

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

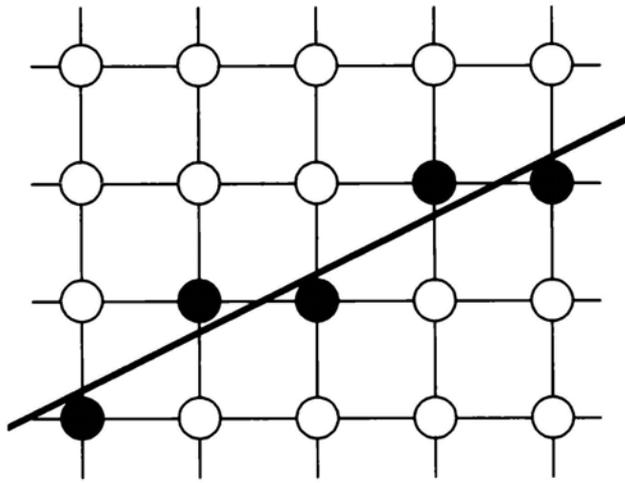


Fig. 3.4 A scan-converted line showing intensified pixels as black circles.

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

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

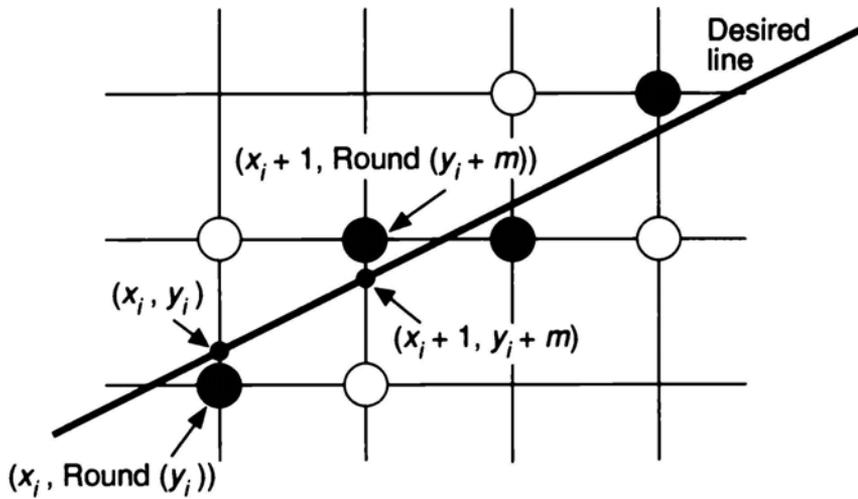


Fig. 3.5 Incremental calculation of (x_i, y_i) .

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

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

```

void Line (
    int x0, int y0,
    int x1, int y1,
    int value)
{
    int x;

    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);
        y += m;
    }
} /* Line */

```

/* Assumes $-1 \leq m \leq 1$, $x_0 < x_1$ */
/* Left endpoint */
/* Right endpoint */
/* Value to place in line's pixels */
/* x runs from x0 to x1 in unit increments. */
/* Set pixel to value */
/* Step y by slope m */

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

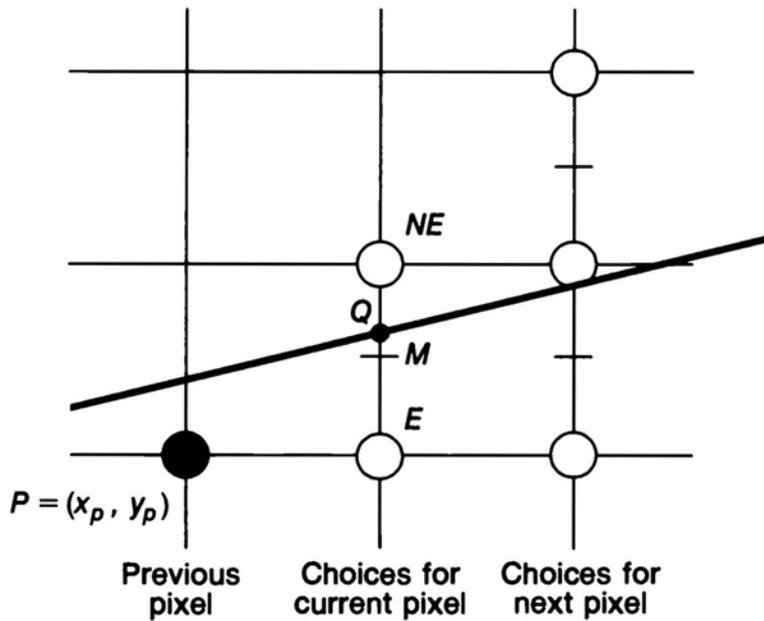


Fig. 3.7 The pixel grid for the midpoint line algorithm, showing the midpoint M , and the E and NE pixels to choose between.

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 function³ 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

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

⁴It 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,

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

but

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

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

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,

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

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

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

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

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

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.

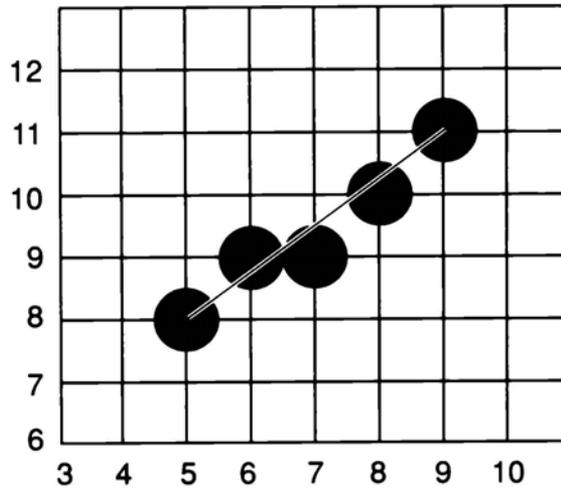


Fig. 3.9 The midpoint line from point (5, 8) to point (9, 11).

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

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

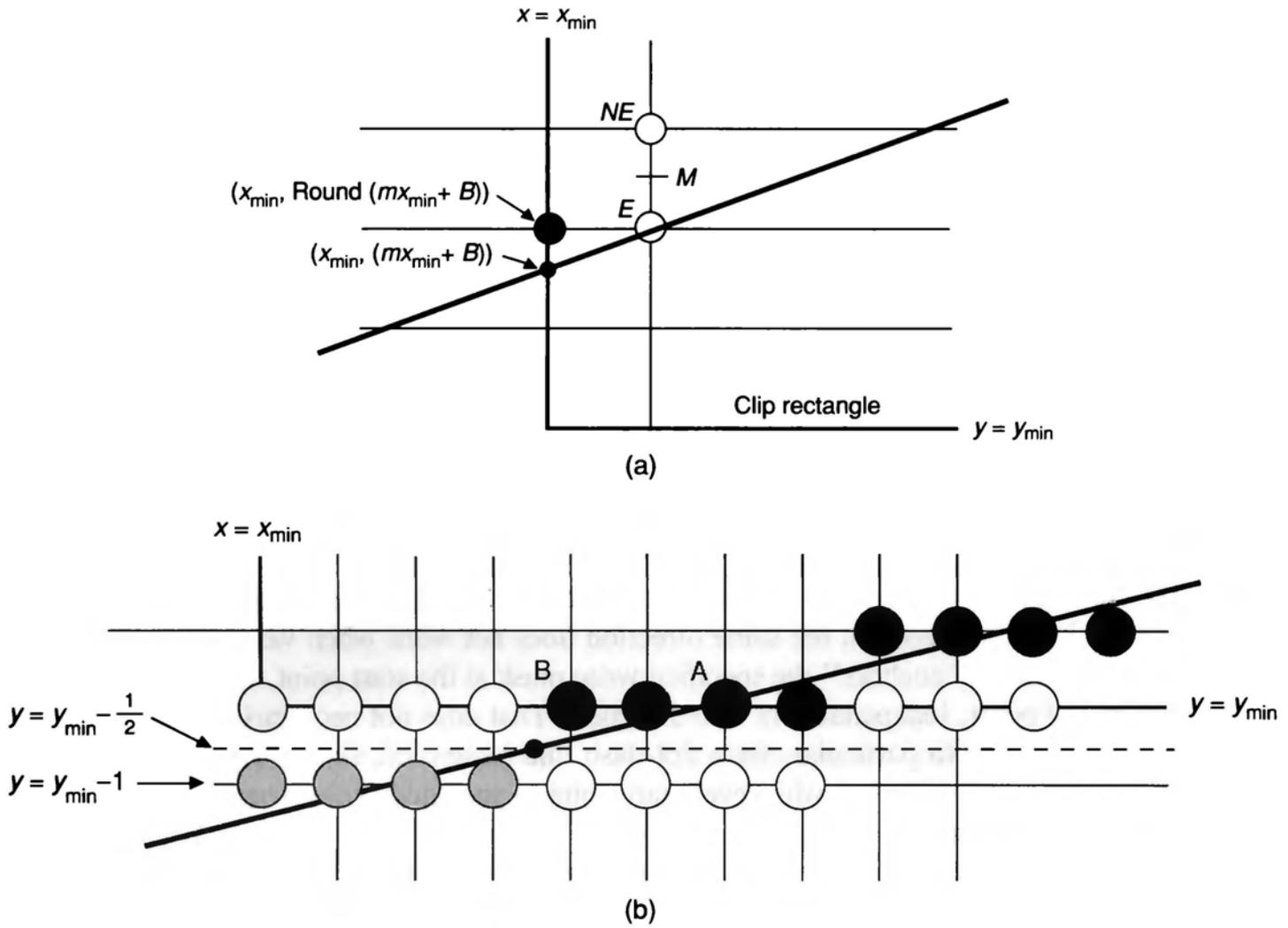


Fig. 3.10 Starting the line at a clip boundary. (a) Intersection with a vertical edge. (b) Intersection with a horizontal edge (gray pixels are on the line but are outside the clip rectangle).

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 $(Round(x_{y_{min}-\frac{1}{2}}), y_{min})$.

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

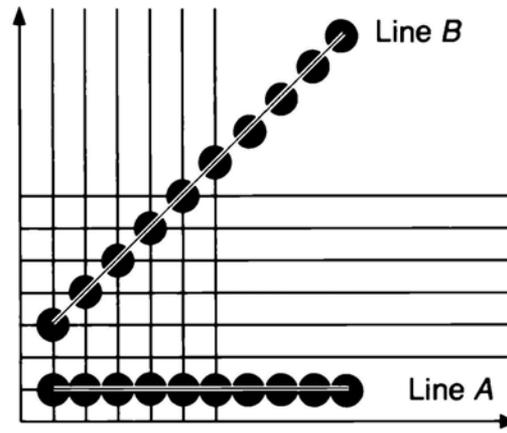


Fig. 3.11 Varying intensity of raster lines as a function of slope.

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.

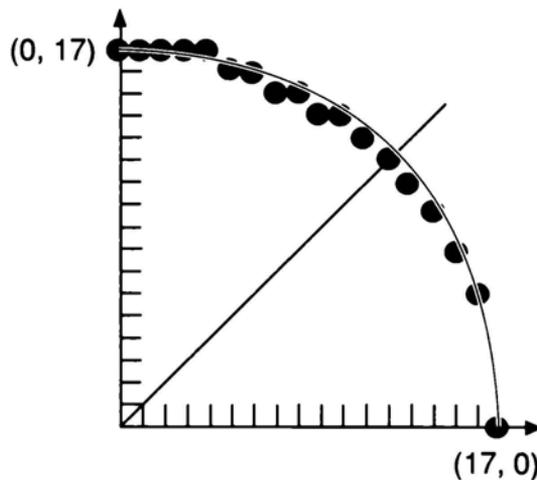


Fig. 3.12 A quarter circle generated with unit steps in x , and with y calculated and then rounded. Unique values of y for each x produce gaps.

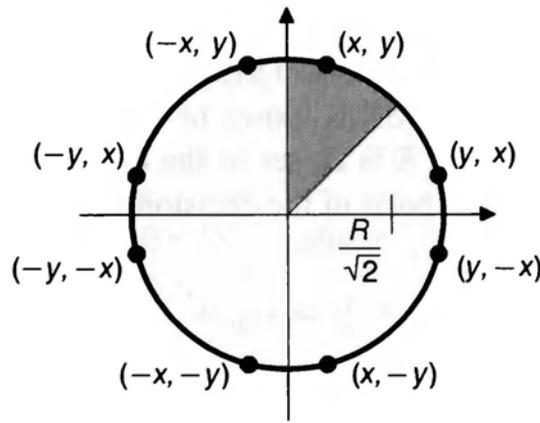


Fig. 3.13 Eight symmetrical points on a circle.

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).

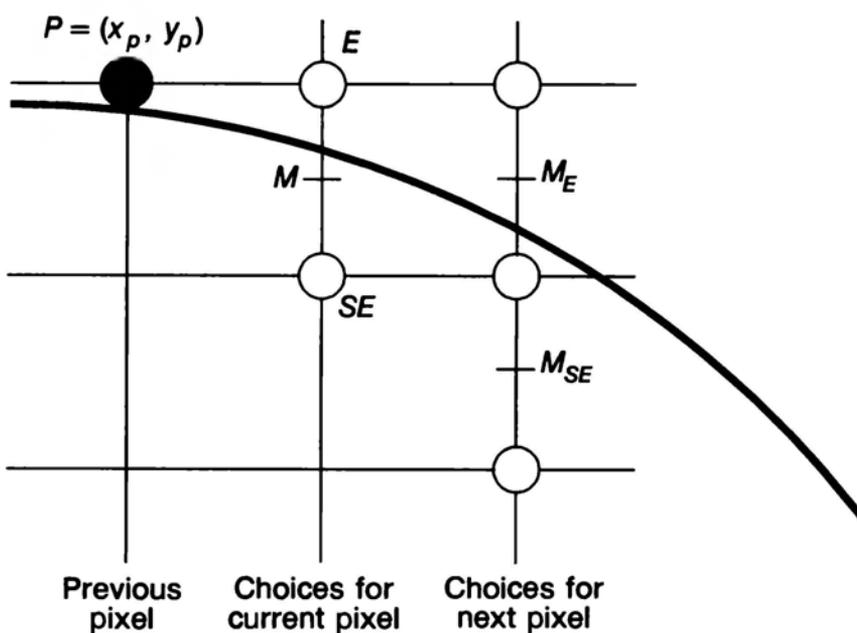


Fig. 3.14 The pixel grid for the midpoint circle algorithm showing M and the pixels E and SE to choose between.

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,

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

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

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

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

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

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 initialization is $h = 1 - R$, and the comparison $d < 0$ becomes $h < -\frac{1}{4}$.

⁶Choosing 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 Δ s 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.

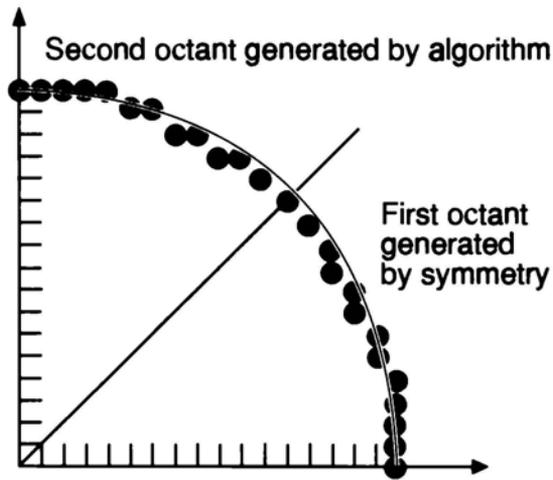


Fig. 3.17 Second octant of circle generated with midpoint algorithm, and first octant generated by symmetry.

```

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.