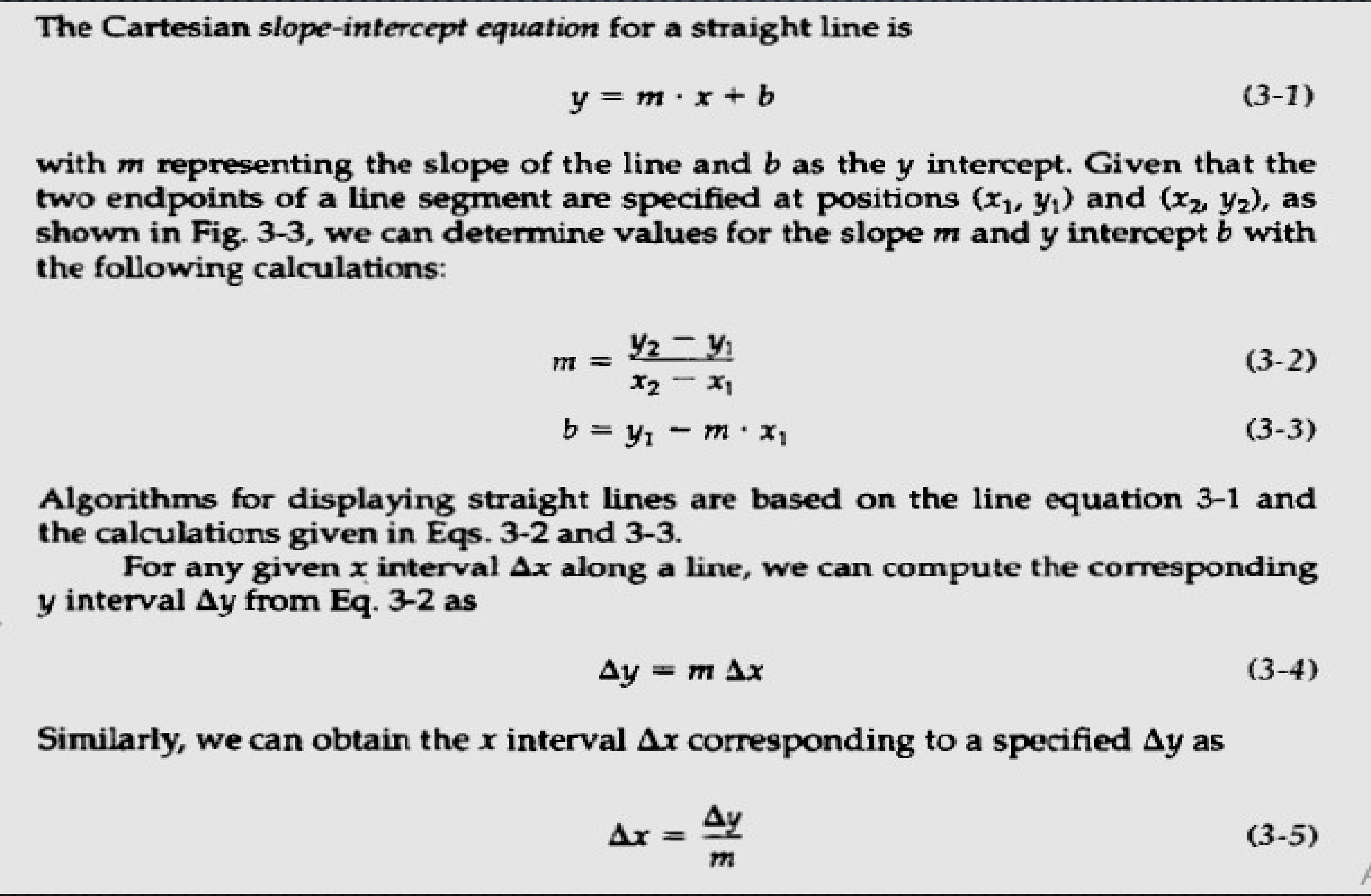
LINE-DRAWING ALGORITHMS:



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

$$y = m \cdot x + b \tag{3-1}$$

with m representing the slope of the line and b as the y intercept. Given that the two endpoints of a line segment are specified at positions (x_1, y_1) and (x_2, y_2), as shown in Fig. 3-3, we can determine values for the slope m and y intercept b with the following calculations:

$$m = \frac{y_2 - y_1}{x_2 - x_1} \tag{3-2}$$

$$b = y_1 - m \cdot x_1 \tag{3-3}$$

Algorithms for displaying straight lines are based on the line equation 3-1 and the calculations given in Eqs. 3-2 and 3-3.

For any given x interval Δx along a line, we can compute the corresponding y interval Δy from Eq. 3-2 as

$$\Delta y = m \, \Delta x \tag{3-4}$$

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

$$\Delta x = \frac{\Delta y}{m} \tag{3-5}$$

These equations form the basis for determining deflection voltages in analog devices. For lines with slope magnitudes $|m| < 1$, Δx can be set proportional to a small horizontal deflection voltage and the corresponding vertical deflection is then set proportional to Δy as calculated from Eq. 3-4.

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

For lines with $m = 1$, $\Delta x = \Delta y$ and the horizontal and vertical deflections voltages are equal. In each case, a smooth line with slope m is generated between the specified endpoints.

On raster systems, lines are plotted with pixels, and step sizes in the horizontal and vertical directions are constrained by pixel separations. That is, we must "sample" a line at discrete positions and determine the nearest pixel to the line at each sampled position. This scan conversion process for straight lines is illustrated in Fig. 3-4, for a near horizontal line with discrete sample positions along the x axis.

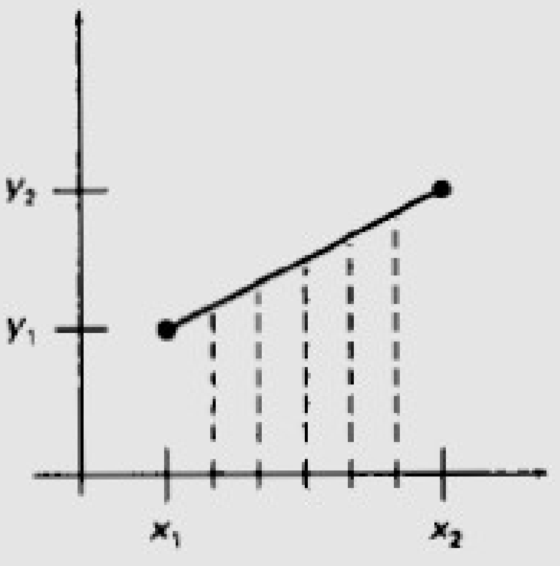


Figure 3-4
Straight line segment with five sampling positions along the x axis between x_1 and x_2.

DDA Algorithm:

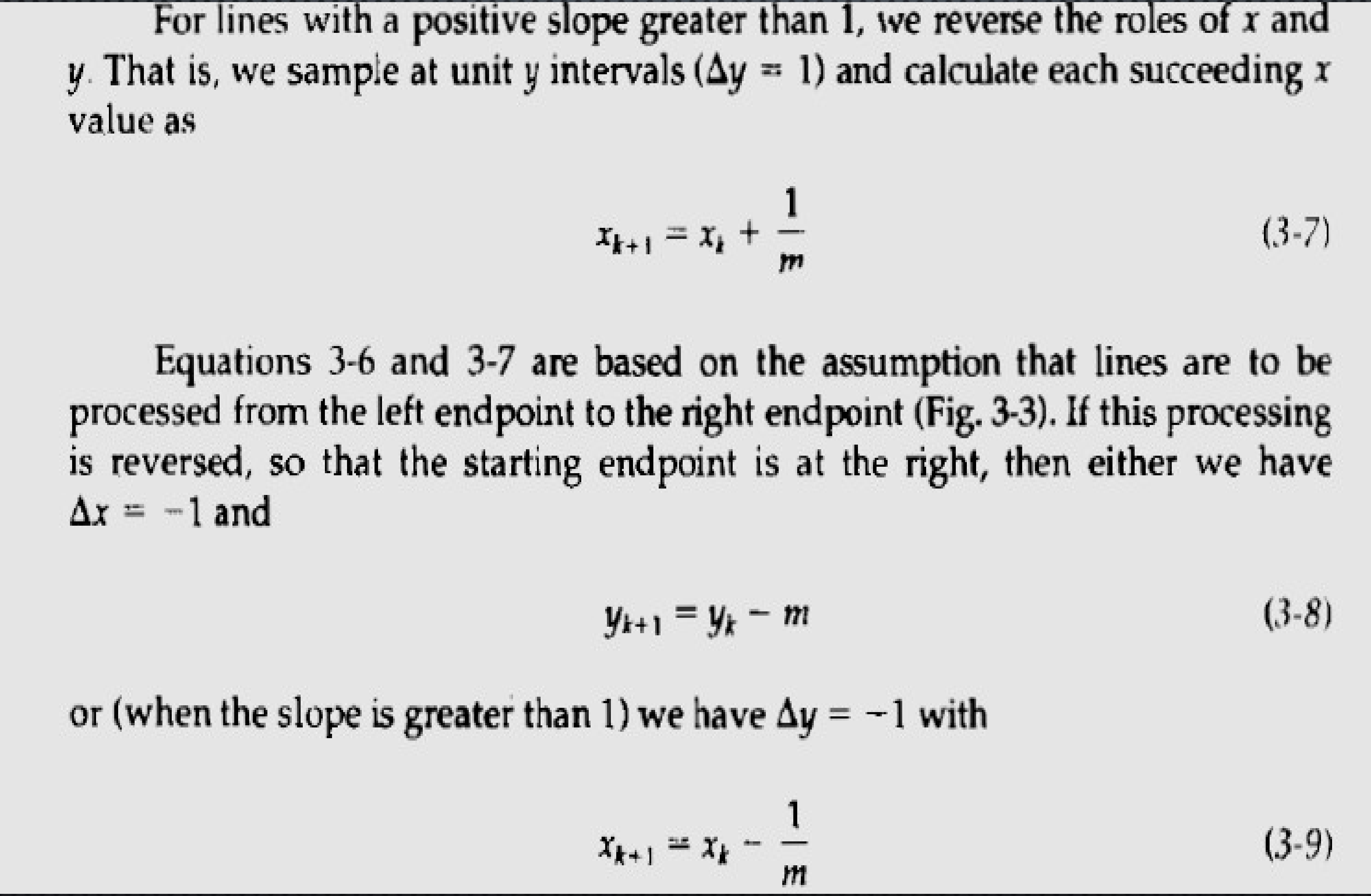
The digital differential analyzer (DDA) is a scan-conversion line algorithm based on figure 3-4 calculating either Δy or Δx, using Eq. 3-4 or Eq. 3-5. We sample the line at unit intervals in one coordinate and determine corresponding integer values nearest the line path for the other coordinate.

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 ($\Delta x = 1$) and compute each successive y value as

$$y_{k+1} = y_k + m$$

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

DDA Algorithm:



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

$$x_{k+1} = x_k + \frac{1}{m} \tag{3-7}$$

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

$$y_{k+1} = y_k - m \tag{3-8}$$

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

$$x_{k+1} = x_k - \frac{1}{m} \tag{3-9}$$

DDA Algorithm:

Equations 3-6 through 3-9 can also be used to calculate pixel positions along a line with negative slope. If the absolute value of the slope is less than 1 and the start endpoint is at the left, we set $\Delta x = 1$ and calculate y values with Eq. 3-6.

When the start endpoint is at the right (for the same slope), we set $\Delta x = -1$ and obtain y positions from Eq. 3-8. Similarly, when the absolute value of a negative slope is greater than 1, we use $\Delta y = -1$ and Eq. 3-9 or we use $\Delta y = 1$ and Eq. 3-7.

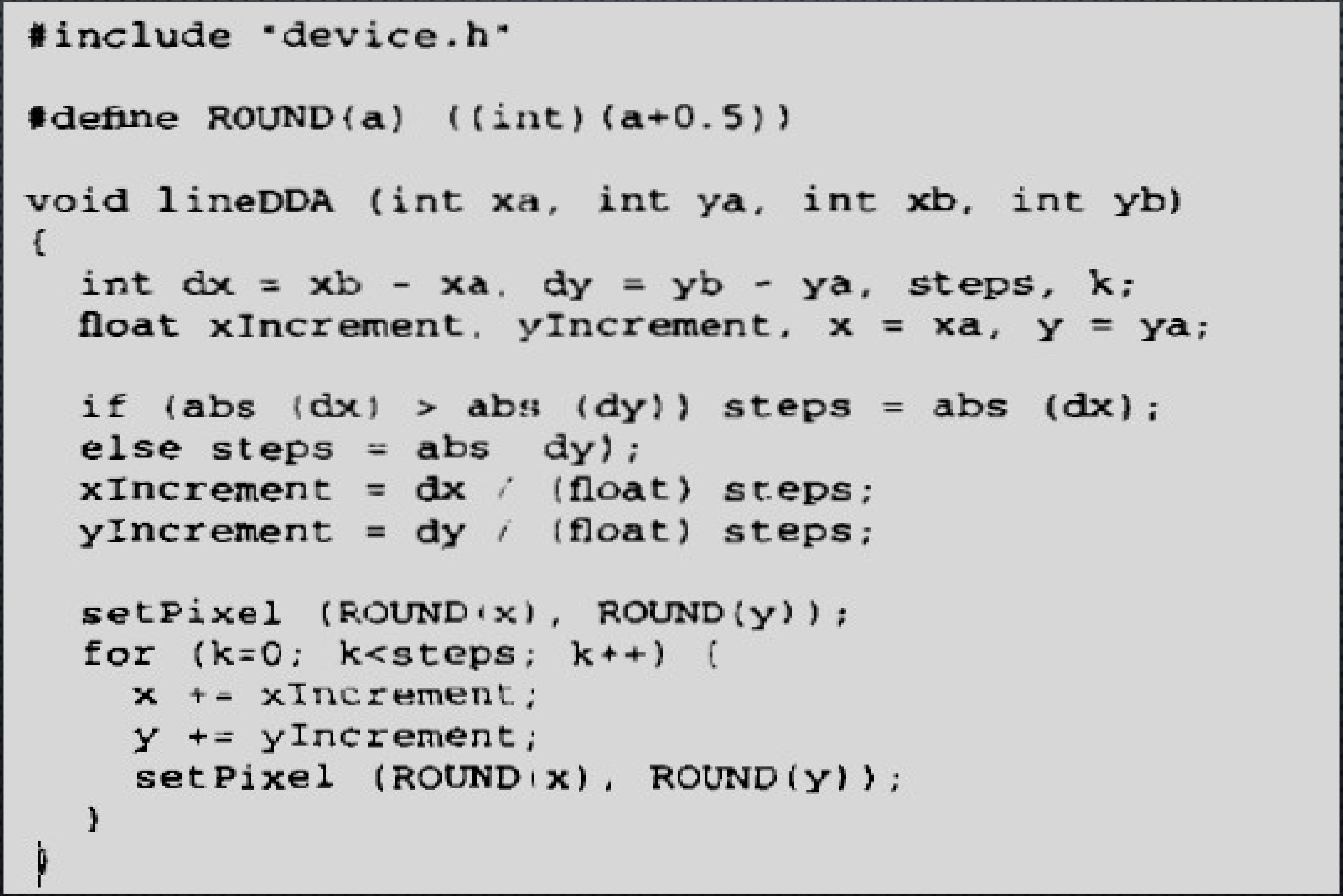
DDA Algorithm:

This algorithm is summarized in the following procedure, which accepts as input the two endpoint pixel positions. Horizontal and vertical differences between the endpoint positions are assigned to parameters dx and dy. The difference with the greater magnitude determines the value of parameter steps. Starting with pixel position (x_a, y_a), we determine the offset needed at each step to generate the next pixel position along the line path.

We loop through this process steps times. If the magnitude of dx is greater than the magnitude of dy and x_a is less than x_b, the values of the increments in the x and y directions are 1 and m, respectively. If the greater change is in the x direction, but x_a is greater than x_b, then the decrements -1 and -m are used to generate each new point on the line.

Otherwise, we use a unit increment (or decrement) in the y direction and an x increment (or decrement) of 1/m .

DDA Algorithm:



```
#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));
  }
}
```

DDA 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. The accumulation of roundoff error in successive additions of the floating-point increment, however, can cause the calculated pixel positions to drift away from the true line path for long line segments.

Furthermore, the rounding operations and floating-point arithmetic in procedure lineDDA are still time-consuming. We can improve the performance of the DDA algorithm by separating the increments m and 1/m into integer and fractional parts so that all calculations are reduced to integer operations.

Bresenham's Line Algorithm :

An accurate and efficient raster line-generating algorithm, developed by Bresenham, scan converts lines using only incremental 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.

Bresenham's Line Algorithm :

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 (51,50) 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.

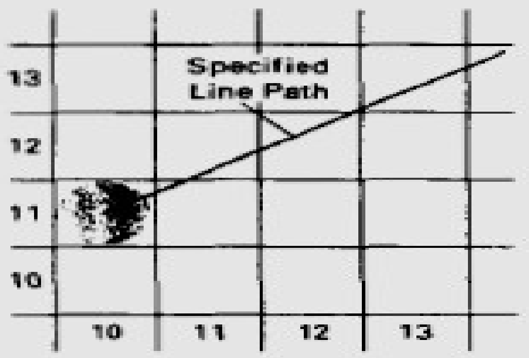


Figure 3-5
Section of a display screen where a straight line segment is to be plotted, starting from the pixel at column 10 on scan line 11.

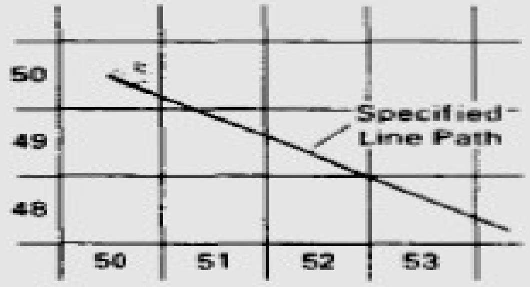


Figure 3-6
Section of a display screen where a negative slope line segment is to be plotted, starting from the pixel at column 50 on scan line 50.

Bresenham's Line Algorithm :

To illustrate Bresenham's approach, we first consider the scan-conversion process for lines with positive slope less than 1. Pixel positions along a line path are then determined by sampling at unit x intervals. Starting from the left endpoint (x_0, y_0) of a given line, we step to each successive column (x position) and plot the pixel whose scan-line y value is closest to the line path. Figure 3-7 demonstrates the k^{th} step in this process. Assuming we have determined that the pixel at (x_k, y_k) is to be displayed, we next need to decide which pixel to plot in column x_k+1. Our choices are the pixels at positions (x_k+1, yk) and (x_k+l, y_k+l).

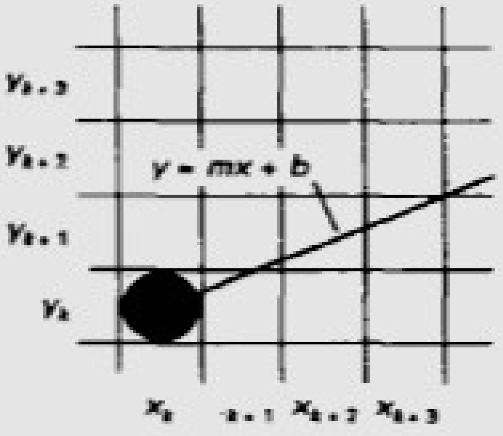
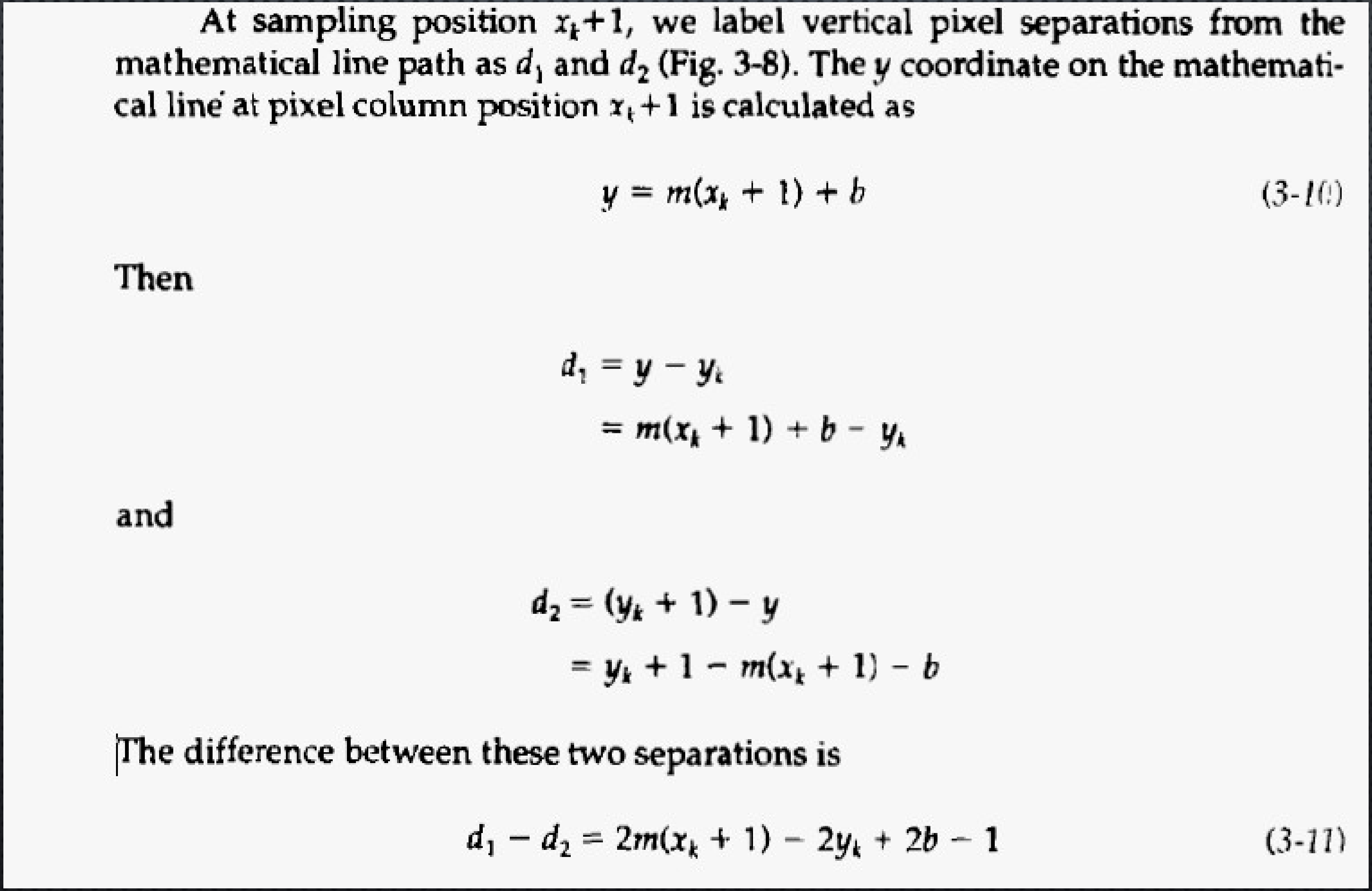


Figure 3-7
Section of the screen grid showing a pixel in column x_k on scan line y_k that is to be plotted along the path of a line segment with slope $0 < m < 1$.

Bresenham's Line Algorithm :



At sampling position x_k+1, we label vertical pixel separations from the mathematical line path as d_1 and d_2 (Fig. 3-8). The y coordinate on the mathematical line at pixel column position x_k+1 is calculated as

$$y = m(x_k + 1) + b \tag{3-10}$$

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

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

Bresenham's Line Algorithm :

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 = Δy / Δx, 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 d1 – d2, since* $\Delta x > 0$ *for our example. Parameter* c is constant and has the value 2Δy + Δx(2b - l), 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, d1 < d2), then decision parameter p_k is negative. In that case, we plot the lower pixel; otherwise, we plot the upper pixel.

Bresenham's Line Algorithm :

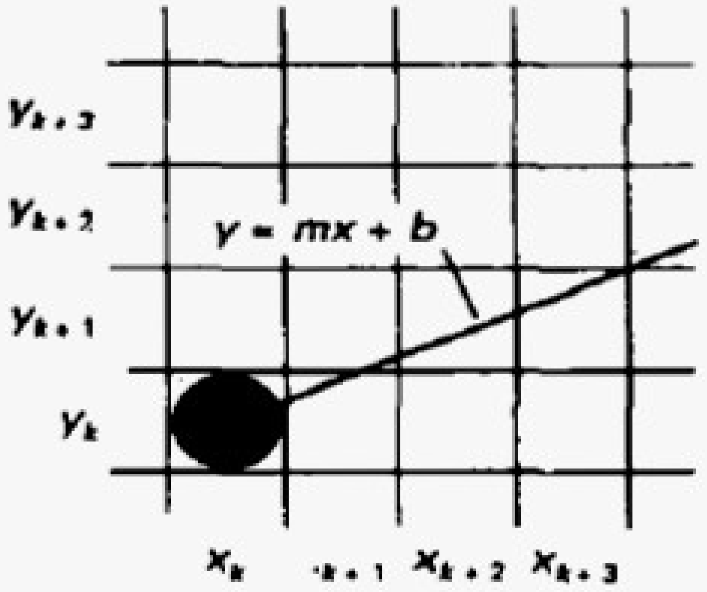


Figure 3-7
Section of the screen grid showing a pixel in column x_k on scan line y_k that is to be plotted along the path of a line segment with slope $0 < m < 1$.

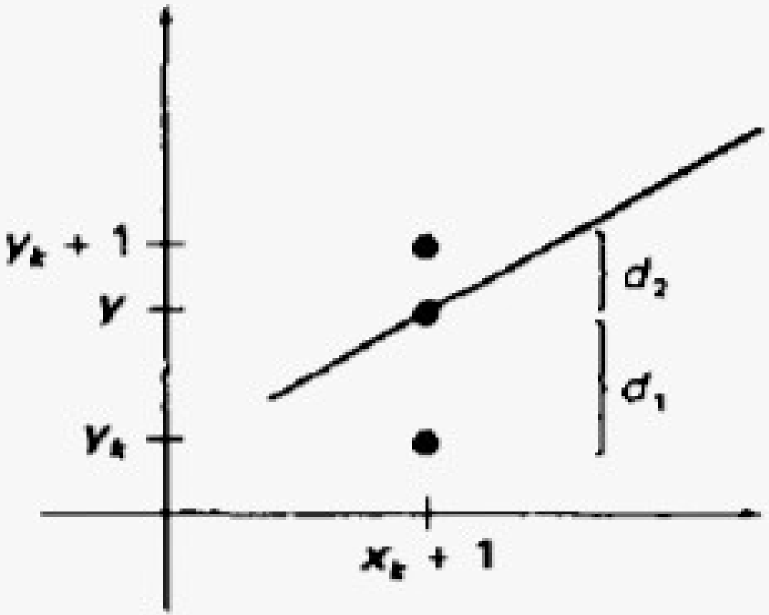
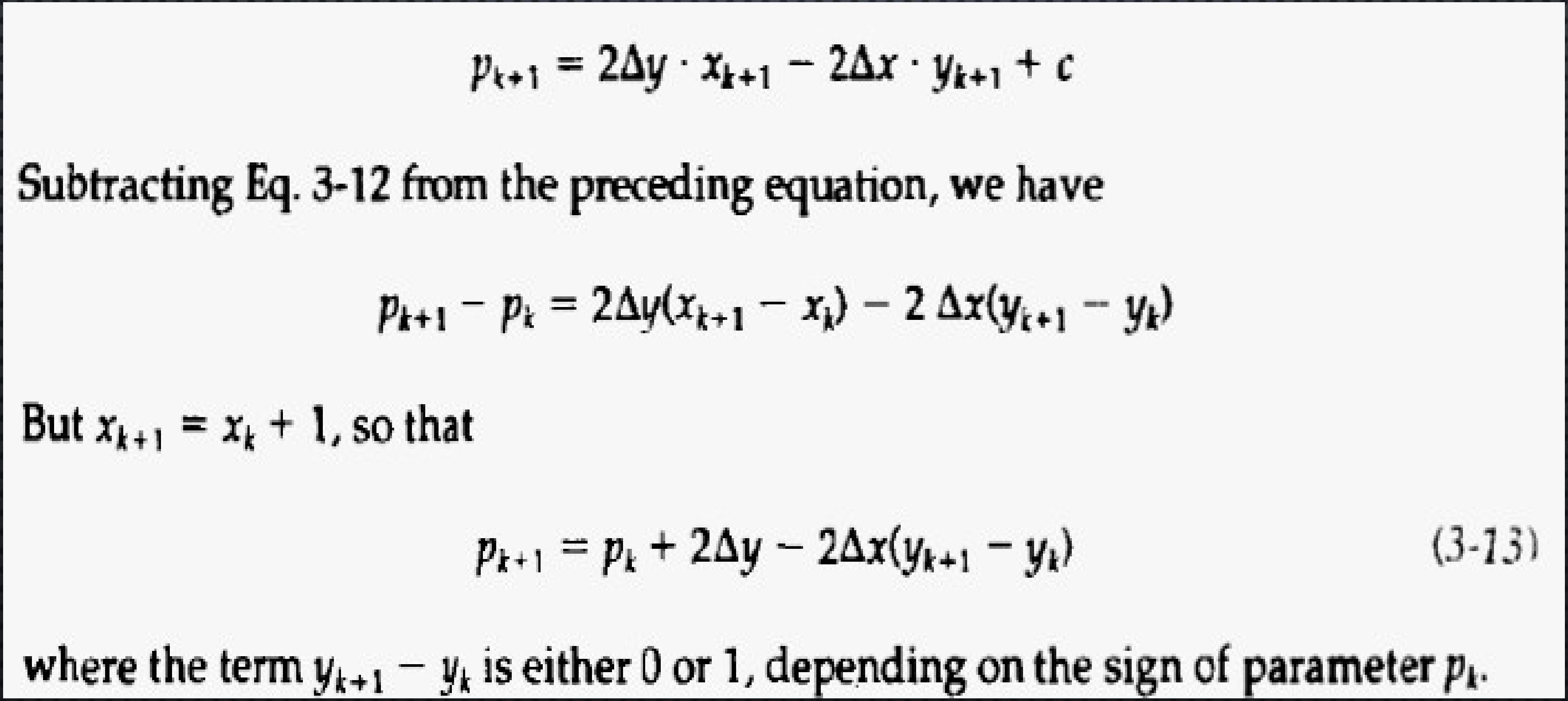


Figure 3-8
Distances between pixel positions and the line y coordinate at sampling position $x_k + 1$.

Bresenham's Line Algorithm :

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

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

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

Bresenham's Line Algorithm :

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 $\Delta y / \Delta x$:

$$P_0 = 2\,\Delta y - \Delta x \qquad \text{........ (3.14)}$$

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 Algorithm :

$P_k = \Delta X\,(d1 - d2)$

$= \Delta X\,(2\,m(X_k + 1) - 2\,Y_k + 2b - 1\,)$

$= \Delta X\,(2mX_k + 2m - 2\,Y_k + 2b - 1\,)$

$= \Delta X\,(2mX_0 + 2m - 2\,Y_0 + 2b - 1\,)$ [put k=0]

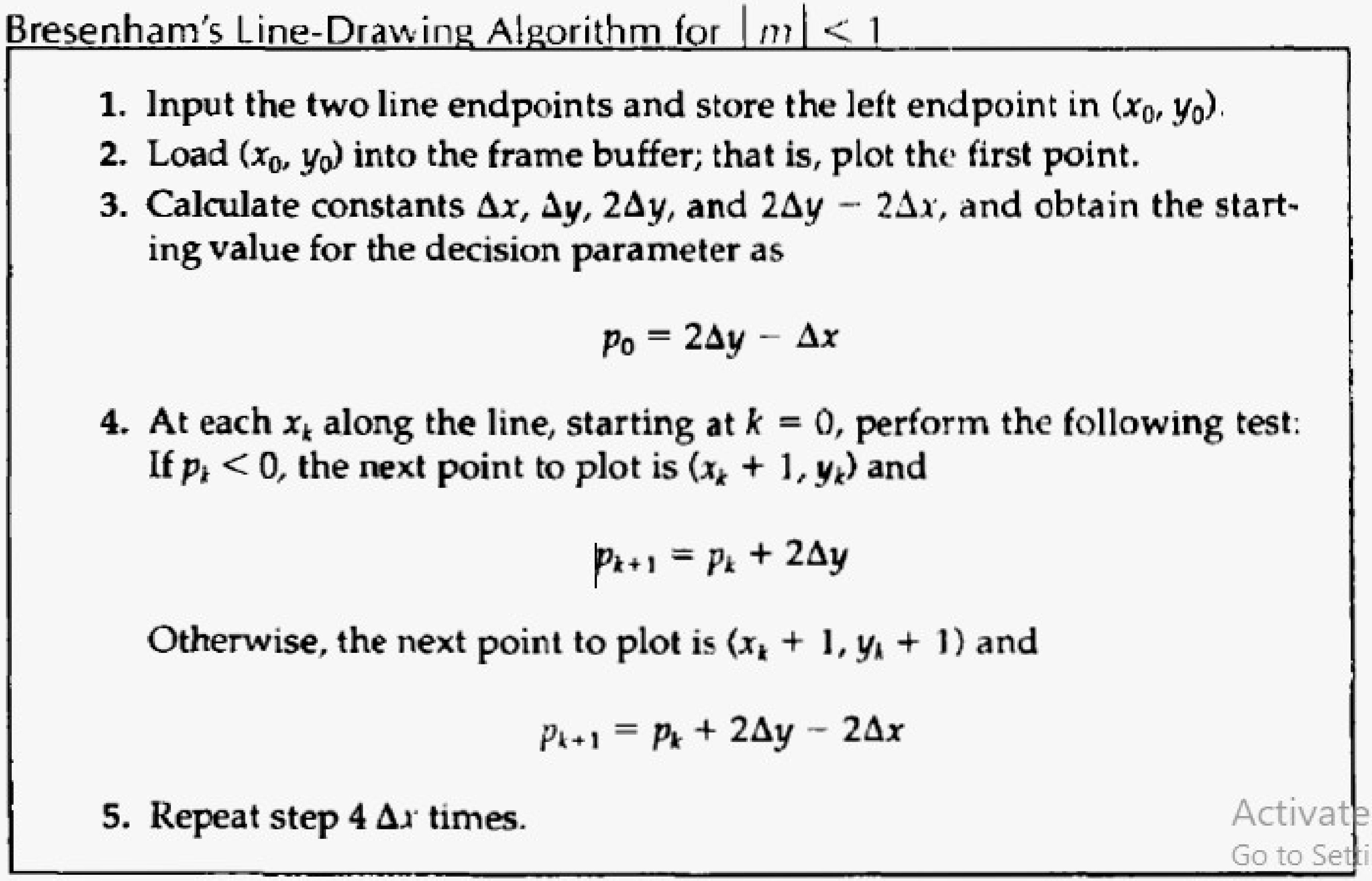
$= \Delta X\,(2(mX_0 + b - Y_0\,) + 2m - 1\,)$

$= \Delta X(2m - 1)$ (Because $mX_0 + b - Y_0 = 0$)

$= \Delta X\,(2\,\Delta Y / \Delta X - 1)$

$P_0 = 2\,\Delta y - \Delta X$ [decision parameter]

Bresenham's Line Algorithm :



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.

Bresenham's Line Algorithm :

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

$$p_0 = 2\Delta y - \Delta x$$
$$= 6$$

and the increments for calculating successive decision parameters are

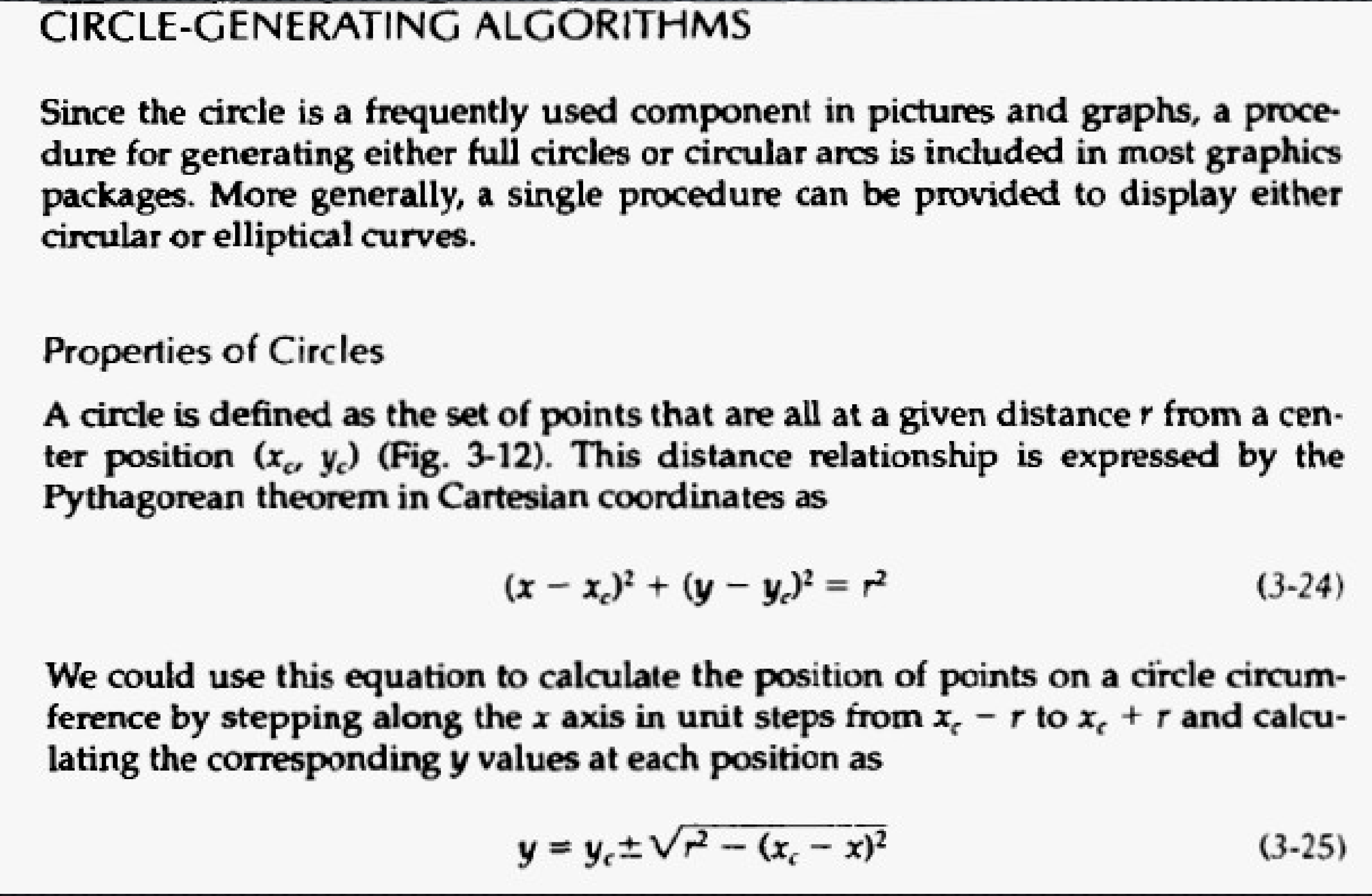
$$2\Delta y = 16, \qquad 2\Delta y - 2\Delta x = -4$$

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)

A plot of the pixels generated along this line path is shown in Fig. 3-9.

Circle-Generating Algorithms:



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.

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

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

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

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

Circle-Generating Algorithms:

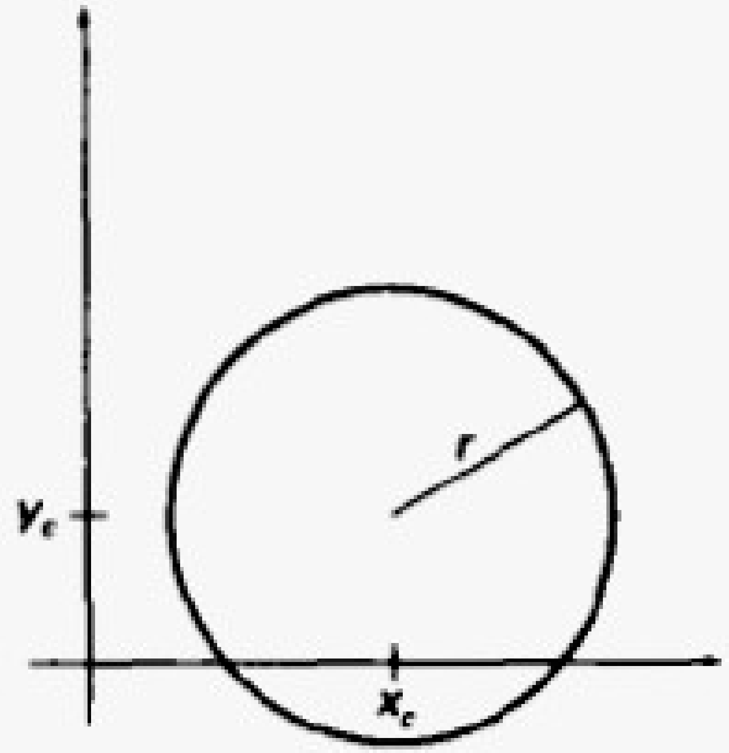


Figure 3-12
Circle with center coordinates (x_c, y_c) and radius r.

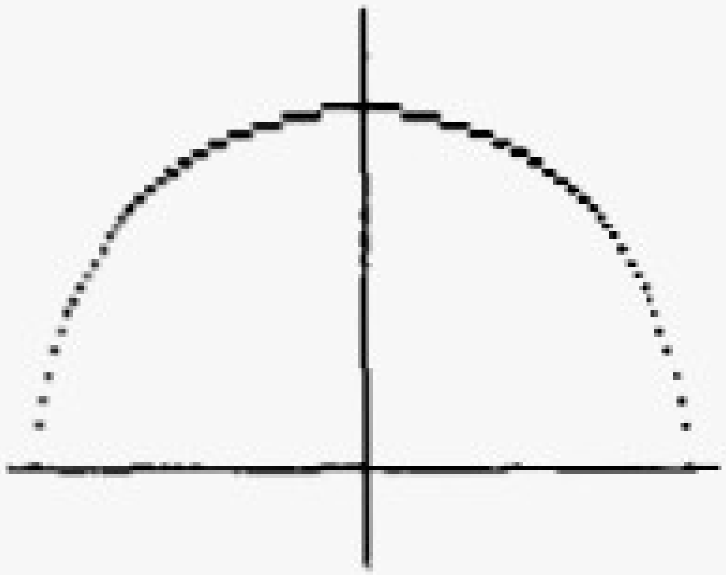


Figure 3-13
Positive half of a circle plotted with Eq. 3-25 and with $(x_c, y_c) = (0, 0)$.

Circle-Generating Algorithms:

But this is not the best method for generating a circle. One problem with this approach is that it involves considerable computation at each step. Moreover, the spacing between plotted pixel positions is not uniform, as demonstrated in Fig. 3-13. 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 θ (Fig.3-12). Expressing the circle equation in parametric polar form yields the pair of equations

$$\begin{aligned} x &= x_c + r\cos\theta \\ y &= y_c + r\sin\theta \end{aligned} \qquad (3\text{-}26)$$

Circle-Generating Algorithms:

When a display is generated with these equations using a fixed angular step size, a circle is plotted with equally spaced points along the circumference. The step size chosen for θ depends on the application and the display device. Larger angular separations along the circumference can be connected with straight line segments to approximate the circular path. For a more continuous boundary on a raster display, we can set the step size at 1/r. This plots pixel positions that are approximately one unit apart.

Circle-Generating Algorithms:

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 the second quadrant of the xy plain by noting that the two circle sections are symmetric with respect to the y 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 octants within one quadrant are symmetric with respect to the 45^0 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 symmetry in this way we can generate all pixel positions around a circle by calculating only the points within the sector from $x = 0$ to $x = y$.

Circle-Generating Algorithms:

Determining pixel positions along a circle circumference using either Eq. 3-24 or Eq. 3-26 still requires a good deal of computation time. The Cartesian equation 3-24 involves multiplications and square-root calculations, while the parametric equations contain multiplications and trigonometric calculations.
More efficient circle algorithms are based on incremental calculation of decision parameters, as in the Bresenham line algorithm, which involves only simple integer operations,

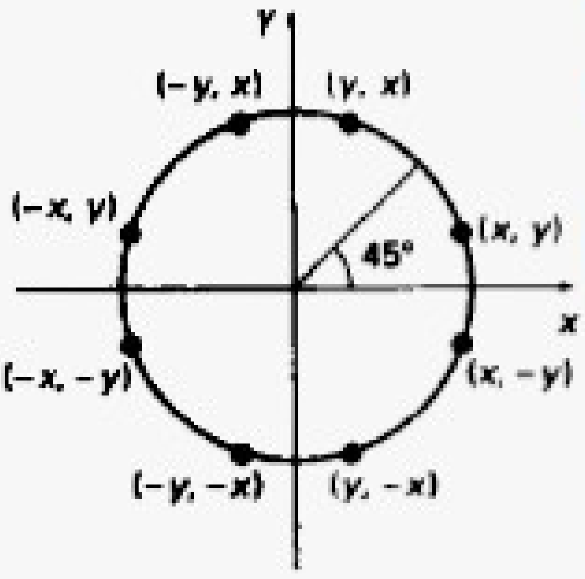


Figure 3-14
Symmetry of a circle. Calculation of a circle point (x, y) in one octant yields the circle points shown for the other seven octants.

Circle-Generating Algorithms:

Bresenham's line algorithm for raster displays is adapted to circle generation by setting up decision parameters for finding the closest pixel to the circumference at each sampling step. The circle equation 3-24, however, is nonlinear, so that square-root evaluations would be required to compute pixel distances from a circular path. Bresenham's circle algorithm avoids these square-root calculations by comparing the squares of the pixel separation distances.

A method for direct distance comparison is to test the halfway position between two pixels to determine if this midpoint is inside or outside the circle boundary. This method is more easily applied to other conics; and for an integer circle radius, the midpoint approach generates the same pixel positions as the Bresenham circle algorithm. Also, the error involved in locating pixel positions along any conic section using the midpoint test is limited to one-half the pixel separation.

Midpoint Circle Algorithm:

As in the raster line algorithm, we sample at unit intervals and determine the closest pixel position to the specified circle path at each step. For a given radius r and screen center position (x_c, y_c), we can first set up our algorithm to calculate pixel positions around a circle path centered at the coordinate origin (0,0). Then each calculated position (x, y) is moved to its proper screen position by adding x_c to x and y_c to y.

Along the circle section from x = 0 to x = y in the first quadrant, the slope of the curve varies from 0 to -1. Therefore, we can take unit steps in the positive x direction over this octant and use a decision parameter to determine which of the two possible y positions is closer to the circle path at each step.
Positions of the other seven octants are then obtained by symmetry.

Midpoint Circle Algorithm:

To apply the midpoint method, we define a circle function:

$$f_{circle}(x, y) = x^2 + y^2 - r^2 \quad \ldots\ldots\ldots\ldots \quad (3\text{-}27)$$

Any point (x , y) on the boundary of the circle with radius r satisfies the equation $f_{circle}(x, y) = 0$. If the point is in the interior of the circle, the circle function is negative. And if the point is outside the circle, the circle function is positive.
To summarize, the relative position of any point (x. y) can be determined by checking the sign of the circle function:

Midpoint Circle Algorithm:

$$f_{circle}(x, y) \begin{cases} < 0, & \text{if } (x, y) \text{ is inside the circle boundary} \\ = 0, & \text{if } (x, y) \text{ is on the circle boundary} \\ > 0, & \text{if } (x, y) \text{ is outside the circle boundary} \end{cases} \qquad (3\text{-}28)$$

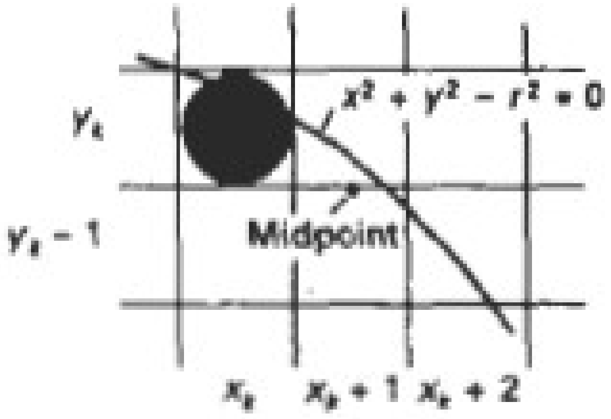


Figure 3-15
Midpoint between candidate pixels at sampling position x_k+1 along a circular path.

Midpoint Circle Algorithm:

The circle-function tests in 3-28 are performed for the mid positions between pixels near the circle path at each sampling step. Thus, the circle function is the decision parameter in the midpoint algorithm, and we can set up incremental calculations for this function as we did in the line algorithm.

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:

Midpoint Circle Algorithm:

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

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 $x_{k+1} + 1 = x_k + 2$:

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

or

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

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

Now find successive decision parameter (P_{k+1})

Replace k as k+1 in eq. no (3)

$P_{k+1} = (X_{k+1}+1)^2 + (Y_{k+1} - ½)^2 - r^2$

$P_{k+1} = (X_k + 2)^2 + (Y_{k+1} - ½)^2 - r^2$ ...(4)

Now,

$P_{k+1} - P_k = [(X_k + 2)^2 + (Y_{k+1} - ½)^2 - r^2] - [(X_k+1)^2 + (Y_k - ½)^2 - r^2]$

$P_{k+1} = P_k + 2(X_k + 1) + (Y^2_{k+1} - Y^2_k) - (Y_{k+1} - Y_k) + 1$ ...(5)

Where Y_{k+1} is either Y_k or Y_{k-1}

- if $P_k < 0$ then midpoint is inside the circle and pixel on scan line Y_k is closer to the circle boundary. So select upper pixel. i.e. Y_k

Then from eq. no (5)

$$P_{k+1} = P_k + 2(X_k + 1) + (Y^2_{k+1} - Y^2_k) - (Y_{k+1} - Y_k) + 1$$

$$P_{k+1} = P_k + 2(X_k + 1) + (Y^2_k - Y^2_k) - (Y_k - Y_k) + 1$$

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

First decision parameter coordinate at (0,r)

$P_0 = f_{circle}(1, r – ½)$

$P_0 = (0 + 1)^2 + (r – ½)^2 –r^2$

$P_0 = 5/4 – r$

but Bresenham algorithm involve only integer

Midpoint Circle Algorithm:

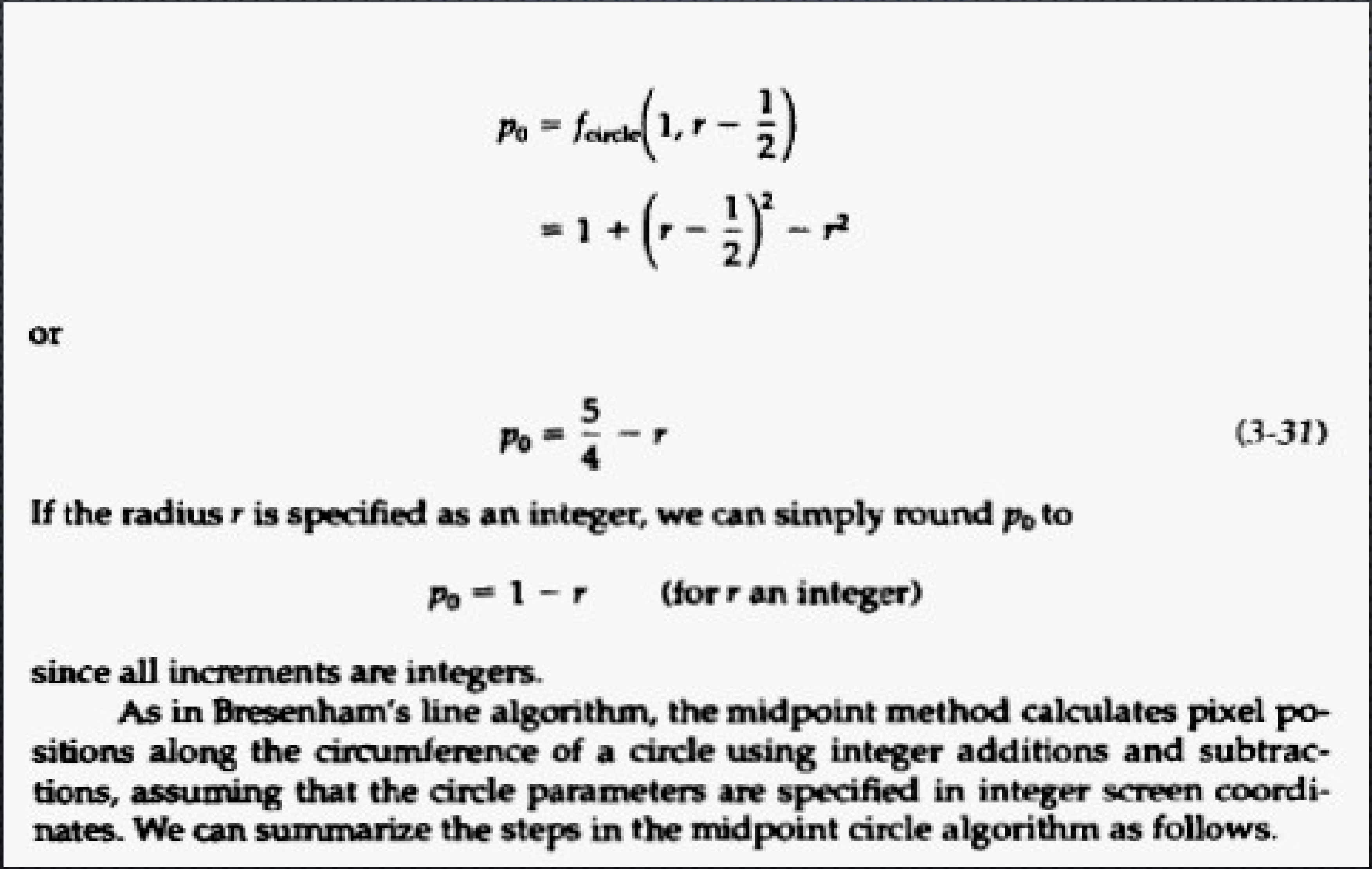
Increments for obtaining p_{k+1} are either $2x_{k+1} + 1$ (if p_k is negative) or $2x_{k+1} + 1 - 2y_{k+1}$. Evaluation of the terms $2x_{k+1}$ and $2y_{k+1}$ can also be done incrementally as

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

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

Midpoint Circle Algorithm:



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

or

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

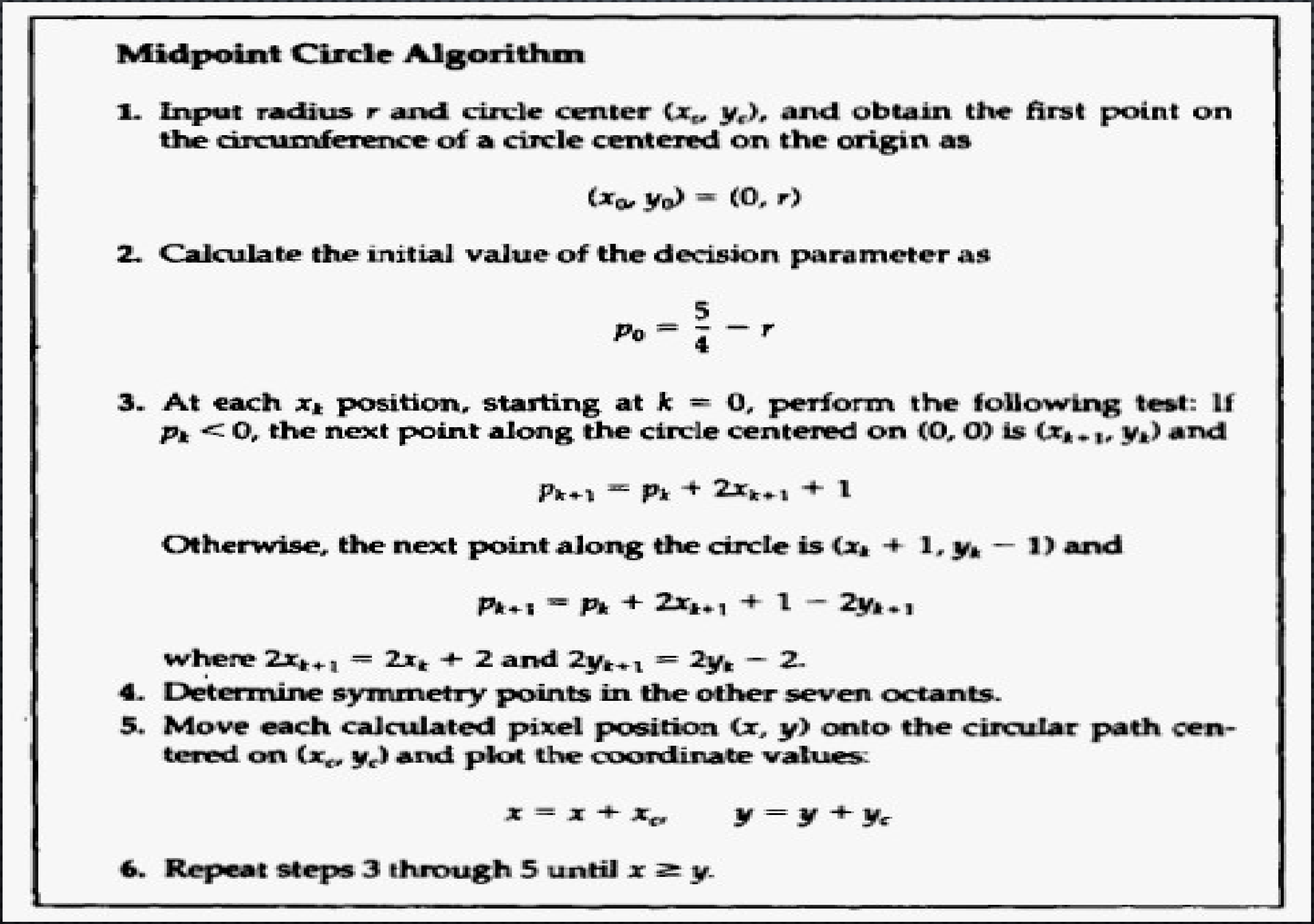
If the radius r is specified as an integer, we can simply round p_0 to

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

since all increments are integers.

As in Bresenham's line algorithm, the midpoint method calculates pixel positions along the circumference of a circle using integer additions and subtractions, assuming that the circle parameters are specified in integer screen coordinates. We can summarize the steps in the midpoint circle algorithm as follows.

Midpoint Circle Algorithm:



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

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

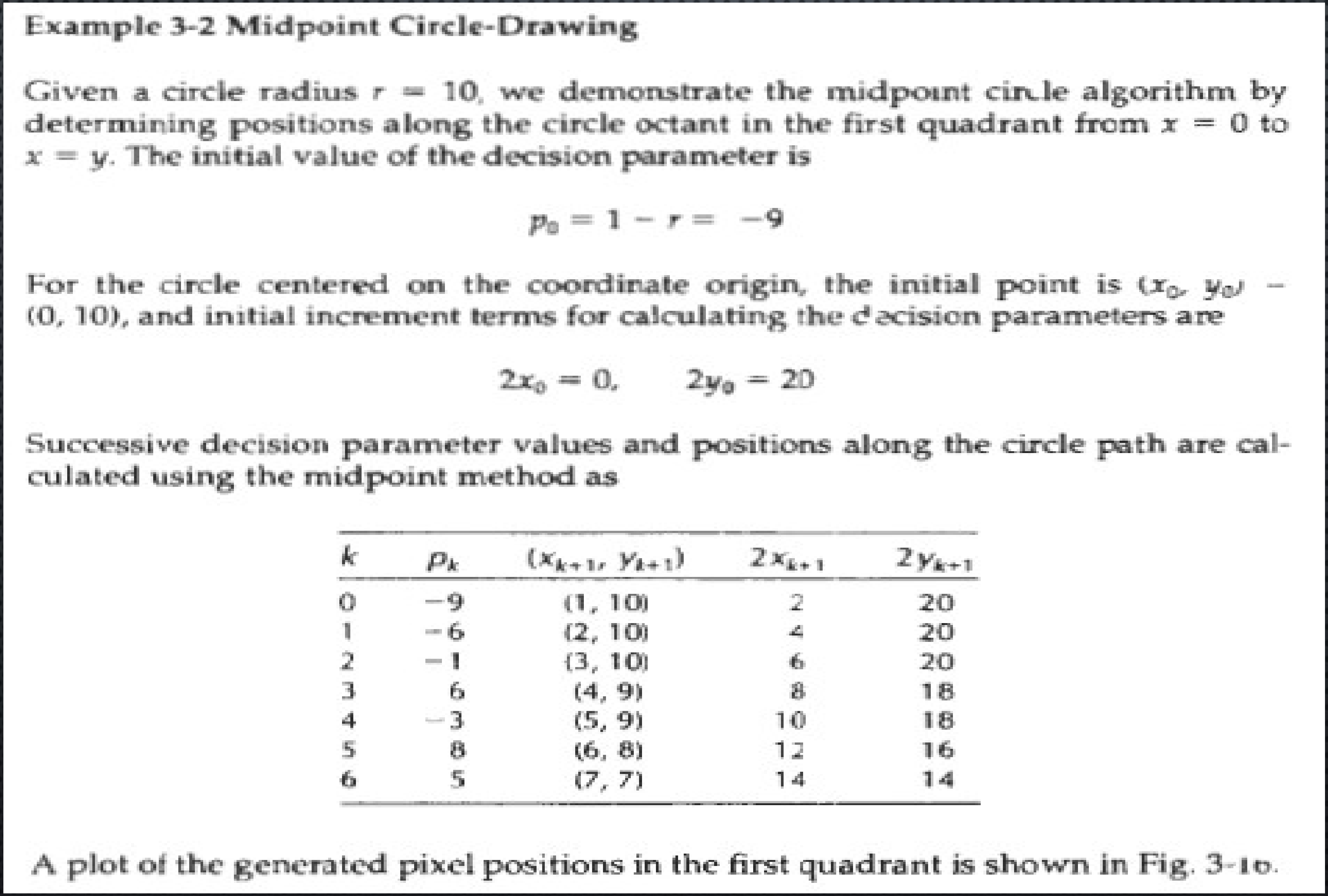
where $2x_{k+1} = 2x_k + 2$ and $2y_{k+1} = 2y_k - 2$.

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 $x \geq y$.

Midpoint Circle Algorithm:



Example 3-2 Midpoint Circle-Drawing

Given a circle radius $r = 10$, we demonstrate the midpoint circle 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.

Midpoint Circle Algorithm:

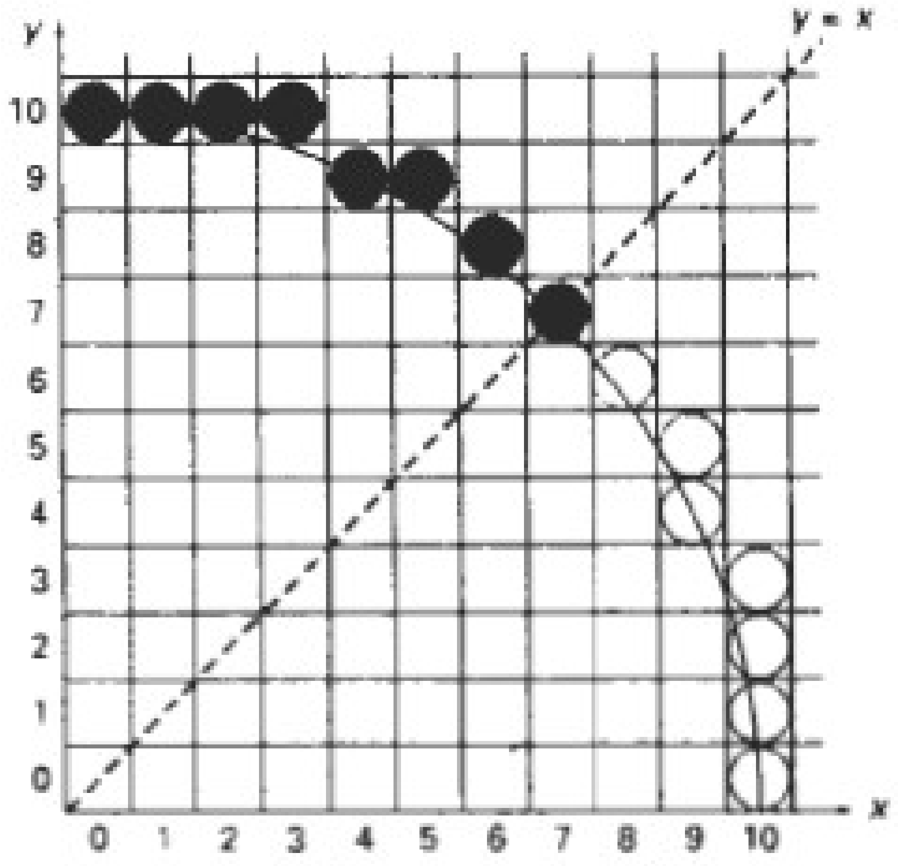


Figure 3-16
Selected pixel positions (solid circles) along a circle path with radius $r = 10$ centered on the origin, using the midpoint circle algorithm. Open circles show the symmetry positions in the first quadrant.

Character Generation:

Letters, numbers, and other characters can be displayed in a variety of sizes and styles. The overall design style for a set (or family) of characters is called a type face. Today, there are hundreds of typefaces available for computer applications.

Examples of a few common typefaces are Courier, Helvetica, New York, Palatino, and Zapf Chancery. Originally, the term font referred to a set of cast metal character forms in a particular size and format, such as 10-point Courier Italic or 12-point Palatino Bold. Now, the terms font and typeface are often used interchangeably, since printing is no longer done with cast metal forms.

Character Generation:

Typefaces (or fonts) can be divided into two broad groups: serif and sans serif. Serif type has small lines or accents at the ends of the main character strokes, while sans-serif type does not have accents. For example, the text in this book is set in a serif font (Palatino). But this sentence is printed in a sans-serif font (Optima).

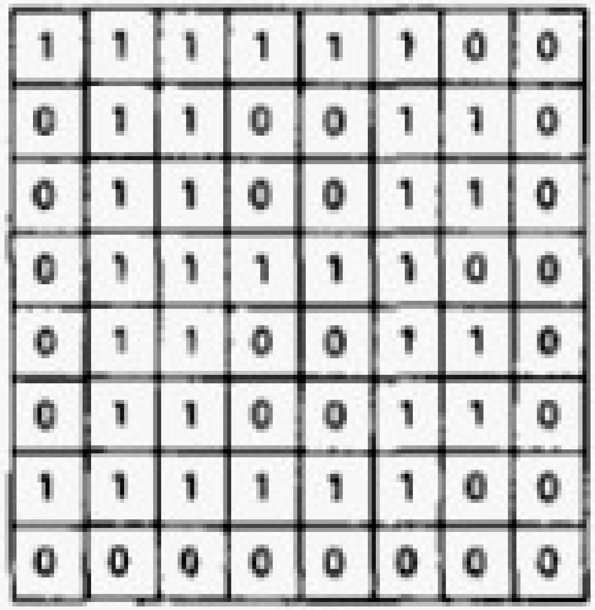
Serif type is generally more readable; that is, it is easier to read in longer blocks of text. On the other hand, the individual characters in sans-serif type are easier to recognize. For this reason, sans-serif type is said to be more legible. Since sans-serif characters can be quickly recognized, this typeface is good for labeling and short headings.

Character Generation:

Two different representations are used for storing computer fonts. A simple method for representing the character shapes in a particular typeface is to use rectangular grid patterns. The set of characters are then referred to as a bitmap font (or bitmapped font).

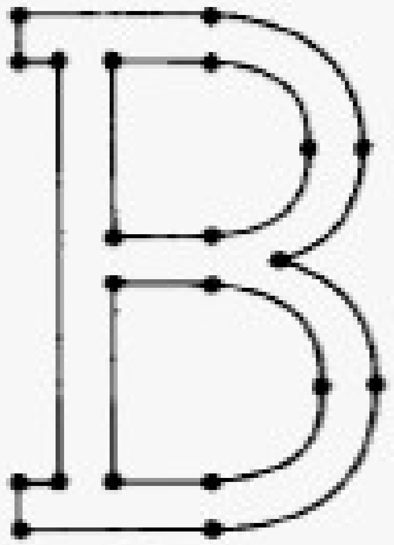
Another, more flexible, scheme is to describe character shapes using straight-line and curve sections, as in PostScript, for example. In this case, the set of characters is called an outline font. Figure 3-48 illustrates the two methods for character representation. When the pattern in Fig. 3-48(a) is copied to an area of the frame buffer, the 1 bits designate which pixel positions are to be displayed on the monitor.

Character Generation:



1	1	1	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
0	1	1	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0

(a)



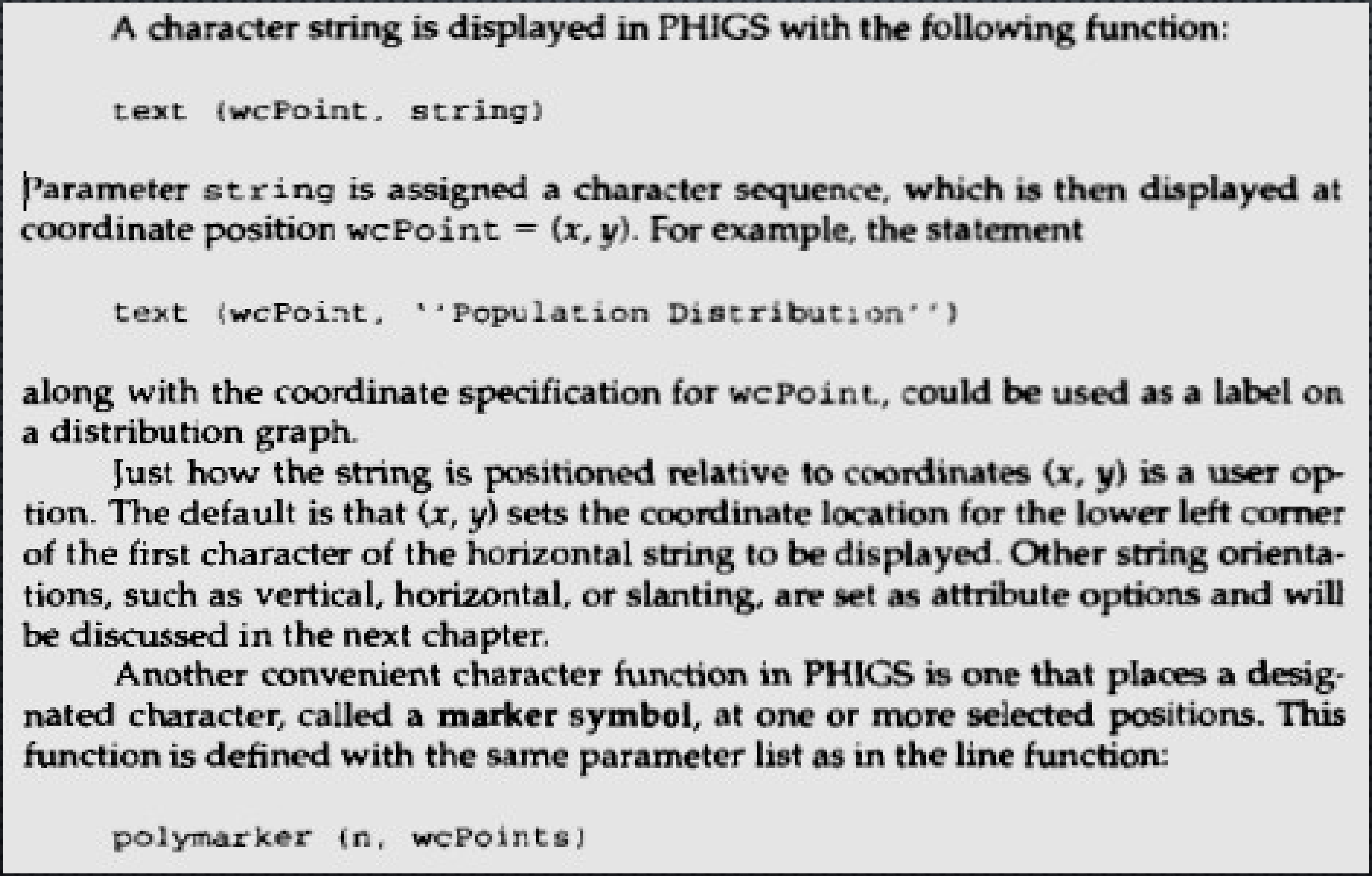
(b)

Figure 3-48
The letter B represented in (a) with an 8 by 8 bilevel bitmap pattern and in (b) with an outline shape defined with straight-line and curve segments.

Character Generation:

Bitmap fonts are the simplest to define and display: The character grid only needs to be mapped to a frame-buffer position. In general, however, bitmap fonts require more space, because each variation (size and format) must be stored in a font cache. It is possible to generate different sizes and other variations, such as bold and italic, from one set, but this usually does not produce good results.
In contrast to bitmap fonts, outline fonts require less storage since each variation does not require a distinct font cache. We can produce boldface, italic, or different sizes by manipulating the curve definitions for the character outlines. But it does take more time to process the outline fonts, because they must be scan converted into the frame buffer.

Character Generation:



A character string is displayed in PHIGS with the following function:

```
text (wcPoint, string)
```

Parameter `string` is assigned a character sequence, which is then displayed at coordinate position `wcPoint` = (x, y). For example, the statement

```
text (wcPoint, ''Population Distribution'')
```

along with the coordinate specification for `wcPoint`, could be used as a label on a distribution graph.

Just how the string is positioned relative to coordinates (x, y) is a user option. The default is that (x, y) sets the coordinate location for the lower left corner of the first character of the horizontal string to be displayed. Other string orientations, such as vertical, horizontal, or slanting, are set as attribute options and will be discussed in the next chapter.

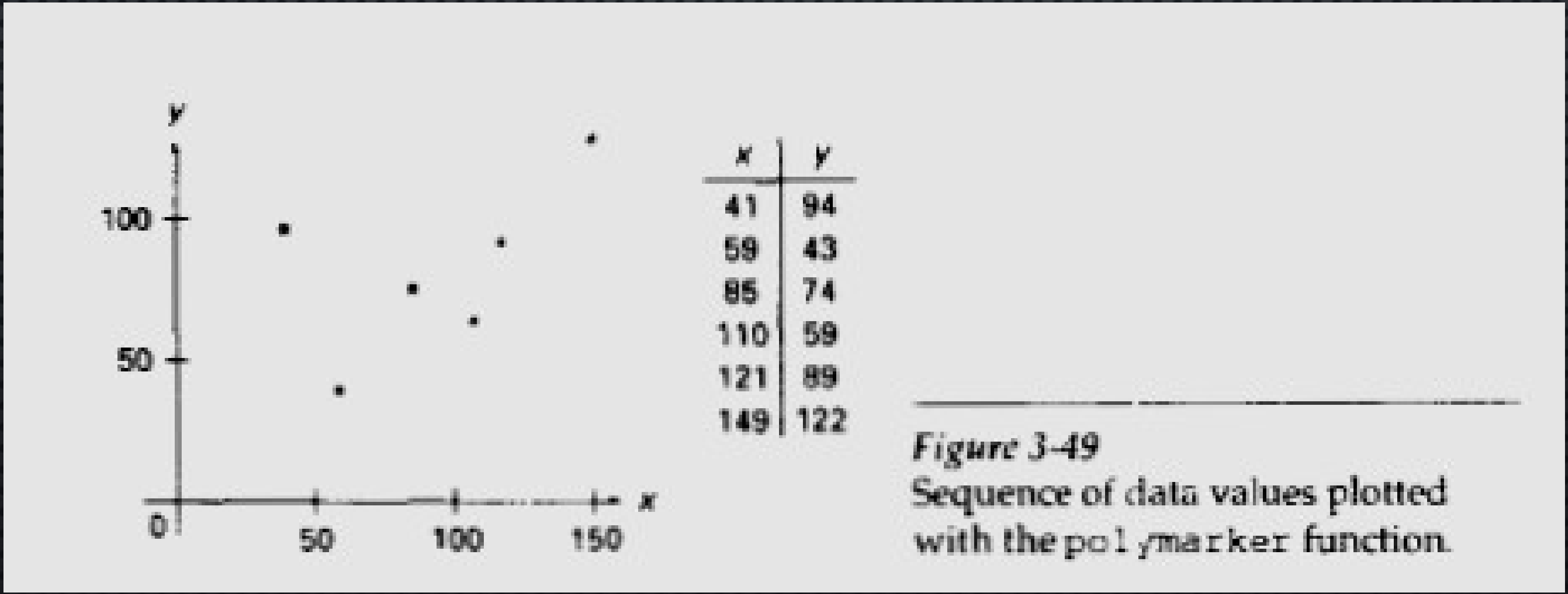
Another convenient character function in PHIGS is one that places a designated character, called a **marker symbol**, at one or more selected positions. This function is defined with the same parameter list as in the line function:

```
polymarker (n, wcPoints)
```

Character Generation:

A predefined character is then centered at each of the n coordinate positions in the list wcpoints. The default symbol displayed by polymarker depends on the particular implementation, but we assume for now that an asterisk is to be used. Figure 3-49 illustrates plotting of a data set with the statement

polymarker (6, wcpoints)



x	y
41	94
59	43
85	74
110	59
121	89
149	122

Figure 3-49
Sequence of data values plotted with the polymarker function.

Attributes of Output Primitives

Any parameter that affects the way a primitive is to be displayed is referred to as an attribute parameter. Some attribute parameters, such as color and size, determine the fundamental characteristics of a primitive. Others specify how the primitive is to be displayed under special conditions. Examples of attributes in this class include depth information for three-dimensional viewing and visibility or detectability options for interactive object-selection programs.

Here, we consider only those attributes that control the basic display properties of primitives, without regard for special situations. For example, lines can be dotted or dashed, fat or thin, and blue or orange. Areas might be filled with one color or with a multicolor pattern. Text can appear reading from left to right, slanted diagonally across the screen, or in vertical columns. Individual characters can be displayed in different fonts, colors, and sizes. And we can apply intensity variations at the edges of objects to smooth out the raster stairstep effect.

Attributes of Output Primitives

- With GKS and PHIGS standards, attributes settings are accomplished with separate functions that updates a system attribute list.
- **Four types of attributes:**

1. **Line Attribute**

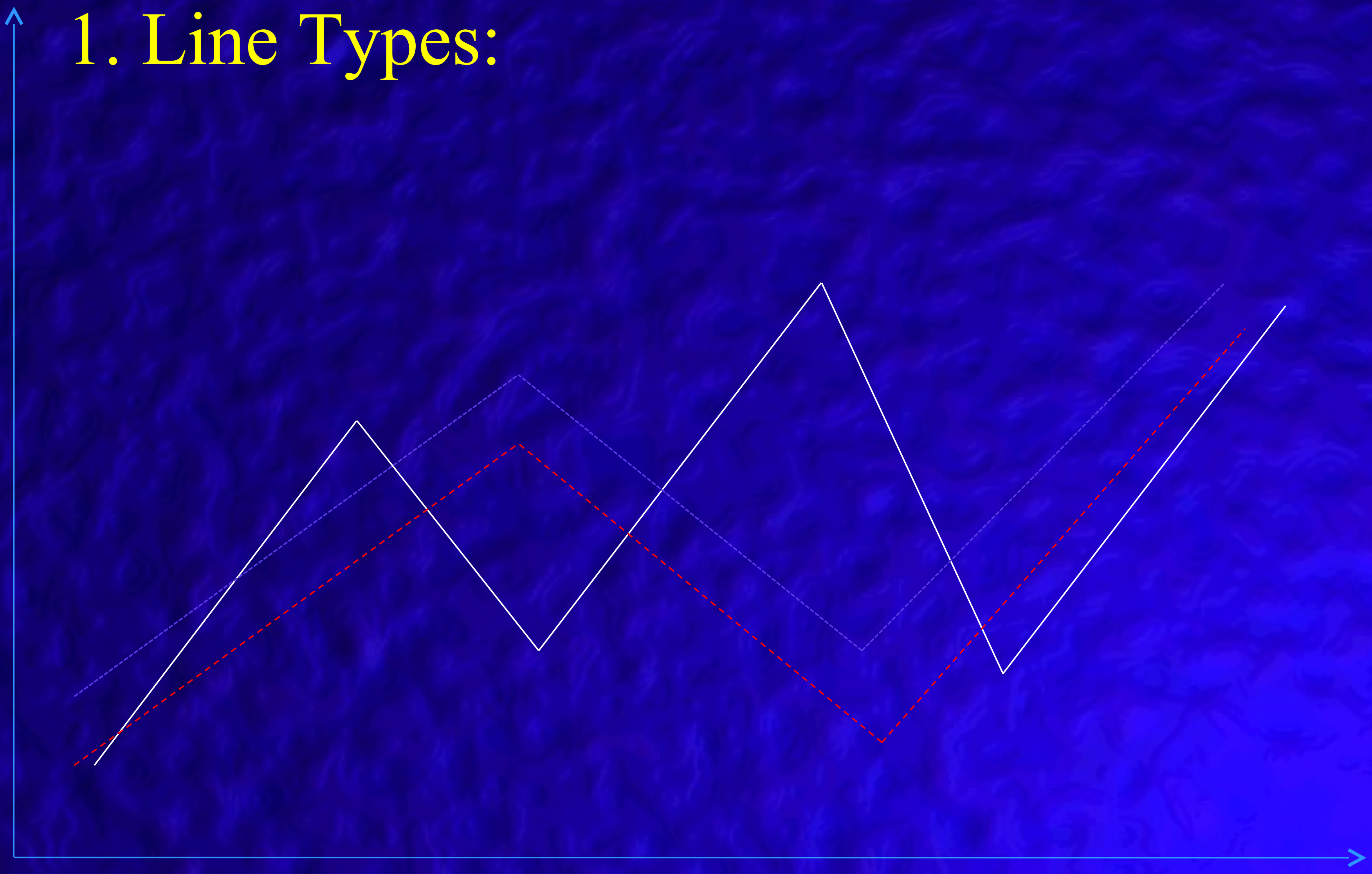
Line Attribute

There are three basic Line Attributes :

1. **Line type (Solid, Dashed, Dotted)**
2. **Line width**
3. **Line color**

A dashed line could be displayed by generating an interdash spacing that is equal to the length of the solid sections. Both the length of the dashes and the interdash spacing are often specified as user options. **A dotted line can be displayed by** generating very short dashes with the spacing equal to or greater than the dash size. Similar methods are used to produce other line-type variations.

1. Line Types:



2. Line Width :

- The implementation of line width option depends on the capabilities of the output devices.
- In Raster Scan, a standard width is generated with single pixel at each sample position.
- In raster scan, the lines of width > 1 are drawn by plotting the additional pixel which are adjacent to the line.
- If the width is 2, the line with (x,y) values is drawn with one more line with (x,y+1) values

Problems:

1. The horizontal and vertical lines can be perfectly drawn, but the diagonal lines will be drawn thinner.
2. Another problem is that the shape of line at the ends is horizontal or vertical regardless of the slope of the line.

Solution:

We can adjust the shape of line ends by using the

Line caps: Three types of line caps.

1. **Butt cap**
2. **Round cap**
3. **Projection square cap**

Butt cap: Butt cap is obtained by adjusting the end position of the component parallel lines, so that the thick line is displayed with square ends that are perpendicular to the line path.

Round cap : Obtained by adding a filled

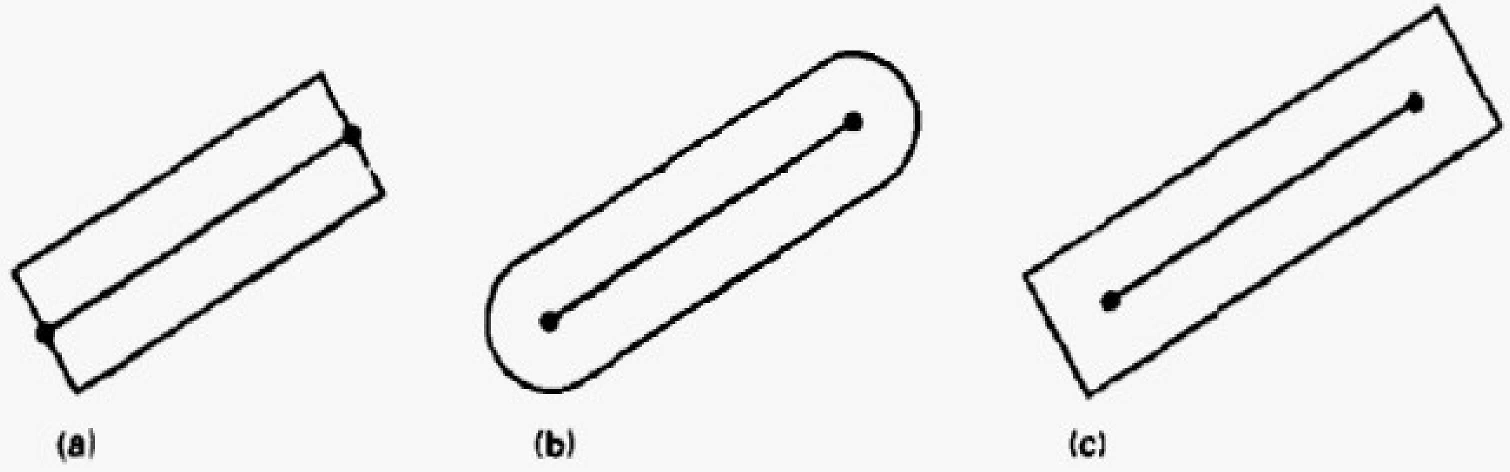
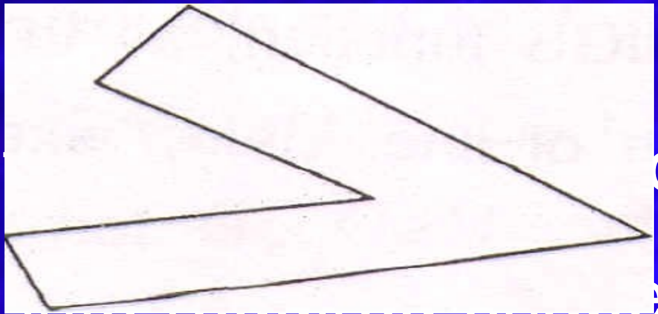


Figure 4-5
Thick lines drawn with (a) butt caps, (b) round caps, and (c) projecting square caps.

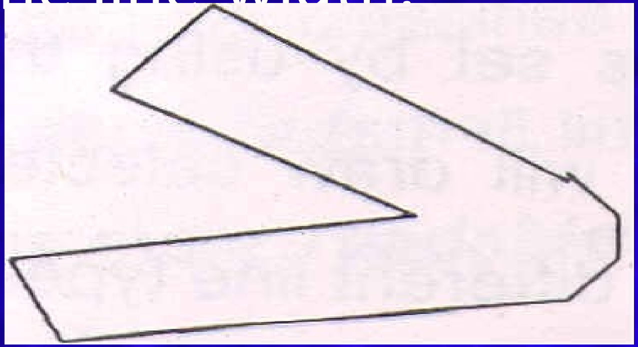
- Displaying thick lines using horizontal and vertical pixel spans, leaves pixel gaps at the boundaries between lines of different slopes where there is a shift from horizontal spans to vertical spans.



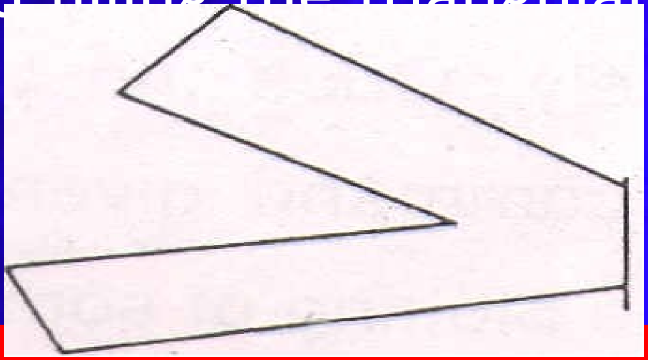
- When two thick lines are in o we may want to set the join into required shape. The **three types of joins** are as below:

  1. Miter join

2. Round join: It is produced by capping the connection between the two segments with a circular boundary whose diameter is equal to the line-width.



3. Bevel join: It is generated by displaying the line segment with butt caps and filling the triangular gap where the segments meet.



- **3. Line color :** When a system provides color (or intensity) options, a parameter giving the current color index is included in the list of system-attribute values. A polyline routine displays a line in the current color by setting this color value in the frame buffer at pixel locations along the line path using the **setpixel procedure.** **The** number of color choices depends on the number of bits available per pixel in the frame buffer.