Introduction

In computer graphics, the midpoint circle algorithm is an algorithm used to determine the points needed for drawing a circle. The algorithm is a variant of Bresenham's line algorithm, and is thus sometimes known as Bresenham's circle algorithm, although not actually invented by Jack E. Bresenham. The algorithm can be generalized to conic sections.

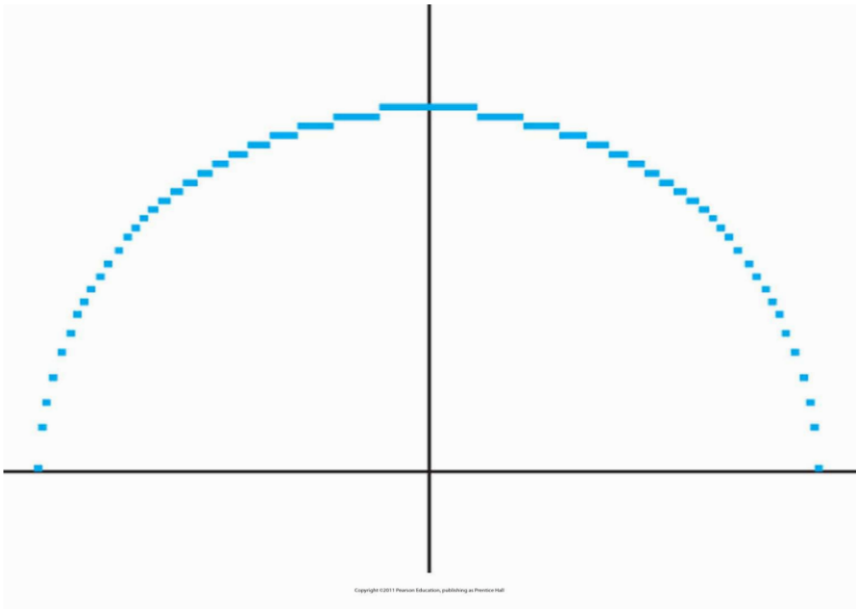
The purpose of Bresenham's circle algorithm is to generate the set of points that approximate a circle on a pixel-based display. We generate the points in the first octant of the circle and then use symmetry to generate the other seven octants. We start at (r, 0). This pixel is chosen trivially. Next we need to find the pixel for the y = 1 row. We must make a decision between two candidate points.

Need for the mid-point algorithm

We can write a simple circle drawing algorithm by solving the equation for y at unit x intervals using:

$$y = \pm\sqrt{r^2 - x^2}$$

This direct circle algorithm involves mathematically costly square root operation and squaring. It also does not make use of the symmetrical structure of the circle wasting some processing time. And since square root operation mostly yields floating point numbers a lot of approximations are to be done which causes the circle to lose its smooth curvature. Figure shown below is a semi-circle plotted by using the above method.



Derivation

The algorithm starts with the circle equation $x^2 + y^2 = r^2$. For simplicity, assume the center of the circle is at (0,0) . We consider first only the first octant and draw a curve which starts at point (r,0) and proceeds counterclockwise, reaching the angle of 45.

The "fast" direction here (the basis vector with the greater increase in value) is the y direction. The algorithm always takes a step in the positive y direction (upwards), and occasionally takes a step in the "slow" direction (the negative x direction).

From the circle equation we obtain the transformed equation $x^2 + y^2 - r^2 = 0$, where r^2 is computed only a single time during initialization.

Let the points on the circle be a sequence of coordinates of the vector to the point (in the usual basis). Points are numbered according to the order in which they are drawn, with n =1 assigned to the first point (r,0).

For each point, the following holds:

$$x_n^2 + y_n^2 = r^2$$

This can be rearranged as follows:

$$x_n^2 = r^2 - y_n^2$$

And likewise for the next point:

$$x_{n+1}^2 = r^2 - y_{n+1}^2$$

In general, it is true that:

$$\begin{aligned} y_{n+1}^2 &= (y_n + 1)^2 \\ &= y_n^2 + 2y_n + 1 \\ x_{n+1}^2 &= r^2 - y_n^2 - 2y_n - 1 \end{aligned}$$

So we refashion our next-point-equation into a recursive one by substituting

$$x_n^2 = r^2 - y_n^2:$$

$$x_{n+1}^2 = x_n^2 - 2y_n - 1$$

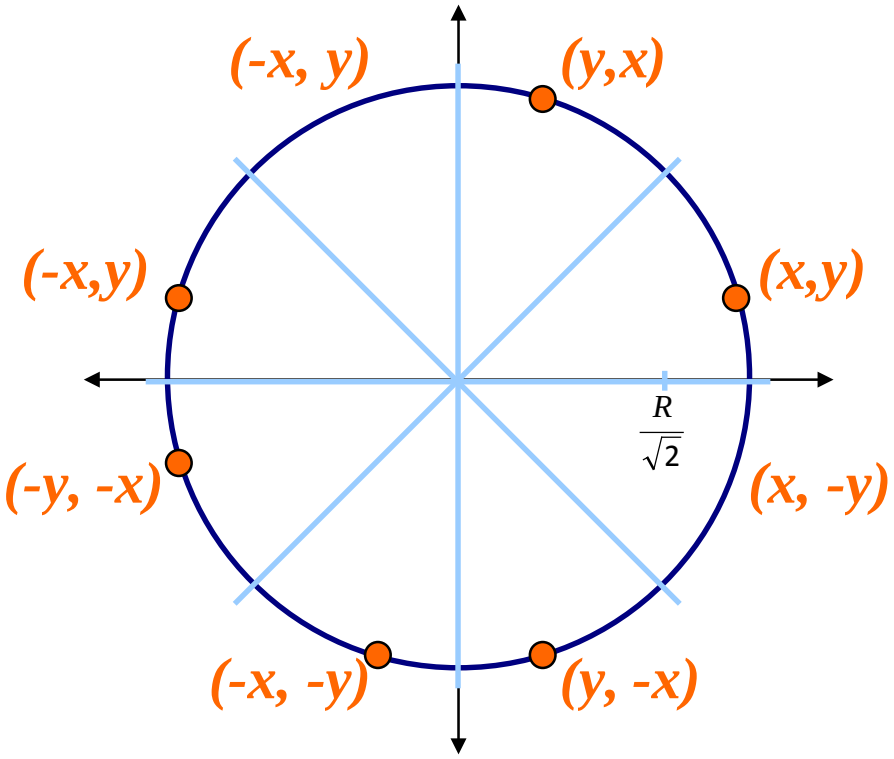
Because of the continuity of a circle and because the maxima along both axes is the same, we know we will not be skipping x points as we advance in the sequence. Usually we will stay on the same x coordinate, and sometimes advance by one.

The resulting co-ordinate is then translated by adding the midpoint coordinates. These frequent integer additions do not limit the performance much, as we can spare those square (root)

computations in the inner loop in turn. Again, the zero in the transformed circle equation is replaced by the error term.

The initialization of the error term is derived from an offset of ½ pixel at the start. Until the intersection with the perpendicular line, this leads to an accumulated value of r in the error term, so that this value is used for initialization.

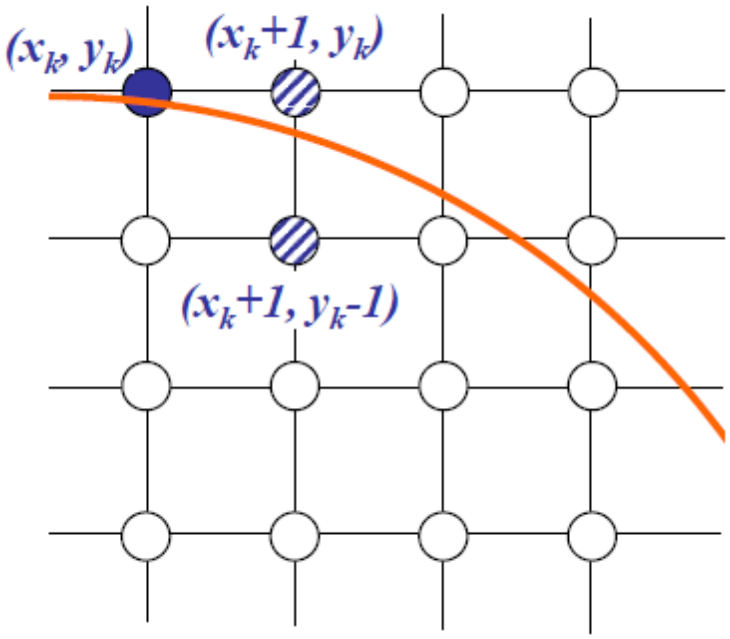
The frequent computations of squares in the circle equation, trigonometric expressions and square roots can again be avoided by dissolving everything into single steps and using recursive computation of the quadratic terms from the preceding iterations.



In the mid-point circle algorithm we use eight-way symmetry so only ever calculate the points for the top right eighth of a circle, and then use symmetry to get the rest of the points.

Assume that we have just plotted point (x_k, y_k). The next point is a choice between (x_k+1, y_k) and (x_k+1, y_k-1)

We would like to choose the point that is nearest to the actual circle



Rewriting the circle equation:

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

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

By evaluating this function at the midpoint between the candidate pixels we can make our decision.

To ensure things are as efficient as possible we can do all of our calculations incrementally.

First consider:

$$p_{k+1} = f_{circ}\left(x_{k+1}+1, y_{k+1}-\tfrac{1}{2}\right)$$

or:

$$= [(x_k+1)+1]^2 + \left(y_{k+1}-\tfrac{1}{2}\right)^2 - r^2$$

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

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

The first decision variable is given as:

$$p_0 = f_{circ}(1, r-\tfrac{1}{2})$$
$$= 1 + (r-\tfrac{1}{2})^2 - r^2$$
$$= \tfrac{5}{4} - r$$

Then if $p_k < 0$ then the next decision variable is given as:

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

If $p_k > 0$ then the decision variable is:

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

Algorithm

1. Input radius *r*and circle centre *(xc, yc)*, then set the coordinates for the first point on the circumference of a circle centred 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. Starting with *k = 0* at each position *xk*, perform the following test. If *pk < 0*, the next point along the circle centred on *(0, 0)*is *(xk+1, yk)* and:

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

   Otherwise the next point along the circle is *(xk+1, yk-1)* and:

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

4. Determine symmetry points in the other seven octants

5. Move each calculated pixel position *(x, y)*onto the circular path centred at *(xc, yc)*to plot the coordinate values:

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

6. Repeat steps 3 to 5 until *x >= y*

Arc

To draw arcs from angle a to angle b we make use of parametric equation form of a circle.

$x = r \cos(a)$

$y = r \sin(a)$

We start from $(x_c+r\cos(a), y_c+r\sin(a))$ and go on upto $(x_c+r\cos(b), y_c+r\sin(b))$ and plot all the points along the path in angular steps of a small value say 0.1.

Source Code

```
#include <stdio.h>
#include <GL/glut.h>
#include <math.h>

int xCenter, yCenter, radius;
float theta1, theta2;
int ch;
int c,d;
void drawCircle(float,float,float);
void drawArc(float,float,float,float,float);

void mouse(int btn, int state, int x, int y)
{
      static int count=0;
      float temp,rs;
    if(btn==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
    {count++;


            if(btn==GLUT_LEFT_BUTTON && state==GLUT_DOWN && (count%2==1) )
                  {
                        xCenter=x; yCenter=499-y;
                                    }

            if(btn==GLUT_LEFT_BUTTON && state==GLUT_DOWN &&
((count%2==0)&&(count!=0)))
                  {
                        c=x; d=499-y;
                        rs=(c-xCenter)*(c-xCenter)+(d-yCenter)*(d-yCenter);
                        temp=sqrt(abs(rs));
                        radius=temp;
                        if(ch == 2)
                        {
                              printf("\nEnter theta1 and theta2: \n");
                              scanf("%f %f",&theta1,&theta2);
```

```
                              drawArc((float)xCenter,(float)yCenter,theta1,
theta2,radius);
                        } else {
                        drawCircle((float)xCenter,(float)yCenter,radius);
                        glFlush();
                        }
                  }
            }

    if(btn==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
    {
        exit(1);
    }

}

void drawpoint(int x, int y)
{
      glColor3f(1,1,1);
      glBegin(GL_POINTS);
      glVertex2i(x,y);
      glEnd();
}

void drawArc(float xCenter, float yCenter,float theta1,float theta2,float
radius)
{
      if(theta1 > 360)
            while(theta1 >=360)
                  theta1-=360;

      if(theta2 > 360)
            while(theta2 >=360)
                  theta2-=360;

      if(theta1 > theta2)
      {
            for(float i = 0; i<=3.1417*theta2/180.0; i = i+0.005)
            drawpoint(xCenter+radius*cos(i), yCenter+radius*sin(i));

            for(float i = 3.1417*theta1/180.0; i<=2*3.1417; i = i+0.005)
            drawpoint(xCenter+radius*cos(i), yCenter+radius*sin(i));
      }
      else
      for(float i = 3.1417*theta1/180.0; i<=3.1417*theta2/180.0; i = i+0.005)
            drawpoint(xCenter+radius*cos(i), yCenter+radius*sin(i));

      glFlush();
}
```

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

void drawCircle(float xCenter, float yCenter, float radius)
{
	int x = 0;
	int y = radius;
	int p = 1 - radius;
	while(x < y) {
		x++;
		if(p < 0)
			p += 2 * x + 1;
		else {
			y--;
			p += 2 * (x-y) + 1;
		}

			circlePlotPoints(x, y);
		}
}

void display()
{
	glFlush();
}

void myinit()
{
	glClearColor(0,0,0,0);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluOrtho2D(0.0,499.0,0.0,499.0);
}

void main(int argc, char** argv)
{
	printf("1.Circle  2.Arc  3.Exit \n");
	scanf("%d",&ch);
	if(ch !=1 && ch!=2 )
	exit(0);
```

```
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowSize(500,500);
        glutInitWindowPosition(0,0);
        glutCreateWindow("Bresenham's circle");
        glutDisplayFunc(display);
        glutMouseFunc(mouse);
        myinit();
        glutMainLoop();
}
```