DDA Algorithm

The Cartesian ***slope-intercept equation*** for a straight line is

$$y = m \cdot x + b \qquad (3\text{-}1)$$

With m representing the slope of the line and b as they intercept. Given that the **two** endpoints of a h e segment are specified at positions ***(x1,*** **y1)** and ***(x2,*** y2), as shown in Fig. **3-3,** we can determine values for the slope rn and y intercept b with the following calculations:

$$m = \frac{y_2 - y_1}{x_2 - x_1} \qquad (3\text{-}2)$$

$$b = y_1 - m \cdot x_1 \qquad (3\text{-}3)$$

For any given ***x*** interval long a line, we can compute the corresponding y interval **from**

$$\Delta y = m \, \Delta x \qquad (3\text{-}4)$$

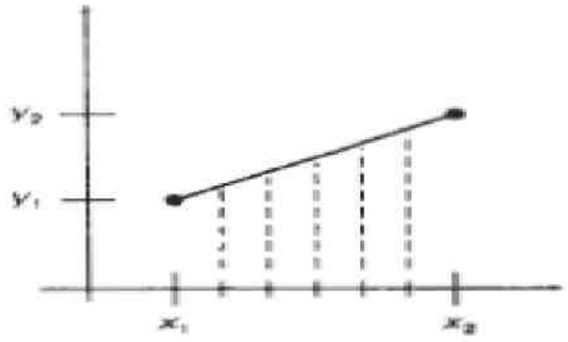
Similarly, we can obtain the x interval Δx corresponding to a specified Δy as

$$\Delta x = \frac{\Delta y}{m} \qquad (3\text{-}5)$$

For lines with slope magnitudes |m| < **1, can be** set proportional to a small horizontal deflection voltage and the corresponding vertical deflection is then set proportional to as calculated from **Eq.** 3-4.

For lines whose slopes have magnitudes |***m*** | > **1,** can **be** set proportional to a small vertical deflection voltage with the corresponding horizontal deflection voltage set proportional to , calculated from Eq. 3-5.

For lines with |*m/* = 1, = and the horizontal and vertical deflections voltages are equal. In each case, a smooth line with slope m is generated between the specified endpoints.



The *digital* ***deferential*** analyzer **(DDA)** is a scan-conversion line algorithm based on calculating either Ay or ***Ax,*** using Eq. **3-4** or Eq. **3-5.** We sample the line at unit intervals in one coordinate and determine corresponding integer values nearest the line path for the other coordinate. Consider first a line with positive slope, as shown in Fig. **3-3.** If the slope is
less than or equal to **1,** we sample at unit ***x*** intervals ***(*** = 1) and compute each successive **y** value as

$$y_{k+1} = y_k + m \qquad (3\text{-}6)$$

Subscript ***k*** takes integer values starting from 1, for the first point, and increases by 1 until the final endpoint is reached. Since ***m*** an **be** any real number between 0 and 1, the calculated y values must be rounded to the n e m t integer.

For lines with a positive slope greater than 1, we reverse the roles of x and . That is, we sample at unit y intervals (= 1) and calculate each succeeding x alue as

$$x_{k+1} = x_k + \frac{1}{m} \qquad (3\text{-}7)$$

Equations 3-6 and 3-7 are based on the assumption that lines are to be processed from the left endpoint to the right endpoint (Fig. 3-3). If this processing is reversed, so that the starting endpoint is at the right, then either we have $\Delta x = -1$ and

$$y_{k+1} = y_k - m \qquad (3\text{-}8)$$

or (when the slope is greater than 1) we have $\Delta y = -1$ with

$$x_{k+1} = x_k - \frac{1}{m} \qquad (3\text{-}9)$$

```
#include "device.h"

#define ROUND(a) ((int)(a+0.5))

void lineDDA (int xa, int ya, int xb, int yb)
{
  int dx = xb - xa, dy = yb - ya, steps, k;
  float xIncrement, yIncrement, x = xa, y = ya;

  if (abs (dx) > abs (dy)) steps = abs (dx);
  else steps = abs (dy);
  xIncrement = dx / (float) steps;
  yIncrement = dy / (float) steps;

  setPixel (ROUND(x), ROUND(y));
  for (k=0; k<steps; k++) {
    x += xIncrement;
    y += yIncrement;
    setPixel (ROUND(x), ROUND(y));
  }
}
```

Advantages of DDA

- DDA is the simplest line drawing algorithm
- The DDA algorithm is a faster method for calculating pixel positions than the direct use of Eq. 3-1. It eliminates the multiplication in Eq. 3-1 by making use of raster characteristics, so that appropriate increments are applied in the x or y direction to step to pixel positions along the line path.

Disadvantages of DDA

- Not very efficient
- Round operation is expensive
- Furthermore, the rounding operations and floating-point arithmetic in procedure **lineDDA** are still time-consuming.

Bresenham's Line Algorithm

An accurate and efficient raster line-generating algorithm, developed by Bresenham, scan converts lines using only incrementa1 integer calculations that can be adapted to display circles and other curves. Figures 3-5 and 3-6 illustrate sections of a display screen where straight line segments are to be drawn. The vertical axes show-scan-line positions, and the horizontal axes identify pixel columns. Sampling at unit *x* intervals in these examples, we need to decide which of two possible pixel positions is closer to the line path at each sample step. Starting from the left endpoint shown in Fig. **3-5,** we need to determine at the next sample position whether to plot the pixel at position (11, 11) or the one at (11, 12). Similarly, Fig. 3-6 shows-a negative slope-line path starting from the left endpoint at pixel position (50, 50). In this one, do we select the next pixel position as (51,501 or as (51,49)? These questions are answered with Bresenham's line algorithm by testing the sign of an integer parameter, whose value is proportional to the difference between the separations of the two pixel positions from the actual line path.

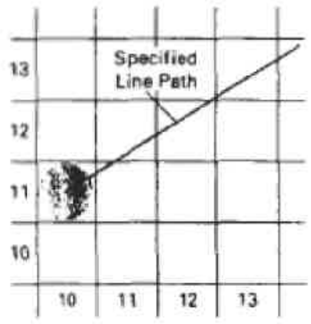


Figure 3-5
Section of a display screen where a straight line segment is to be plotted, starting from the pixel at column 10 on scan line 11.

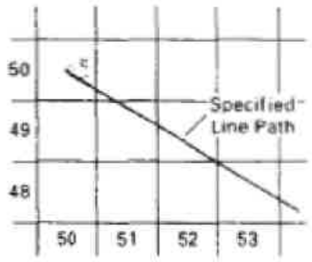


Figure 3-6
Section of a display screen where a negative slope line segment is to be plotted, starting from the pixel at column 50 on scan line 50.

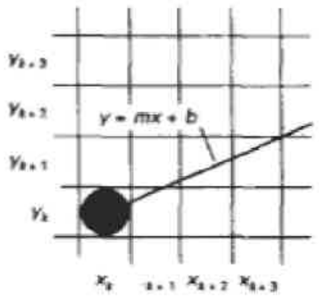


Figure 3-7
Section of the screen grid showing a pixel in column x_k on scan line y_k that is to be plotted along the path of a line segment with slope $0 < m < 1$.

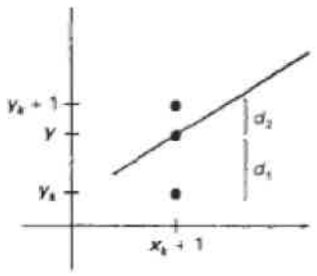


Figure 3-8
Distances between pixel positions and the line y coordinate at sampling position $x_k + 1$.

Starting from the left end point ($x0$,$y0$) of a given line , we step to each successive column (x position) and plot the pixel whose scan-line y value closest to the line path

Assuming we have determined that the pixel at (xk,yk) is to be displayed, we next need to decide which pixel to plot in column xk+1.

$$y = m(x_k + 1) + b$$

Then

$$\begin{aligned} d_1 &= y - y_k \\ &= m(x_k + 1) + b - y_k \end{aligned}$$

and

$$\begin{aligned} d_2 &= (y_k + 1) - y \\ &= y_k + 1 - m(x_k + 1) - b \end{aligned}$$

The difference between these two separations is

$$d_1 - d_2 = 2m(x_k + 1) - 2y_k + 2b - 1 \qquad (3\text{-}11)$$

A decision parameter p_k for the kth step in the line algorithm can be obtained by rearranging Eq. 3-11 so that it involves only integer calculations. We accomplish this by substituting $m = \Delta y/\Delta x$, where Δy and Δx are the vertical and horizontal separations of the endpoint positions, and defining:

$$\begin{aligned} p_k &= \Delta x(d_1 - d_2) \\ &= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c \qquad (3\text{-}12) \end{aligned}$$

The sign of p_k is the same as the sign of $d_1 - d_2$, since $\Delta x > 0$ for our example. Parameter c is constant and has the value $2\Delta y + \Delta x(2b - 1)$, which is independent of pixel position and will be eliminated in the recursive calculations for p_k. If the pixel at y_k is closer to the line path than the pixel at y_k+1 (that is, $d_1 < d_2$), then decision parameter p_k is negative. In that case, we plot the lower pixel; otherwise, we plot the upper pixel.

Coordinate changes along the line occur in unit steps in either the x or y directions. Therefore, we can obtain the values of successive decision parameters using incremental integer calculations. At step $k + 1$, the decision parameter is evaluated from Eq. 3-12 as

$$p_{k+1} = 2\Delta y \cdot x_{k+1} - 2\Delta x \cdot y_{k+1} + c$$

Subtracting Eq. 3-12 from the preceding equation, we have

$$p_{k+1} - p_k = 2\Delta y(x_{k+1} - x_k) - 2\,\Delta x(y_{k+1} - y_k)$$

But $x_{k+1} = x_k + 1$, so that

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k) \tag{3-13}$$

where the term $y_{k+1} - y_k$ is either 0 or 1, depending on the sign of parameter p_k.

This recursive calculation of decision parameters is performed at each integer x position, starting at the left coordinate endpoint of the line. The first parameter, p_0, is evaluated from Eq. 3-12 at the starting pixel position (x_0, y_0) and with m evaluated as $\Delta y/\Delta x$:

$$p_0 = 2\Delta y - \Delta x \tag{3-14}$$

We can summarize Bresenham line drawing for a line with a positive slope less than 1 in the following listed steps. The constants $2\Delta y$ and $2\Delta y - 2\Delta x$ are calculated once for each line to be scan converted, so the arithmetic involves only integer addition and subtraction of these two constants.

Bresenham's Line-Drawing Algorithm for $|m| < 1$

1. Input the two line endpoints and store the left endpoint in (x_0, y_0).
2. Load (x_0, y_0) into the frame buffer; that is, plot the first point.
3. Calculate constants Δx, Δy, $2\Delta y$, and $2\Delta y - 2\Delta x$, and obtain the starting value for the decision parameter as

$$p_0 = 2\Delta y - \Delta x$$

4. At each x_k along the line, starting at $k = 0$, perform the following test: If $p_k < 0$, the next point to plot is $(x_k + 1, y_k)$ and

$$p_{k+1} = p_k + 2\Delta y$$

Otherwise, the next point to plot is $(x_k + 1, y_k + 1)$ and

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

5. Repeat step 4 Δx times.

Example 3-1 Bresenham Line Drawing

To illustrate the algorithm, we digitize the line with endpoints (20, 10) and (30, 18). This line has a slope of 0.8, with

$$\Delta x = 10, \qquad \Delta y = 8$$

The initial decision parameter has the value

$$\begin{aligned} p_0 &= 2\Delta y - \Delta x \\ &= 6 \end{aligned}$$

and the increments for calculating successive decision parameters are

$$2\Delta y = 16, \qquad 2\Delta y - 2\Delta x = -4$$

We plot the initial point $(x_0, y_0) = (20, 10)$, and determine successive pixel positions along the line path from the decision parameter as

k	p_k	(x_{k+1}, y_{k+1})	k	p_k	(x_{k+1}, y_{k+1})
0	6	(21, 11)	5	6	(26, 15)
1	2	(22, 12)	6	2	(27, 16)
2	−2	(23, 12)	7	−2	(28, 16)
3	14	(24, 13)	8	14	(29, 17)
4	10	(25, 14)	9	10	(30, 18)

```
void lineBres (int xa, int ya, int xb, int yb)
{
  int dx = abs (xa - xb), dy = abs (ya - yb);
  int p = 2 * dy - dx;
  int twoDy = 2 * dy, twoDyDx = 2 * (dy - dx);
  int x, y, xEnd;

  /* Determine which point to use as start, which as end */
  if (xa > xb) {
    x = xb;
    y = yb;
    xEnd = xa;
  }
  else {
    x = xa;
    y = ya;
    xEnd = xb;
  }
  setPixel (x, y);

  while (x < xEnd) {
    x++;
    if (p < 0)
      p += twoDy;
    else {
      y++;
      p += twoDyDx;
    }
    setPixel (x, y);
  }
}
```

Bresenham's algorithm is generalized to lines with arbitrary slope by considering the symmetry between the various octants and quadrants of the xy plane. For a line with positive slope greater than 1, we intelrhange the roles of the x and y directions. That is, we step along they direction in unit steps and calculate successive x values nearest the line path. Also, we could revise the program to plot pixels starting from either endpoint. If the initial position for a line with positive slope is the right endpoint, both x and y decrease as we step from right to left. To ensure that the same pixels are plotted regardless of the starting endpoint, we always choose the upper (or the lower) of the two candidate pixels whenever the two vertical separations from the line path are equal (d1 = d2). For negative slopes, the procedures are similar, except that now one coordinate decreases as the other increases. Finally, specla1 cases can be handled separately: Horizontal lines (=) vertical lines (=), and diagonal lines with = each can be loaded directly into the frame buffer without processing them through the line-plotting algorithm

LOADING THE **FRAME** BUFFER

When straight line segments and other objects are scan converted for display with a raster system, frame-buffer positions must be calculated. We have assumed that this is accomplished with the **setpixel** procedure, which stores intensity values for the pixels at corresponding addresses within the frame-buffer array. Scan-conversion algorithms generate pixel positions at successive unit intervals. This allows us to use incremental methods to calculate frame-buffer addresses.

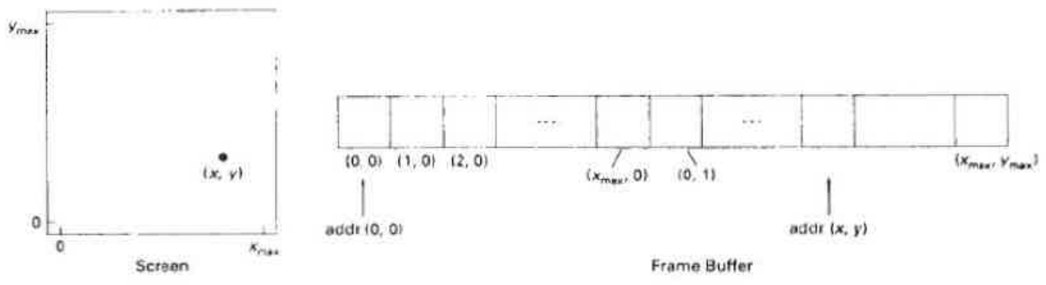


Figure 3-11
Pixel screen positions stored linearly in row-major order within the frame buffer.

As a specific example, suppose the frame-buffer array is addressed in row-major order and that pixel positions vary from (0, 0) at the lower left screen corner to (x_{max}, y_{max}) at the top right corner (Fig. 3-11). For a bilevel system (1 bit per pixel), the frame-buffer bit address for pixel position (x, y) is calculated as

$$\text{addr}(x, y) = \text{addr}(0, 0) + y(x_{max} + 1) + x \qquad (3\text{-}21)$$

Moving across a scan line, we can calculate the frame-buffer address for the pixel at $(x + 1, y)$ as the following offset from the address for position (x, y):

$$\text{addr}(x + 1, y) = \text{addr}(x, y) + 1 \qquad (3\text{-}22)$$

Stepping diagonally up to the next scan line from (x, y), we get to the frame-buffer address of $(x + 1, y + 1)$ with the calculation

$$\text{addr}(x + 1, y + 1) = \text{addr}(x, y) + x_{max} + 2 \qquad (3\text{-}23)$$

LINE FUNCTION

A procedure for specifying straight-line segments can be set up in a number of different forms. The two-dimensional line function is

```
polyline (n, wcPoints)
```

where parameter n is assigned an integer value equal to the number of coordinate positions to be input, and **wcpoints** is the array of input world coordinate values for line segment endpoints. This function is used to define a set of n – 1 connected straight line segments. Because series of connected line segments occur more often than isolated line segments in graphics applications, **polyline** provides a more general line function. To display a single straight-line segment, we set n =2 and list the ***x*** and y values of the two endpoint coordinates in wcpoints.

As an example of the use of **polyline,** the following statements generate two connected line segments, with endpoints at (50, 100) (150, **250)** and (250,**100)**

```
wcPoints[1].x = 50;
wcPoints[1].y = 100;
wcPoints[2].x = 150;
wcPoints[2].y = 250;
wcPoints[3].x = 250;
wcPoints[3].y = 100;
polyline (3, wcPoints);
```

CIRCLE-GENERATING ALGORITHMS

Since the circle is a frequently used component in pictures and graphs, a procedure for generating either full circles or circular arcs is included in most graphics packages. More generally, a single procedure can be provided to display either circular or elliptical **curves.**

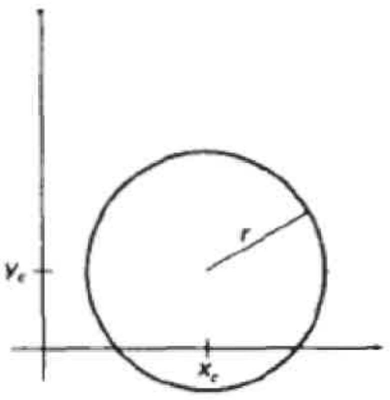


Figure 3-12
Circle with center coordinates (x_c, y_c) and radius r.

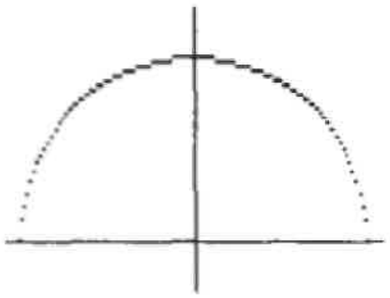


Figure 3-13
Positive half of a circle plotted with Eq. 3-25 and with $(x_c, y_c) = (0, 0)$.

Properties of Circles

A circle is defined as the set of points that are all at a given distance r from a center position (x_c, y_c) (Fig. 3-12). This distance relationship is expressed by the Pythagorean theorem in Cartesian coordinates as

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \qquad (3\text{-}24)$$

We could use this equation to calculate the position of points on a circle circumference by stepping along the x axis in unit steps from $x_c - r$ to $x_c + r$ and calculating the corresponding y values at each position as

$$y = y_c \pm \sqrt{r^2 - (x_c - x)^2} \qquad (3\text{-}25)$$

- This is not the best method:
  - *Considerable amount of computation*
  - *Spacing between plotted pixels is not uniform*

We could adjust the spacing by interchanging ***x*** and y (stepping through y values and calculating x values) whenever the absolute value of the slope of the circle is greater than 1. But this simply increases the computation and processing required by the algorithm.

Another way to eliminate the unequal spacing shown in Fig. **3-13** is to calculate points along the circular boundary using polar coordinates r and ***8*** (Fig. **3-12).** Expressing the circle equation in parametric polar form yields the pair of equations.

$$x = x_c + r\cos\theta$$
$$y = y_c + r\sin\theta$$

Computation can be reduced by considering the symmetry of circles. The shape of the circle is similar in each quadrant. We can generate the circle section in (he second quadrant of the **xy** plane by noting that the two circle sections are symmetric with respect to they axis. And circle sections in the third and fourth quadrants can be obtained from sections in the first and second quadrants by considering symmetry about the ***x*** axis. We can take this one step further and note that there is also symmetry between octants. Circle sections in adjacent ***oc***tants within one quadrant are symmetric with respect to the 45' line dividing the two octants. These symmetry conditions are illustrated in Fig.3-14, where a point at position ***(x,*** y) on a one-eighth circle sector is mapped into the seven circle points in the other octants of the **xy** plane. Taking advantage of the circle symme**try** in this way we can generate all pixel positions around a circle by calculating only the points within the sector from ***x*** = ***0*** to **x** = y.

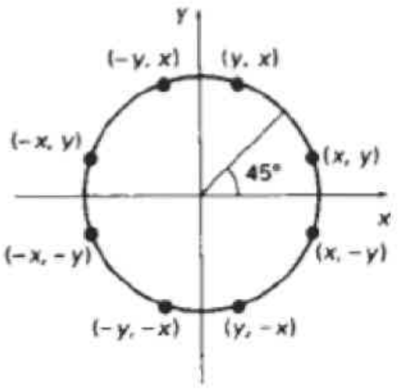


Figure 3-14
Symmetry of a circle. Calculation of a circle point (x, y) in one octant yields the circle points shown for the other seven octants.

Midpoint Circle Algorithm

- We will first calculate pixel positions for a circle centered around the origin (0,0). Then, each calculated position (x,y) is moved to its proper screen position by adding xc to x and yc to y

- Note that along the circle section from x=0 to x=y in the first octant, the slope of the curve varies from 0 to -1

- Circle function around the origin is given by fcircle(x,y) = x2 + y2 – r2

- Any point (x,y) on the boundary of the circle satisfies the equation and circle function is zero

- For a point in the interior of the circle, the circle function is negative and for a point outside the circle, the function is positive
- Thus,
  - $f_{circle}(x,y) < 0$ if (x,y) is inside the circle boundary
  - $f_{circle}(x,y) = 0$ if (x,y) is on the circle boundary
  - $f_{circle}(x,y) > 0$ if (x,y) is outside the circle boundary

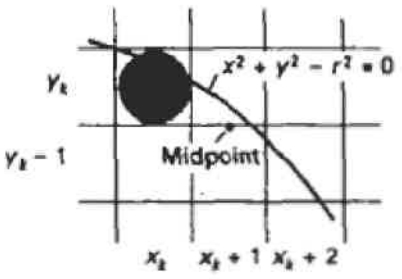


Figure 3-15
Midpoint between candidate pixels at sampling position x_k+1 along a circular path.

Figure 3-15 shows the midpoint between the two candidate pixels at sampling position $x_k + 1$. Assuming we have just plotted the pixel at (x_k, y_k), we next need to determine whether the pixel at position $(x_k + 1, y_k)$ or the one at position $(x_k + 1, y_k - 1)$ is closer to the circle. Our decision parameter is the circle function 3-27 evaluated at the midpoint between these two pixels:

$$\begin{aligned} p_k &= f_{\text{circle}}\left(x_k + 1, y_k - \frac{1}{2}\right) \\ &= (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2 \end{aligned} \tag{3-29}$$

If $p_k < 0$, this midpoint is inside the circle and the pixel on scan line y_k is closer to the circle boundary. Otherwise, the midposition is outside or on the circle boundary, and we select the pixel on scanline $y_k - 1$.

Successive decision parameters are obtained using incremental calculations. We obtain a recursive expression for the next decision parameter by evaluating the circle function at sampling position $x_{k+1} + 1 = x_k + 2$:

$$\begin{aligned} p_{k+1} &= f_{\text{circle}}\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right) \\ &= [(x_k + 1) + 1]^2 + \left(y_{k+1} - \frac{1}{2}\right)^2 - r^2 \end{aligned}$$

or

$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1 \tag{3-30}$$

where y_{k+1} is either y_k or y_{k-1}, depending on the sign of p_k.

Increments for obtaining p_{k+1} are either $2x_{k+1} + 1$ (if p_k is negative) or $2x_{k+1} + 1 - 2y_{k+1}$. Evaluation of the terms $2x_{k+1}$ and $2y_{k-1}$ can also be done incrementally as

$$\begin{aligned} 2x_{k+1} &= 2x_k + 2 \\ 2y_{k+1} &= 2y_k - 2 \end{aligned}$$

At the start position $(0, r)$, these two terms have the values 0 and $2r$, respectively. Each successive value is obtained by adding 2 to the previous value of $2x$ and subtracting 2 from the previous value of $2y$.

The initial decision parameter is obtained by evaluating the circle function at the start position $(x_0, y_0) = (0, r)$:

$$p_0 = f_{circle}\left(1, r - \frac{1}{2}\right)$$
$$= 1 + \left(r - \frac{1}{2}\right)^2 - r^2$$

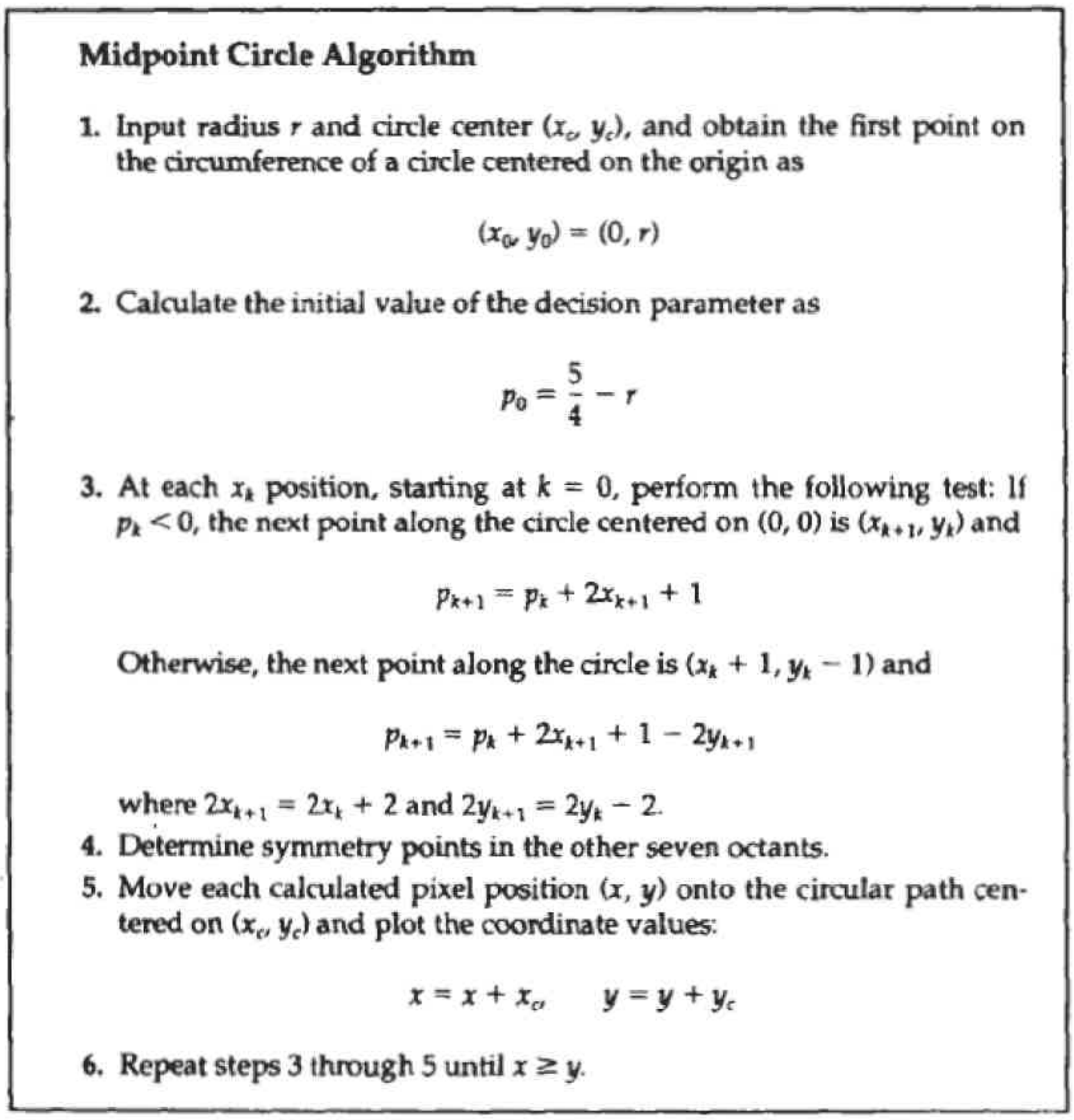
or

$$p_0 = \frac{5}{4} - r \qquad (3\text{-}31)$$

If the radius r is specified as an integer, we can simply round p_0 to

$$p_0 = 1 - r \qquad \text{(for } r \text{ an integer)}$$

since all increments are integers.



Midpoint Circle Algorithm

1. Input radius r and circle center (x_c, y_c), and obtain the first point on the circumference of a circle centered on the origin as

$$(x_0, y_0) = (0, r)$$

2. Calculate the initial value of the decision parameter as

$$p_0 = \frac{5}{4} - r$$

3. At each x_k position, starting at $k = 0$, perform the following test: If $p_k < 0$, the next point along the circle centered on $(0, 0)$ is (x_{k+1}, y_k) and

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

   Otherwise, the next point along the circle is $(x_k + 1, y_k - 1)$ and

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$

   where $2x_{k+1} = 2x_k + 2$ and $2y_{k+1} = 2y_k - 2$.

4. Determine symmetry points in the other seven octants.
5. Move each calculated pixel position (x, y) onto the circular path centered on (x_c, y_c) and plot the coordinate values:

$$x = x + x_c, \qquad y = y + y_c$$

6. Repeat steps 3 through 5 until $x \geq y$.

Given a circle radius $r = 10$, we demonstrate the midpoint cirle algorithm by determining positions along the circle octant in the first quadrant from $x = 0$ to $x = y$. The initial value of the decision parameter is

$$p_0 = 1 - r = -9$$

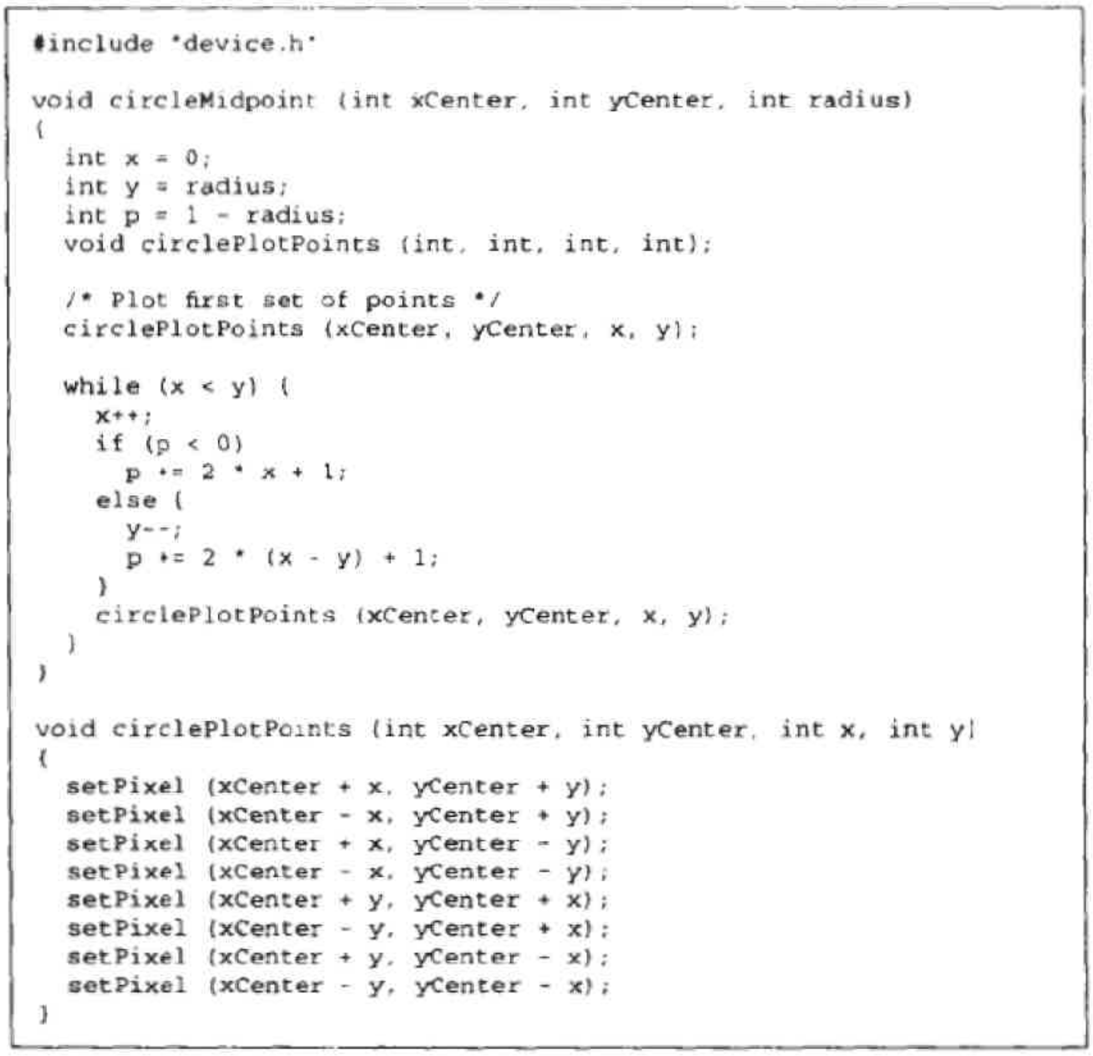
For the circle centered on the coordinate origin, the initial point is $(x_0, y_0) = (0, 10)$, and initial increment terms for calculating the decision parameters are

$$2x_0 = 0, \qquad 2y_0 = 20$$

Successive decision parameter values and positions along the circle path are calculated using the midpoint method as

k	p_k	(x_{k+1}, y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
0	−9	(1, 10)	2	20
1	−6	(2, 10)	4	20
2	−1	(3, 10)	6	20
3	6	(4, 9)	8	18
4	−3	(5, 9)	10	18
5	8	(6, 8)	12	16
6	5	(7, 7)	14	14

A plot of the generated pixel positions in the first quadrant is shown in Fig. 3-10.



```
#include "device.h"

void circleMidpoint (int xCenter, int yCenter, int radius)
{
  int x = 0;
  int y = radius;
  int p = 1 - radius;
  void circlePlotPoints (int, int, int, int);

  /* Plot first set of points */
  circlePlotPoints (xCenter, yCenter, x, y);

  while (x < y) {
    x++;
    if (p < 0)
      p += 2 * x + 1;
    else {
      y--;
      p += 2 * (x - y) + 1;
    }
    circlePlotPoints (xCenter, yCenter, x, y);
  }
}

void circlePlotPoints (int xCenter, int yCenter, int x, int y)
{
  setPixel (xCenter + x, yCenter + y);
  setPixel (xCenter - x, yCenter + y);
  setPixel (xCenter + x, yCenter - y);
  setPixel (xCenter - x, yCenter - y);
  setPixel (xCenter + y, yCenter + x);
  setPixel (xCenter - y, yCenter + x);
  setPixel (xCenter + y, yCenter - x);
  setPixel (xCenter - y, yCenter - x);
}
```

ELLIPSE-GENERATING ALGORITHMS

Loosely **stated,** an ellipse is an elongated circle. Therefore, elliptical ***curves*** can be generated by modifying circle-drawing procedures to take into account the different dimensions of an ellipse along the major and minor axes.

Properties of Ellipses

An ellipse is defined as the set of points such that the sum of the distances from two **fixed** positions (foci) is the **same** for **all** points. **If** the distances to the two foci from any point P = ***(x, y)*** on the ellipse are labeled **dl** and **d2,** then the general equation of an ellipse can be stated as

$$d_1 + d_2 = \text{constant} \tag{3-32}$$

Expressing distances d_1 and d_2 in terms of the focal coordinates $\mathbf{F}_1 = (x_1, y_1)$ and $\mathbf{F}_2 = (x_2, y_2)$, we have

$$\sqrt{(x - x_1)^2 + (y - y_1)^2} + \sqrt{(x - x_2)^2 + (y - y_2)^2} = \text{constant} \tag{3-33}$$

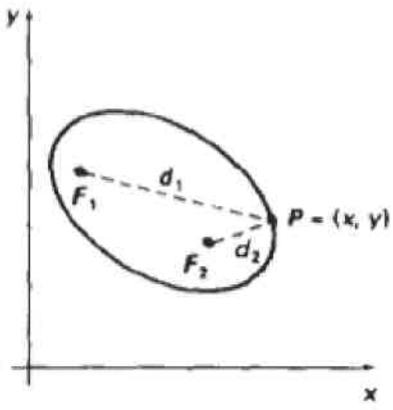


Figure 3-17
Ellipse generated about foci $\mathbf{F}_1$ and $\mathbf{F}_2$.

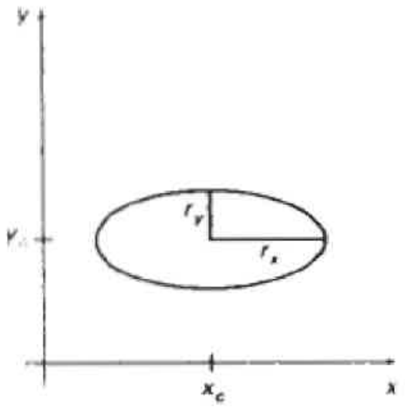


Figure 3-18
Ellipse centered at (x_c, y_c) with semimajor axis r_x and semiminor axis r_y.

The major axis is the straight line segment extending from one side of the ellipse to the other through the foci. The minor axis spans the shorter dimension of the ellipse, bisecting the major axis at the halfway position (ellipse center) between the two foci.

$$\left(\frac{x - x_c}{r_x}\right)^2 + \left(\frac{y - y_c}{r_y}\right)^2 = 1 \tag{3-35}$$

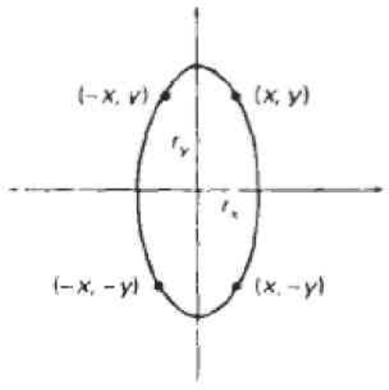
Using polar coordinates r and θ, we can also describe the ellipse in standard position with the parametric equations:

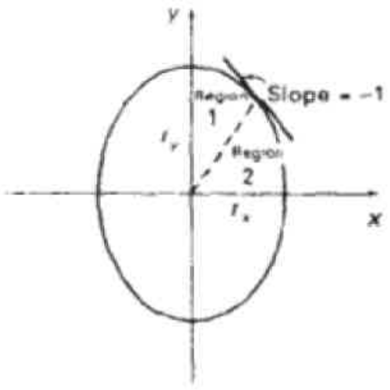
$$\begin{aligned} x &= x_c + r_x \cos\theta \\ y &= y_c + r_y \sin\theta \end{aligned} \tag{3-36}$$

Our approach here is similar *to* that used in displaying d raster circle. Given parameters we determine points ***(x,*** y) for an ellipse in standard position centered on the origin, and then we shift the points so the ellipse is centered at ***(xc ,*** yc).

The midpoint ellipse method is applied throughout the first quadrant in two parts. Figure sows the division of the first quadrant according to the slope of an ellipse with $r < r$ We process this quadrant by taking unit steps in x direction slope of the curve has a magnitude less than 1, and taking unit steps in the y direction where the slop has a magnitude greater than 1.

Regions 1 and 2 (Fig. **3-20),** can he processed in various ways. We can start at position (0. **r) and** step clockwise along the elliptical path in the first quadrant, shifting from unit steps in ***x*** to unit steps in y when the slope becomes less than -1.





We define an ellipse function from Eq. 3-35 with $(x_c, y_c) = (0, 0)$ as

$$f_{\text{ellipse}}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2 \tag{3-37}$$

which has the following properties:

$$f_{\text{ellipse}}(x, y) \begin{cases} < 0, & \text{if } (x, y) \text{ is inside the ellipse boundary} \\ = 0, & \text{if } (x, y) \text{ is on the ellipse boundary} \\ > 0, & \text{if } (x, y) \text{ is outside the ellipse boundary} \end{cases} \tag{3-38}$$

Thus, the ellipse function $f_{\text{ellipse}}(x, y)$ serves as the decision parameter in the midpoint algorithm. At each sampling position, we select the next pixel along the ellipse path according to the sign of the ellipse function evaluated at the midpoint between the two candidate pixels.

Starting at $(0, r_y)$, we take unit steps in the x direction until we reach the boundary between region 1 and region 2 (Fig. 3-20). Then we switch to unit steps in the y direction over the remainder of the curve in the first quadrant. At each step, we need to test the value of the slope of the curve. The ellipse slope is calculated from Eq. 3-37 as

$$\frac{dy}{dx} = -\frac{2r_y^2 x}{2r_x^2 y} \tag{3-39}$$

At the boundary between region 1 and region 2, $dy/dx = -1$ and

$$2r_y^2 x = 2r_x^2 y$$

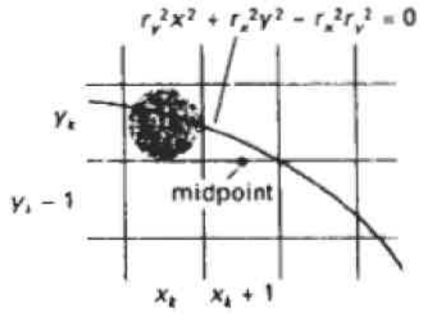
Therefore, we move out of region 1 whenever

$$2r_y^2 x \geq 2r_x^2 y \tag{3-40}$$

Figure 3-21 shows the midpoint between the two candidate pixels at sampling position $x_k + 1$ in the first region. Assuming position (x_k, y_k) has been selected at the previous step, we determine the next position along the ellipse path by evaluating the decision parameter (that is, the ellipse function 3-37) at this midpoint:

$$\begin{aligned} p1_k &= f_{\text{ellipse}}\left(x_k + 1, y_k - \frac{1}{2}\right) \\ &= r_y^2(x_k + 1)^2 + r_x^2\left(y_k - \frac{1}{2}\right)^2 - r_x^2 r_y^2 \end{aligned} \tag{3-41}$$

If $p1_k < 0$, the midpoint is inside the ellipse and the pixel on scan line y_k is closer to the ellipse boundary. Otherwise, the midposition is outside or on the ellipse boundary, and we select the pixel on scan line $y_k - 1$.



At the next sampling position ($x_{k+1} + 1 = x_k + 2$), the decision parameter for region 1 is evaluated as

$$p1_{k+1} = f_{\text{ellipse}}\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right)$$
$$= r_y^2[(x_k + 1) + 1]^2 + r_x^2\left(y_{k+1} - \frac{1}{2}\right)^2 - r_x^2r_y^2$$

or

$$p1_{k+1} = p1_k + 2r_y^2(x_k + 1) + r_y^2 + r_x^2\left[\left(y_{k+1} - \frac{1}{2}\right)^2 - \left(y_k - \frac{1}{2}\right)^2\right] \qquad (3\text{-}42)$$

where y_{k+1} is either y_k or $y_k - 1$, depending on the sign of $p1_k$.

Decision parameters are incremented by the following amounts:

$$\text{increment} = \begin{cases} 2r_y^2x_{k+1} + r_y^2, & \text{if } p1_k < 0 \\ 2r_y^2x_{k+1} + r_y^2 - 2r_x^2y_{k+1}, & \text{if } p1_k \geq 0 \end{cases}$$

In region 1, the initial value of the decision parameter is obtained by evaluating the ellipse function at the start position $(x_0, y_0) = (0, r_y)$:

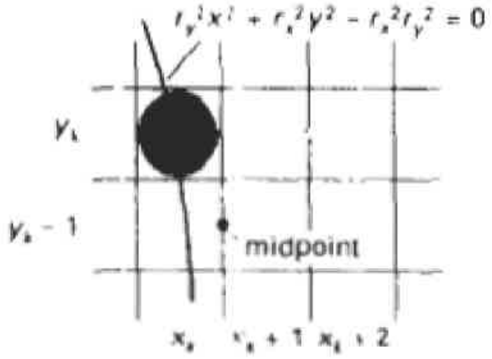
$$p1_0 = f_{\text{ellipse}}\left(1, r_y - \frac{1}{2}\right)$$
$$= r_y^2 + r_x^2\left(r_y - \frac{1}{2}\right)^2 - r_x^2r_y^2$$

or

$$p1_0 = r_y^2 - r_x^2r_y + \frac{1}{4}r_x^2 \qquad (3\text{-}45)$$

Over region 2, we sample at unit steps in the negative y direction, and the midpoint is now taken between horizontal pixels at each step (Fig. 3-22). For this region, the decision parameter is evaluated as

$$\begin{aligned} p2_k &= f_{\text{ellipse}}\left(x_k + \frac{1}{2},\ y_k - 1\right) \\ &= r_y^2\left(x_k + \frac{1}{2}\right)^2 + r_x^2(y_k - 1)^2 - r_x^2 r_y^2 \end{aligned} \tag{3-46}$$



If $p2_k > 0$, the midposition is outside the ellipse boundary, and we select the pixel at x_k. If $p2_k \leq 0$, the midpoint is inside or on the ellipse boundary, and we select pixel position x_{k+1}.

To determine the relationship between successive decision parameters in region 2, we evaluate the ellipse function at the next sampling step $y_{k+1} - 1 = y_k - 2$:

$$\begin{aligned} p2_{k+1} &= f_{\text{ellipse}}\left(x_{k+1} + \frac{1}{2}, y_{k+1} - 1\right) \\ &= r_y^2\left(x_{k+1} + \frac{1}{2}\right)^2 + r_x^2[(y_k - 1) - 1]^2 - r_x^2 r_y^2 \end{aligned} \tag{3-47}$$

or

$$p2_{k+1} = p2_k - 2r_x^2(y_k - 1) + r_x^2 + r_y^2\left[\left(x_{k+1} + \frac{1}{2}\right)^2 - \left(x_k + \frac{1}{2}\right)^2\right] \tag{3-48}$$

with x_{k+1} set either to x_k or to $x_k + 1$, depending on the sign of $p2_k$.

When we enter region 2, the initial position (x_0, y_0) is taken as the last position selected in region 1 and the initial decision parameter in region 2 is then

$$\begin{aligned} p2_0 &= f_{\text{ellipse}}\left(x_0 + \frac{1}{2}, y_0 - 1\right) \\ &= r_y^2\left(x_0 + \frac{1}{2}\right)^2 + r_x^2(y_0 - 1)^2 - r_x^2 r_y^2 \end{aligned} \tag{3-49}$$

the x-intersection value x_{k+1} on the upper scan line can be determined from the x-intersection value x_k on the preceding scan line as

$$x_{k+1} = x_k + \frac{1}{m} \tag{3-60}$$

Each successive x intercept can thus be calculated by adding the inverse of the slope and rounding to the nearest integer.

An obvious parallel implementation of the fill algorithm is to assign each scan line crossing the polygon area to a separate processor. Edge-intersection calculations are then performed independently. Along an edge with slope m, the intersection x_k value for scan line k above the initial scan line can be calculated as

$$x_k = x_0 + \frac{k}{m} \tag{3-61}$$

In a sequential fill algorithm, the increment of x values by the amount $1/m$ along an edge can be accomplished with integer operations by recalling that the slope m is the ratio of two integers:

$$m = \frac{\Delta y}{\Delta x}$$

where Δx and Δy are the differences between the edge endpoint x and y coordinate values. Thus, incremental calculations of x intercepts along an edge for successive scan lines can be expressed as

$$x_{k+1} = x_k + \frac{\Delta x}{\Delta y} \tag{3-62}$$

Boundary-Fill Algorithm

If the boundary is specified in a single color, the fill algorithm proceeds outward pixel by pixel until the boundary color is encountered. **This** method, called the boundary-till algorithm, is particularly useful in interactive painting packages, where interior points ***are*** easiiy selected. Using a graphics tablet or other interactive device, an artist or designer can sketch a figure outline, select a fill color or pattern from a color menu, and pick an interior point. The system then paints the figure interior. To display a solid color region (with no border), the designer can choose the fill color to **be** the same as the boundary color.

A boundary-fill procedure accepts as input the coordinates of an interior point ***(x,*** y), a fill color, and a boundary color. Starting from **(x, y),** the procedure tests neighboring positions to determine whether they are of the boundary color.If not, they are painted with the fill color, and their neighbors are tested. This process continues until all pixels up to the boundary color for the area have been tested. Both inner and outer boundaries can be set up to specify an area, and some examples of defining regions for boundary fill are shown in Fig. 3-42.

Figure 3-43 shows two methods for proceeding to neighboring pixels from the current test position. In Fig. 3.43(a), four neighboring points are tested. These are the pixel positions that are right, left, above, and below the current pixel. Areas **filled** by this method are called connected. The second method, shown in Fig. 3-43(b), is **used** to fill more complex figures. Here the set of neighboring positions to **be** tested includes the four diagonal pixels.

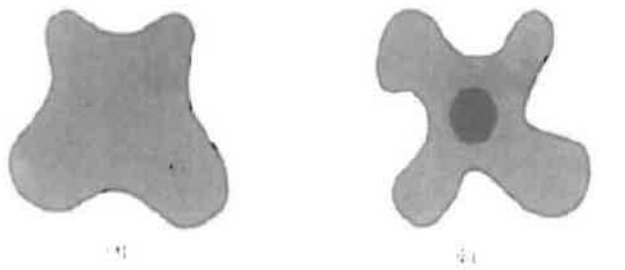


Figure 3-42

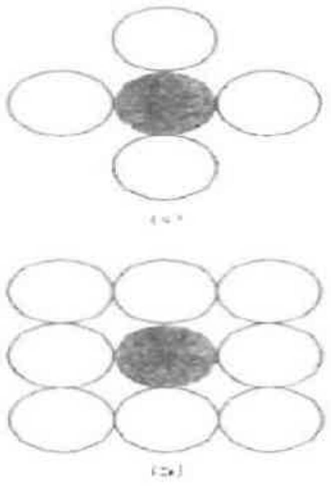


Figure 3-43

The following procedure illustrates a recursive method for filling a **4** connected area with an intensity specified in parameter **f** ill up to a boundary color specified with parameter boundary. We can extend this procedure to fill an 8 connected region by including four additional statements to test diagonal positions, such is ***(x*** + 1, **y** + **1)**.

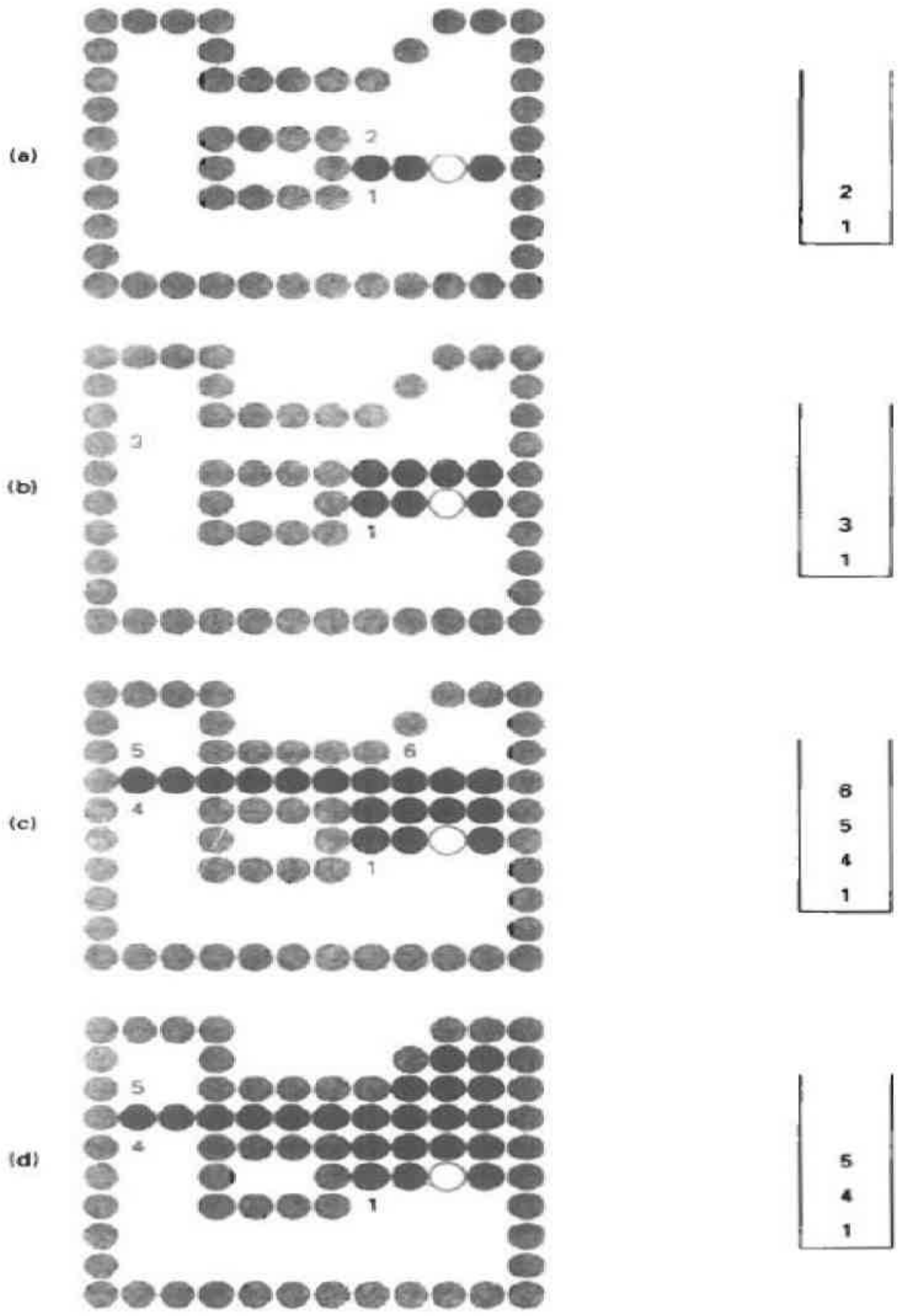
```
void boundaryFill4 (int x, int y, int fill, int boundary)
{
  int current;

  current = getPixel (x, y);
  if ((current != boundary) && (current != fill)) {
    setColor (fill);
    setPixel (x, y);
    boundaryFill4 (x+1, y, fill, boundary);
    boundaryFill4 (x-1, y, fill, boundary);
    boundaryFill4 (x, y+1, fill, boundary);
    boundaryFill4 (x, y-1, fill, boundary);
  }
}
```

Recursive boundary-fill algorithms may not fill regions correctly if some interior pixels are already displayed in the fill color. This occurs because the algorithm checks next pixels both for boundary color and for fill color. Encountering a pixel with the fill color can cause a recursive branch to terminate, leaving other interior pixels unfilled. To avoid this, we can first change the color of any interior pixels that are initially set to the fill color before applying the boundary-fill procedure.

Also, since this procedure **requires** considerable stacking of neighboring points, more efficient methods are generally employed. These methods fill horizontal pixel spans across scan lines, instead of proceeding to 4-connected or 8-connected neighboring points. Then we need only stack a beginning position for each horizontal pixel span, instead of stacking all unprocessed neighboring positions around the current position. Starting **from** the initial interior point with this method, we first fill in the contiguous span of pixels on this starting scan line. Then we locate and stack starting positions for spans on the adjacent scan lines, whew spans are defined as the contiguous horizontal string of positions bounded by pixels displayed in the area border color. **At** each subsequent step, we unstack the next start position and repeat the process.

An example of how pixel spans could be filled using this approach is illustrated for the 4-connected fill region in Fig. 3-45. In this example, we first process scan lines successively from the start line to the top boundary. After all upper scan lines are processed, we fill in the pixel spans on the remaining scan lines in order down to the bottom boundary. The leftmost pixel position for each horizontal span is located and stacked, in left to right order across successive scan lines, as shown in Fig. 3-45.



Flood-Fill Algorithm

Sometimes we want to fill in (or recolor) an area that is not defined within a single color boundary. Figure 3-46 shows an area bordered by several different color regions. We can paint such areas by replacing a specified interior color instead of searching for a boundary color value. This approach is called a flood-fill algorithm. We start from a specified interior point ***(x, y)*** and reassign all pixel values that are currently set to a given interior color with the desired fill color. If the area we want to paint has more than one interior color, we can first reassign pixel values ***so*** that all interior points have the same color. Using either a 4 connected or 8-connected approach, we then step through pixel positions until all interior points have been repainted. The following procedure flood fills a connected region recursively, starting from the input position.

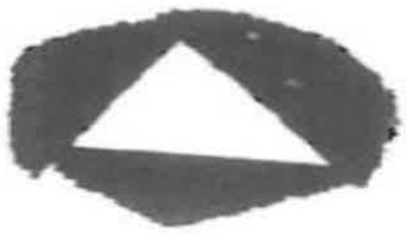
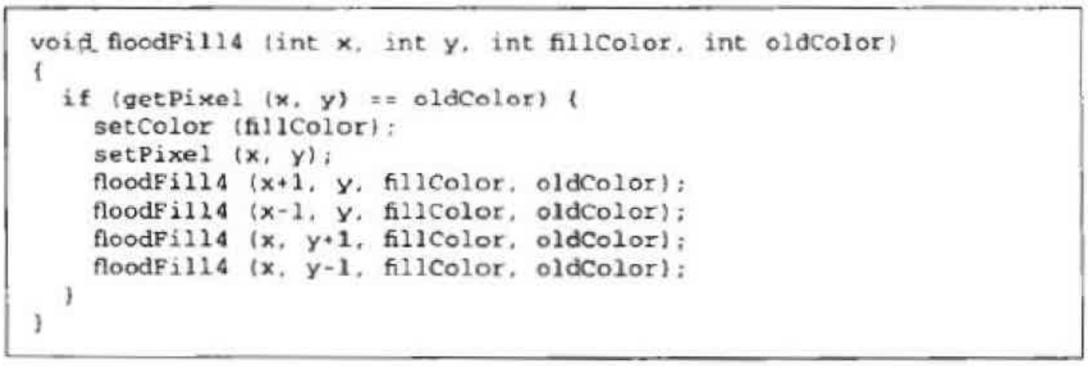


Figure 3-46
An area defined within multiple color boundaries.



```
void floodFill4 (int x, int y, int fillColor, int oldColor)
{
  if (getPixel (x, y) == oldColor) {
    setColor (fillColor);
    setPixel (x, y);
    floodFill4 (x+1, y, fillColor, oldColor);
    floodFill4 (x-1, y, fillColor, oldColor);
    floodFill4 (x, y+1, fillColor, oldColor);
    floodFill4 (x, y-1, fillColor, oldColor);
  }
}
```

Scan line Polygon Fill Algorithm

Figure **3-35** illustrates the scan-line procedure for solid filling of polygon areas. For each scan line crossing a polygon, the area-fill algorithm locates the intersection points of the scan line with the polygon edges. These intersection points are then sorted from left to right, and the corresponding frame-buffer positions between each intersection pair are set to the specified fill color. In the example of Fig. **3-35,** the four pixel intersection positions with the polygon boundaries define two stretches of interior pixels from $x = 10$ to $x = 14$ and from $x = 18$ to $\mathbf{x} = 24$.

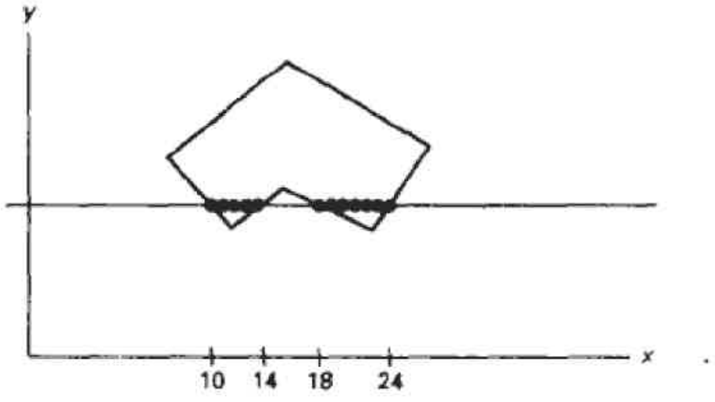


Figure 3-35
Interior pixels along a scan line
passing through a polygon area

Some scan-line intersections at polygon vertices require special handling. **A** scan line passing through a vertex intersects two edges at that position, adding two points to the list of intersections for the scan line. Figure **3-36** shows two scan lines at positions y and y' that intersect edge endpoints. Scan line y intersects five polygon edges. Scan line y', however, intersects an even number of edges although it also passes through a vertex. Intersection points along scan line **y'** correctly identify the interior pixel spans. But with scan line y, we need to do some additional processing to determine the correct interior points.

The topological difference between scan line y and scan line y' in Fig. 3-36 is identified by noting the position of the intersecting edges relative to the scan line. For scan line y, the two intersecting edges sharing a vertex are on opposite sides of the scan line. But for scan line **y',** the two intersecting edges are both above the scan line. Thus, the vertices that require additional processing are those that have connecting edges on opposite sides of the scan line. We can identify these vertices by tracing around the polygon boundary either in clockwise or counterclockwise order and observing the relative changes in vertex y coordinates as we move from one edge to the next. If the endpoint y values of two consecutive edges monotonically increase or decrease, we need to count the middle vertex as a single intersection point for any scan line passing through that vertex. Otherwise, the shared vertex represents a local extremum (minimum or maximum) on the **poly**gon boundary, and the two edge intersections with the scan line passing through that vertex can be added to the intersection list.

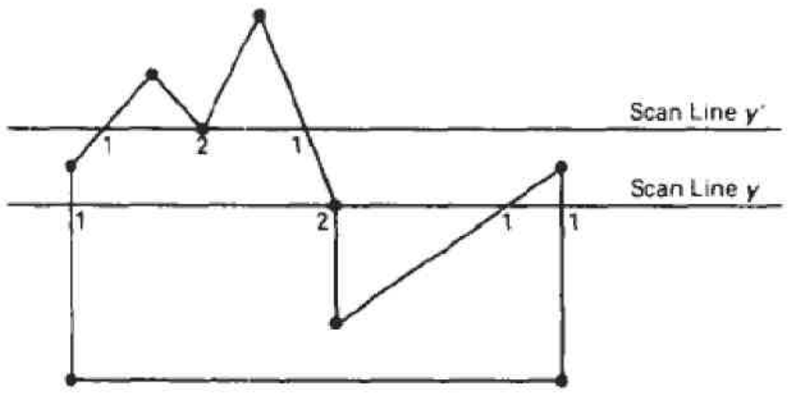


Figure 3-36

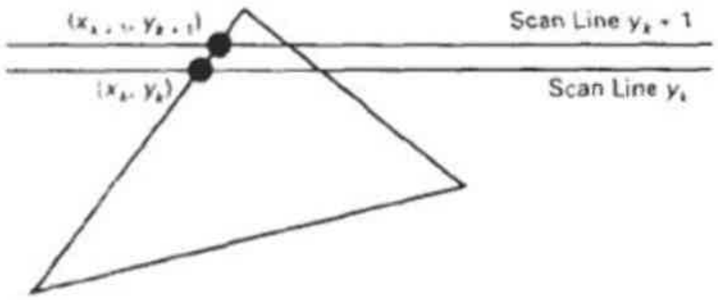
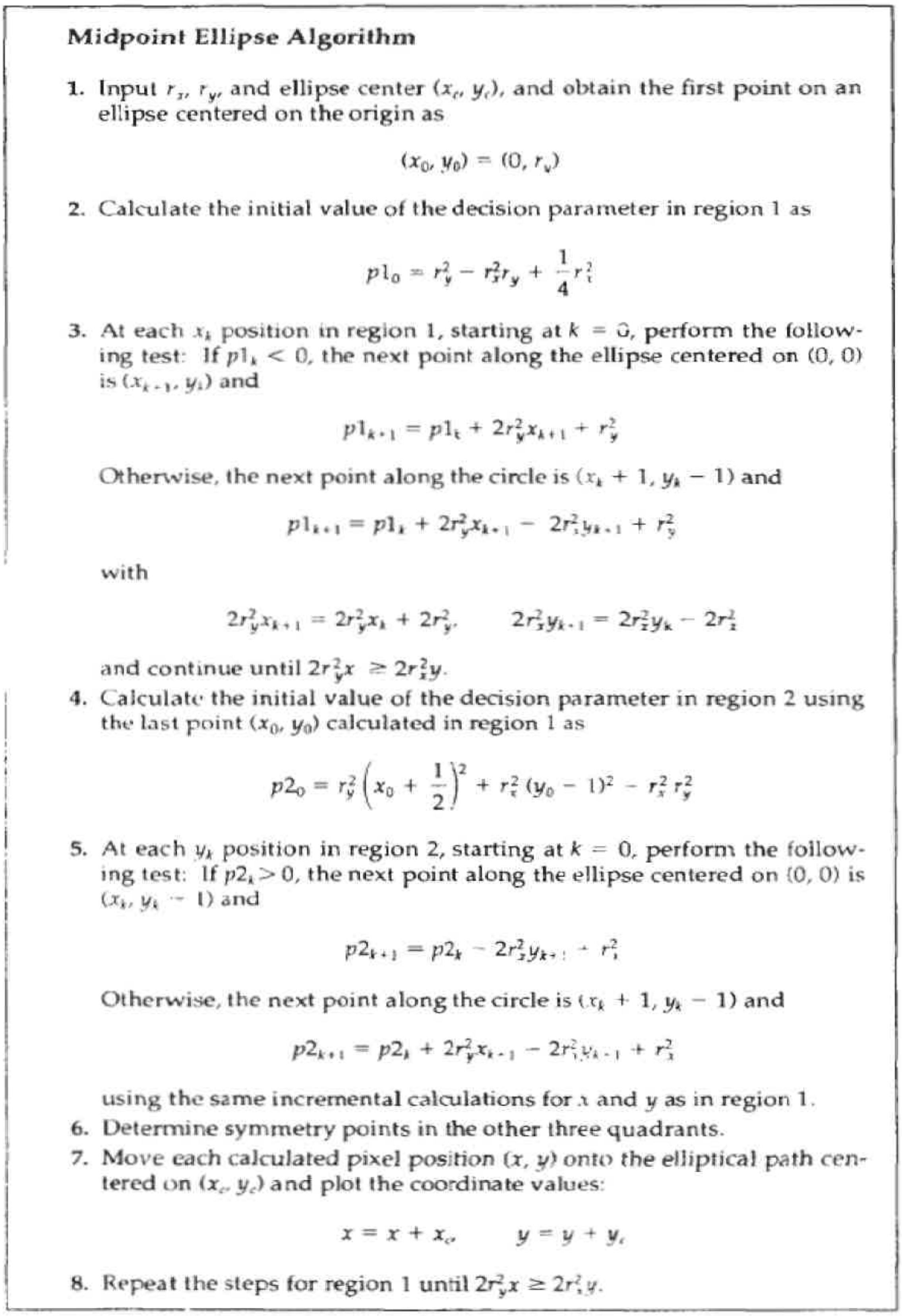


Figure 3-38
Two successive scan lines intersecting a polygon boundary.



Midpoint Ellipse Algorithm

1. Input r_x, r_y, and ellipse center (x_c, y_c), and obtain the first point on an ellipse centered on the origin as

$$(x_0, y_0) = (0, r_y)$$

2. Calculate the initial value of the decision parameter in region 1 as

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2$$

3. At each x_k position in region 1, starting at $k = 0$, perform the following test: If $p1_k < 0$, the next point along the ellipse centered on (0, 0) is (x_{k+1}, y_k) and

$$p1_{k+1} = p1_k + 2r_y^2 x_{k+1} + r_y^2$$

Otherwise, the next point along the circle is $(x_k + 1, y_k - 1)$ and

$$p1_{k+1} = p1_k + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_y^2$$

with

$$2r_y^2 x_{k+1} = 2r_y^2 x_k + 2r_y^2, \qquad 2r_x^2 y_{k+1} = 2r_x^2 y_k - 2r_x^2$$

and continue until $2r_y^2 x \geq 2r_x^2 y$.

4. Calculate the initial value of the decision parameter in region 2 using the last point (x_0, y_0) calculated in region 1 as

$$p2_0 = r_y^2 \left(x_0 + \frac{1}{2}\right)^2 + r_x^2 (y_0 - 1)^2 - r_x^2 r_y^2$$

5. At each y_k position in region 2, starting at $k = 0$, perform the following test: If $p2_k > 0$, the next point along the ellipse centered on (0, 0) is $(x_k, y_k - 1)$ and

$$p2_{k+1} = p2_k - 2r_x^2 y_{k+1} + r_x^2$$

Otherwise, the next point along the circle is $(x_k + 1, y_k - 1)$ and

$$p2_{k+1} = p2_k + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_x^2$$

using the same incremental calculations for x and y as in region 1.

6. Determine symmetry points in the other three quadrants.
7. Move each calculated pixel position (x, y) onto the elliptical path centered on (x_c, y_c) and plot the coordinate values:

$$x = x + x_c, \qquad y = y + y_c$$

8. Repeat the steps for region 1 until $2r_y^2 x \geq 2r_x^2 y$.