Chapter 5
Line Drawing

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

- Coordinates systems.
- Polar coordinate.
- DDA algorithm.
- Bresenham algorithm.

5.1 Introduction

The line-drawing algorithm attempts to find edges of solid bodies and intersections with other bodies. The lines in three-dimensional space are the intersection of two surfaces. In body geometry, the intersection of two surfaces belonging to the same body is explicitly known. The intersection of two surfaces belonging to different bodies is generated by points given by the intersection of "scanning" rays, that are confined to the surface of one body, with other bodies. For each point of intersection, there is a path back to the viewpoint. The intersection of these paths with an imaginary viewplane produces the three-dimensional projection of the geometry.

The path to the viewpoint may be traced to determine if the point is hidden by an opaque body; if so, the line segment(s) connected to that point is considered obscured and not drawn. All projections are determined before the plot is displayed so that the extent of the image on the viewplane can be calculated and the image thereby scaled to fill the window; a priori knowledge of the spatial bounds is not required.

Line drawing works well for simple, body-defined models. The time required increases as the square of the number of bodies, and will exceed the time required for a ray-traced rendering of a complex model. Drawing may be limited to a subset of defined bodies, known as a drawset, thereby increasing rendering speed. In this chapter, we present geometric representation of line. Also, line drawing algorithms such as digital differential (DDA) and Bresenham's Line algorithms are obtained.

This chapter can be organized as follows: In Section 2, a 2D Cartesian representation points are described. 2D polar coordinate system is presented in Section 3. Section 4 represents 3D coordinate systems. In Section 5, line drawing algorithms are presented.

5.2 2D Cartesian coordinate system

A Cartesian coordinate system, also known as rectangular coordinate system, can be used to plot points and graph lines. The following is an example of rectangular coordinate system.

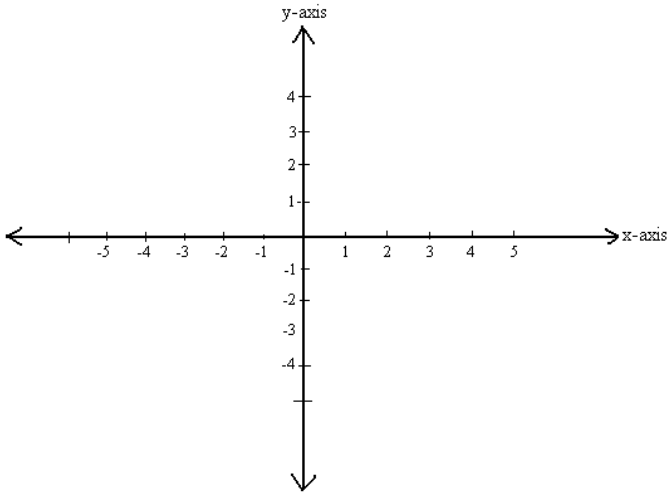


Fig.5-1: Rectangular coordinate system.

It is basically, a set of two number lines. The horizontal line is called x-axis and the vertical line is called y-axis. A good real life example of a vertical number line or y-axis is a thermometer.

A point is represented by a pair of numbers (x, y) where x stands for any value on the x-axis and y stands for any value on the y-axis. An example of a point is (1, 5). Let us now plot some points. Locate point (2, 3), x = 2 and y = 3. Draw a vertical line at x = 2 and draw a horizontal line at y = 3. Where the two lines meet or intersect, is your point (in red).

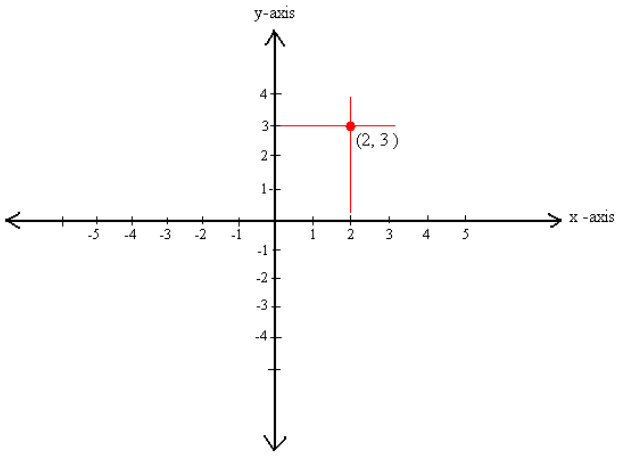


Fig.5-2: Point P(2,3).

Locate point (-3, 1). Draw a vertical line at x = -3 and draw a horizontal line at y=1. The intersection of the two lines is your point.

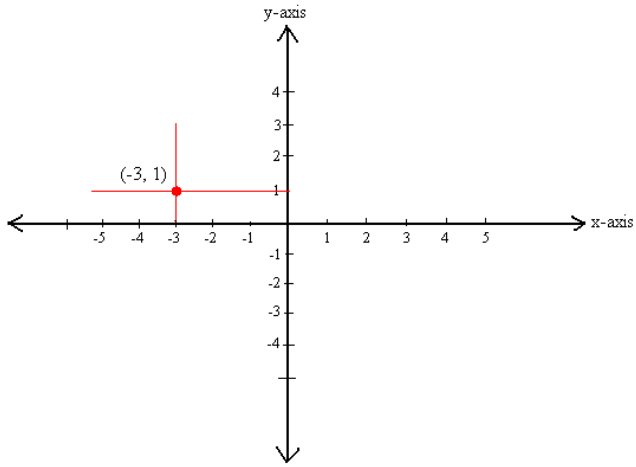


Fig.5-3: Point P(-3,1).

Locate point (-4, -4). Draw a vertical line at x = -4 and draw a horizontal line at y=-4.

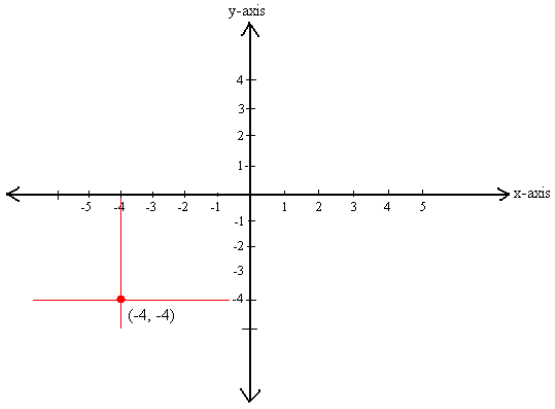


Fig.5-4: Point P(-4,4).

Tricky cases are when points are located on the x-axis or the y-axis. Students usually get confused, so study the following two examples carefully. Locate point (2, 0). Draw a vertical line at x = 2 and draw a horizontal line at y = 0. Notice that when y = 0, your horizontal line is on the x-axis. Important point to remember: When y = 0, your point is always located on the x-axis.

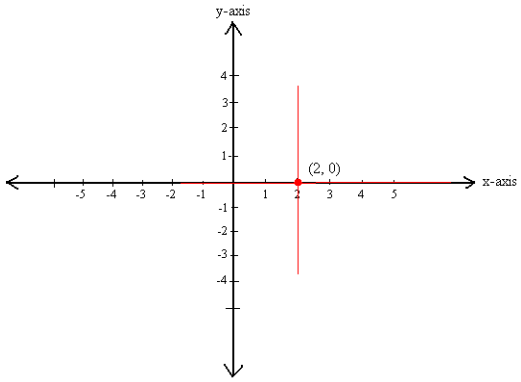


Fig.5-5: Point P(2,0).

Locate point (0, -3). Draw a vertical line at x = 0 and draw a horizontal line at y = 3. Notice that when x = 0, your vertical line is on

the x-axis. Important point to remember: when x = 0, your point is always located on the y-axis.

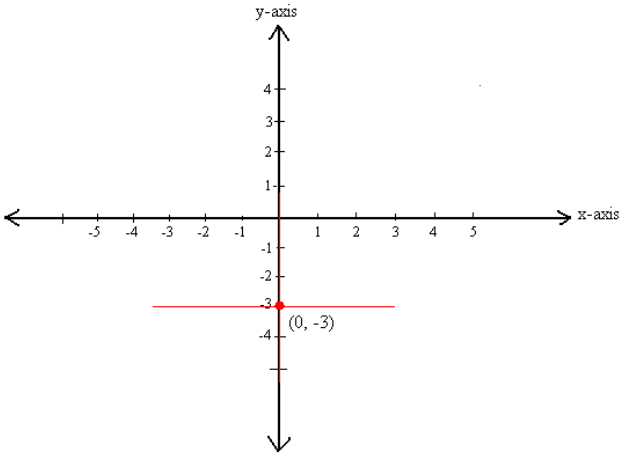


Fig.5-6: Point P(0,-3).

Other examples:

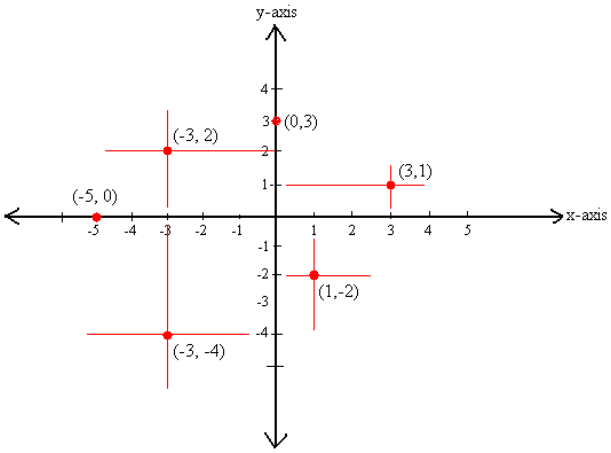


Fig.5-7: Different points.

5.3 2D polar coordinate system

The polar coordinates can be represented by r and θ coordinates.

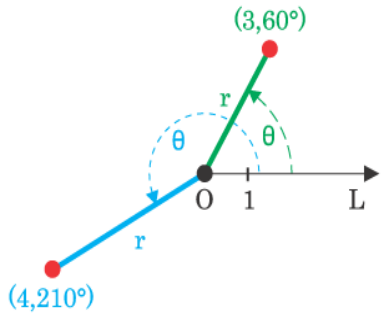


Fig.5-8:2D polar coordinate.

where x and y represented as:

$x = r\cos\theta$

$y = r\sin\theta$

(see Fig.5-9)

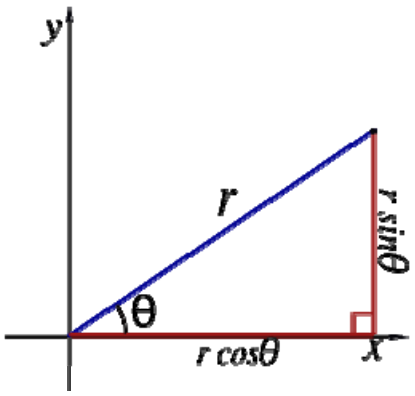


Fig.5-9:Point P(x,y)in polar coordinate.

5.4 3D coordinate system

Map points in our world to 3 real numbers (x,y,z) as in Fig.5-10. The Cartesian coordinates for a point P can be represented in Fig.5-11.

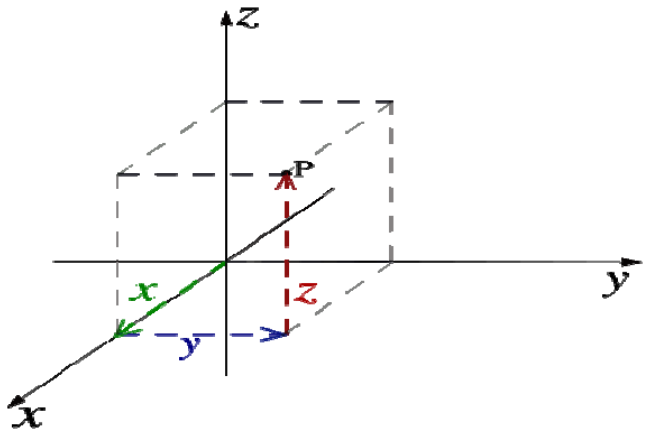


Fig.5-10: Map points in our world to 3.

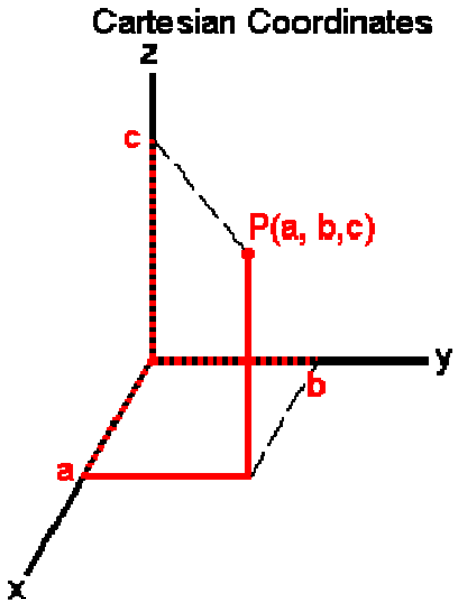


Fig.5-11: Cartesian coordinates for a point P.

In a 3D Cartesian coordinate system, a point P is referred to by three real numbers (coordinates), indicating the positions of the perpendicular projections from the point to three fixed, and perpendicular, graduated lines, called the axes which intersect at the origin. Often the x-axis is imagined to be horizontal and pointing roughly toward the viewer (out of the page), the y-axis is also horizontal and pointing to the right and the z-axis is vertical, pointing up. The system is called right-handed if it can be rotated so that the three axes are in the position as shown in the Fig.5-11. The x-coordinate of the point P in the Fig. 5-11 is a, the y-coordinate is b, and the z-coordinate is c.

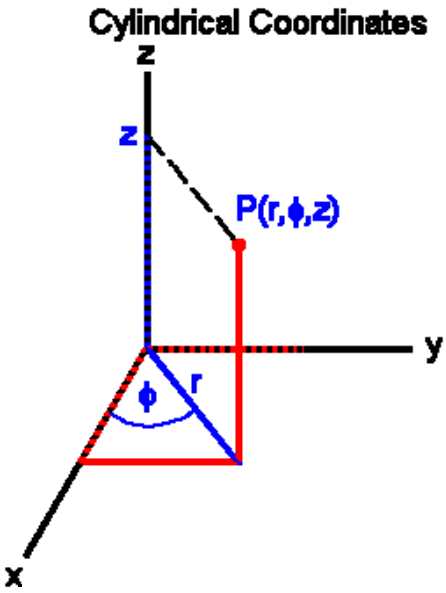


Fig.5-12: Cylindrical coordinate of point P.

To define a cylindrical coordinate system, we take an axis (usually called the z-axis) and a perpendicular plane, on which we choose a ray (the initial ray) originating at the intersection of the plane and the axis (the origin). The coordinates of a point P are the polar coordinates (r, ϕ) of the projection of P on the plane, and the coordinate z of the

projection of P on the z-axis. The coordinate r is always positive and the range of ϕ is from 0 to 2π (360^o) (see Fig.5-12).

To transform from Cartesian to cylindrical coordinates and vice versa, we use the transformation equations:

$$x = r \cos \phi,\ y = r \sin \phi,\ z = z,$$

$$r = (x^2 + y^2)^{1/2},\ \phi = \tan^{-1}(y/x),\ z = z.$$

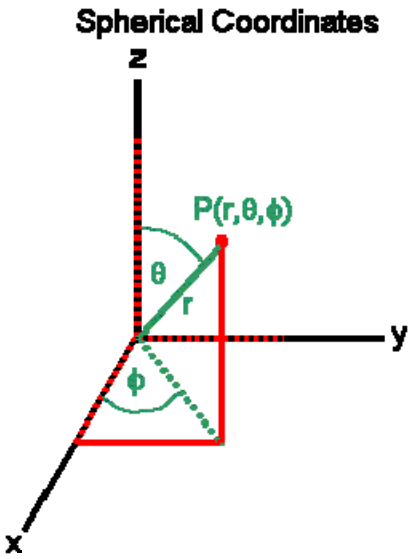


Fig.5-13: Spherical coordinate of point P.

To define spherical coordinates, we take an axis (the polar axis) and a perpendicular plane (the equatorial plane), on which we choose a ray (the initial ray) originating at the intersection of the plane and the axis (the origin O). The coordinates of a point P are the distance r from P to the origin; the angle θ (zenith) between the line OP and the positive polar axis; and the angle ϕ (azimuth) between the initial ray and the projection of OP onto the equatorial plane. The range of ϕ is from 0 to 2p (360^o), and the range of θ is from 0 to p (180^o) (see Fig.5-13).

To transform from Cartesian to spherical coordinates and vice versa, we use the transformation equations:

$$x = r \sin\theta \cos\phi,\ y = r \sin\theta \sin\phi,\ z = r \cos\theta,$$

$$r = (x^2 + y^2 + z^2)^{1/2},\ \theta = \tan^{-1}(z/(x^2+y^2)^{1/2}),\ \phi = \tan^{-1}(y/x).$$

5.5 Cylindrical coordinate system

The parameter in this system is angle and radius.

$x = r\cos\theta$

$y = r\sin\theta$

Fig.5-14 shows the point P in the cylindrical system.

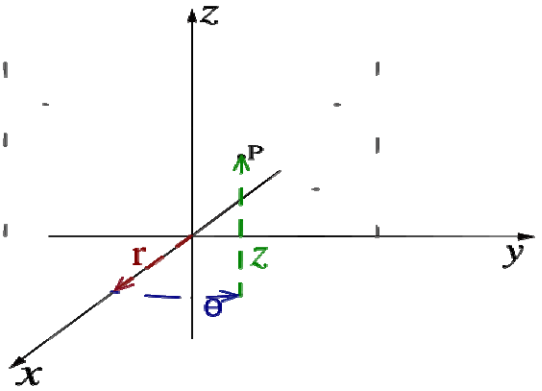


Fig.5-14: Cylindrical 3D coordinate of point P.

Spherical coordinate system

Point P*(x,y,z)* can be represented in polar coordinates (see in Fig.5-16 in Fig.5-15) as:

$x = r\sin\varphi\cos\theta$

$y = r\sin\varphi\sin\theta$

$z = r\cos\varphi$

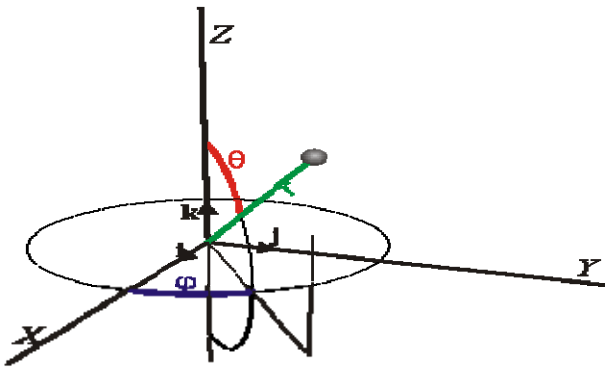


Fig.5-15: Spherical Coordinate System 3D point.

Scene

3D scene can be shown as in Fig.5-16.

- Scene = list of objects
- Object = list of surfaces
- surface = list of polygons
- Polygon = list of vertices
- Vertex = a point in 3D

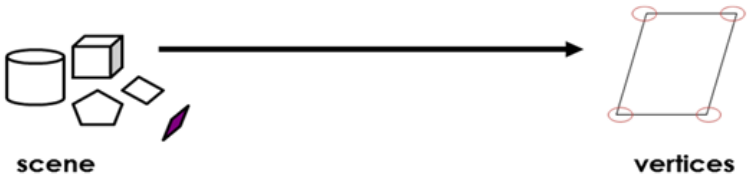


Fig.5-16: 3D scene.

5.6 Line-drawing algorithms

A straight-line segment in a scene is defined by the coordinate positions for the endpoints of the segment. To display the line on a raster monitor, the graphics system must first project the endpoints to integer screen coordinates and determine the nearest pixel positions along the line path between the two endpoints. Then the line color is loaded into the frame buffer at the corresponding pixel coordinates. Reading from the frame buffer, the video controller plots the screen pixels. This process digitizes the line into a set of discrete integer positions that, in general, only approximates the actual line path. A computed line position of (10.48, 20.51) for example, is converted to pixel position (10, 21). This rounding of coordinate values to integer's causes all but horizontal and vertical lines to be displayed with a stair-step appearance ("the jaggies"), as represented in Fig. 5-17. The characteristic stair-step shape of raster lines is particularly noticeable on systems with low resolution, and we can improve their appearance somewhat by displaying them on high-resolution systems. More effective techniques for smoothing a raster line are based on adjusting pixel intensities along the line path (Section 4-17).

Fig.5-17: Stair-step effect (jaggies produced when a line is generated as a series of pixel positions).

5.6.1. Line equations

We determine pixel positions along a straight-line path from the geometric properties of the line. The Cartesian *slope-intercept equation* for a straight line is

$$y = m \cdot x + c \tag{1}$$

with m as 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_0, y_0) and (x_{end}, y_{end}), as shown in Fig. 5-18, we can determine values for the slope m and y intercept b with the following calculations:

$$m = \frac{y_{end} - y_0}{x_{end} - x_0} \tag{2}$$

$$b = y_0 - m \cdot x_0 \tag{3}$$

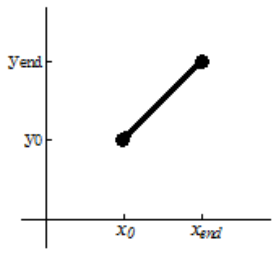


Fig.5-18 : Line path between endpoint (x_0, y_0) and (x_{end}, y_{end}).

Algorithms for displaying straight lines are based on the line equation 1 and the calculations given in Eqs.2 and 3. For any given x interval δx along a line, we can compute the corresponding y interval δy from Eq. 2 as

$$\delta y = m \cdot \delta x \tag{4}$$

Similarly, we can obtain x interval δx corresponding to a specified δy as

$$\delta x = \frac{\delta y}{m} \tag{5}$$

These equations from the basis for determining deflection voltages in analog displays such as a vector-scan system, where arbitrarily small changes in deflection voltage are possible. 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. 4. For lines whose slopes have magnitudes $|m|<1$, δy can be set proportional to a small vertical deflection voltage with corresponding horizontal deflection voltage set proportional to δx, calculated from Eq.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 sample position. This scan-conversion process for straight lines is illustrated in Fig. 5-18 with discrete sample positions along the x axis.

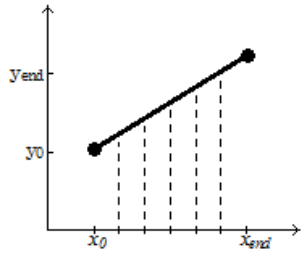


Fig.5-19 Straight-line segment with five sampling positions along the x axis between x_0 and x_{end}.

5.6.2. DDA algorithm

The *digital differential analyzer* (DDA) is a scan-conversion line algorithm based on calculating either δx or δy, using Eq. 4 or Eq. 5. A

line is sampled at unit interval in one coordinate and the corresponding integer values nearest the line path are determined for the other coordinate.

We consider first a line with positive slope, as shown in Fig5-19. If the slope is less than or equal to 1, we sample at unit x intervals (δx= 1) and compute successive y values as

$$y_{k+1} = y_k + m \tag{6}$$

Subscript k takes integer values starting from 0, for the first point, and increases by 1 until the final endpoint is reached. Since m can be any real number between 0.0 and 1.0, each calculated y value must be rounded to the nearest integer corresponding to a screen pixel in the x column we are processing.

For lines with a positive slope greater than 1.0, we reverse the roles of x and y. That is, we sample at unit y intervals (δy=1) and calculate consecutive x values as:

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

In this case, each computed x value is rounded to the nearest pixel position along the current y scan line.

Equations 6 and 7 are based on the assumption that lines are to be processed from left endpoint to the right endpoint (Fig. 5-19). If this processing is reversed, so that the starting endpoint is at the right, then either we have δx= -1 and:

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

or (when the slope is greater than 1) we have δy= -1 with:

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

Similar calculations are carried out using equations 6 through 9 to determine pixel positions along a line with negative slope. Thus, if the

absolute value of the slope is less than 1 and the starting endpoint is at the left, we set $\delta x = 1$ and calculate y values with Eq. 6. When the starting endpoint is at the right (for the same slope), we set $\delta x = -1$ and obtain y positions using Eq. 8. For a negative slope with absolute value greater than 1, we use $\delta y = -1$ and Eq. 9 or we use $\delta y = 1$ and Eq. 7. This algorithm is summarized as the following procedure, which accepts as input two integer screen positions for the endpoints of a line segment. 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 (x0, y0), 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 x0 is less than xEnd, the values for the increments in x and y directions are 1 and m, respectively. If the greater change is in the x direction but x0 is greater than xEnd, 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) in the y direction and an x increment (or decrement) of $1/m$.

Example 5-1: DDA code: void line DDA

```
inline int round (const float a){(return int
(a+0.5);)}
void lineDDA (int x0, int y0, int xEnd, int yEnd)
{
       Int dx = xEnd – x0, dy = yEnd-y0, steps, k;
           Float xIncrement, yIncrement, x = x0, y
= y0;
             If (fabs (dx)>fabs (dy));
           steps = fabs (dx);
         Else
       steps = fabs (dy);
     xIncrement = float (dx)/float (steps);
       yIncrement = float (dy)/float (steps);
     setPixel (round (x), round (y));
```

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

Example 5-2: Numerical digitization

Generate five points from the begin of line slope m = 0.5 , start point (0 , 0)?

***Ans*:**

Start points (x_0 , y_0) = (0 , 0)

Slope m = 0.5

$y_1 = y_0 + m = 0 + 0.5 = 0.5$

$x_1 = Xx_0 + 1 = 0 + 1 = 1$

(1 , 0.5)

$y_2 = y_1 + m = 0.5 + 0.5 = 1$

$x_2 = x_1 + 1 = 1 + 1 = 2$

Similar, we can get the next points are: (2 , 1), (3 , 1.5), (4 , 2), (5 , 2.5).

5.6.3. Bresenham's line algorithm

In this section, we introduce an accurate and efficient raster line-generating algorithm, developed by Bresenham that uses only incremental integer calculations. In addition, Bresenham's line algorithm can be adapted to display circles and other curves.

To illustrate Bresenham's approach, we first consider the scan-conversion process for lines with positive slope less than 1.0. 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. 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 $x_{k+1} = x_k + 1$. Our choices are the pixels at positions $(x_k + 1, y_k)$ and $(x_k + 1, y_k + 1)$.
At sampling position $x_k + 1$, we label vertical pixel separations from the mathematical line path as d_{lower} and d_{upper}. The y coordinate on the mathematical line at pixel column position $x_k + 1$ is calculated as

$$y = m(x_{k+1}) + b \tag{10}$$

Then

$$d_{lower} = y - y_k = m(x_k + 1) + b - y_k \tag{11}$$

and

$$d_{upper} = (y_k + 1) - y = y_k + 1 - m(x_k + 1) - b \tag{12}$$

To determine which of the two pixels is closed to the line path, we can set up an efficient test that is based on the difference between the two pixel separations:

$$d_{lower} - d_{upper} = 2m(x_k + 1) - 2y_k + 2b - 1 \tag{13}$$

A decision parameter p_k for the kth step in the line algorithm can be obtained by rearranging Eq. 13 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 the decision parameter as

$$p_k = \Delta x(d_{lower} - d_{upper}) = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c \tag{14}$$

The sign of p_k is the same as the sign of $d_{\text{lower}} - d_{\text{upper}}$, 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 the 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, $d_{\text{lower}} < d_{\text{upper}}$), 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 evaluated from Eq. 14 as

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

Subtracting Eq. 14 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

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k) \qquad (15)$$

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. 14 at the starting pixel position (x_0, y_0) and with *m* evaluated as $\Delta y/\Delta x$:

$$p_0 = 2\Delta y - \Delta x \qquad (16)$$

We summarize Bresenham line drawing for a line with a positive slope less than 1 in the following outline of the algorithm. 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| < 1.0

1. Input the two line endpoints and store the left endpoint in (x_0, y_0).

2. Set the color for frame-buffer position (x_0, y_0); i.e., plot the first point.
3. Calculate the 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. Perform step 4, $\Delta x - 1$ times

Example 5-3: Bresenham line drawing

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

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

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})
0	6	(21,11)
1	2	(22,12)
2	-2	(23,12)
3	14	(24,13)
4	10	(25,14)

k	p_k	(x_{k+1}, y_{k+1})
5	6	(26,15)
6	2	(27,16)
7	-2	(28,16)
8	14	(29,17)
9	10	(30,18)

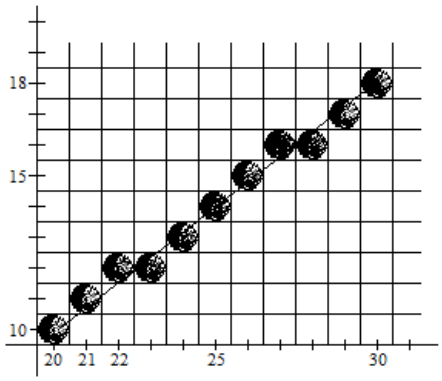


Fig.5-20: Pixel positions along the line path between end points (20, 10) and (30, 18) plotted with Bresenham's line algorithm.

A plot of the pixels generated along this line path is shown in Fig. 5-20. An implementation of Bresenham line drawing for slopes in the range $0 < m < 1.0$ is given in the following procedure. Endpoint pixel positions for the line are passed to this procedure, and pixels are plotted from the left endpoint to the right endpoint.

Example 5-4: Bresenham line-drawing procedure: Void lineBres

```
/* Bresenham line-drawing procedure for |m| < 1.0
*/
Void lineBres (int x0, int y0, int xEnd, int yEnd)
{
    Int dx = fabs(xEnd – x0), dy = fabs(yEnd - y0);
       Int p = 2*dy-dx;
```

```
        Int twoDy = 2*dy, twoDyMinusDx = 2*(dy-dx);
/* Determine which endpoint to use as start
position.  */
              if (x0 > XEnd) {
              x = xEnd;
              y = yEnd;
              xEnd = x0;
               }
        Else {
     x = x0;
    y = y0;
     }
   setPixel(x,y);
while (x < xEnd) {
       x++;
if (p < 0)
              p += twoDy;
           else {
       y++;
      p += twoDy;
     else {
     Y++;
    P += twoDyMinusDx;
}
setPixel (x,y);
}
}
```

Bibliography

1- Donald D. Hearn, M. Pauline Baker, “Computer Graphics with OpenGL”, Prentice Hall; 3rd edition 2003.
2- http://www.whiterockscience.com/sabrina/help/features/Fline_draw_intro.html

Exercises

1- List the code needed to set up an OpenGl display window whose lower-right corner is at pixel position (150,150) with windows width of 100 pixels and a height of 75 pixels.

2- Generate 5 points from the beginning of line (y_0 =0, x_0=0) and 5 points from the end of line (y_n =5, x_n=10) when m=0.5 respectively, using
   i. DDA Algorithm.
   ii. Cartesian coordinates

3- Digitize only five points of the line using Bresenham algorithm with endpoints (20, 10), (30, 18), and slop 0.5.

4- Use the DDA algorithm to:
   i. Draw a line from point (2,1) to (12,6)(m=0.5)
   ii. Draw a line from point (1,6) to (11,0)(m=-0.6)

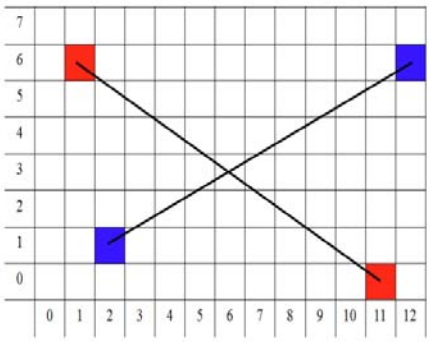


Fig.5-21: Two lines.

5- Write a program to draw line using DDA algorithm, start point=(0,0) and the slop=1.

6- Write a program to draw line using Bresenham algorithm, start point=(0,0), end point =(10,10).

7- Describe pseudo codes: for drawing line using DDA algorithm.

8- Describe pseudo codes: for drawing line using Bresenham algorithm.

Solved Problems

1. Digitize the line using the DDA algorithm, given any number(n) of input points. A single point is plotted when n=1 when $(x_0,y_0)=(0,0)$, m=1.5.

Solution

We will generate the points when n=5, m=1.5

DDA equations are $x_{k+1}= x_k+(1/m)$ and $y_{k+1}= y_k+1$

First point: $x_1= 0+(1/1.5)=0.67$ and $y_1=0+1=1 \Rightarrow(0.67,1)$

Second point: $x_2= 0.67+(1/1.5)=1.34$ and $y_2= 1+1=2 \Rightarrow (1.34,2)$

Third point: $x_3= 1.34+(1/1.5)=2$ and $y_3=2+1=3 \Rightarrow(2,3)$

Fourth point: $x_4= 2+(1/1.5)=2.67$ and $y_4=3+1=4 \Rightarrow(2.67,4)$

Fifth point: $x_5= 2.67+(1/1.5)=3.34$ and $y_5= 4+1=5 \Rightarrow(3.34,5)$

---

2. Write a program to generate five points along the line m=1.5, endpoint=(10.5,10) using DDA algorithm.

Solution

In this example, we only apply the DDA algorithm when m>1 and generating point start by the endpoint. The program is stated as follows:

```
#include<windows.h>
#include<GL/glut.h>
#include<GL/gl.h>
#include<math.h>
void Display()
{
    GLfloat x_k=10.50,y_k=10,M=1.05;
    glClear(GL_COLOR_BUFFER_BIT);
     int count;
     for (count=1;count<=5;count++)
    {
        x_k=x_k-1/M;
        y_k=y_k-1/M;
        glBegin(GL_POINTS);
        glVertex2f(x_k,y_k);
        glEnd();
     }
    glFlush();}
void Init()
{
    glClearColor(1.0,1.0,1.0,0);
    glColor3f(1.0,0.0,1.0);
    glPointSize(3.0);
    gluOrtho2D(0,15,0,15);
}
int main(int argc,char**argv)
{
    glutInit (&argc,argv);
    glutInitWindowSize(300,300);
    glutCreateWindow("generate five points with DDA");
    Init();
    glutDisplayFunc(Display);
    glutMainLoop();}
```

Output:

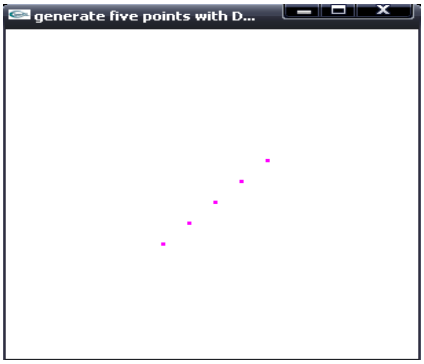


Fig.5-22: The output of DDA program.

3. Implement a line function using the DDA algorithm given any number (n) of input points. A single point is plotted when n=1?

Program

```
#include<windows.h>
#include<GL/gl.h>
#include<GL/glut.h>

 void Init(){
 glClearColor(1.0,1.0,1.0,0);
 glColor3f(0.0,0.0,0.0);
 glPointSize(5.0);
 gluOrtho2D(0,100,0,100);}

 void display(){
 GLfloat x=1,y=1,m=1.5;
 glClear(GL_COLOR_BUFFER_BIT);
 int n;
 if (m>=1)
 for(n=1;n<=50;n++){
 x=x+1;
 y=y+1/m;
 glBegin(GL_POINTS);
 glVertex2f(x,y);
 glEnd();}

 else{

 for(n=1;n<=50;n++){
 x=x/m;
 y=y+1;
 glBegin(GL_POINTS);
 glVertex2f(x,y);
 glEnd();}}
 glFlush();}

 void main(int argc,char**argv){
 glutInit (&argc,argv);
 glutInitWindowSize(250,250);
 glutCreateWindow("DDA");
 glutDisplayFunc(display);
 Init();
 glutMainLoop();
 }
```

Output:

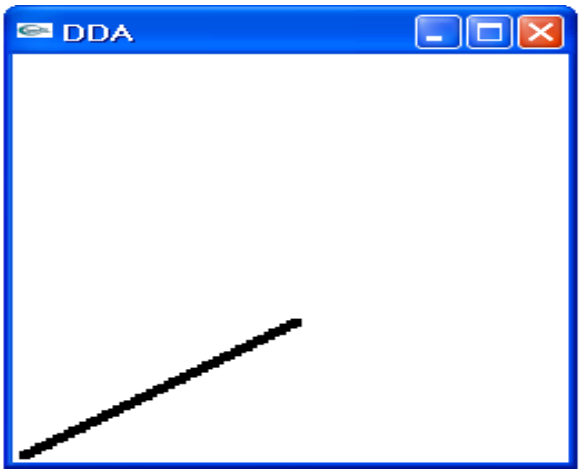


Fig.5-23: The output of DDA program.

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.

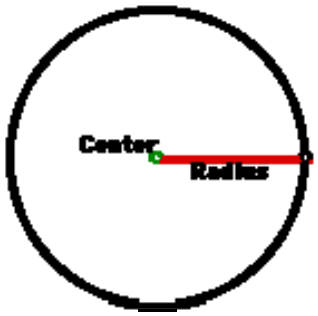


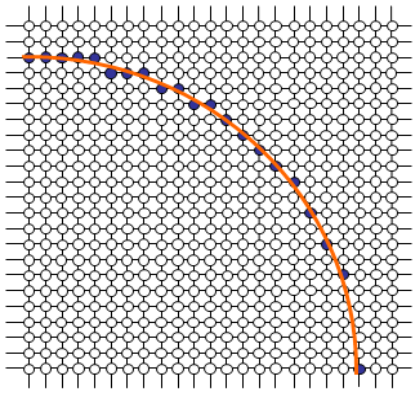
Fig.6-1: Generating circle.

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 $-r \le x \le r$

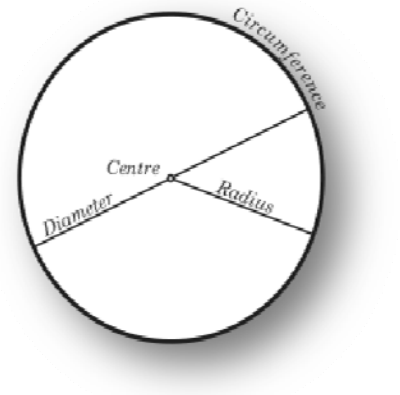


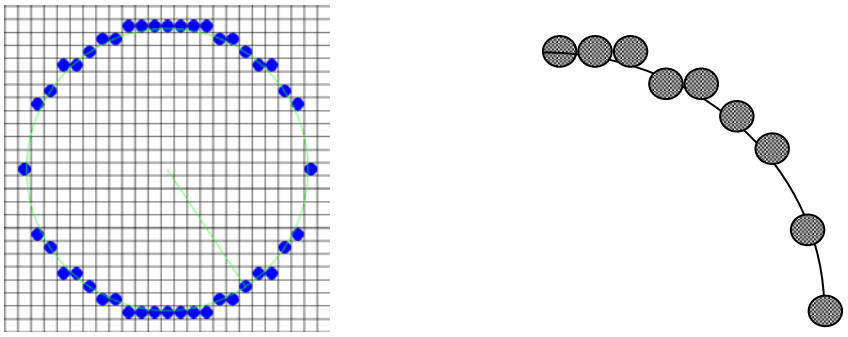
Fig.6-3: Circle in general form.

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).

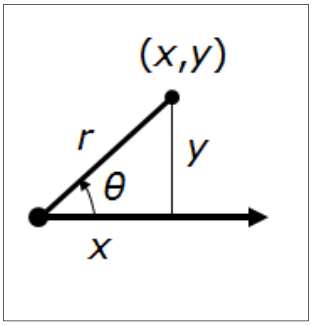


Fig.6-5: Representing a point (x,y) in polar coordinates.

- 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.

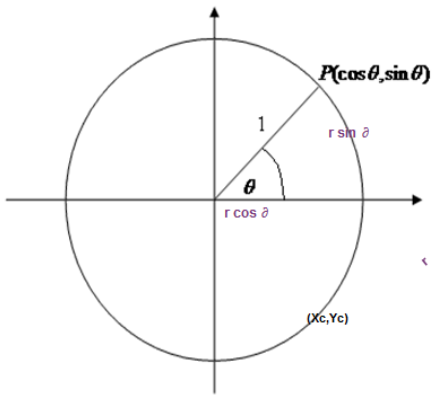


Fig.6-6: Point p on the circle.

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.

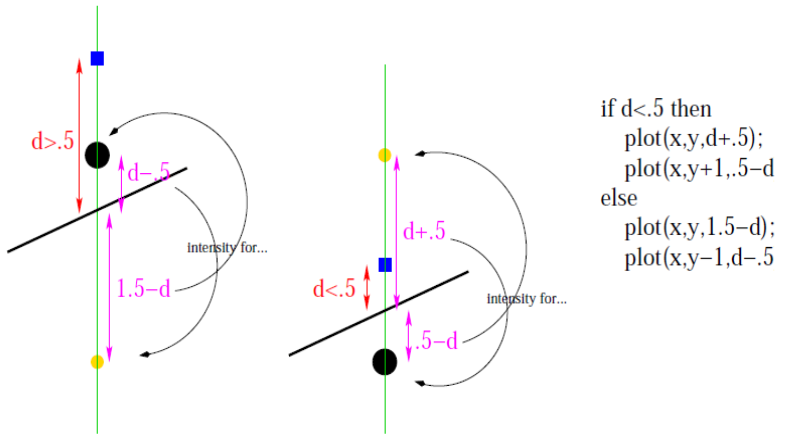


Fig.6-7: Drawing circle using *D(x,y)* inequality.

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.

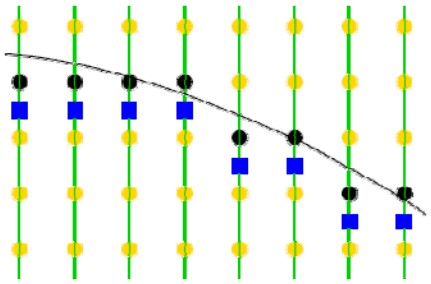


Fig. 6-8: Points are plotted using Bresenham's algorithm.

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

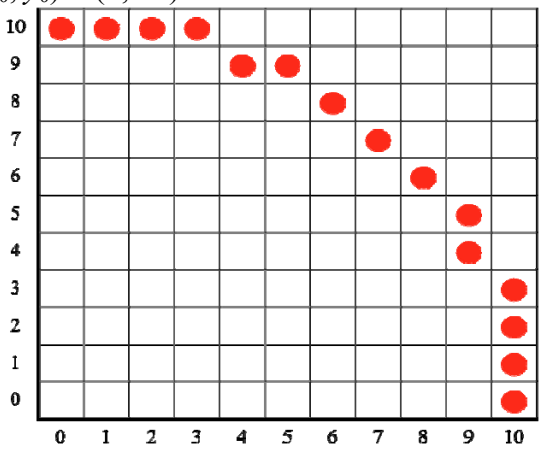
Draw a circle with radius r = 10 and center (0, 0) using Bresenham's algorithm.

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

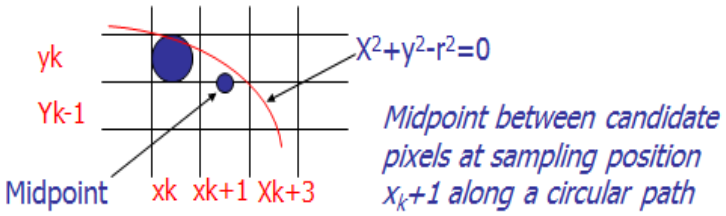
It is an algorithm used to determine the points needed for drawing a circle. The description of midpoint algorithm can be shown in Fig.6-9 and Fig.6-10.

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

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

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

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

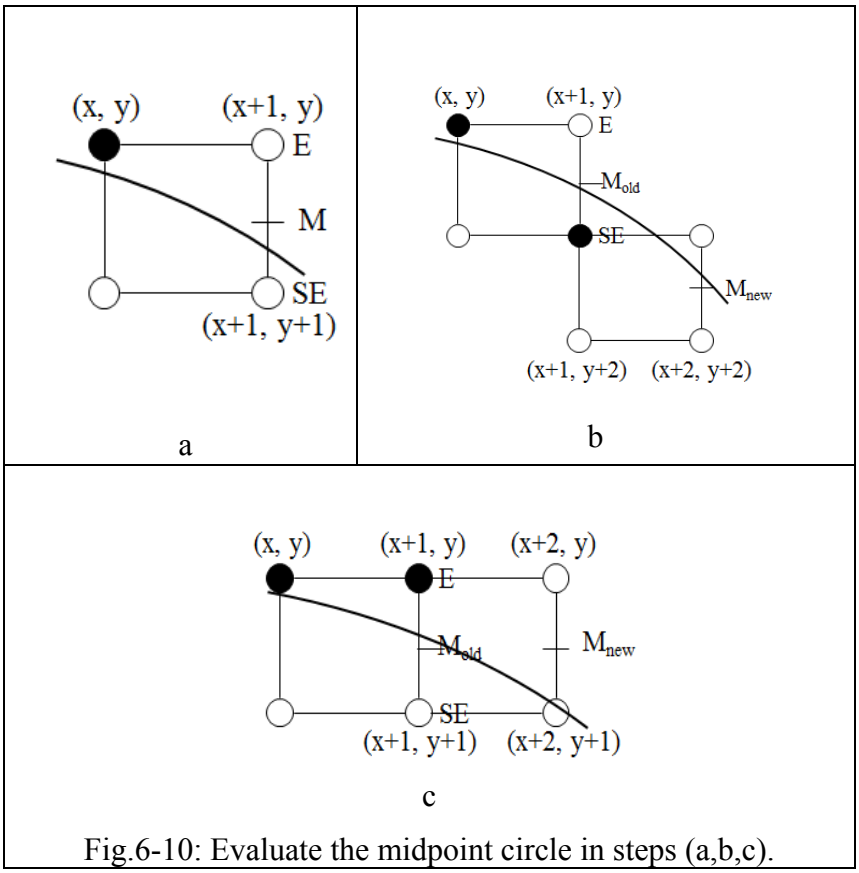


Fig.6-10: Evaluate the midpoint circle in steps (a,b,c).

- 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);
        }
}
```

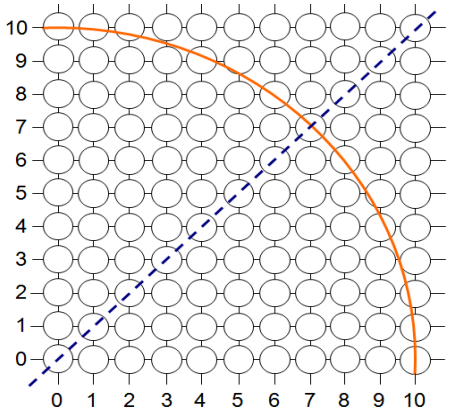
Bibliography

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:

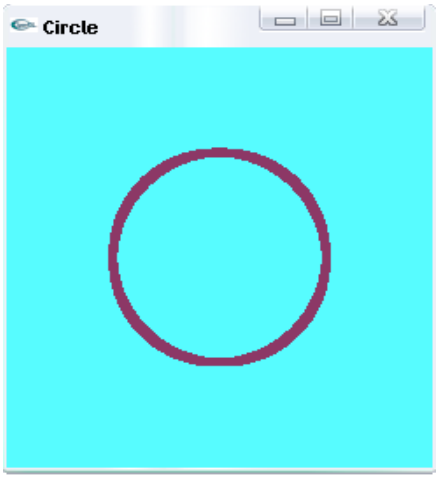


Fig.6-11: The output circle.

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:

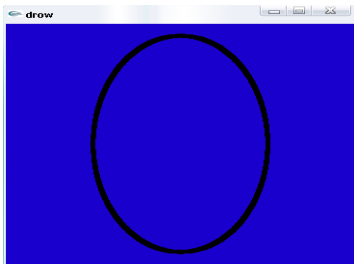


Fig.6-12: The output ellipse.