3.3 SCAN-CONVERTING A CIRCLE

A circle is a symmetrical figure. Any circle-generating algorithm can take advantage of the circle's symmetry to plot eight points for each value that the algorithm calculates. Eight-way symmetry is used by reflecting each calculated point around each 45° axis. For example, if point 1 in Fig. 3-4 were calculated with a circle algorithm, seven more points could be found by reflection. The reflection is accomplished by reversing the x, y coordinates as in point 2, reversing the x, y coordinates and reflecting about the y axis as in point 3, reflecting about the y axis as in point 4, switching the signs of x and y as in point 5, reversing the x, y coordinates, reflecting about the y axis and reflecting about the x axis as in point 6, reversing the x, y coordinates and reflecting about the y axis as in point 7, and reflecting about the x axis as in point 8.

To summarize:

$$
\begin{aligned}
P_1 &= (x, y) & P_5 &= (-x, -y) \\
P_2 &= (y, x) & P_6 &= (-y, -x) \\
P_3 &= (-y, x) & P_7 &= (y, -x) \\
P_4 &= (-x, y) & P_8 &= (x, -y)
\end{aligned}
$$

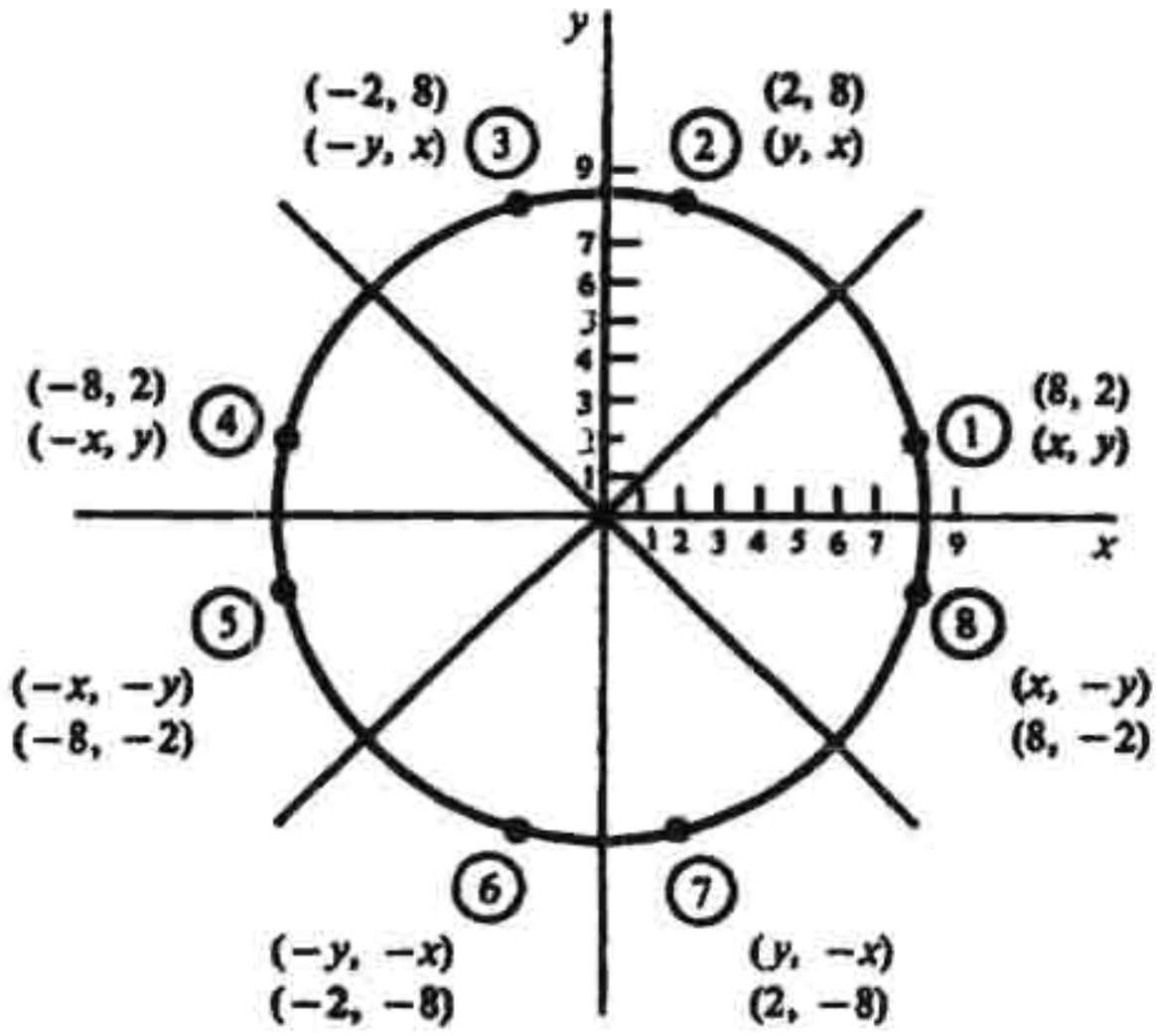

Fig. 3-4 Eight-way symmetry of a circle.

Defining a Circle

There are two standard methods of mathematically defining a circle centered at the origin. The first method defines a circle with the second-order polynomial equation (see Fig. 3-5)

$$y^2 = r^2 - x^3$$

where x = the x coordinate

y = the y coordinate

r = the circle radius

With this method, each x coordinate in the sector, from 90° to 45°, is found by stepping x from 0 to $r/\sqrt{2}$, and each y coordinate is found by evaluating $\sqrt{r^2 - x^2}$ for each step of x. This is a very inefficient method, however, because for each point both x and r must be squared and subtracted from each other; then the square root of the result must be found.

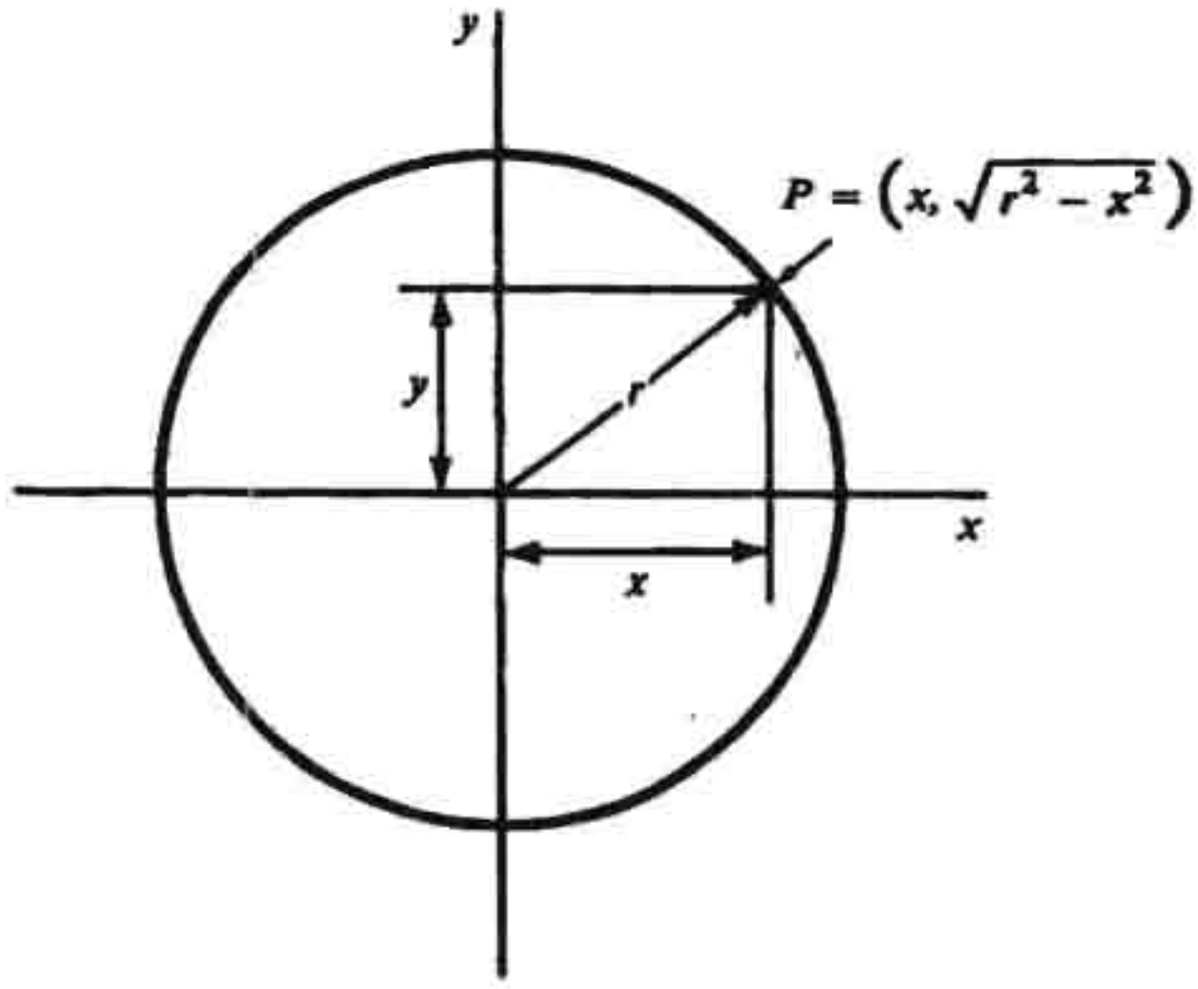

Fig. 3-5 Circle defined with a second-degree polynomial equation.

The second method of defining a circle makes use of trigonometric functios (see Fig. 3-6):

$$x = r\cos\theta \qquad y = r\sin\theta$$

where θ = current angle
r = circle radius
$x = x$ coordinate
$y = y$ coordinate

By this method, θ is stepped from θ to $\pi/4$, and each value of x and y is calculated. However, computation of the values of $\sin\theta$ and $\cos\theta$ is even more time-consuming than the calculations required by the first method.

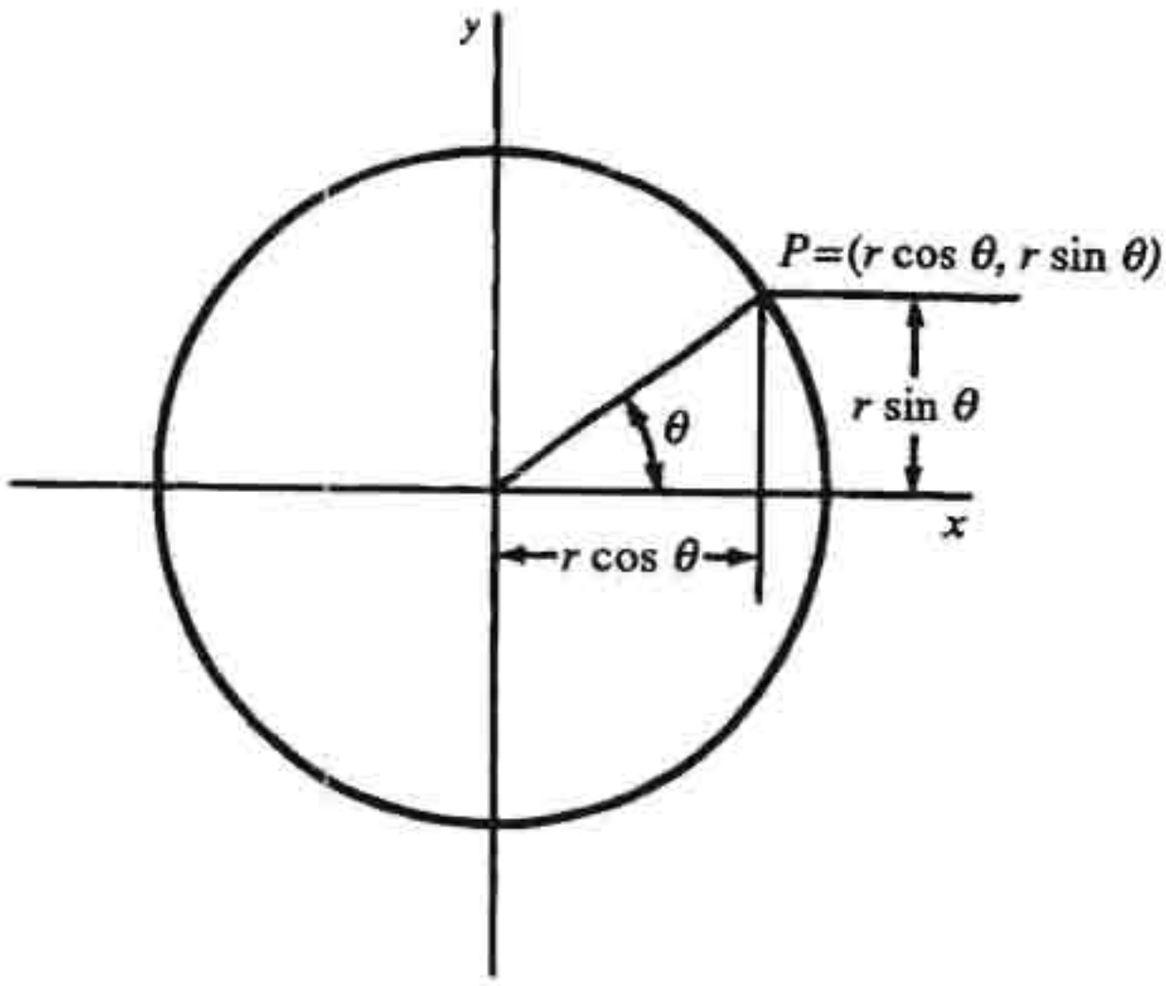

Fig. 3-6 Circle defined with trigonometric functions.

Bresenham's Circle Algorithm

If a circle is to be plotted efficiently, the use of trigonometric and power functions must be avoided. And, as with the generation of a straight line, it is also desirable to perform the calculations necessary to find the scan-converted points with only integer addition, subtraction, and multiplication by powers of 2. *Bresenham's circle algorithm* allows these goals to be met.

Scan-converting a circle using Bresenham's algorithm works as follows. If the eight-way symmetry of a circle is used to generate a circle, points will only have to be generated through a 45° angle. And, if points are generated from 90° to 45°, moves will be made only in the $+x$ and $-y$ directions (see Fig. 3-7).

The best approximation of the true circle will be described by those pixels in the raster that fall the least distance from the true circle. Examine Fig. 3-8. Notice that, if points are generated from 90° and 45°, each new point closest to the true circle can be found by taking either of two actions: (1) move in the x direction one unit or (2) move in the x direction one unit and move in the negative y direction one unit. Therefore, a method of selecting between these two choices is all that is necessary to find the points closest to the true circle.

Assume that (x_i, y_i) are the coordinates of the last scan-converted pixel upon entering step i (see Fig. 3-8). Let the distance from the origin to pixel T squared minus the distance to the true circle squared = D(T). Then let the distance from the origin to pixel S squared minus the distance to the true

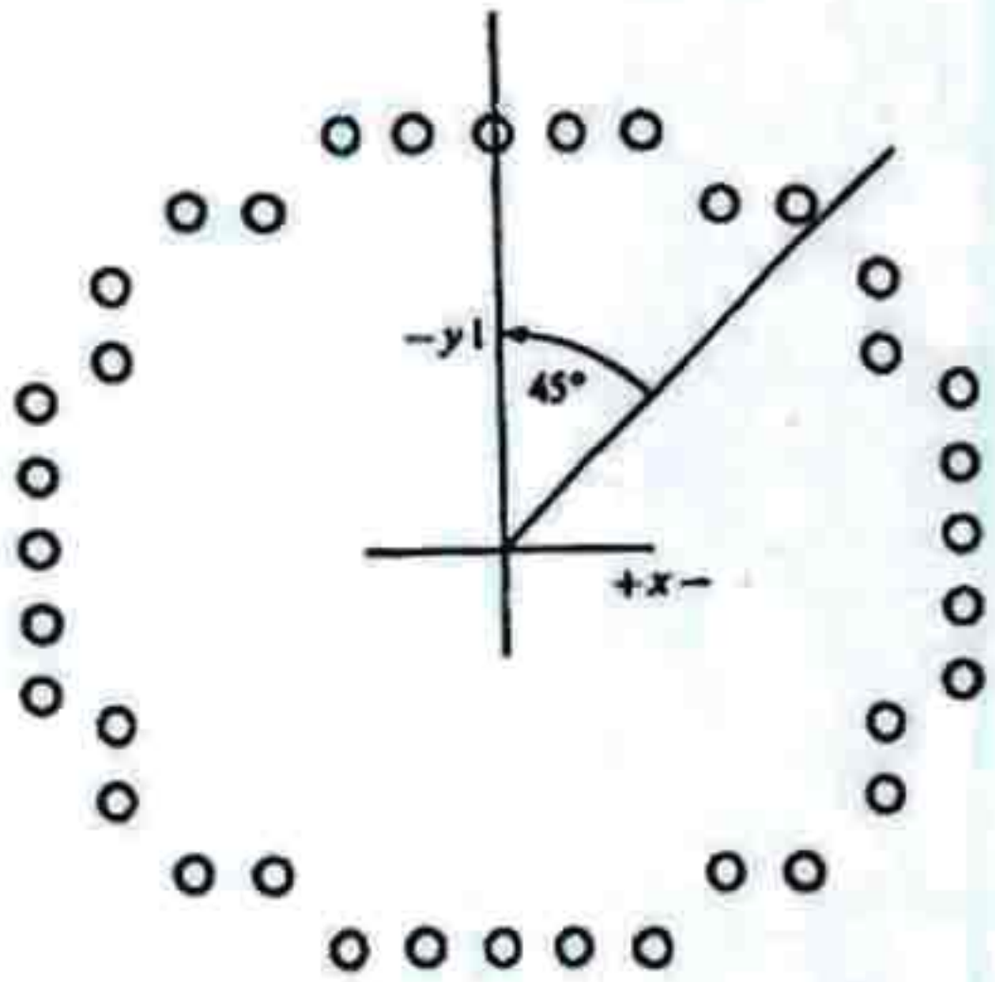

Fig. 3-7 Circle scan-converted with Bresenham's algorithm.

circle squared = $D(S)$. As the coordinates of T are $(x_i + 1, y_i)$ and those of S are $(x_i + 1, y_i - 1)$, the following expressions can be developed:

$$D(T) = (x_i + 1)^2 + y_i^2 - r^2 \qquad D(S) = (x_i + 1)^2 + (y_i - 1)^2 - r^2$$

This function D provides a relative measurement of the distance from the center of a pixel to the true circle. Since $D(T)$ will always be positive (T is outside the true circle) and $D(S)$ will always be negative (S is inside the true circle), a decision variable d_i may be defined as follows:

$$d_i = D(T) + D(S)$$

Therefore

$$d_i = 2(x_i + 1)^2 + y_i^2 + (y_i - 1)^2 - 2r^2$$

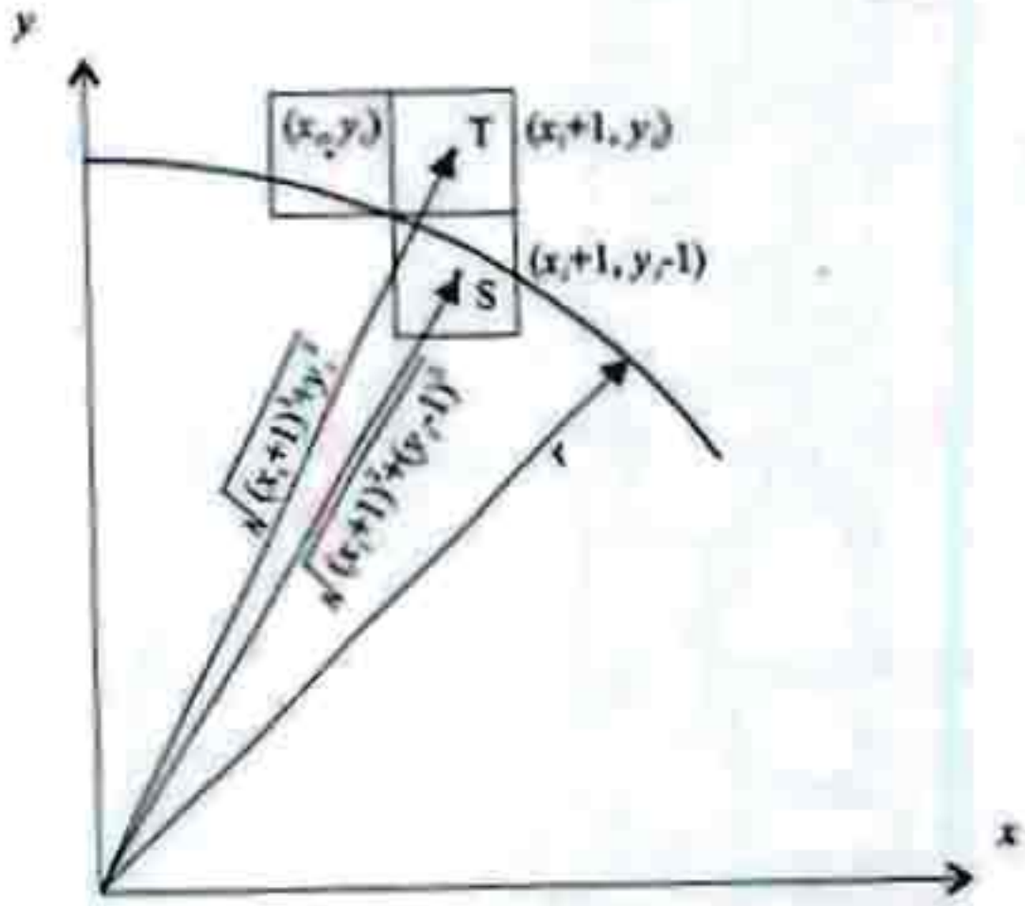

Fig. 3-8 Choosing pixels in Bresenham's circle algorithm.

When $d_i < 0$, we have $|D(\mathrm{T})| < |D(\mathrm{S})|$ and pixel T is chosen. When $d_i \geq 0$, we have $|D(\mathrm{T})| \geq |D(\mathrm{S})|$ and pixel S is selected. We can also write the decision variable d_{i+1} for the next step:

$$d_{i+1} = 2(x_{i+1}+1)^2 + y_{i+1}^2 + (y_{i+1}-1)^2 - 2r^2$$

Hence

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

Since $x_{i+1} = x_i + 1$, we have

$$d_{i+1} = d_i + 4x_i + 2(y_{i+1}^2 - y_i^2) - 2(y_{i+1} - y_i) + 6$$

If T is the chosen pixel (meaning that $d_i < 0$) then $y_{i+1} = y_i$ and so

$$d_{i+1} = d_i + 4x_i + 6$$

On the other hand, if S is the chosen pixel (meaning that $d_i \geq 0$) then $y_{i+1} = y_i - 1$ and so

$$d_{i+1} = d_i + 4(x_i - y_i) + 10$$

Hence we have

$$d_{i+1} = \begin{cases} d_i + 4x_i + 6 & \text{if } d_i < 0 \\ d_i + 4(x_i - y_i) + 10 & \text{if } d_i \geq 0 \end{cases}$$

Finally, we set $(0, r)$ to be the starting pixel coordinates and compute the base case value d_1 for this recursive formula from the original definition of d_i:

$$d_1 = 2(0+1)^2 + r^2 + (r-1)^2 - 2r^2 = 3 - 2r$$

We can now summarize the algorithm for generating all the pixel coordinates in the 90° to 45° octant that are needed when scan-converting a circle of radius r:

```
int x = 0, y = r, d = 3 - 2r;
while (x <= y) {
   setPixel(x, y);
   if (d < 0)
      d = d + 4x + 6;
   else {
      d = d + 4(x - y) + 10;
      y--;
   }
   x++;
}
```

Note that during each iteration of the while loop we first set a pixel whose position has already been determined, starting with $(0, r)$. We then test the current value of decision variable d in order to update d and determine the proper y coordinate of the next pixel. Finally we increment x.

Midpoint Circle Algorithm

We present another incremental circle algorithm that is very similar to Bresenham's approach. It is based on the following function for testing the spatial relationship between an arbitrary point (x, y) and a circle of radius r centered at the origin:

$$f(x,y) = x^2 + y^2 - r^2 \begin{cases} < 0 & (x,y) \text{ inside the circle} \\ = 0 & (x,y) \text{ on the circle} \\ > 0 & (x,y) \text{ outside the circle} \end{cases}$$

Now consider the coordinates of the point halfway between pixel T and pixel S in Fig. 3-8: $(x_i + 1, y_i - \frac{1}{2})$. This is called the midpoint and we use it to define a decision parameter:

$$p_i = f(x_i + 1, y_i - \tfrac{1}{2}) = (x_i + 1)^2 + (y_i - \tfrac{1}{2})^2 - r^2$$

If p_i is negative, the midpoint is inside the circle, and we choose pixel T. On the other hand, if p_i is positive (or equal to zero), the midpoint is outside the circle (or on the circle), and we choose pixel S. Similarly, the decision parameter for the next step is

$$p_{i+1} = (x_{i+1} + 1)^2 + (y_{i+1} - \tfrac{1}{2})^2 - r^2$$

Since $x_{i+1} = x_i + 1$, we have

$$p_{i+1} - p_i = [(x_i + 1) + 1]^2 - (x_i + 1)^2 + (y_{i+1} - \tfrac{1}{2})^2 - (y_i - \tfrac{1}{2})^2$$

Hence

$$p_{i+1} = p_i + 2(x_i + 1) + 1 + (y_{i+1}^2 - y_i^2) - (y_{i+1} - y_i)$$

If pixel T is chosen (meaning $p_i < 0$), we have $y_{i+1} = y_i$. On the other hand, if pixel S is chosen (meaning $p_i \geq 0$), we have $y_{i+1} = y_i - 1$. Thus

$$p_{i+1} = \begin{cases} p_i + 2(x_i + 1) + 1 & \text{if } p_i < 0 \\ p_i + 2(x_i + 1) + 1 - 2(y_i - 1) & \text{if } p_i \geq 0 \end{cases}$$

We can continue to simplify this in terms of (x_i, y_i) and get

$$p_{i+1} = \begin{cases} p_i + 2x_i + 3 & \text{if } p_i < 0 \\ p_i + 2(x_i - y_i) + 5 & \text{if } p_i \geq 0 \end{cases}$$

Or we can write it in terms of (x_{i+1}, y_{i+1}) and have

$$p_{i+1} = \begin{cases} p_i + 2x_{i+1} + 1 & \text{if } p_i < 0 \\ p_i + 2(x_{i+1} - y_{i+1}) + 1 & \text{if } p_i \geq 0 \end{cases}$$

Finally, we compute the initial value for the decision parameter using the original definition of p_i and $(0, r)$:

$$p_i = (0 + 1)^2 + (r - \tfrac{1}{2})^2 - r^2 = \tfrac{5}{4} - r$$

One can see that this is not really integer computation. However, when r is an integer we can simply set $p_1 = 1 - r$. The error of being $\frac{1}{4}$ less than the precise value does not prevent p_1 from getting the appropriate sign. It does not affect the rest of the scan-conversion process either, because the decision variable is only updated with integer increments in subsequent steps.

The following is a description of this midpoint circle algorithm that generates the pixel coordinates in the 90° to 45° octant:

```
int x = 0, y = r, p = 1 − r;
while (x <= y) {
   setPixel(x, y);
   if (p < 0)
      p = p + 2x + 3;
   else {
      p = p + 2(x − y) + 5;
      y−−;

   x++;
```