Raster Algorithms and Software

11.1 INTRODUCTION

We now turn our attention away from classical vector graphics and toward the new and rapidly expanding area of raster graphics. The growth of raster graphics has been driven by the microelectronics revolution, which allows processors and large amounts of random-access memory to be manufactured on small silicon chips. The processor and a few memory chips are used for the image creation system, which scan-converts output primitives such as lines, characters, polygons, etc. Many more memory chips are used for the refresh buffer, from which the image is displayed, one scan line at a time. Chapter 1 gave some background on the growth of raster graphics, while Chapter 3 outlined the basic structure of a typical raster display. In the first part of this chapter we present some of the many scan-conversion algorithms, and in the second part we discuss capabilities which are useful in a raster graphics subroutine package. The next chapter is a detailed discussion of raster graphics hardware.

The scan-conversion algorithms used in a raster display will be invoked quite often—typically hundreds or even thousands of times each time an image is created or modified. Hence, they must not only create visually satisfactory images, but must also execute as rapidly as possible. Indeed, speed versus image quality is the basic trade-off in selecting scan-conversion algorithms: some are fast and give jagged edges, while others are slower but give smoother edges. However, whichever way the trade-off is resolved, faster is better. Thus, the algorithms use incremental methods which minimize the number of calculations (especially multiplies and divides) performed during each iteration. Speed can be increased even further by using multiple processors, all simultaneously scan-converting output primitives into a multiported refresh buffer.

11.2 SCAN-CONVERTING LINES

The basic task of a scan-conversion algorithm for lines is to compute the coordinates of the pixels which lie near the line on a two-dimensional raster grid. In discussing this task we assume that the starting and ending points for the line have integer coordinates (the generalization is left as an exercise). The basic strategy used by the line scan-conversion algorithm in Chapter 3 is to increment x, calculate $y = mx + b$, and intensify the pixel at $(x, ROUND(y))$. This calculation of m times x takes time, however, and slows the scan-conversion process. Furthermore, floating-point (or binary-fraction) data representation must be used to ensure sufficient accuracy.

11.2.1 The Basic Incremental Algorithm

We can eliminate the multiplication by noting that if $\Delta x = 1$, then $m = \Delta y/\Delta x$ reduces to $m = \Delta y$, that is, a unit change in x changes y by m, which is the slope of the line. Thus 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 next values of x and y are defined in terms of their previous values, as shown in Fig. 11.1. If $m > 1$, then a step in x will create 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 $\Delta x = \Delta y/m = 1/m$. This is what is meant by an *incremental algorithm:* at each step we make incremental calculations based on the preceding step. The procedure *LINE* to implement the technique, limited to the case of $-1 < m < 1$, appears below. The procedure *WRITE_PIXEL*, used by *LINE*, places a value into the refresh buffer pixel whose coordinates are given as the first two arguments.

```
procedure LINE(                          {assumes slope between +1 and -1}
  x1, y1,                                {start point}
  x2, y2,                                {end point}
  x
  value : integer);                      {value to place in pixels near line}
  var dy, dx, y, m: real;
begin
  if x1 <> x2
    then begin
      dy := y2 - y1;
      dx := x2 - x1;
      m := dy/dx;
      y := y1;
      for x := x1 to x2 do
        begin
          WRITE_PIXEL(x, ROUND(y), value);     {sets pixel to value}
          y := y + m                           {step y by slope m}
        end
    end
    {if "line" really a point, plot it; else, error}
    else if y1 = y2 then WRITE_PIXEL(x1, y1)
                    else ERROR
end    {LINE}
```

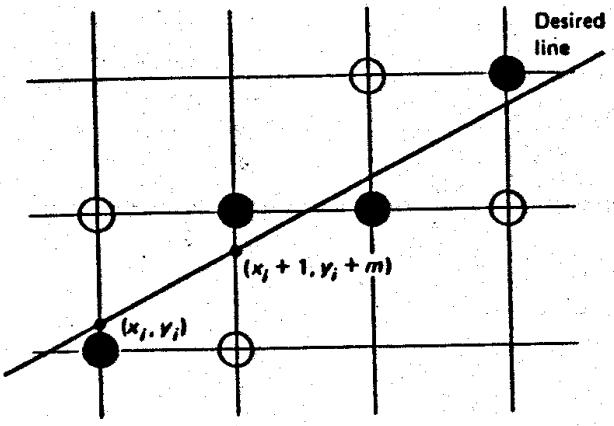

Fig. 11.1 Incremental calculation of y by rounding y to select pixel (designated by black circle).

11.2.2 Bresenham's Line Algorithm

The difficulties with *LINE* are that rounding y to an integer takes time, and the variables y and m must be real or fractional binary rather than integer, because the slope is a fraction. *Bresenham's algorithm* [BRES65] is attractive because it uses only integer arithmetic. No real variables are used, and hence rounding is not needed. We assume, for simplicity, that the slope of the line is between 0 and 1. The algorithm uses a decision variable d_i which at each step is proportional to the difference between s and t shown in Fig. 11.2. The figure depicts the ith step, at which the pixel p_{i-1} has been determined to be closest to the actual line being drawn, and we now want to decide whether the next pixel to be set should be T_i or S_i. If $s < t$, then S_i is closer to the desired line and should be set; else T_i is closer and should be set. Said differently, we choose S_i if $s - t < 0$, otherwise we choose T_i.

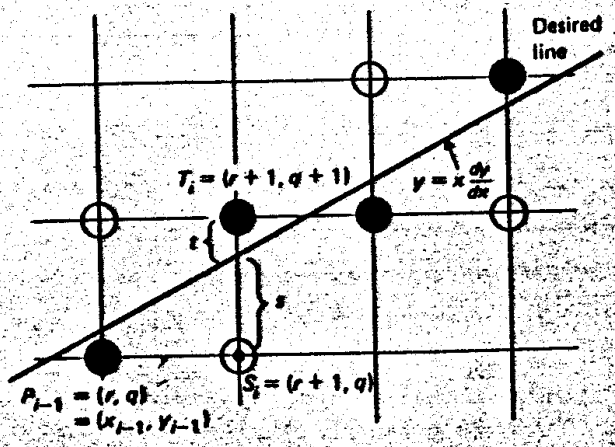

Fig. 11.2 Geometry for Bresenham's algorithm. Black circles are pixels selected by Bresenham's algorithm.

The line being drawn is from $(x1, y1)$ to $(x2, y2)$. Assuming that the first point is nearer the origin, we translate both points by $T(-x1, -y1)$, so it becomes the line from (0, 0) to (dx, dy), where $dx = x2 - x1$ and $dy = y2 - y1$. The equation of the line is now $y = (dy/dx)x$. Referring to Fig. 11.2, we represent the coordinates (after the translation) of P_{i-1} as (r, q). Then $S_i = (r + 1, q)$ and $T_i = (r + 1, q + 1)$.

From the examination of Fig. 11.2 we can write

$$s = \frac{dy}{dx}(r + 1) - q, \qquad t = q + 1 - \frac{dy}{dx}(r + 1).$$

Therefore

$$s - t = 2\frac{dy}{dx}(r + 1) - 2q - 1. \tag{11.1}$$

When $s - t < 0$, we choose S_i. Manipulating (11.1), we have

$$dx(s - t) = 2(r \cdot dy - q \cdot dx) + 2dy - dx.$$

Now dx is positive, so we can use $dx(s - t) < 0$ as the test for choosing S_i. We define this as d_i; then

$$d_i = 2(r \cdot dy - q \cdot dx) + 2dy - dx.$$

With $r = x_{i-1}$ and $q = y_{i-1}$, this is

$$d_i = 2x_{i-1}dy - 2y_{i-1}dx + 2dy - dx. \tag{11.2}$$

Adding 1 to each index gives:

$$d_{i+1} = 2x_i \cdot dy - 2y_i \cdot dx + 2dy - dx.$$

Subtracting d_i from d_{i+1}, we get

$$d_{i+1} - d_i = 2dy(x_i - x_{i-1}) - 2dx(y_i - y_{i-1}).$$

We know that $x_i - x_{i-1} = 1$. Rewriting this, we get

$$d_{i+1} = d_i + 2dy - 2dx(y_i - y_{i-1}).$$

If $d_i \geq 0$, then T_i is selected, so $y_i = y_{i-1} + 1$ and

$$d_{i+1} = d_i + 2(dy - dx). \tag{11.3}$$

If $d_i < 0$, then S_i is selected, so $y_i = y_{i-1}$ and

$$d_{i+1} = d_i + 2dy. \tag{11.4}$$

Hence we have an iterative way to calculate d_{i+1} from the previous d_i and to make the selection between S_i and T_i. The initial starting value d_1 is found by evaluating (11.2) for $i = 1$, knowing that $(x_0, y_0) = (0, 0)$. Then

$$d_1 = 2dy - dx. \tag{11.5}$$

The arithmetic needed to evaluate (11.3), (11.4), and (11.5) is minimal: it involves addition, subtraction and left shift (to multiply by 2). This is important, because time-consuming multiplication is avoided. Further, the actual inner loop is quite simple, as seen in the following Bresenham's algorithm (note that this version works only for lines with slope between 0 and 1; generalizing the algorithm is left as an exercise for the reader):

```
procedure BRESENHAM(x1, y1, x2, y2, value: integer);
  var dx, dy, incr1, incr2, d, x, y, xend: integer;
begin
  dx := ABS(x2 - x1);
  dy := ABS(y2 - y1);
  d := 2 * dy - dx;                    {initial value for d from (11.5)}
  incr1 := 2 * dy;                     {constant used for increment if d < 0}
  incr2 := 2 * (dy - dx);              {constant used for increment if d ≥ 0}
  if x1 > x2
    then begin                         {start at point with smaller x}
      x := x2;
      y := y2;
      xend := x1
      end
    else begin
      x := x1;
      y := y1;
      xend := x2
      end
  WRITE_PIXEL(x, y, value);            {first point on line}
  while x < xend do begin
    x := x + 1;
    if d < 0
      then d := d + incr1              {choose S_i—no change in y}
      else begin                       {choose T_i—y is incremented}
        y := y + 1;
        d := d + incr2
        end
      WRITE_PIXEL(x, y, value)         {the selected point near the line}
  end    {while}
end    {BRESENHAM}
```

For a line from point (5, 8) to point (9, 11), the successive values of d are 2, 0, -2, 4, and 2. Figure 11.3 shows which pixels are set and the ideal path of the line.

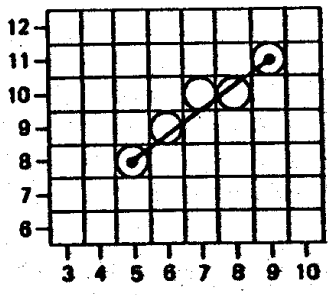

Fig. 11.3 Line from point (5, 8) to point (9, 11) drawn with Bresenham's algorithm.

The line appears jagged, in part because of the enlarged scale of the drawing and in part due to the approximations involved in attempting to draw a line on a discrete grid of points.

11.4 SCAN-CONVERTING CIRCLES

There are several very easy but inefficient ways to scan-convert a circle. Consider, for simplicity, the circle centered at the origin, for which

$$x^2 + y^2 = R^2.$$

Solving for y, we get

$$y = \pm\sqrt{R^2 - x^2}. \tag{11.10}$$

To draw a quarter circle, we can increment x from 0 to R in unit steps, solving for $+y$ at each step (the other quarters are drawn by symmetry). This works, but is inefficient because of the multiply and square-root operations. Furthermore, there will be large gaps in the circle for values of x close to R because the slope of the circle becomes infinite as x approaches R (see Fig. 11.10). A similar inefficient method, which does avoid the large gaps, is to plot $R\cos\theta$ or $R\sin\theta$ by stepping θ from 0 to 90°.

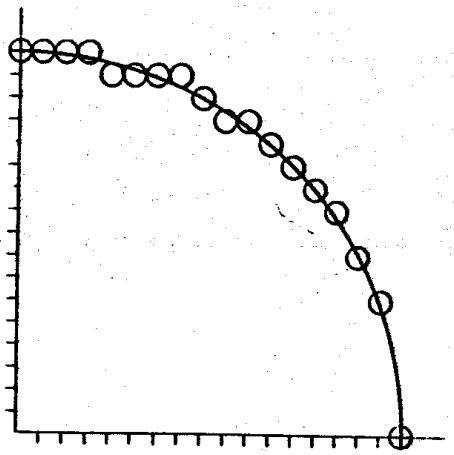

Fig. 11.10 A quarter-circle generated with unit steps in x and with y calculated and then rounded.

11.4.1 Eight-Way Symmetry

This process can be improved somewhat by taking greater advantage of the symmetry in a circle. Consider first a circle 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. 11.11. Therefore if we use Eq. (11.10) or some other more efficient mechanism to compute y for values of x between 0 and $R/\sqrt{2}$ (the point at which $x = y$), seven additional points on the circle are also available; this range of x corresponds to the 45° segment of the circle in the figure. For a circle centered at the origin, the points can be displayed with procedure *CIRCLE_POINTS* (the procedure is easily generalized to the case of circles with arbitrary origins):

```
procedure CIRCLE_POINTS(x, y, value: integer);
begin
  WRITE_PIXEL(x, y, value);
  WRITE_PIXEL(y, x, value);
  WRITE_PIXEL(y, -x, value);
  WRITE_PIXEL(x, -y, value);
  WRITE_PIXEL(-x, -y, value);
  WRITE_PIXEL(-y, -x, value);
  WRITE_PIXEL(-y, x, value);
  WRITE_PIXEL(-x, y, value)
end    {CIRCLE_POINTS}
```

11.4.2 Bresenham's Circle Algorithm

Bresenham [BRES77] has developed an incremental circle generator which is more efficient than either of the above methods. Conceived for use with pen plotters, the algorithm generates all points on a circle centered at the origin by incrementing

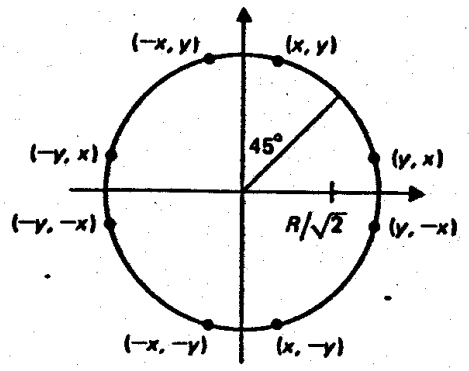

Fig. 11.11 Eight symmetrical points on a circle.

360° around the circle. We present an adaptation of the algorithm which increments through only 45° of a circle, from $x = 0$ to $x = R/\sqrt{2}$, and uses the *CIRCLE_POINTS* procedure to display points on the entire circle.

At each step, the algorithm selects the point $P_i(x_i, y_i)$ which is closest to the true circle and which therefore makes the error term

$$D(P_i) = (x_i^2 + y_i^2) - R^2$$

closest to zero; that is, $|D(P_i)|$ is minimized at each step. As with Bresenham's line-drawing algorithm, the fundamental strategy is to select the nearest point by using decision variables whose values can be incrementally calculated with only a few adds, subtracts, and shifts. The signs of the variables are used to make the decisions.

What decisions are to be made? Consider Fig. 11.12 which shows a small part of the pixel grid and the various possible ways (A to G) that the true circle might cut through the grid.* Assume that the point P_{i-1} has been determined to be the closest to the circle for $x = x_{i-1}$. Now for $x = x_{i-1} + 1$ we must determine whether T_i or S_i is closer to the circle.

Let us define

$$D(S_i) = [(x_{i-1} + 1)^2 + (y_{i-1})^2] - R^2, \tag{11.11}$$

$$D(T_i) = [(x_{i-1} + 1)^2 + (y_{i-1} - 1)^2] - R^2. \tag{11.12}$$

These are the differences between the squared distances from the origin (the center of the circle) to S_i (or to T_i) and to the actual circle. If $|D(S_i)| \geq |D(T_i)|$, then T_i is closer (or equidistant) to the actual circle than is S_i. Conversely, if $|D(S_i)| < |D(T_i)|$, then S_i is closer to the actual circle than is T_i.

*Bresenham's original circle algorithm was not limited to the 45° segment being examined here and therefore considers cases F and G, for which the point directly below P_{i-1} might be selected.

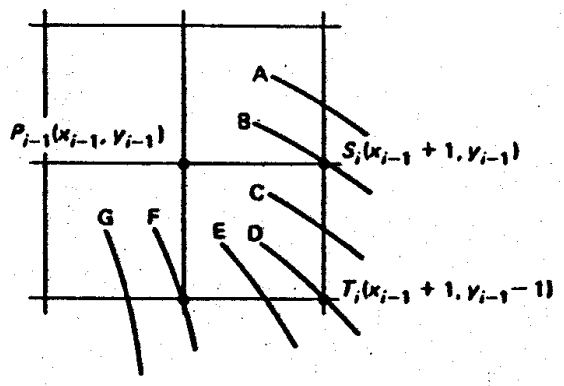

Fig. 11.12 Decision points for Bresenham's circle generator.

Now if we define

$$d_i = |D(S_i)| - |D(T_i)|,$$

then the point T_i is selected when $d_i \geq 0$; otherwise the point S_i is selected.

In case C, we have $D(S_i) > 0$, because S_i lies outside the circle, and $D(T_i) < 0$, because T_i lies inside the circle, so

$$d_i = D(S_i) + D(T_i). \tag{11.13}$$

Now if $d_i \geq 0$, then T_i is selected; otherwise, $d_i < 0$, so S_i is selected.

Consider now cases A and B and the corresponding value of d_i from (11.13). It is clear that $D(T_i) < 0$, because T_i is inside of the true circle. Similarly, $D(S_i) \leq 0$ (the equality occurs in case B; the inequality in case A). Therefore, $d_i < 0$ for cases A and B. The same selection rules applied to the preceding discussion of case C will therefore properly lead to the choice of S_i, using (11.13).

Finally, consider cases D and E. First, $D(S_i) > 0$, because S_i is outside the true circle. Similarly, $D(T_i) \geq 0$ (the equality is for case D; the inequality for case E). Therefore, $d_i \geq 0$ for cases D and E, so the decision rules developed above for case C apply here also: if $d_i \geq 0$, choose T_i.

We are not finished yet: calculating the decision variable (11.13) as it is currently expressed requires several multiplications. However, a series of algebraic manipulations shows that

$$d_1 = 3 - 2R.$$

If S_i is chosen (because $d_i < 0$), then

$$d_{i+1} = d_i + 4x_{i-1} + 6;$$

if T_i is chosen (because $d_i \geq 0$), then

$$d_{i+1} = d_i + 4(x_{i-1} - y_{i-1}) + 10.$$

These specific algebraic results and consequent algorithm were derived by J. Michener by applying Bresenham's methodology. Expression (11.13) for d_i is expanded by using (11.11) and (11.12). By substituting $i - 1$ for i, an expression for d_{i-1} is found; then the difference $d_i - d_{i-1}$ is formed and evaluated for each of the two possible moves. The following procedure is based on these results:*

```
procedure MICH_CIRCLE(radius, value: integer);
  {assumes center of circle is at origin}
  var x, y, d: integer;
begin
  x := 0;
  y := radius;
  d := 3 - 2 * radius;
  while x < y do begin
    CIRCLE_POINTS (x, y, value);
    if d < 0
      then d := d + 4 * x + 6              {select S}
      else begin                           {select T— decrement y}
        d := d + 4 * (x - y) + 10;
        y: = y - 1
      end
    x := x + 1
  end     {while}
  if x = y then CIRCLE_POINTS (x,y,value);
end     {MICH_CIRCLE}
```

If *CIRCLE_POINTS* is called when either $x = y$ or $r = 1$, then each of four pixels is set twice. On a raster display, this is no problem. If the algorithm were used with a film recorder, however, double exposure of these points to the light source would cause their intensity to be greater than that of the other points. Figure 11.13 shows one octant of a circle of radius 17 generated with the algorithm (compare the results to Fig. 11.10).

Other techniques have been developed for drawing circles [BADL77, DORO79, HORN76, SUEN79] and for more general curves than circles. Jordan, Lennon, and Holm [JORD73b] developed a general and efficient method for most curves which can be expressed as $f(x, y) = 0$ and have continuous derivatives. The method was later shown to have a few limitations [BELS76, RAMO76]. Special cases of such curves include conic sections (particularly the circle) and straight lines. While the algorithm can be simplified in these special cases, there is still slightly more work per iteration involved than for Bresenham's line and circle algorithms. This is important because of the stringent speed requirements for our scan-conversion algorithms. On the other hand, the algorithm of Pitteway [PITT67], while more complicated to derive, requires even less work per iteration than does Bresenham's general formulation for 360° circles, and equivalent work for the 45° formulation given here.

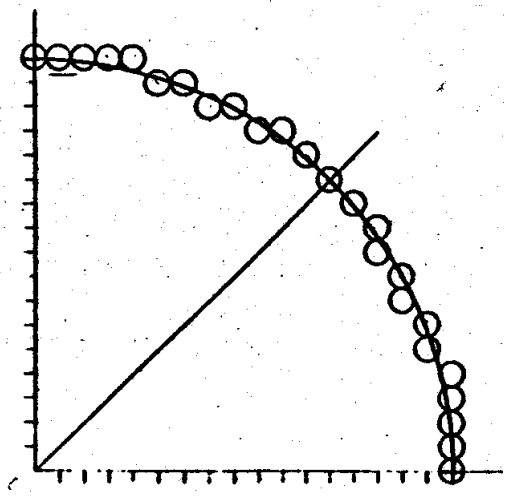

Fig. 11.13 Octant of circle generated with Bresenham's algorithm; second octant generated by symmetry.