                i = Intersect (s, p, clipBoundary);
                Output (i, outLength, outVertexArray);
                Output (p, outLength, outVertexArray);
            }
        } else                                     /* Cases 2 and 3 */
            if (Inside (s, clipBoundary)) {        /* Case 2 */
                i = Intersect (s, p, clipBoundary);
                Output (i, outLength, outVertexArray);
            }                                      /* No action for case 3 */
        s = p;                                     /* Advance to next pair of vertices */
    }  /* for */
}  /* SutherlandHodgmanPolygonClip */

/* Adds newVertex to outVertexArray and then updates outLength */
static void Output (vertex newVertex, int *outLength, vertexArray outVertexArray)
{
...
}
/* Checks whether the vertex lies inside the clip edge or not */
static boolean Inside (vertex testVertex, edge clipBoundary)
```

Fig. 3.49 *(Cont.)*.

```
{
...
}

/* Clips polygon edge (first, second) against clipBoundary, outputs the new point */
static vertex Intersect (vertex first, vertex second, edge clipBoundary)
{
...
}
```

Fig. 3.49 Sutherland–Hodgman polygon-clipping algorithm.

of the cache, along with discrete instances of the same characters at low resolution. Each loaded font is described by a struct (declared in Fig. 3.51) containing a reference to the canvas that stores the characters' images, along with information on the height of the characters and the amount of space to be placed between adjacent characters in a text string. (Some packages store the space between characters as part of a character's width, to allow variable intercharacter spacing.)

As described in Section 2.1.5, descender height and total height are constants for a given font—the former is the number of rows of pixels at the bottom of the font cache used only by descenders, and the latter is simply the height of the font-cache canvas. The width of a character, on the other hand, is not considered a constant; thus, a character can occupy the space that suits it, rather than being forced into a fixed-width character box. SRGP puts a fixed amount of space between characters when it draws a text string, the amount being specified as part of each font's descriptor. A word-processing application can display lines of text by using SRGP to display individual words of text, and can right-justify lines by using variable spacing between words and after punctuation to fill out lines so that their rightmost characters are aligned at the right margin. This involves using the text-extent inquiry facilities to determine where the right edge of each word is, in order to calculate the start of the next word. Needless to say, SRGP's text-handling facilities are really too crude for sophisticated word processing, let alone for typesetting programs, since such applications require far finer control over the spacing of individual letters to deal with such effects as sub- and superscripting, kerning, and printing text that is not horizontally aligned.

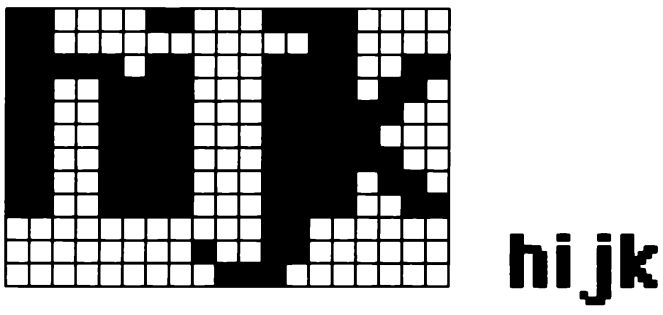

Fig. 3.50 Portion of an example of a font cache.

```
typedef struct {
    int leftX, width;                    /* Horizontal location, width of image in font cache */
} charLocation;

typedef struct {
    canvasID cache;
    int descenderHeight, totalHeight;    /* Height is a constant; width varies */
    int interCharacterSpacing;           /* Measured in pixels */
    charLocation locationTable[128];     /* Explained in the text */
} fontCacheDescriptor;
```

Fig. 3.51 Type declarations for the font cache.

3.15.2 Implementation of a Text Output Primitive

In the code of Fig. 3.52, we show how SRGP text is implemented internally: Each character in the given string is placed individually, and the space between characters is dictated by the appropriate field in the font descriptor. Note that complexities such as dealing with mixed fonts in a string must be handled by the application program.

```
void SRGP_characterText (
        point origin,                    /* Where to place the character in the current canvas */
        char *stringToPrint,
        fontCacheDescriptor fontInfo)
{
    int i;

    /* Origin specified by the application is for baseline and does not include descender. */
    origin.y -= fontInfo.descenderHeight;

    for (i = 0; i < strlen (stringToPrint); i++) {
        rectangle fontCacheRectangle;
        char charToPrint = stringToPrint[i];
        /* Find the rectangular region within the cache wherein the character lies */
        charLocation *fip = &fontInfo.locationTable[charToPrint];

        fontCacheRectangle.bottomLeft = SRGP_defPoint (fip->leftX, 0);
        fontCacheRectangle.topRight = SRGP_defPoint (fip->leftX + fip->width - 1,
                                                     fontInfo.totalHeight - 1);

        SRGP_copyPixel (fontInfo.cache, fontCacheRectangle, origin);
        /* Update the origin to move past the new character plus intercharacter spacing */
        origin.x += fip->width + interCharacterSpacing;
    }
} /* SRGP_characterText */
```

Fig. 3.52 Implementation of character placement for SRGP's text primitive.

We mentioned that the bitmap technique requires a distinct font cache for each combination of font, size, and face for each different resolution of display or output device supported. A single font in eight different point sizes and four faces (normal, bold, italic, bold italic) thus requires 32 font caches! Figure 3.53(a) shows a common way of allowing a single font cache to support multiple face variations: The italic face is approximated by splitting the font's image into regions with horizontal "cuts," then offsetting the regions while performing a sequence of calls to SRGP_copyPixel.

This crude approximation does not make pleasing characters; for example, the dot over the "*i*" is noncircular. More of a problem for the online user, the method distorts the intercharacter spacing and makes picking much more difficult. A similar trick to achieve boldface is to copy the image twice in **or** mode with a slight horizontal offset (Fig. 3.53b). These techniques are not particularly satisfactory, in that they can produce illegible characters, especially when combined with subscripting.

A better way to solve the storage problem is to store characters in an abstract, device-independent form using polygonal or curved outlines of their shapes defined with floating-point parameters, and then to transform them appropriately. Polynomial functions called *splines* (see Chapter 11) provide smooth curves with continuous first and higher derivatives and are commonly used to encode text outlines. Although each character definition takes up more space than its representation in a font cache, multiple sizes may be derived from a single stored representation by suitable scaling; also, italics may be quickly approximated by shearing the outline. Another major advantage of storing characters in a completely device-independent form is that the outlines may be arbitrarily translated, rotated, scaled, and clipped (or used as clipping regions themselves).

The storage economy of splined characters is not quite so great as this description suggests. For instance, not all point sizes for a character may be obtained by scaling a single abstract shape, because the shape for an aesthetically pleasing font is typically a function of point size; therefore, each shape suffices for only a limited range of point sizes. Moreover, scan conversion of splined text requires far more processing than the simple copyPixel implementation, because the device-independent form must be converted to pixel coordinates on the basis of the current size, face, and transformation attributes. Thus, the font-cache technique is still the most common for personal computers and even is used for many workstations. A strategy that offers the best of both methods is to store the fonts in outline form but to convert the ones being used in a given application to their bitmap equivalents—for example, to build a font cache on the fly. We discuss processing of splined text in more detail in Section 19.4.

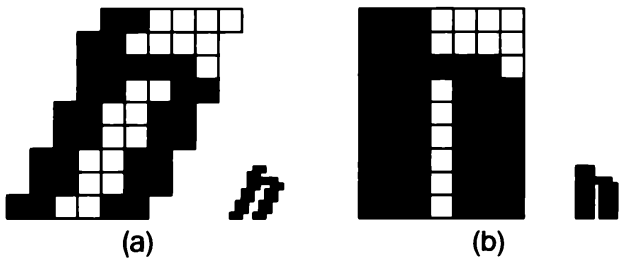

Fig. 3.53 Tricks for creating different faces for a font. (a) Italic. (b) Bold.

3.16 SRGP_copyPixel

If only WritePixel and ReadPixel low-level procedures are available, the SRGP_copyPixel procedure can be implemented as a doubly nested **for** loop for each pixel. For simplicity, assume first that we are working with a bilevel display and do not need to deal with the low-level considerations of writing bits that are not word-aligned; in Section 19.6, we cover some of these more realistic issues that take hardware-memory organization into account. In the inner loop of our simple SRGP_copyPixel, we do a ReadPixel of the source and destination pixels, logically combine them according to the SRGP write mode, and then WritePixel the result. Treating **replace** mode, the most common write mode, as a special case allows a simpler inner loop that does only a ReadPixel/WritePixel of the source into the destination, without having to do a logical operation. The clip rectangle is used during address calculation to restrict the region into which destination pixels are written.

3.17 ANTIALIASING

3.17.1 Increasing Resolution

The primitives drawn so far have a common problem: They have jagged edges. This undesirable effect, known as *the jaggies* or *staircasing*, is the result of an all-or-nothing approach to scan conversion in which each pixel either is replaced with the primitive's color or is left unchanged. Jaggies are an instance of a phenomenon known as *aliasing*. The application of techniques that reduce or eliminate aliasing is referred to as *antialiasing*, and primitives or images produced using these techniques are said to be *antialiased*. In Chapter 14, we discuss basic ideas from signal processing that explain how aliasing got its name, why it occurs, and how to reduce or eliminate it when creating pictures. Here, we content ourselves with a more intuitive explanation of why SRGP's primitives exhibit aliasing, and describe how to modify the line scan-conversion algorithm developed in this chapter to generate antialiased lines.

Consider using the midpoint algorithm to draw a 1-pixel-thick black line, with slope between 0 and 1, on a white background. In each column through which the line passes, the algorithm sets the color of the pixel that is closest to the line. Each time the line moves between columns in which the pixels closest to the line are not in the same row, there is a sharp jag in the line drawn into the canvas, as is clear in Fig. 3.54(a). The same is true for other scan-converted primitives that can assign only one of two intensity values to pixels.

Suppose we now use a display device with twice the horizontal and vertical resolution. As shown in Fig. 3.54 (b), the line passes through twice as many columns and therefore has twice as many jags, but each jag is half as large in x and in y. Although the resulting picture looks better, the improvement comes at the price of quadrupling the memory cost, memory bandwidth, and scan-conversion time. Increasing resolution is an expensive solution that only diminishes the problem of jaggies—it does not eliminate the problem. In the following sections, we look at antialiasing techniques that are less costly, yet result in significantly better images.

3.17.2 Unweighted Area Sampling

The first approach to improving picture quality can be developed by recognizing that, although an ideal primitive such as the line has zero width, the primitive we are drawing has

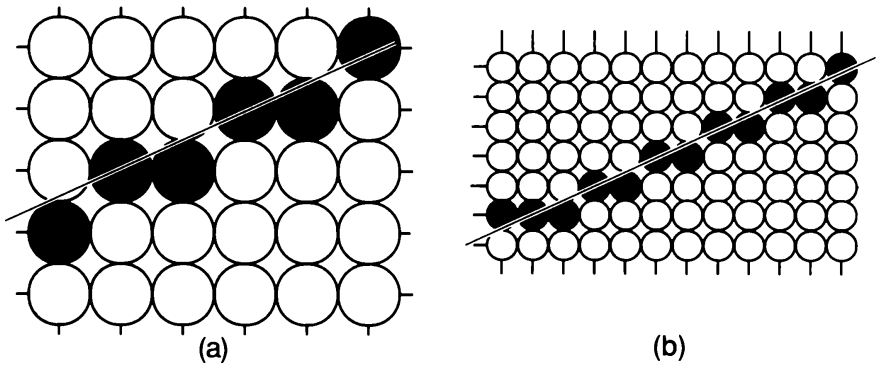

Fig. 3.54 (a) Standard midpoint line on a bilevel display. (b) Same line on a display that has twice the linear resolution.

nonzero width. A scan-converted primitive occupies a finite area on the screen—even the thinnest horizontal or vertical line on a display surface is 1 pixel thick and lines at other angles have width that varies over the primitive. Thus, we think of any line as a rectangle of a desired thickness covering a portion of the grid, as shown in Fig. 3.55. It follows that a line should not set the intensity of only a single pixel in a column to black, but rather should contribute some amount of intensity to each pixel in the columns whose area it intersects. (Such varying intensity can be shown on only those displays with multiple bits per pixel, of course.) Then, for 1-pixel-thick lines, only horizontal and vertical lines would affect exactly 1 pixel in their column or row. For lines at other angles, more than 1 pixel would now be set in a column or row, each to an appropriate intensity.

But what is the geometry of a pixel? How large is it? How much intensity should a line contribute to each pixel it intersects? It is computationally simple to assume that the pixels form an array of nonoverlapping square tiles covering the screen, centered on grid points. (When we refer to a primitive overlapping all or a portion of a pixel, we mean that it covers (part of) the tile; to emphasize this we sometimes refer to the square as the *area represented by the pixel*.) We also assume that a line contributes to each pixel's intensity an amount

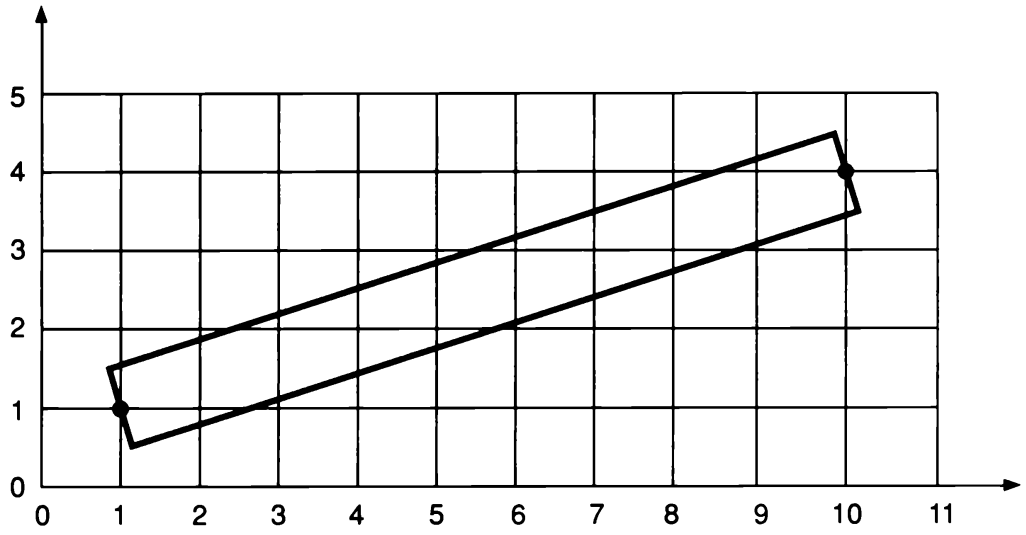

Figure 3.55 Line of nonzero width from point (1,1) to point (10,4).

proportional to the percentage of the pixel's tile it covers. A fully covered pixel on a black and white display will be colored black, whereas a partially covered pixel will be colored a gray whose intensity depends on the line's coverage of the pixel. This technique, as applied to the line shown in Fig. 3.55, is shown in Fig. 3.56.

For a black line on a white background, pixel (2, 1) is about 70 percent black, whereas pixel (2, 2) is about 25 percent black. Pixels not intersected by the line, such as (2, 3), are completely white. Setting a pixel's intensity in proportion to the amount of its area covered by the primitive softens the harsh, on–off characteristic of the edge of the primitive and yields a more gradual transition between full on and full off. This blurring makes a line look better at a distance, despite the fact that it spreads the on–off transition over multiple pixels in a column or row. A rough approximation to the area overlap can be found by dividing the pixel into a finer grid of rectangular subpixels, then counting the number of subpixels inside the line—for example, below the line's top edge or above its bottom edge (see Exercise 3.32).

We call the technique of setting intensity proportional to the amount of area covered *unweighted area sampling*. This technique produces noticeably better results than does setting pixels to full intensity or zero intensity, but there is an even more effective strategy called *weighted area sampling*. To explain the difference between the two forms of area sampling, we note that unweighted area sampling has the following three properties. First, the intensity of a pixel intersected by a line edge decreases as the distance between the pixel center and the edge increases: The farther away a primitive is, the less influence it has on a pixel's intensity. This relation obviously holds because the intensity decreases as the area of overlap decreases, and that area decreases as the line's edge moves away from the pixel's center and toward the boundary of the pixel. When the line covers the pixel completely, the overlap area and therefore the intensity are at a maximum; when the primitive edge is just tangent to the boundary, the area and therefore the intensity are zero.

A second property of unweighted area sampling is that a primitive cannot influence the intensity at a pixel at all if the primitive does not intersect the pixel—that is, if it does not intersect the square tile represented by the pixel. A third property of unweighted area

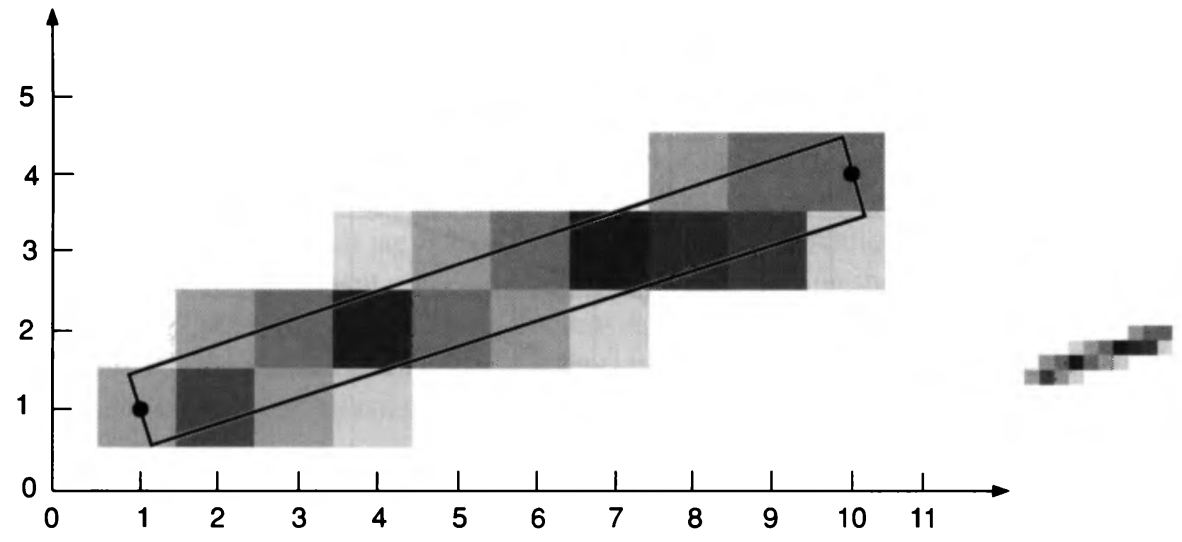

Fig. 3.56 Intensity proportional to area covered.

sampling is that equal areas contribute equal intensity, regardless of the distance between the pixel's center and the area; only the total amount of overlapped area matters. Thus, a small area in the corner of the pixel contributes just as much as does an equal-sized area near the pixel's center.

3.17.3 Weighted Area Sampling

In weighted area sampling, we keep unweighted area sampling's first and second properties (intensity decreases with decreased area overlap, and primitives contribute only if they overlap the area represented by the pixel), but we alter the third property. We let equal areas contribute unequally: A small area closer to the pixel center has greater influence than does one at a greater distance. A theoretical basis for this change is given in Chapter 14, where we discuss weighted area sampling in the context of filtering theory.

To retain the second property, we must make the following change in the geometry of the pixel. In unweighted area sampling, if an edge of a primitive is quite close to the boundary of the square tile we have used to represent a pixel until now, but does not actually intersect this boundary, it will not contribute to the pixel's intensity. In our new approach, the pixel represents a circular area larger than the square tile; the primitive *will* intersect this larger area; hence, it will contribute to the intensity of the pixel.

To explain the origin of the adjectives *unweighted* and *weighted*, we define a *weighting function* that determines the influence on the intensity of a pixel of a given small area dA of a primitive, as a function of dA's distance from the center of the pixel. This function is constant for unweighted area sampling, and decreases with increasing distance for weighted area sampling. Think of the weighting function as a function, $W(x, y)$, on the plane, whose height above the (x, y) plane gives the weight for the area dA at (x, y). For unweighted area sampling with the pixels represented as square tiles, the graph of W is a box, as shown in Fig. 3.57.

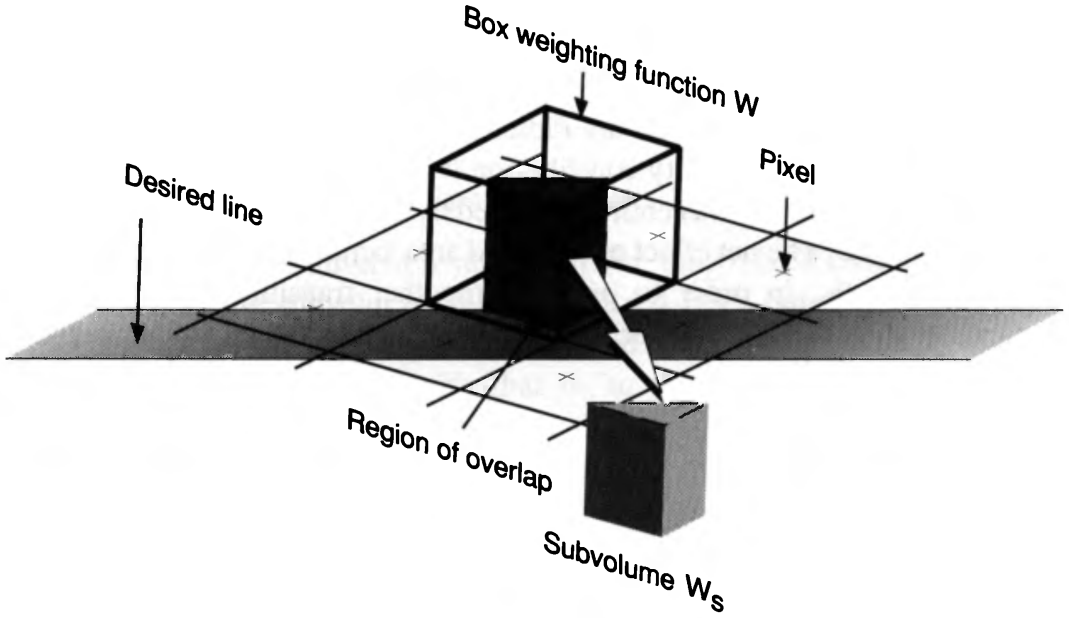

Fig. 3.57 Box filter for square pixel.

The figure shows square pixels, with centers indicated by crosses at the intersections of grid lines; the weighting function is shown as a box whose base is that of the current pixel. The intensity contributed by the area of the pixel covered by the primitive is the total of intensity contributions from all small areas in the region of overlap between the primitive and the pixel. The intensity contributed by each small area is proportional to the area multiplied by the weight. Therefore, the total intensity is the integral of the weighting function over the area of overlap. The volume represented by this integral, W_S, is always a fraction between 0 and 1, and the pixel's intensity I is $I_{max} \cdot W_S$. In Fig. 3.57, W_S is a wedge of the box. The weighting function is also called a *filter function*, and the box is also called a *box filter*. For unweighted area sampling, the height of the box is normalized to 1, so that the box's volume is 1, which causes a thick line covering the entire pixel to have an intensity $I = I_{max} \cdot 1 = I_{max}$.

Now let us construct a weighting function for weighted area sampling; it must give less weight to small areas farther away from the pixel center than it does to those closer. Let's pick a weighting function that is the simplest decreasing function of distance; for example, we choose a function that has a maximum at the center of the pixel and decreases linearly with increasing distance from the center. Because of rotational symmetry, the graph of this function forms a circular cone. The circular base of the cone (often called the *support* of the filter) should have a radius larger than you might expect; the filtering theory of Chapter 14 shows that a good choice for the radius is the unit distance of the integer grid. Thus, a primitive fairly far from a pixel's center can still influence that pixel's intensity; also, the supports associated with neighboring pixels overlap, and therefore a single small piece of a primitive may actually contribute to several different pixels (see Fig. 3.58). This overlap also ensures that there are no areas of the grid not covered by some pixel, which would be the case if the circular pixels had a radius of only one-half of a grid unit.[12]

As with the box filter, the sum of all intensity contributions for the cone filter is the volume under the cone and above the intersection of the cone's base and the primitive; this volume W_S is a vertical section of the cone, as shown in Fig. 3.58. As with the box filter, the height of the cone is first normalized so that the volume under the entire cone is 1; this allows a pixel whose support is completely covered by a primitive to be displayed at maximum intensity. Although contributions from areas of the primitive far from the pixel's center but still intersecting the support are rather small, a pixel whose center is sufficiently close to a line receives some intensity contribution from that line. Conversely, a pixel that, in the square-geometry model, was entirely covered by a line of unit thickness[13] is not quite as bright as it used to be. The net effect of weighted area sampling is to decrease the contrast between adjacent pixels, in order to provide smoother transitions. In particular, with weighted area sampling, a horizontal or vertical line of unit thickness has more than 1 pixel

[12]As noted in Section 3.2.1, pixels displayed on a CRT are roughly circular in cross-section, and adjacent pixels typically overlap; the model of overlapping circles used in weighted area sampling, however, is not directly related to this fact and holds even for display technologies, such as the plasma panel, in which the physical pixels are actually nonoverlapping square tiles.

[13]We now say a "a line of unit thickness" rather than "a line 1 pixel thick" to make it clear that the unit of line width is still that of the SRGP grid, whereas the pixel's support has grown to have a two-unit diameter.

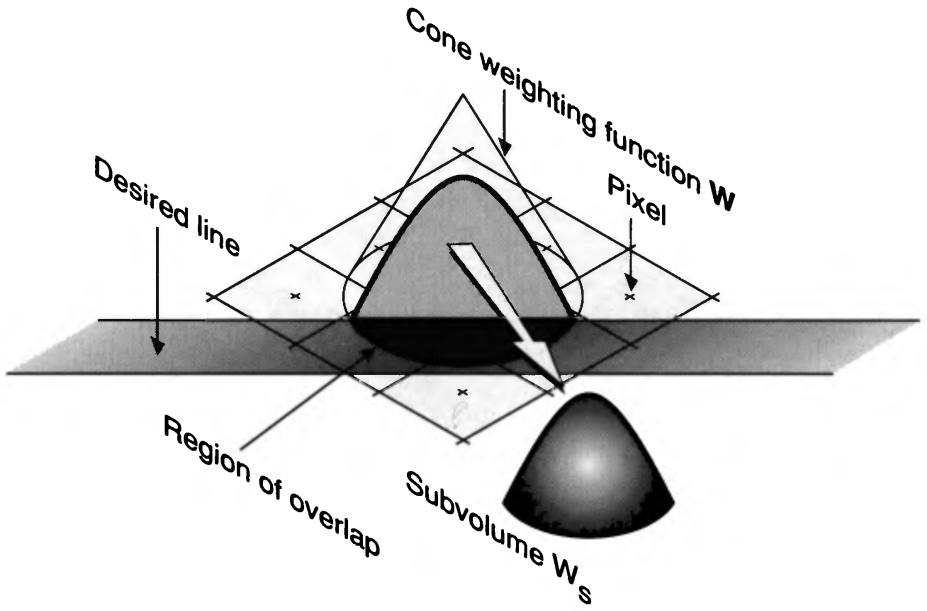

Fig. 3.58 Cone filter for circular pixel with diameter of two grid units.

intensified in each column or row, which would not be the case for unweighted area sampling.

The conical filter has two useful properties: rotational symmetry and linear decrease of the function with radial distance. We prefer rotational symmetry because it not only makes area calculations independent of the angle of the line, but also is theoretically optimal, as shown in Chapter 14. We also show there, however, that the cone's linear slope (and its radius) are only an approximation to the optimal filter function, although the cone filter is still better than the box filter. Optimal filters are computationally most expensive, box filters least, and therefore cone filters are a very reasonable compromise between cost and quality. The dramatic difference between an unfiltered and filtered line drawing is shown in Fig. 3.59. Notice how the problems of indistinct lines and moiré patterns are greatly ameliorated by filtering. Now we need to integrate the cone filter into our scan-conversion algorithms.

3.17.4 Gupta–Sproull Antialiased Lines

The Gupta–Sproull scan-conversion algorithm for lines [GUPT81a] described in this section precomputes the subvolume of a normalized filter function defined by lines at various directed distances from the pixel center, and stores them in a table. We use a pixel area with radius equal to a grid unit—that is, to the distance between adjacent pixel centers—so that a line of unit thickness with slope less than 1 typically intersects three supports in a column, minimally two and maximally five, as shown in Fig. 3.60. For a radius of 1, each circle partially covers the circles of its neighboring pixels.

Figure 3.61 shows the geometry of the overlap between line and pixel that is used for table lookup of a function Filter (D, t). Here D is the (angle-independent) distance between pixel and line centers, t is a constant for lines of a given thickness, and function Filter() is dependent on the shape of the filter function. Gupta and Sproull's paper gives the table for a cone filter for a 4-bit display; it contains fractional values of Filter (D, t) for equal

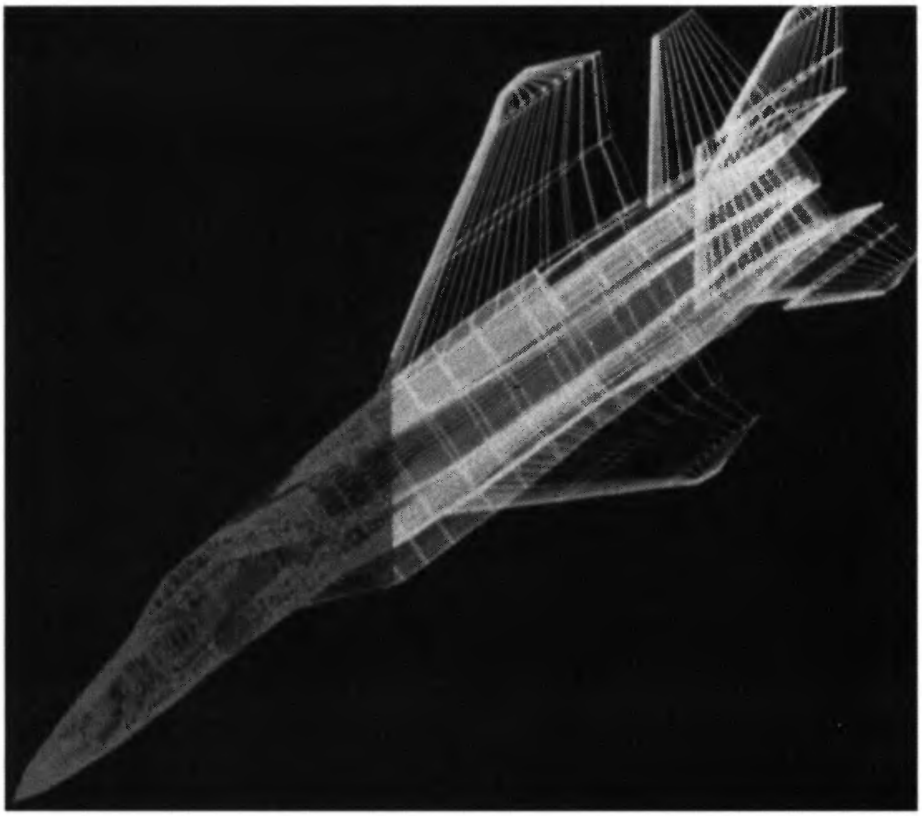

Fig. 3.59 Filtered line drawing. The left half is unfiltered; the right half is filtered. (Courtesy of Branko Gerovac, Digital Equipment Corporation.)

increments of D ranging from 0 to 1.5 and for $t = 1$. The function, by definition, is 0 outside the support, for $D \geq 1 + \frac{1}{2} = \frac{24}{16}$ in this case. The precision of the distance is only 1 in 16, because it need not be greater than that of the intensity—4 bits, in this case.

Now we are ready to modify the midpoint line scan-conversion algorithm. As before, we use the decision variable d to choose between E and NE pixels, but must then set the intensity of the chosen pixel and its two vertical neighbors, on the basis of the distances from these pixels to the line. Figure 3.61 shows the relevant geometry; we can calculate the

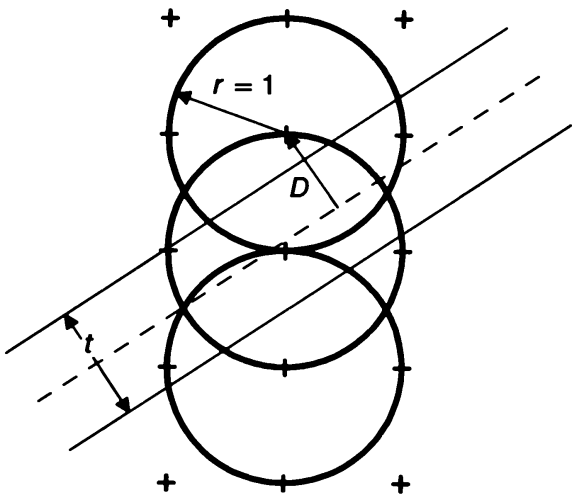

Fig. 3.60 One-unit-thick line intersects 3-pixel supports.

true, perpendicular distance D from the vertical distance v, using simple trigonometry.

Using similar triangles and knowing that the slope of the line is dy/dx, we can see from the diagram that

$$D = v \cos\phi = \frac{vdx}{\sqrt{dx^2 + dy^2}} \tag{3.2}$$

The vertical distance v between a point on the line and the chosen pixel with the same x coordinate is just the difference between their y coordinates. It is important to note that this distance is a signed value. That is, if the line passes below the chosen pixel, v is negative; if it passes above the chosen pixel, v is positive. We should therefore pass the absolute value of the distance to the filter function. The chosen pixel is also the middle of the 3 pixels that must be intensified. The pixel above the chosen one is a vertical distance $1 - v$ from the line, whereas the pixel below is a vertical distance $1 + v$ from the line. You may want to verify that these distances are valid regardless of the relative position of the line and the pixels, because the distance v is a signed quantity.

Rather than computing v directly, our strategy is to use the incremental computation of $d = F(M) = F(x_P + 1, y_P + \frac{1}{2})$. In general, if we know the x coordinate of a point on the line, we can compute that point's y coordinate using the relation developed in Section 3.2.2, $F(x, y) = 2(ax + by + c) = 0$:

$$y = (ax + c)/-b.$$

For pixel E, $x = x_P + 1$, and $y = y_P$, and $v = y - y_P$; thus

$$v = ((a(x_P + 1) + c)/-b) - y_P.$$

Now multiplying both sides by $-b$ and collecting terms,

$$-bv = a(x_P + 1) + by_P + c = F(x_P + 1, y_P)/2.$$

But $b = -dx$. Therefore, $vdx = F(x_P + 1, y_P)/2$. Note that vdx is the numerator of Eq. (3.2) for D, and that the denominator is a constant that can be precomputed. Therefore, we

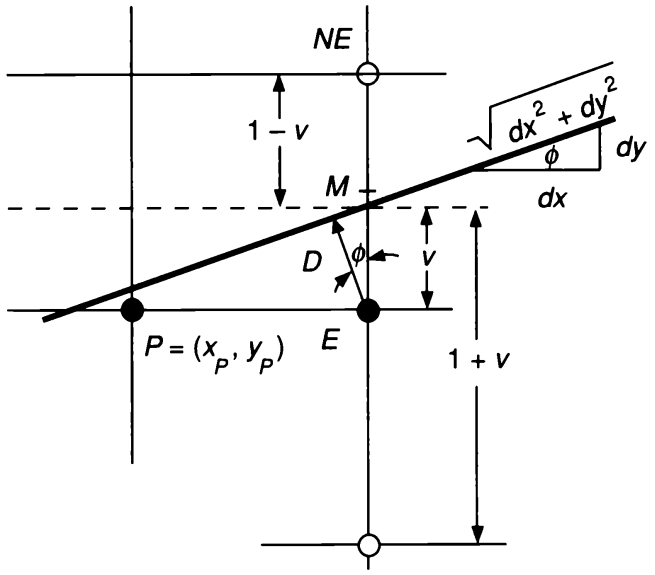

Fig. 3.61 Calculating distances to line in midpoint algorithm.

would like to compute vdx incrementally from the prior computation of $d = F(M)$, and avoid division by 2 to preserve integer arithmetic. Thus, for the pixel E,

$$\begin{aligned} 2vdx = \mathrm{F}(x_P + 1, y_P) &= 2a(x_P + 1) + 2by_P + 2c \\ &= 2a(x_P + 1) + 2b(y_P + \tfrac{1}{2}) - 2b/2 + 2c \\ &= d + dx. \end{aligned}$$

Thus

$$D = \frac{d + dx}{2\sqrt{x^2 + y^2}}$$

and the constant denominator is $1/(2\sqrt{x^2 + y^2})$. The corresponding numerators for the pixels at $y_P + 1$ and $y_P - 1$ are then easily obtained as $2(1 - v)dx = 2dx - 2vdx$, and $2(1 + v)dx = 2dx + 2vdx$, respectively.

Similarly, for pixel NE,

$$\begin{aligned} 2vdx = F(x_P + 1, y_P + 1) &= 2a(x_P + 1) + 2b(y_P + \tfrac{1}{2}) + 2b/2 + 2c, \\ &= d - dx, \end{aligned}$$

and the corresponding numerators for the pixels at $y_P + 2$ and y_P are again $2(1 - v)dx = 2dx - 2vdx$ and $2(1 + v)dx = 2dx + 2vdx$, respectively.

We have put a dot in front of the statements added to the midpoint algorithm of Section 3.2.2 to create the revised midpoint algorithm shown in Fig. 3.62. The WritePixel of E or NE has been replaced by a call to IntensifyPixel for the chosen pixel and its vertical neighbors; IntensifyPixel does the table lookup that converts the absolute value of the distance to weighted area overlap, a fraction of maximum intensity. In an actual implementation, this simple code, of course, would be inline.

The Gupta–Sproull algorithm provides an efficient incremental method for antialiasing lines, although fractional arithmetic is used in this version. The extension to antialiasing endpoints by using a separate look-up table is covered in Section 19.3.5. Since the lookup works for the intersection with the edge of the line, it can also be used for the edge of an arbitrary polygon. The disadvantage is that a single look-up table applies to lines of a given thickness only. Section 19.3.1 discusses more general techniques that consider any line as two parallel edges an arbitrary distance apart. We also can antialias characters either by filtering them (see Section 19.4) or, more crudely, by taking their scanned-in bitmaps and manually softening pixels at edges.

3.18 SUMMARY

In this chapter, we have taken our first look at the fundamental clipping and scan-conversion algorithms that are the meat and potatoes of raster graphics packages. We have covered only the basics here; many elaborations and special cases must be considered for robust implementations. Chapter 19 discusses some of these, as well as such topics as general regions and region filling. Other algorithms that operate on bitmaps or pixmaps are discussed in Chapter 17, and a fuller treatment of the theory and practice of antialiasing is found in Chapters 14 and 19.

```
static void IntensifyPixel (int, int, double)

void AntiAliasedLineMidpoint (int x0, int y0, int x1, int y1)
/* This algorithm uses Gupta-Sproull's table of intensity as a function of area */
/* coverage for a circular support in the IntensifyPixel function. Note that */
/* overflow may occur in the computation of the denominator for 16-bit integers, */
/* because of the squares. */
{
    int dx = x1 - x0;
    int dy = y1 - y0;
    int d = 2 * dy - dx;                          /* Initial value d_start as before */
    int incrE = 2 * dy;                           /* Increment used for move to E */
    int incrNE = 2 * (dy - dx);                   /* Increment used for move to NE */
•   int two_v_dx = 0;                             /* Numerator; v = 0 for start pixel */
•   double invDenom = 1.0 /
        (2.0 * sqrt (dx * dx + dy * dy));         /* Precomputed inverse denominator */
•   double two_dx_invDenom =                      /* Precomputed constant */
        2.0 * dx * invDenom;
    int x = x0;
    int y = y0;
•   IntensifyPixel (x, y, 0);                     /* Start pixel */
•   IntensifyPixel (x, y + 1, two_dx_invDenom);   /* Neighbor */
•   IntensifyPixel (x, y - 1, two_dx_invDenom);   /* Neighbor */
    while (x < x1) {
        if (d < 0) {                              /* Choose E */
•           two_v_dx = d + dx;
            d += incrE;
            x++;
        } else {                                  /* Choose NE */
•           two_v_dx = d - dx;
            d += incrNE;
            x++;
            y++;
        }
        /* Now set chosen pixel and its neighbors */
•       IntensifyPixel (x, y, two_v_dx * invDenom);
•       IntensifyPixel (x, y + 1, two_v_dx_invDenom - two_v_dx * invDenom);
•       IntensifyPixel (x, y - 1, two_v_dx_invDenom + two_v_dx * invDenom);
    }
} /* AntiAliasedLineMidpoint */

void IntensifyPixel (int x, int y, double distance)
{
    double intensity = Filter (Round (fabs (distance)));
    /* Table lookup done on an integer index; thickness 1 */
    WritePixel (x, y, intensity);
} /* IntensifyPixel */
```

Fig. 3.62 Gupta–Sproull algorithm for antialiased scan conversion of lines.

The most important idea of this chapter is that, since speed is essential in interactive raster graphics, incremental scan-conversion algorithms using only integer operations in their inner loops are usually the best. The basic algorithms can be extended to handle thickness, as well as patterns for boundaries or for filling areas. Whereas the basic algorithms that convert single-pixel-wide primitives try to minimize the error between chosen pixels on the Cartesian grid and the ideal primitive defined on the plane, the algorithms for thick primitives can trade off quality and "correctness" for speed. Although much of 2D raster graphics today still operates, even on color displays, with single-bit-per-pixel primitives, we expect that techniques for real-time antialiasing will soon become prevalent.

EXERCISES

3.1 Implement the special-case code for scan converting horizontal and vertical lines, and lines with slopes of ± 1.

3.2 Modify the midpoint algorithm for scan converting lines (Fig. 3.8) to handle lines at any angle.

3.3 Show why the point-to-line error is always $\leq \frac{1}{2}$ for the midpoint line scan-conversion algorithm.

3.4 Modify the midpoint algorithm for scan converting lines of Exercise 3.2 to handle endpoint order and intersections with clip edges, as discussed in Section 3.2.3.

3.5 Modify the midpoint algorithm for scan converting lines (Exercise 3.2) to write pixels with varying intensity as a function of line slope.

3.6 Modify the midpoint algorithm for scan converting lines (Exercise 3.2) to deal with endpoints that do not have integer coordinates—this is easiest if you use floating point throughout your algorithm. As a more difficult exercise, handle lines of *rational* endpoints using only integers.

3.7 Determine whether the midpoint algorithm for scan converting lines (Exercise 3.2) can take advantage of symmetry by using the decision variable d to draw simultaneously from both ends of the line toward the center. Does your algorithm consistently accommodate the case of equal error on an arbitrary choice that arises when dx and dy have a largest common factor c and dx/c is even and dy/c is odd ($0 < dy < dx$), as in the line between (0, 0) and (24, 9)? Does it deal with the subset case in which dx is an integer multiple of $2dy$, such as for the line between (0, 0) and (16, 4)? (Contributed by J. Bresenham.)

3.8 Show how polylines may share more than vertex pixels. Develop an algorithm that avoids writing pixels twice. Hint: Consider scan conversion and writing to the canvas in **xor** mode as separate phases.

3.9 Expand the pseudocode for midpoint ellipse scan conversion of Fig. 3.21 to code that tests properly for various conditions that may arise.

3.10 Apply the technique of forward differencing shown for circles in Section 3.3.2 to develop the second-order forward differences for scan converting standard ellipses. Write the code that implements this technique.

3.11 Develop an alternative to the midpoint circle scan-conversion algorithm of Section 3.3.2 based on a piecewise-linear approximation of the circle with a polyline.

3.12 Develop an algorithm for scan converting unfilled rounded rectangles with a specified radius for the quarter-circle corners.

3.13 Write a scan-conversion procedure for solidly filled upright rectangles at arbitrary screen positions that writes a bilevel frame buffer efficiently, an entire word of pixels at a time.

3.14 Construct examples of pixels that are "missing" or written multiple times, using the rules of Section 3.6. Try to develop alternative, possibly more complex, rules that do not draw shared pixels on shared edges twice, yet do not cause pixels to be missing. Are these rules worth the added overhead?

3.15 Implement the pseudocode of Section 3.6 for polygon scan conversion, taking into account in the span bookkeeping of potential sliver polygons.

3.16 Develop scan-conversion algorithms for triangles and trapezoids that take advantage of the simple nature of these shapes. Such algorithms are common in hardware.

3.17 Investigate triangulation algorithms for decomposing an arbitrary, possibly concave or self-intersecting, polygon into a mesh of triangles whose vertices are shared. Does it help to restrict the polygon to being, at worse, concave without self-intersections or interior holes? (See also [PREP85].)

3.18 Extend the midpoint algorithm for scan converting circles (Fig. 3.16) to handle filled circles and circular wedges (for pie charts), using span tables.

3.19 Extend the midpoint algorithm for scan converting ellipses (Fig. 3.21) to handle filled elliptical wedges, using span tables.

3.20 Implement both absolute and relative anchor algorithms for polygon pattern filling, discussed in Section 3.9, and contrast them in terms of visual effect and computational efficiency.

3.21 Apply the technique of Fig. 3.30 for writing characters filled with patterns in opaque mode. Show how having a copyPixel with a write mask may be used to good advantage for this class of problems.

3.22 Implement a technique for drawing various symbols such as cursor icons represented by small bitmaps so that they can be seen regardless of the background on which they are written. Hint: Define a mask for each symbol that "encloses" the symbol—that is, that covers more pixels than the symbol—and that draws masks and symbols in separate passes.

3.23 Implement thick-line algorithms using the techniques listed in Section 3.9. Contrast their efficiency and the quality of the results they produced.

3.24 Extend the midpoint algorithm for scan converting circles (Fig. 3.16) to handle thick circles.

3.25 Implement a thick-line algorithm that accommodates line style as well as pen style and pattern.

3.26 Implement scissoring as part of scan converting lines and unfilled polygons, using the fast-scan-plus-backtracking technique of checking every ith pixel. Apply the technique to filled and thick lines and to filled polygons. For these primitives, contrast the efficiency of this type of on-the-fly clipping with that of analytical clipping.

3.27 Implement scissoring as part of scan converting unfilled and filled circles and ellipses. For these primitives, contrast the feasibility and efficiency of this type of on-the-fly clipping with that of analytical clipping.

3.28 Modify the Cohen–Sutherland line-clipping algorithm of Fig. 3.41 to avoid recalculation of slopes during successive passes.

3.29 Contrast the efficiency of the Sutherland–Cohen and Cyrus–Beck algorithms for several typical and atypical cases, using instruction counting. Are horizontal and vertical lines handled optimally?

3.30 Consider a convex polygon with n vertices being clipped against a clip rectangle. What is the maximum number of vertices in the resulting clipped polygon? What is the minimum number? Consider the same problem for a concave polygon. How many polygons might result? If a single polygon results, what is the largest number of vertices it might have?

3.31 Explain why the Sutherland–Hodgman polygon-clipping algorithm works for only convex clipping regions.

3.32 Devise a strategy for subdividing a pixel and counting the number of subpixels covered (at least to a significant degree) by a line, as part of a line-drawing algorithm using unweighted area sampling.

3.33 Create tables with various decreasing functions of the distance between pixel center and line center. Use them in the antialiased line algorithm of Fig. 3.62. Contrast the results produced with those produced by a box-filtered line.

3.34 Generalize the antialiasing techniques for lines to polygons. How might you handle nonpolygonal boundaries of curved primitives and characters?