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

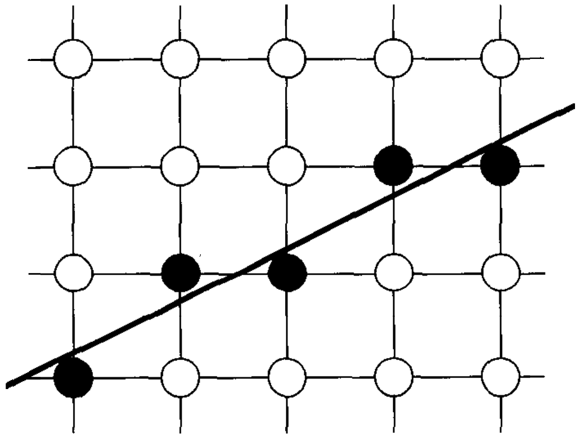

Fig. 3.4 A scan-converted line showing intensified pixels as black circles.

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*).

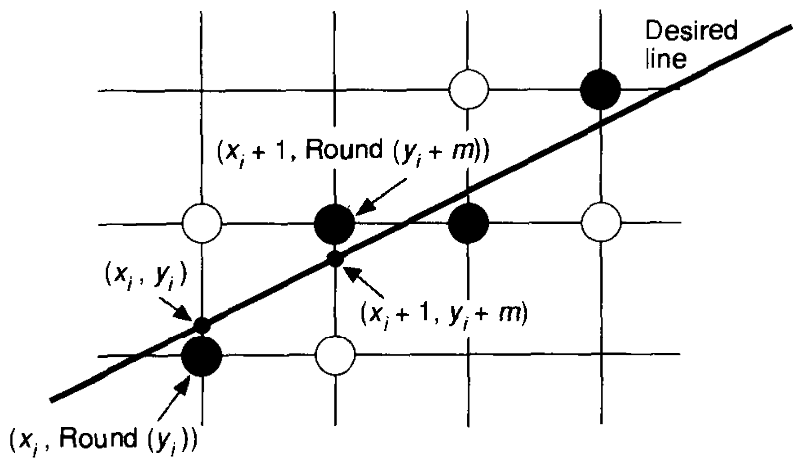

Fig. 3.5 Incremental calculation of (x_i, y_i).

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 $-1 \le m \le 1$, 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 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.

```
procedure Line (                              {Assumes -1 ≤ m ≤ 1, x0 < x1}
    x0, y0,                                   {Left endpoint}
    x1, y1,                                   {Right endpoint}
    value : integer);                         {Value to place in line's pixels}
var
    x : integer;                              {x runs from x0 to x1 in unit increments.}
    dy, dx, y, m : real;
begin
    dy := y1 - y0;
    dx := x1 - x0;
    m := dy/dx;
    y := y0;
    for x := x0 to x1 do
      begin
        WritePixel (x, Round (y), value);     {Set pixel to value}
        y := y + m                            {Step y by slope m}
      end
end; {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

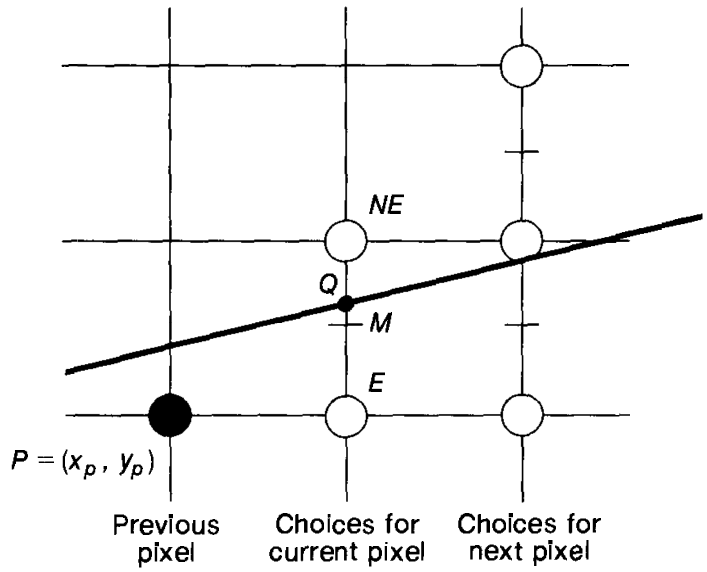

Fig. 3.7 The pixel grid for the midpoint line algorithm, showing the midpoint *M*, and the *E* and *NE* pixels to choose between.

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,

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

but

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

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

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,

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

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

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

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 $(x_0 + 1, y_0 + \tfrac{1}{2})$, 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.

```
procedure MidpointLine (x0, y0, x1, y1, value: integer);
var
    dx, dy, incrE, incrNE, d, x, y : integer;
begin
    dx := x1 - x0;
    dy := y1 - y0;
    d := 2 * dy - dx;              {Initial value of d}
    incrE := 2 * dy;               {Increment used for move to E}
    incrNE := 2 * (dy - dx);       {Increment used for move to NE}
    x := x0;
    y := y0;
    WritePixel (x, y, value);      {The start pixel}
    while x < x1 do
      begin
        if d <= 0 then
          begin                    {Choose E}
            d := d + incrE;
            x := x + 1
          end
        else
          begin                    {Choose NE}
            d := d + incrNE;
            x := x + 1;
            y := y + 1
          end;
        WritePixel (x, y, value)   {The selected pixel closest to the line}
      end {while}
end;{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.