Chapter 6
Circle Drawing

After reading this chapter, you'll be able to understand and implement circle drawing algorithms using:

- Cartesian coordinate.
- Polar coordinate.
- Midpoint.
- Bresenham.

6.1 Introduction

A picture is completely specified by the set of intensities for the pixel positions in the display. Shapes and colors of the objects can be described internally with pixel arrays into the frame buffer or with the set of the basic geometric – structure such as circle or line segments and polygon color areas. To describe structure of basic object is referred to as output primitives. Each output primitive is specified with input co-ordinate data and other information about the way that objects is to be displayed. Additional output primitives that can be used to constant a picture include conic sections, quadric surfaces, spline curves and surfaces, polygon floor areas and character string.
In this chapter, we describe the algorithms of circle such as generating circle in Section 2. The drawing a circle using: Cartesian coordinates as in Section 3, polar coordinates as in Section 4, Bresenham's circle algorithm as in Section 5, and midpoint circle algorithm in Section 6.

6.2 Generating Circle

A circle is a simple shape of Euclidean geometry consisting of those points which are the same distance from a given point called the center. The common distance of the points of a circle from its center is called its radius.

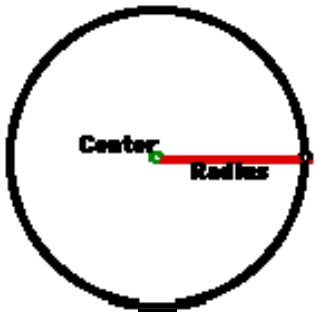


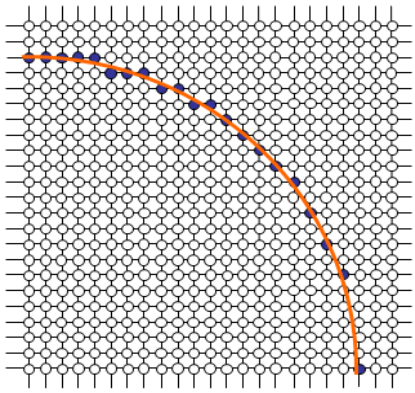
Fig.6-1: Generating circle.

There is several ways to draw a circle:

- Generate circle using Cartesian coordinate
- Generate circle using polar coordinate
- Midpoint circle algorithm.
- Bresenham's circle algorithm

6.3 Generating circle using Cartesian coordinate

The equation for a circle is: $x^2 + y^2 = r^2$ where r is the radius and (0, 0) is centre of the circle. So we can write a simple circle (see Fig.6-2) drawing algorithm by solving the equation for y at unit *x* intervals using: $y = \pm\sqrt{r^2 - x^2}$.



$$y_0 = \sqrt{20^2 - 0^2} \approx 20$$
$$y_1 = \sqrt{20^2 - 1^2} \approx 20$$
$$y_2 = \sqrt{20^2 - 2^2} \approx 20$$
$$\vdots$$
$$y_{19} = \sqrt{20^2 - 19^2} \approx 6$$
$$y_{20} = \sqrt{20^2 - 20^2} \approx 0$$

Fig.6-2: Generating simple circle using Cartesian coordinates.

We can describe the circle using general Cartesian form. Let we have circle with center (x_c, y_c) anmd radius *r* (see Fig.6-3), ggenerally becomes equivalent circuit as follows:

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

$$\therefore\ y_1 = y_c + \sqrt{r^2 - (x - x_c)^2}$$

$$y_2 = y_c - \sqrt{r^2 - (x - x_c)^2}$$

for $-r \le x \le r$

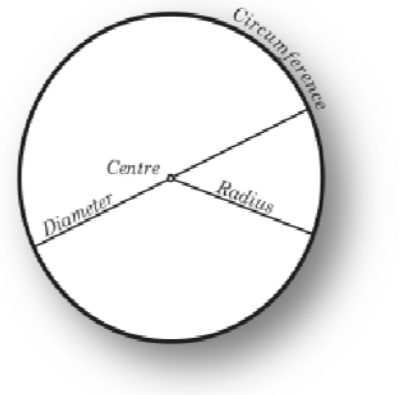


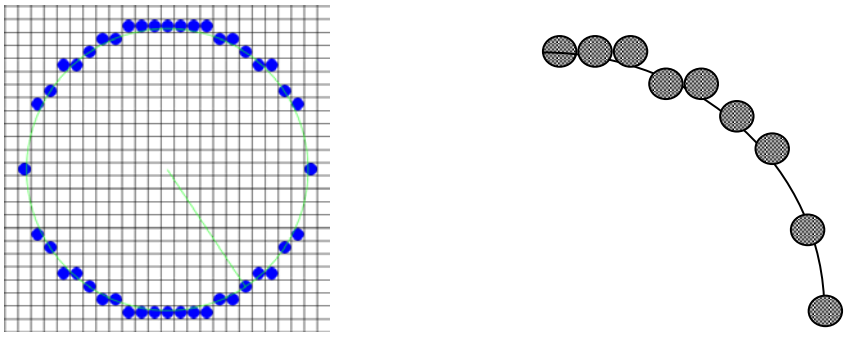
Fig.6-3: Circle in general form.

Note: This method is not very good because

- the resulting circle has large gaps.
- the calculations are not very efficient with square root operations

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

It also creates large gaps in the circle for values of *x* close to *r* (and clumping for *x* near 0) (see Fig.6-4).



a b

Fig.6-4: Generating general circle using Cartesian coordinates.

The algorithm

1. Input radius r and circle center (x_c, y_c)
2. Generate x where $-r \le x \le r$
3. Compute (y_1, y_2)
   where: $y_1 = y_c + \sqrt{r^2 - (x - x_c)^2}$
   and $y_2 = y_c - \sqrt{r^2 - (x - x_c)^2}$
4. glVertex2fv (x,y_1) , glVertex2fv (x,y_2)
5. Display the circle.

Example 6-1: Numeric Cartesian example

Generate 6 points from circle with radius $r = 5$, $(x_c , y_c) = (0, 0)$

$x = 0.1$

$y_1 = y_c + \sqrt{r^2 - (x - x_c)^2} = 0 + \sqrt{5^2 - (0.1 - 0)^2}$

$= 4.998 \quad \Rightarrow (0.1, 4.998)$

$y_2 = y_c - \sqrt{r^2 - (x - x_c)^2} = 0 - \sqrt{5^2 - (0.1 - 0)^2}$

$= -4.998 \quad \Rightarrow (0.1, -4.998)$

$x = 0.2$

$y_1 = 0 + \sqrt{5^2 - (0.2 - 0)^2}$

$= 4.995$

$\Rightarrow (0.1, 4.995)$

$$y_2 = 0 - \sqrt{5^2 - (0.2 - 0)^2}$$
$$= -4.995$$
$$\Rightarrow (0.1, -4.995)$$

$$x = 0.3$$
$$y_1 = 0 + \sqrt{5^2 - (0.3 - 0)^2}$$
$$= 4.990$$
$$\Rightarrow (0.1, 4.990)$$
$$y_2 = 0 - \sqrt{5^2 - (0.3 - 0)^2}$$
$$= -4.990$$
$$\Rightarrow (0.1, -4.990)$$

Example 6-2: Code of Cartesian coordinate

```
void circleSimple(int xCenter, int yCenter, int
radius)
     {
          int x, y, r2;
            r2 = radius * radius;
           for (x = -radius; x <= radius; x++) {
          y = (double) (sqrt(r2 - x*x));
      setPixel(xCenter + x, round( yCenter + y));
setPixel(xCenter + x, round( yCenter – y));
}}
```

6.4 Generating circle using polar coordinate

From the right angle triangle in the picture one immediately gets the following correspondence between the Cartesian coordinates (x,y) and the polar coordinates (r,θ) assuming the Pole of the polar Coordinates is the origin of the Cartesian coordinates and the polar axis is the positive x-axis (see Fig.6-5).

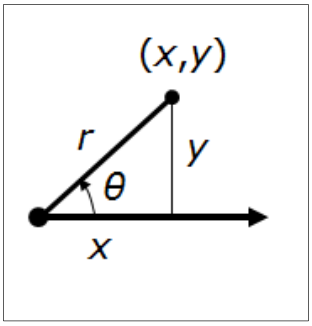


Fig.6-5: Representing a point (x,y) in polar coordinates.

- If the circle is centered at the origin (a, b) :
  - $\boldsymbol{sin\theta = \frac{y-y_c}{r} \rightarrow y = y_c + r * sin\theta}$
  - $\boldsymbol{cos\theta = \frac{x-x_c}{r} \rightarrow x = x_c + r * cos\theta}$

- If the circle is centered at the origin (0, 0), then the equation simplifies to
  - $\boldsymbol{y = rsin\theta}$
  - $\boldsymbol{x = rcos\theta}$

$$\textbf{where} -2\pi \leq \theta \leq 2\pi$$

Using above equation circle can be plotted by calculating x and y coordinates as θ takes values from 0 to 360 degrees or 0 to 2π (see Fig.6-6). 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.

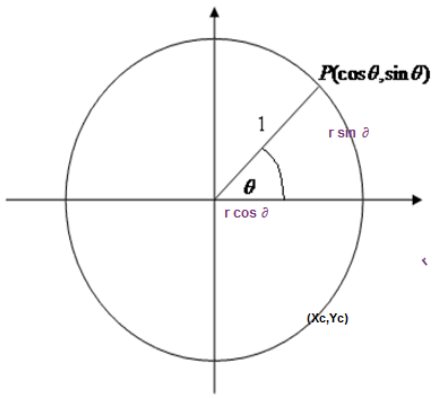


Fig.6-6: Point p on the circle.

The algorithm

1. Input radius r and circle center (x_c, y_c)
2. Generate θ where: $-2\pi \leq \theta \leq 2\pi$
3. Compute x, y
   where: $y = y_c + r * sin\theta$
   and $x = x_c + r * cos\theta$
4. Display(x, y)
5. Repeat steps 2 through 4

Example 6-3: Numeric polar example

Generate 5 points from circle with radius $r = 5$, $(x_c , y_c) = (0, 0)$

$\theta = -6.28$
$x = x_c + r\cos\theta = 0 + 5\cos(-6.28) = 4.96$
$y = y_c + r\sin\theta = 0 + 5\sin(-6.28) = -0.54$ $\Rightarrow (4.96, -0.54)$

$\theta = 6.28$
$x = 0 + 5\cos(6.28) = 4.96$
$y = 0 + 5\sin(6.28) = 0.54$ $\Rightarrow (4.96, 0.54)$

$\theta = 0$
$x = 0 + 5\cos(0) = 5$
$y = 0 + 5\sin(0) = 0 \quad \Rightarrow (5,0)$

$\theta = -6.18$
$x = 0 + 5\cos(-6.18) = 4.97$
$y = 0 + 5\sin(-6.18) = -0.53 \quad \Rightarrow (4.97, -0.53)$

$\theta = 6.18$
$x = 0 + 5\cos(6.18) = 4.97$
$y = 0 + 5\sin(6.18) = 0.53 \quad \Rightarrow (4.97,0.53)$

Example 6-4: Code of polar coordinate

```
void Polar_circle(GLfloat xc,GLfloat yc ,GLfloat
rc)
{     GLfloat xi,yi;
      theta=-2*pi;
        while(theta<(2*pi))
             {
            x_i=x_c+r*cos(theta)
             y_i=y_c+r*sin(theta);
               glBegin(GL_POINTS);
               glVertex2d(x_i,y_i);
            glEnd();
          theta+=0.01;
      }
      glFlush();

}
```

6.5 Bresenham's circle algorithm

Let's say we want to scan-convert a circle centered at (0, 0) with an integer radius *r*. First of all, notice that the interior of the circle is characterized by the inequality:

$$D(x,y) = x^2 + y^2 \quad | \quad r^2 < 0$$

We can rewrite the equation as

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

We'll use $D(x, y)$ to derive our decision variable. We'll go over vertical scan lines through the centers of the pixels and, for each such scan line, compute the pixel on that line which is the closest to the scan line-circle intersection point (black dots in Fig. 6-7). All such pixels will be plotted by our procedure.

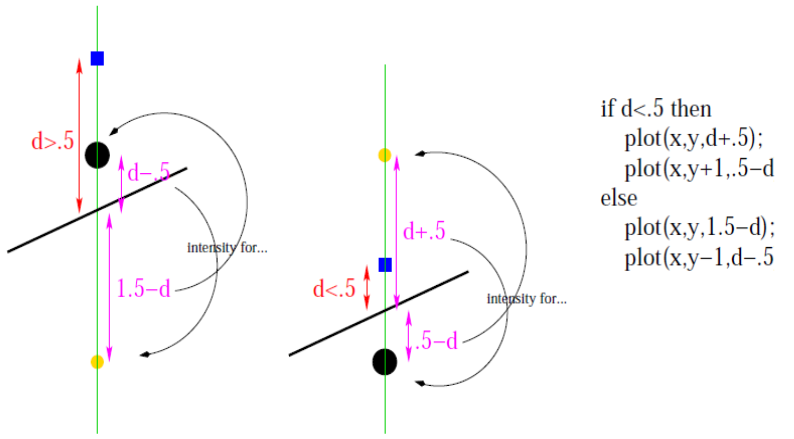


Fig.6-7: Drawing circle using *D(x,y)* inequality.

Two cases which need to be addressed in the antialiased variant of the Bresenham's algorithm: Left: the plotted point is above the line; in this case we need to distribute intensity between this one and the one immediately below it. Center: the plotted point is below the line; in this case we need to distribute intensity between this one and the one immediately below it. Right: here is what needs to replace the plot(*x*, *y*) call if antialiased lines are to be produced (the third argument of

plot is the intensity to be assigned to a pixel). A circle and the description of its interior and exterior can be represented as two quadratic inequalities midpoint between the plotted pixel and the pixel immediately below in Fig.6-8.

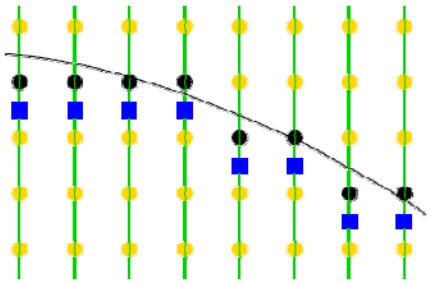


Fig. 6-8: Points are plotted using Bresenham's algorithm.

Bresenham's algorithm steps

1- $x_0 = 0$
2- $y_0 = r$
3- $p_0 = [1^2 + r^2 - r^2] + [1^2 + (r-1)^2 - r^2] = 3 - 2r$
4- if $p_i < 0$ then

$$y_{i+1} = y_i$$

$$p_{i+1} = p_i + 4x_i + 6$$

else *if* $p_i \geq 0$ then

$$y_{i+1} = y_i - 1$$

$$p_{i+1} = p_i + 4(x_i - y_i) + 10$$

5- Stop when $x_i \geq y_i$ and determine symmetry points in the other octants.

Example 6-5: Numeric evaluation

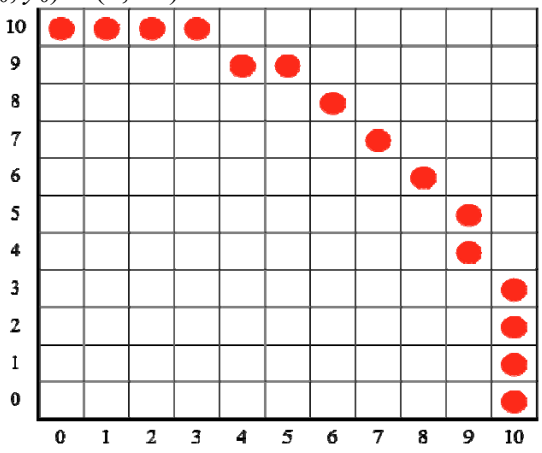
Draw a circle with radius r = 10 and center (0, 0) using Bresenham's algorithm.

Ans:

$r = 10$

$p_0 = 3 - 2r = -17$

Initial point $(x_0, y_0) = (0, 10)$



i	p_i	x_i, y_i
0	-17	(0, 10)
1	-11	(1, 10)
2	-1	(2, 10)
3	13	(3, 10)
4	-5	(4, 9)
5	15	(5, 9)
6	9	(6, 8)
7		(7,7)

Example 6-6: Code of Bresenham circle algorithm

```
void CircleBresenham(int xc, int yc, int r, int
color)
{
    int x = 0;
         int y = r;
         int p = 3 - 2 * r;
            if (!r) return;
       while (y >= x) // only formulate 1/8 of circle
                {
drawpixel(xc-x, yc-y, color); //upper left left
drawpixel(xc-y, yc-x, color); //upper upper left
drawpixel(xc+y, yc-x, color); //upper upper right
 drawpixel(xc+x, yc-y, color); //upper right right
drawpixel(xc-x, yc+y, color); //lower left left
drawpixel(xc-y, yc+x, color); //lower lower left
      drawpixel(xc+y, yc+x, color); //lower lower right
        drawpixel(xc+x, yc+y, color); //lower right right
        if (p < 0) p += 4*x++ + 6;
              else p += 4*(x++ - y--) + 10;
     } }
```

6.6 Midpoint circle algorithm

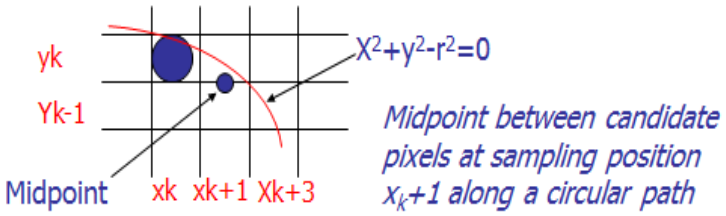
It is an algorithm used to determine the points needed for drawing a circle. The description of midpoint algorithm can be shown in Fig.6-9 and Fig.6-10.

The algorithm can be described as follows:

- We will first calculate pixel positions for a circle centered on the 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$y$
- 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

$$f_{circle}(x,y) = x^2 + y^2 - r^2$$

- 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 boundry
  - $f_{circle}(x,y) = 0\ if\ (x,y)$ is on the circle boundry
  - $f_{circle}(x,y) >$ $0\ if\ (x,y)$ is outside the circle boundry



a

Fig.6-9: Compute mid points along a circular path.

- Choosing the next pixel decision variable *d*

  $d = F(M) = F(x+1, y+1/2)$

  $F(x+1, y+1/2) > 0\ choose$ E

  $F\left(x+1, y+\frac{1}{2}\right) \leq 0\ choose$ SE

- Change of d when E is chosen

  $d_{new} = (x+2)^2 + (y+1/2)^2 - r^2$

  $d_{old} = (x+1)^2 + (y+1/2)^2 - r^2$

  $\Delta d = d_{new} - d_{old} = 2x + 3$

- Change of d when SE is chosen

$d_{new} = (x+2)^2 + (y+3/2)^2 - r^2$

$d_{old} = (x+1)^2 + (y+1/2)^2 - r^2$

$\Delta d = d_{new} - d_{old} = 2x + 2y + 5$

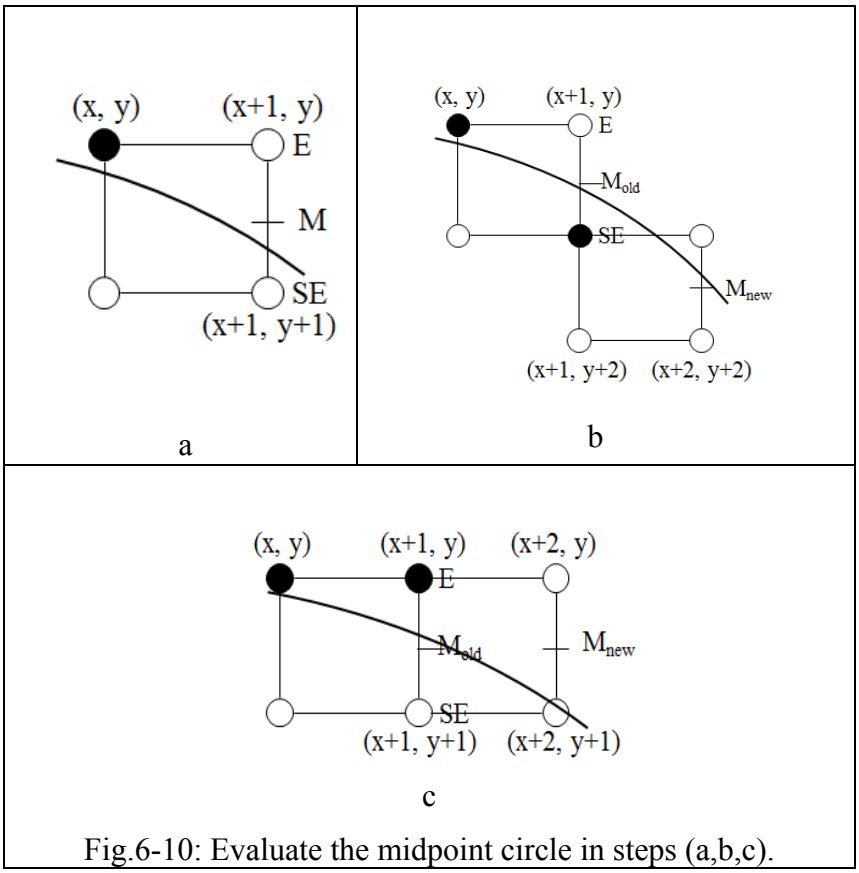


Fig.6-10: Evaluate the midpoint circle in steps (a,b,c).

- Initial value of d

$d_0 = F(M_0)$

$d_0 = F(1, -r + 1/2)$

$d_0 = (1)^2 + (-r + 1/2)^2 - r^2$

$d_0 = 5/4 - r$

- Assuming we have just plotted the pixel at (x_k, y_k)we next need to determine whether the pixel at position $(x_k + 1, y_k - 1)$is closer to the circle.
- Our decision parameter is the circle function evaluated at the midpoint between these two pixels

  $p_k = f_{circle}(x_k + 1, y_k - 1/2)$

  $= (x_k + 1)^2 + (y_k - 1/2)^2 - r^2$
- If $p_k < 0$ this midpoint is inside the circle and the pixel on the scan line y_k is closer to the circle boundary. Otherwise, the mid position is outside or on the circle boundary, and we select the pixel on the scan line $y_k - 1$
- Successive decision parameters are obtained using incremental calculations

  $p_{k+1} = f_{circle}(x_{k+1} + 1, y_{k+1} - 1/2)$

  $= (x_{k+1} + 1)^2 + (y_{k+1} - 1/2)^2 - r^2$

OR: $p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_k + 1 - y_k) + 1$

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}:

  $2x_{k+1} + 1$if$p_k$is negative

  $2x_{k+1} + 1 - 2y_{k+1}$otherwise
- Note that following can also be done incrementally:

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

  $2y_{k+1} = 2y_k - 2$
- At the start position$(0, r)$ these two terms have the values 2 and $2r - 2$ respectively
- Initial decision parameter is obtained by evaluating the circle function at the start position

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

  $p_0 = f_{circle}(1, r - 1/2) = 1 + (r - 1/2)^2 - r^2$

  OR: $p_0 = 5/4 - r$
- If radius r is specified as an integer, we can round $p_0$to

  $p_0 = 1 - r$.

The midpoint circle algorithm steps

1. Input radius r and circle center (x_c, y_c) and obtain the first point on the circumference of the circle centered on the origin as $(x_0, y_0) = (0, r)$
2. Calculate the initial value of the decision parameter as $p_0 = 5/4 - r$
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+1}) 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, y_c) and plot the coordinate values at $x = x + x_c$ and $y = y + y_c$
6. Repeat steps 3 through 5 until $x \geq y$

Example 6-7: Numeric example

Draw a circle with radius r = 10 and center (0, 0) using midpoint circle algorithm.

Ans:

(0 , 10)

p_0 = 1 – 10 = -9

$p_1 = p_0 + 2\ x_0 + 1$ = -9 +2 +1 = -6

$p_1 < 0$

(2 , 10)

$p_2 = p_1 + 2\ x_1 + 1$ = -6 +4 +1 = -1

$p_2 < 0$

(3 , 10)

$p_3 = p_2 + 2\,x_2 + 1 = -1 + 6 + 1 = 6$

$p_3 > 0$

(4 , 9)

$p_4 = P_3 + 2\,x_4 + 1 - 2y_4 = 6 + 8 + 1 - 18 = -3$

$p_4 < 0$

(5 ,9)

$p_5 = -3 + 10 + 1 = 8$

$p_5 > 0$

(6 ,80)

$p_6 = 8 + 12 + 1 - 16 = 5$

$p_6 > 0$

(7 , 7)

$p_7 = 5 + 14 + 1 - 14 = 6$

Stop $x \geq y$: this complete the generation.

p_0	(x,y)
-9	(1,10)
-6	(2,10)
-1	(3,10)
6	(4,9)
-3	(5,9)
8	(6,8)
5	(7,7)

Example 6-8: Code of midpoint circle algorithm

```
void circleMidpoint(GLint xc,GLint yc,GLint radius)
{   screenPt circPt;
    GLint p=1-radius;
     circPt.setCoords(0,radius);
    circlePlotPoints(xc,yc,circPt);
    while (circPt.getx() < circPt.gety())
        {  circPt.incrementx();
            if (p<0) p+=2*circPt.getx() +1;
                else{circPt.decrementy();
                     p+=2*circPt.getx() -
circPt.gety()+1;}
   circlePlotPoints(xc,yc,circPt);
        }
}
```

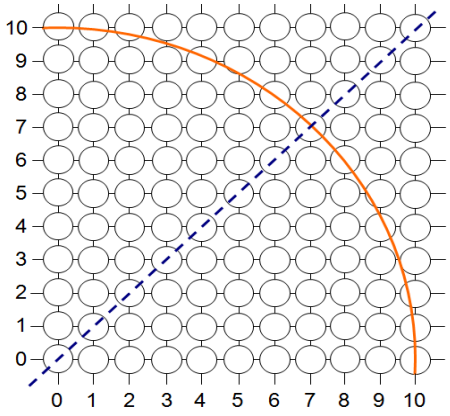
Bibliography

1- Donald D. Hearn, M. Pauline Baker, “Computer Graphics with OpenGL”, Prentice Hall; 3rd edition 2003.
2- http://www.vrarchitect.net/anu/cg/Circle/index.en.html
3- F. S. Hill, Jr. and S. Kelley, "Computer Graphics using OpenGL", 3nd edition, Prentice Hall 2001.
4- A. Watt, "3D Computer Graphics", 3rd edition, Addison Wesley – Pearson Education, 2000.
5- E Angel, "Interactive Computer Graphics: A top-Down approach with OpenGL", 3rd edition, Addison Wesley, 2003.
6- P. Egerton & W. Hall, "Computer Graphics: Mathematical First Steps", Prentice Hall - Pearson Education, 1999.
7- R. Hearn and M.P. Baker, "Computer Graphics with OpenGL", 3rd ed, Prentice Hall - Pearson Education, 2004.

Exercises

1. Use the mid-point circle algorithm to draw the circle centered at (0, 0) with radius 10.

k	p_k	(x_{k+1}, y_{k+1})
0		
1		
2		
3		
4		
5		
- - -		



2. Write OpenGL program to do the following:
   - i- Drawing circle using Cartesian coordinate.
   - ii- Drawing a circle using Polar coordinate.

3. Given a circle radius r=11, the center (5, 5), and the initial point is (0, 2); demonstrate the midpoint circle algorithm for determining only five points along the circle.
4. Given a circle radius r=9, the center (5, 5), and the initial point is (0, 2); demonstrate the midpoint circle algorithm for determining only five points along the circle.
5. Go through the steps of the Bresenham circle drawing algorithm for a circle (in Eq.4).
6. Given a circle radius r=21, demonstrate the midpoint circle algorithm by determining position along the circle octant in the first quadrant from x=y the initial parameter (1, 1).
7. Describe pseudo codes for drawing circle using the midpoint circle algorithm.
8. Draw circle with radius r=15 and centre $(x_c,y_c)=(1,-3)$.

1)-Cartesian coordinate
2)-Polar coordinate
3)-Midpoint circle

Solved Problems

1. Given a circle radius r=10, demonstrate the midpoint circle algorithm by determining position along the circle octant in the first quadrant from x=1 to x=y the initial parameter (x0,y0)=(1,r-1).

Solution

Through applying the midpoint circle algorithm, we obtain the following table:

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

2. Write a program to draw a circle (polar coordinates) with center (0,0) and radius=0.5. The circle equations are as follows:

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

Program:

```
#include<windows.h>
#include<GL/glut.h>
#include<math.h>
#include<GL/gl.h>
#include<math.h>
void Init()
{
glClearColor(0.34,0.99,9.0,0);
glColor3f(0.55,0.22,0.4);
glPointSize(5.0);
}
void display()
{
    GLfloat x_i,y_i,theta=0;
    GLfloat x_c=0,y_c=0,r=0.5;
    glClear(GL_COLOR_BUFFER_BIT);
    int count;
    for(count=1;count<=10000;count++)
    {theta=theta+0.001;
    x_i=x_c+r*cos(theta);
    y_i=y_c+r*sin(theta);
    glBegin(GL_POINTS);
    glVertex2f(x_i,y_i);
    glEnd();}
    glFlush();}
void main( int argc , char**argv )
{
glutInit (&argc,argv);
glutInitWindowSize(500,500);
glutCreateWindow("Circle");
glutDisplayFunc(display);
Init();
glutMainLoop();
}
```

Output:

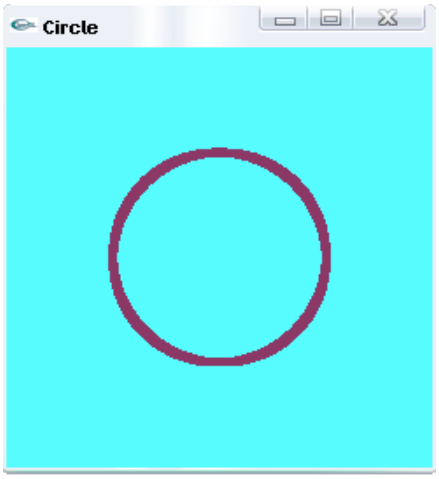


Fig.6-11: The output circle.

3. Implement a program to draw ellipse using OpenGL program for drawing a circle (polar coordinates). The ellipse equations are as follows:

$$x = x_c + r_x * sin\theta$$
$$y = y_c + r_y * cos\theta$$

Program:

```
#include<windows.h>
#include<GL/glut.h>
#include<GL/gl.h>
#include<math.h>
void Init()
{
glClearColor(0.1,0.0,.8,0);
glColor3f(0.0,0.0,0.0);
glPointSize(5.0);
}
void display()
{
    GLfloat x_i,y_i,theta=0;
    GLfloat x_c=0,y_c=0,r_x=0.5,r_y=0.9;
    glClear(GL_COLOR_BUFFER_BIT);
    int count;
    for(count=1;count<=10000;count++)
    {theta=theta+0.001;
    x_i=x_c+r_x*cos(theta);
    y_i=y_c+r_y*sin(theta);
    glBegin(GL_POINTS);
    glVertex2f(x_i,y_i);
    glEnd();}
    glFlush();}
void main( int argc , char**argv )
{
glutInit (&argc,argv);
glutInitWindowSize(500,500);
glutCreateWindow("drow");
glutDisplayFunc(display);
Init();
glutMainLoop();
}
```

Output:

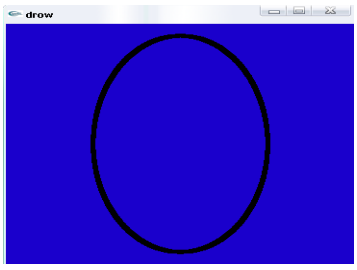


Fig.6-12: The output ellipse.