UNIT 2 Computer Graphics (BCA-407)

A **point** is a dimensionless shape, since it represents a dot only, whereas a line is a one-dimensional shape.

A line connects two points.

Both points are lines used to draw different shapes and sizes in a plane. These shapes can be two-dimensional or three-dimensional.

DDA Algorithm:

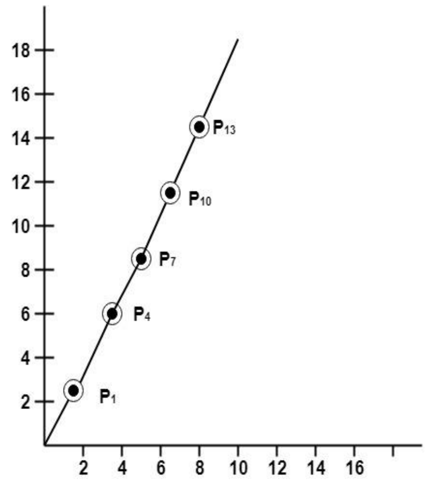
```
Step1: Start Algorithm

Step2: Declare $x_1$,$y_1$,$x_2$,$y_2$,dx,dy,x,y as integer variables.

Step3: Enter value of $x_1$,$y_1$,$x_2$,$y_2$.

Step4: Calculate $dx = x_2-x_1$

Step5: Calculate $dy = y_2-y_1$

Step6: If ABS (dx) > ABS (dy)
        Then step = abs (dx)
        Else

Step7: $x_{inc}$=dx/step
        $y_{inc}$=dy/step
        assign $x = x_1$
        assign $y = y_1$

Step8: Set pixel (x, y)

Step9: $x = x + x_{inc}$
        $y = y + y_{inc}$
        Set pixels (Round (x), Round (y))

Step10: Repeat step 9 until $x = x_2$

Step11: End Algorithm
```

Example: If a line is drawn from (2, 3) to (6, 15) with use of DDA. How many points will needed to generate such line?

Solution: P_1 (2,3) P_{11} (6,15)

x_1=2
y_1=3
x_2= 6
y_2=15
dx = 6 - 2 = 4
dy = 15 - 3 = 12

$P_1(2,3)$	point plotted
$P_2(2\frac{1}{3},4)$	point plotted
$P_3(2\frac{2}{3},5)$	point not plotted
$P_4(3,6)$	point plotted
$P_5(3\frac{1}{3},7)$	point not plotted
$P_6(3\frac{2}{3},8)$	point not plotted
$P_7(4,9)$	point plotted
$P_8(4\frac{1}{3},10)$	point not plotted
$P_9(4\frac{2}{3},11)$	point not plotted
$P_{10}(5,12)$	point plotted
$P_{11}(5\frac{1}{3},13)$	point not plotted
$P_{12}(5\frac{2}{3},14)$	point not plotted
$P_{13}(6,15)$	point plotted



Advantage:

1. It is a faster method than method of using direct use of line equation.
2. This method does not use multiplication theorem.

3. It allows us to detect the change in the value of x and y ,so plotting of same point twice is not possible.
4. This method gives overflow indication when a point is repositioned.
5. It is an easy method because each step involves just two additions.

Disadvantage:

1. It involves floating point additions rounding off is done. Accumulations of round off error cause accumulation of error.
2. Rounding off operations and floating point operations consumes a lot of time.
3. It is more suitable for generating line using the software. But it is less suited for hardware implementation.

Program to implement DDA Line Drawing Algorithm:

```
 #include<graphics.h>
 #include<conio.h>
 #include<stdio.h>
 void main()
 {
  intgd = DETECT ,gm, i;
float x, y,dx,dy,steps;
int x0, x1, y0, y1;
initgraph(&gd, &gm, "C:\\TC\\BGI");
setbkcolor(WHITE);
x0 = 100 , y0 = 200, x1 = 500, y1 = 300;
dx = (float)(x1 - x0);
dy = (float)(y1 - y0);
if(dx>=dy)
      {
   steps = dx;
}
else
      {
   steps = dy;
}
dx = dx/steps;
```

```
    dy = dy/steps;
    x = x0;
    y = y0;
    i = 1;
    while(i<= steps)
    {
        putpixel(x, y, RED);
        x += dx;
        y += dy;
        i=i+1;
    }
    getch();
    closegraph();
}
```

Bresenham's Line Algorithm:

Step1: Start Algorithm

Step2: Declare variable $x_1,x_2,y_1,y_2,d,i_1,i_2,dx,dy$

Step3: Enter value of x_1,y_1,x_2,y_2
Where x_1,y_1 are coordinates of starting point
And x_2,y_2 are coordinates of Ending point

Step4: Calculate $dx = x_2-x_1$
Calculate $dy = y_2-y_1$
Calculate $i_1=2*dy$
Calculate $i_2=2*(dy-dx)$
Calculate $d=i_1-dx$

Step5: Consider (x, y) as starting point and x_{end} as maximum possible value of x.

```
If dx < 0
        Then x = x2
        y = y2
          xend=x1
If dx > 0
    Then x = x1
y = y1
        xend=x2
```

Step6: Generate point at (x,y)coordinates.

Step7: Check if whole line is generated.

```
    If x > = $x_{end}$
    Stop.
```

Step8: Calculate co-ordinates of the next pixel

```
    If d < 0
        Then d = d + $i_1$
    If d ≥ 0
  Then d = d + $i_2$
    Increment y = y + 1
```

Step9: Increment x = x + 1

Step10: Draw a point of latest (x, y) coordinates

Step11: Go to step 7

Step12: End of Algorithm

Example: Starting and Ending position of the line are (1, 1) and (8, 5). Find intermediate points.

Solution: $x_1=1$

$y_1=1$

$x_2=8$

$y_2=5$

$dx= x_2-x_1=8-1=7$

$dy=y_2-y_1=5-1=4$

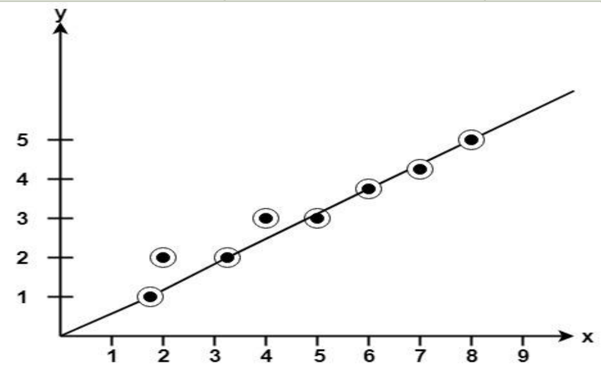
$I_1=2^* \Delta y=2^*4=8$

$I_2=2^*(\Delta y-\Delta x)=2^*(4-7)=-6$

$d = I_1-\Delta x=8-7=1$

x	**y**	**d=d+I_1 or I_2**
1	1	$d+I_2=1+(-6)=-5$
2	2	$d+I_1=-5+8=3$

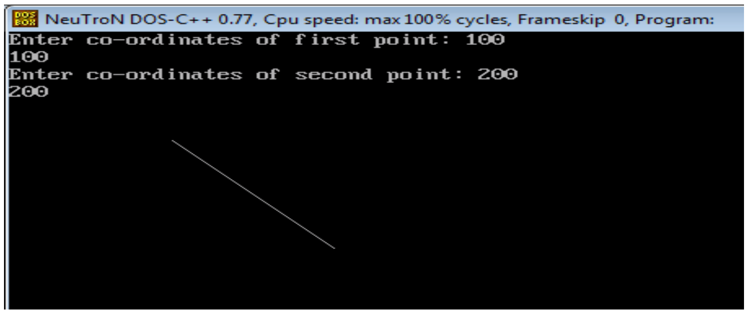
3	2	$d+I_2=3+(-6)=-3$
4	3	$d+I_1=-3+8=5$
5	3	$d+I_2=5+(-6)=-1$
6	4	$d+I_1=-1+8=7$
7	4	$d+I_2=7+(-6)=1$
8	5	



Program to implement Bresenham's Line Drawing Algorithm:

```
 #include<stdio.h>
#include<graphics.h>
void drawline(int x0, int y0, int x1, int y1)
{
 int dx, dy, p, x, y;
 dx=x1-x0;
 dy=y1-y0;
 x=x0;
 y=y0;
 p=2*dy-dx;
 while(x<x1)
 {
    if(p>=0)
    {
        putpixel(x,y,7);
        y=y+1;
        p=p+2*dy-2*dx;
    }
    else
    {
        putpixel(x,y,7);
        p=p+2*dy;}
        x=x+1;
    }
}
    int main()
{
 int gdriver=DETECT, gmode, error, x0, y0, x1, y1;
 initgraph(&gdriver, &gmode, "c:\\turboc3\\bgi");
 printf("Enter co-ordinates of first point: ");
 scanf("%d%d", &x0, &y0);
 printf("Enter co-ordinates of second point: ");
 scanf("%d%d", &x1, &y1);
 drawline(x0, y0, x1, y1);
 return 0;
}
```

Output:


NeuTroN DOS-C++ 0.77, Cpu speed: max 100% cycles, Frameskip 0, Program:

```
Enter co-ordinates of first point: 100
100
Enter co-ordinates of second point: 200
200
```

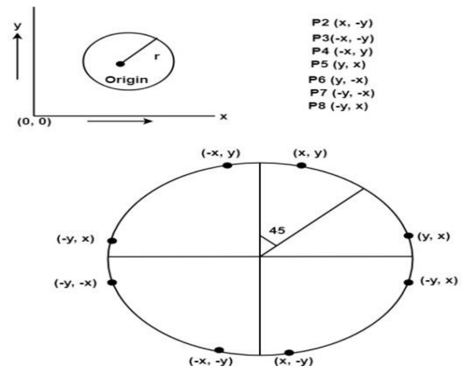
Differentiate between DDA Algorithm and Bresenham's Line Algorithm:

DDA Algorithm	**Bresenham's Line Algorithm**
1. DDA Algorithm use floating point, i.e., Real Arithmetic.	1. Bresenham's Line Algorithm use fixed point, i.e., Integer Arithmetic
3. DDA Algorithm is slowly than Bresenham's Line Algorithm in line drawing because it uses real arithmetic (Floating Point operation)	3. Bresenham's Algorithm is faster than DDA Algorithm in line because it involves only integer arithmetic.
4. DDA Algorithm is not accurate and efficient as Bresenham's Line Algorithm.	4. Bresenham's Line Algorithm is more accurate and efficient at DDA Algorithm.
5.DDA Algorithm can draw circle and curves but are not accurate as Bresenham's Line Algorithm	5. Bresenham's Line Algorithm can draw circle and curves with more accurate than DDA Algorithm.

Defining a Circle:

Circle is an eight-way symmetric figure. The shape of circle is the same in all quadrants. In each quadrant, there are two octants. If the calculation of the point of one octant is done, then the other seven points can be calculated easily by using the concept of eight-way symmetry.

For drawing, circle considers it at the origin. If a point is $P_1(x, y)$, then the other seven points will be



So we will calculate only 45°arc. From which the whole circle can be determined easily.

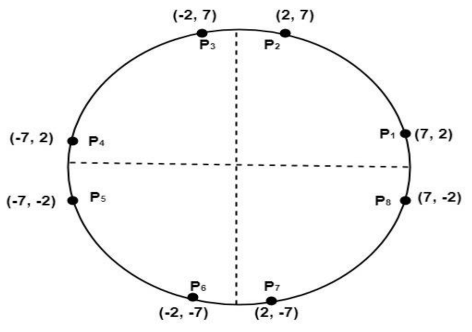
If we want to display circle on screen then the putpixel function is used for eight points as shown below:

```
putpixel (x, y, color)
putpixel (x, -y, color)
putpixel (-x, y, color)
putpixel (-x, -y, color)
putpixel (y, x, color)
putpixel (y, -x, color)
putpixel (-y, x, color)
putpixel (-y, -x, color)
```

Example: Let we determine a point (2, 7) of the circle then other points will be (2, -7), (-2, -7), (-2, 7), (7, 2), (-7, 2), (-7, -2), (7, -2)

These seven points are calculated by using the property of reflection. The reflection is accomplished in the following way:

The reflection is accomplished by reversing x, y co-ordinates.



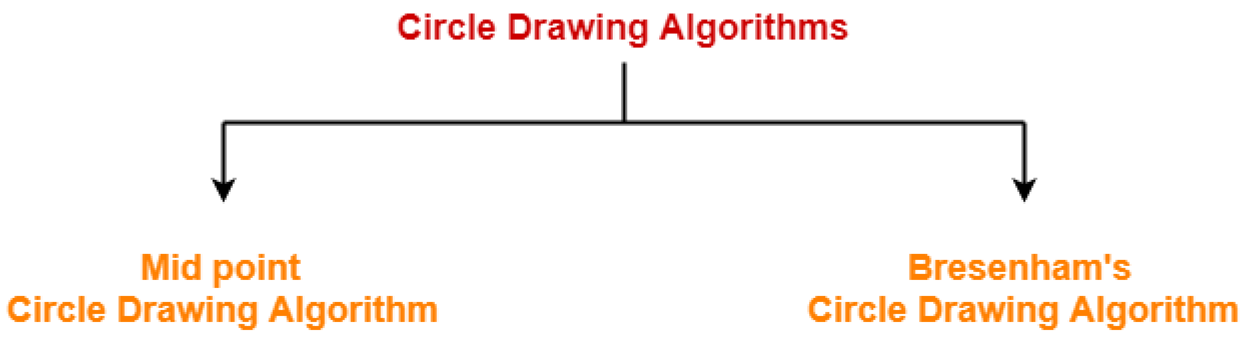
Eight way symmetry of a Circle

There are two standards methods of mathematically defining a circle centered at the origin.

1. Defining a circle using Polynomial Method
2. Defining a circle using Polar Co-ordinates

Circle Drawing Algorithms-

In computer graphics, popular algorithms used to generate circle are-



1. Mid Point Circle Drawing Algorithm
2. Bresenham's Circle Drawing Algorithm

In this article, we will discuss about Mid Point Circle Drawing Algorithm.

Mid Point Circle Drawing Algorithm-

Given the centre point and radius of circle,

Mid Point Circle Drawing Algorithm attempts to generate the points of one octant.

The points for other octants are generated using the eight symmetry property.

Procedure-

Given-

- Centre point of Circle = (X_0, Y_0)
- Radius of Circle = R

The points generation using Mid Point Circle Drawing Algorithm involves the following steps-

Step-01:

Assign the starting point coordinates (X_0, Y_0) as-

- $X_0 = 0$
- $Y_0 = R$

Step-02:

Calculate the value of initial decision parameter P_0 as-

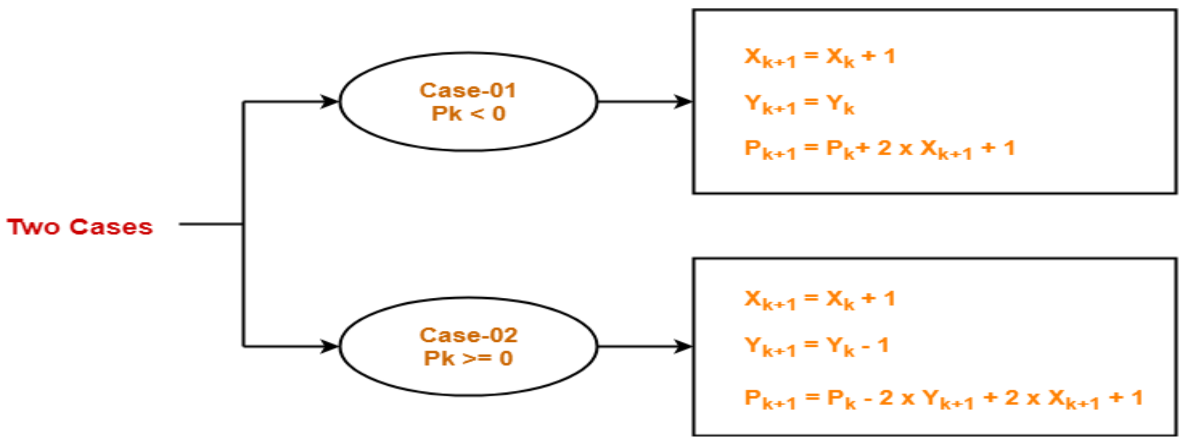
$$P_0 = 1 - R$$

Step-03:

Suppose the current point is (X_k, Y_k) and the next point is (X_{k+1}, Y_{k+1}).

Find the next point of the first octant depending on the value of decision parameter P_k.

Follow the below two cases-



Step-04:

If the given centre point (X_0, Y_0) is not (0, 0), then do the following and plot the point-

- $X_{plot} = X_c + X_0$
- $Y_{plot} = Y_c + Y_0$

Here, (X_c, Y_c) denotes the current value of X and Y coordinates.

Step-05:

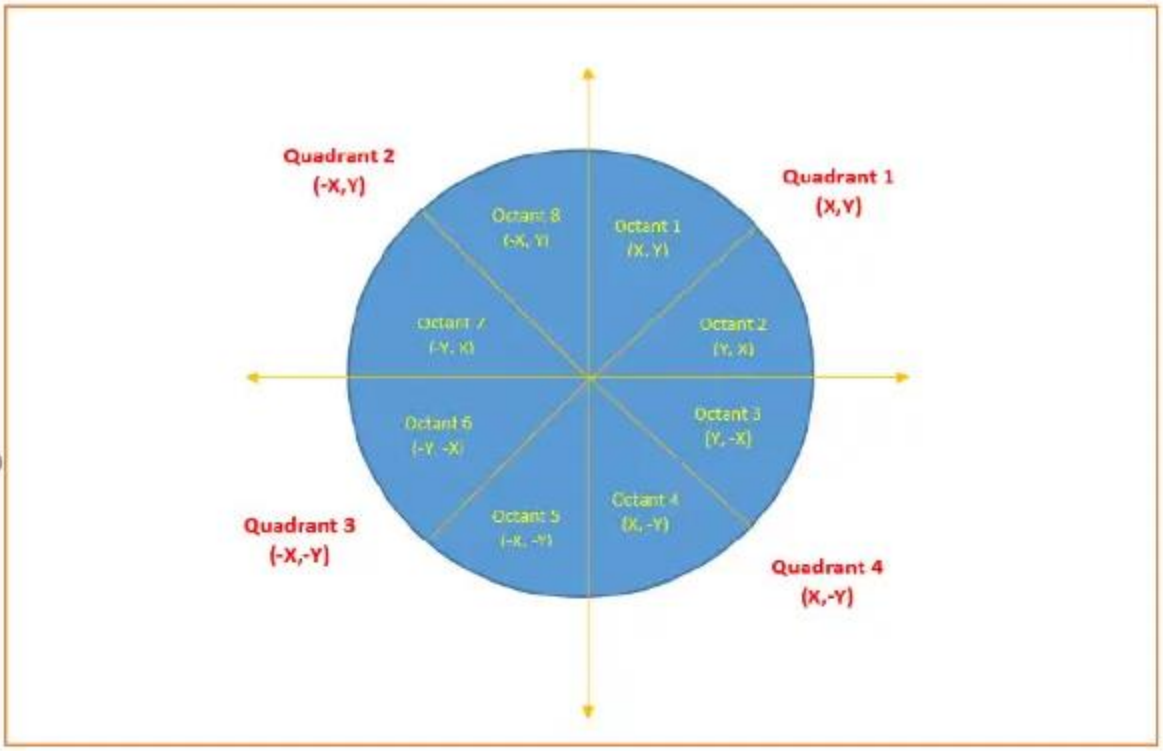
Keep repeating Step-03 and Step-04 until $X_{plot} >= Y_{plot}$.

Step-06:

Step-05 generates all the points for one octant.

To find the points for other seven octants, follow the eight symmetry property of circle.

This is depicted by the following figure-



Problem-01:

Given the centre point coordinates (0, 0) and radius as 10, generate all the points to form a circle.

Solution-

Given-

- Centre Coordinates of Circle $(X_0, Y_0) = (0, 0)$
- Radius of Circle = 10

Step-01:

Assign the starting point coordinates (X_0, Y_0) as-

- $X_0 = 0$
- $Y_0 = R = 10$

Step-02:

Calculate the value of initial decision parameter P_0 as-

$P_0 = 1 – R$

$P_0 = 1 – 10$

$P_0 = -9$

Step-03:

As $P_{initial} < 0$, so case-01 is satisfied.

Thus,

- $X_{k+1} = X_k + 1 = 0 + 1 = 1$
- $Y_{k+1} = Y_k = 10$
- $P_{k+1} = P_k + 2 \times X_{k+1} + 1 = -9 + (2 \times 1) + 1 = -6$

Step-04:

This step is not applicable here as the given centre point coordinates is (0, 0).

Step-05:

Step-03 is executed similarly until $X_{k+1} >= Y_{k+1}$ as follows-

P_k	P_{k+1}	(X_{k+1}, Y_{k+1})
		(0, 10)
-9	-6	(1, 10)
-6	-1	(2, 10)
-1	6	(3, 10)
6	-3	(4, 9)
-3	8	(5, 9)
8	5	(6, 8)

Algorithm Terminates

These are all points for Octant-1.

Algorithm calculates all the points of octant-1 and terminates.

Now, the points of octant-2 are obtained using the mirror effect by swapping X and Y coordinates.

Octant-1 Points	Octant-2 Points
(0, 10)	(8, 6)
(1, 10)	(9, 5)
(2, 10)	(9, 4)
(3, 10)	(10, 3)
(4, 9)	(10, 2)
(5, 9)	(10, 1)
(6, 8)	(10, 0)
These are all points for Quadrant-1.	

Now, the points for rest of the part are generated by following the signs of other quadrants.

The other points can also be generated by calculating each octant separately.

Here, all the points have been generated with respect to quadrant-1-

Quadrant-1 (X,Y)	**Quadrant-2 (-X,Y)**	**Quadrant-3 (-X,-Y)**	**Quadrant-4 (X,-Y)**
(0, 10)	(0, 10)	(0, -10)	(0, -10)
(1, 10)	(-1, 10)	(-1, -10)	(1, -10)
(2, 10)	(-2, 10)	(-2, -10)	(2, -10)
(3, 10)	(-3, 10)	(-3, -10)	(3, -10)
(4, 9)	(-4, 9)	(-4, -9)	(4, -9)
(5, 9)	(-5, 9)	(-5, -9)	(5, -9)
(6, 8)	(-6, 8)	(-6, -8)	(6, -8)
(8, 6)	(-8, 6)	(-8, -6)	(8, -6)
(9, 5)	(-9, 5)	(-9, -5)	(9, -5)
(9, 4)	(-9, 4)	(-9, -4)	(9, -4)
(10, 3)	(-10, 3)	(-10, -3)	(10, -3)
(10, 2)	(-10, 2)	(-10, -2)	(10, -2)
(10, 1)	(-10, 1)	(-10, -1)	(10, -1)
(10, 0)	(-10, 0)	(-10, 0)	(10, 0)
These are all points of the Circle.			

Advantages of Mid Point Circle Drawing Algorithm-

The advantages of Mid Point Circle Drawing Algorithm are-

- It is a powerful and efficient algorithm.
- The entire algorithm is based on the simple equation of circle $X^2 + Y^2 = R^2$.
- It is easy to implement from the programmer’s perspective.

- This algorithm is used to generate curves on raster displays.

Disadvantages of Mid Point Circle Drawing Algorithm-

The disadvantages of Mid Point Circle Drawing Algorithm are-

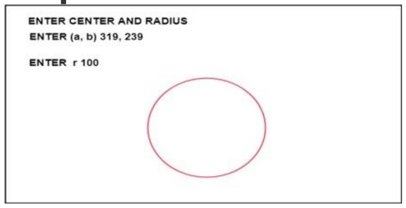
- Accuracy of the generating points is an issue in this algorithm.
- The circle generated by this algorithm is not smooth.
- This algorithm is time consuming.

Program to draw a circle using Midpoint Algorithm:

```
#include <graphics.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <conio.h>
int main ()
{
    int gdriver = DETECT,gmode, errorcode;
  //  int midx, midy, i;
  int xc,yc,r;
   float x, y,p;

     printf("ENTER CENTER ");
     scanf("%d%d",&xc,&yc);
     printf("%d%d",xc,yc);

   printf("ENTER r");
  scanf("%d",&r);
   printf("%d",r);

  p= -r;
printf("%d",p);
initgraph (&gdriver, &gmode, NULL);
    x=0;
    y=r;

    p=(1-r);
    printf("%d",p);
    while (x<=y)
    {
        if (p<0)
```

```
      {

       p= p+(2*x)+1;printf("%d",p);
      }
       else
       {
           p+=(2*(x-y))+1;
           y--;
       }
       x++;
       putpixel(x+xc,y+yc,RED);
             putpixel(x+xc,-y+yc,YELLOW);
             putpixel(-x+xc,-y+yc,GREEN);
             putpixel(-x+xc,y+yc,YELLOW);
             putpixel(y+xc,x+yc,12);
             putpixel(y+xc,-x+yc,14);
             putpixel(-y+xc,-x+yc,15);
             putpixel(-y+xc,x+yc,6);

   }
   getch();
   return 0;
}
```

Output:



```
ENTER CENTER AND RADIUS
ENTER (a, b) 319, 239

ENTER  r 100
```

Bresenham's Circle Algorithm:

Step1: Start Algorithm

Step2: Declare p, q, x, y, r, d variables
        p, q are coordinates of the center of the circle
        r is the radius of the circle

Step3: Enter the value of r

Step4: Calculate d = 3 - 2r

Step5: Initialize x=0
& y= r

Step6: Check if the whole circle is scan converted

```
If x > = y
Stop
```

Step7: Plot eight points by using concepts of eight-way symmetry. The center is at (p, q). Current active pixel is (x, y).

```
putpixel (x+p, y+q)
putpixel (y+p, x+q)
putpixel (-y+p, x+q)
putpixel (-x+p, y+q)
putpixel (-x+p, -y+q)
putpixel (-y+p, -x+q)
putpixel (y+p, -x+q)
putpixel (x+p, -y-q)
```

Step8: Find location of next pixels to be scanned

```
If d < 0
then d = d + 4x + 6
increment x = x + 1
If d ≥ 0
then d = d + 4 (x - y) + 10
increment x = x + 1
decrement y = y - 1
```

Step9: Go to step 6

Step10: Stop Algorithm

Example: Plot 6 points of circle using Bresenham Algorithm. When radius of circle is 10 units. The circle has centre (50, 50).

Solution: Let r = 10 (Given)

Step1: Take initial point (0, 10)

$d = 3 - 2r$

$d = 3 - 2 * 10 = -17$

$d < 0 \therefore d = d + 4x + 6$

$= -17 + 4 (0) + 6$

$= -11$

Step2: Plot (1, 10)

$d = d + 4x + 6$ ($\because d < 0$)

$= -11 + 4 (1) + 6$

$= -1$

Step3: Plot (2, 10)

$d = d + 4x + 6$ ($\because d < 0$)

$= -1 + 4 \times 2 + 6$

$= 13$

Step4: Plot (3, 10) d is > 0 so x = x + 1, y = y - 1

$d = d + 4 (x-y) + 10$ ($\because d > 0$)

$= 13 + 4 (3-10) + 10$

$= 13 + 4 (-7) + 10$

$= 23-28=-5$

Step5: Plot (4, 9)

$d = -1 + 4x + 6$

$= -1 + 4 (4) + 6$

$= 21$

Step6: Plot (5, 8)

$d = d + 4 (x-y) + 10$ ($\because d > 0$)

$= 21 + 4 (5-8) + 10$

$= 21-12 + 10 = 19$

So P_1 (0,0)$\Longrightarrow$(50,50)

P_2 (1,10)$\Longrightarrow$(51,60)

P_3 (2,10)$\Longrightarrow$(52,60)

P_4 (3,9)$\Longrightarrow$(53,59)

P_5 (4,9)$\Longrightarrow$(54,59)

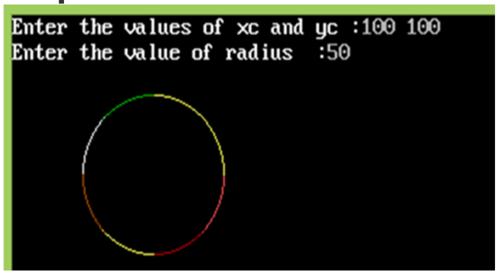
P_6 (5,8)$\Longrightarrow$(55,58)

Program to draw a circle using Bresenham's circle drawing algorithm:

```
#include <graphics.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <conio.h>
void    EightWaySymmetricPlot(int xc,int yc,int x,int y)
  {
   putpixel(x+xc,y+yc,1);
   putpixel(x+xc,-y+yc,2);
   putpixel(-x+xc,-y+yc,3);
   putpixel(-x+xc,y+yc,4);
   putpixel(y+xc,x+yc,1);
   putpixel(y+xc,-x+yc,4);
   putpixel(-y+xc,-x+yc,3);
   putpixel(-y+xc,x+yc,2);
  }

   void BresenhamCircle(int xc,int yc,int r)
  {
   int x=0,y=r,d=3-(2*r);
   EightWaySymmetricPlot(xc,yc,x,y);

   while(x<=y)
    {
     if(d<0)
            {
       d=d+(4*x)+6;
     }
    else
     {
       d=d+(4*x)-(4*y)+10;
       y=y-1;
     }
      x=x+1;
      EightWaySymmetricPlot(xc,yc,x,y);
     }
   }

   int  main()
  {
```

```
int xc,yc,r,gdriver = DETECT, gmode, errorcode;

 initgraph(&gdriver, &gmode, "C:\\TURBOC3\\BGI");


   printf("Enter the values of xc and yc :");
   scanf("%d%d",&xc,&yc);
      printf("%d%d",xc,yc);
   printf("Enter the value of radius  :");
   scanf("%d",&r);
   printf("%d",r);
   BresenhamCircle(xc,yc,r);
 getch();
 closegraph();
 return 0;
}
```

Output:



```
Enter the values of xc and yc :100 100
Enter the value of radius  :50
```

Filled Area Primitives:

Note: For filling a given picture or object with color’s, we can do it in two ways in C programming. The two ways are given below:

i. Using filling algorithms such as Floodfill algorithm, Boundary fill algorithm and scanline polygon fill algorithm, we can color the objects.

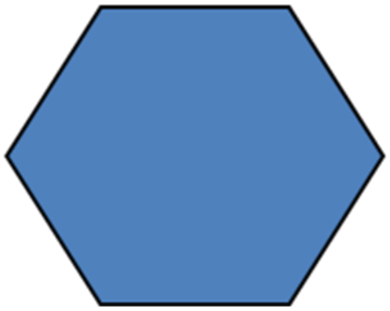
ii. Using inbuilt graphics functions such as floodfill(),setfillstyle() we can fill the object with color’s directly without using any filling algorithm.

Here we will see the **filling algorithms Polygon Filling**

Types of filling

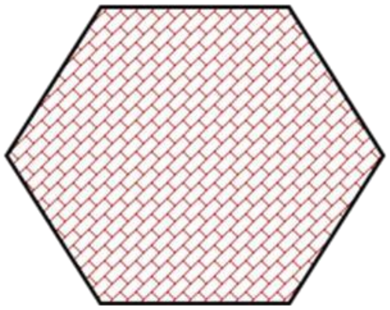
- Solid-fill

All the pixels inside the polygon’s boundary are illuminated.



- Pattern-fill

the polygon is filled with an arbitrary predefined pattern.



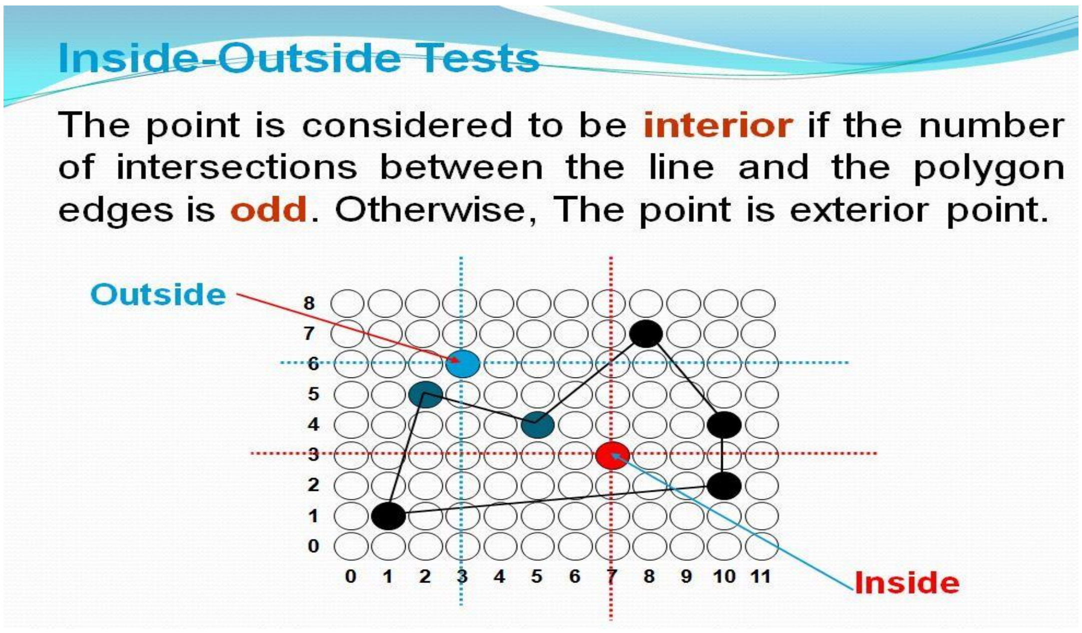
Polygon Representation

- The polygon can be represented by listing its n vertices in an ordered list. P = {(x_1, y_1), (x_2, y_2), ……., (x_n, y_n)}.
- The polygon can be displayed by drawing a line between (x_1, y_1), and (x_2, y_2), then a line between (x_2, y_2), and (x_3, y_3), and so on until the end vertex. In order to close up the polygon, a line between (x_n, y_n), and (x_1, y_1) must be drawn.
- One problem with this representation is that if we wish to translate the polygon, it is necessary to apply the translation transformation to each vertex in order to obtain the translated polygon.

Inside-Outside Tests

- When filling polygons we should decide whether a particular point is interior or exterior to a polygon.
- A rule called the odd-parity (or the odd-even rule) is applied to test whether a point is interior or not.
- To apply this rule, we conceptually draw a line starting from the particular point and extending to a distance point outside the coordinate extends of the object in any direction such that no polygon vertex intersects with the line.

Inside-Outside Tests



Inside-Outside Tests

The point is considered to be **interior** if the number of intersections between the line and the polygon edges is **odd**. Otherwise, The point is exterior point.

The Filling Algorithms are 3:

They are:

Three Algorithms for filling areas:

- 1) Scan Line Polygon Fill Algorithm
- 2) Boundary Fill Algorithm
- 3)Flood fill Algorithm

Boundary Fill Algorithm:

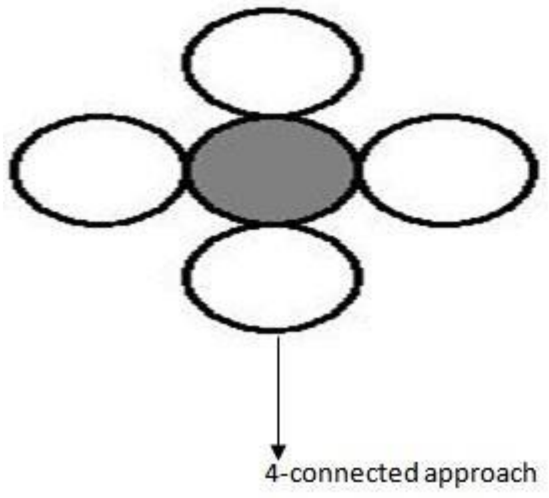
- Start at a point inside a region and paint the interior outward toward the boundary.

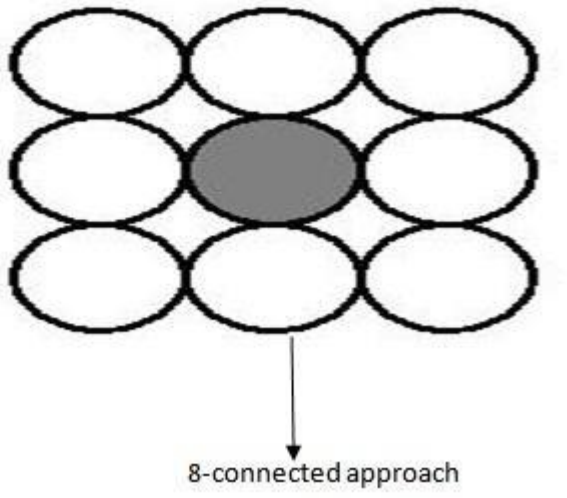
- If the boundary is specified in a single color, the fill algorithm processed outward pixel by pixel until the boundary color is encountered.
- A boundary-fill procedure accepts as input the coordinate of the interior point (x, y), a fill color, and a boundary color.

Algorithm:

The following steps illustrate the idea of the recursive boundary-fill algorithm:

1. Start from an interior point.
2. If the current pixel is not already filled and if it is not an edge point, then set the pixel with the fill color, and store its neighboring pixels (**4 or 8-connected**). Store only neighboring pixel that is not already filled and is not an edge point.
3. Select the next pixel from the stack, and continue with step 2.





In 4 connected approach, we can fill an object in only 4 directions. We have 4 possibilities for proceeding to next pixel from current pixel.	In 8 connected approach, we can fill an object in 8 directions. We have 8 possibilities for proceeding to next pixel from current pixel.

Function for 4 connected approach:

```
Step 1> boundary_fill(int x, int y, int fcolor, int bcolor)


Step 2>if ((getpixel(x, y) != bcolor)
             && (getpixel(x, y) != fcolor))
                        delay(10);
Step 3> putpixel(x, y, fcolor);
Step 4> boundary_fill(x + 1, y, fcolor, bcolor);
             boundary_fill(x - 1, y, fcolor, bcolor);
             boundary_fill(x, y + 1, fcolor, bcolor);
             boundary_fill(x, y - 1, fcolor, bcolor);
```

Function for 8 connected approach:

```
Step 1>boundary_fill(int x, int y, int fcolor, int bcolor)


Step 2>if ((getpixel(x, y) != bcolor)
            && (getpixel(x, y) != fcolor))
            delay(10);
Step 3>putpixel(x, y, fcolor);
Step 4>boundary_fill(x + 1, y, fcolor, bcolor);
            boundary_fill(x , y+1, fcolor, bcolor);
            boundary_fill(x+1, y + 1, fcolor, bcolor);
           boundary_fill(x-1, y - 1, fcolor, bcolor);
           boundary_fill(x-1, y, fcolor, bcolor);
           boundary_fill(x , y-1, fcolor, bcolor);
           boundary_fill(x-1, y + 1, fcolor, bcolor);
           boundary_fill(x+1, y - 1, fcolor, bcolor);
```

Advantages:

- The boundary fill algorithm is used to create attractive paintings.

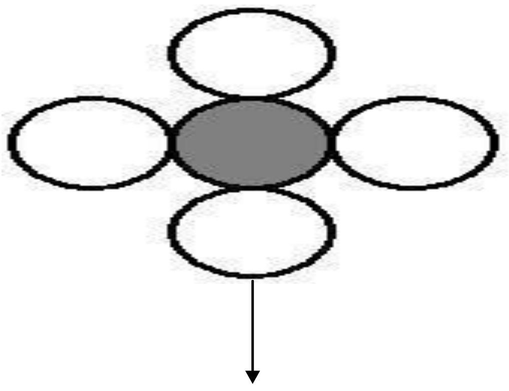
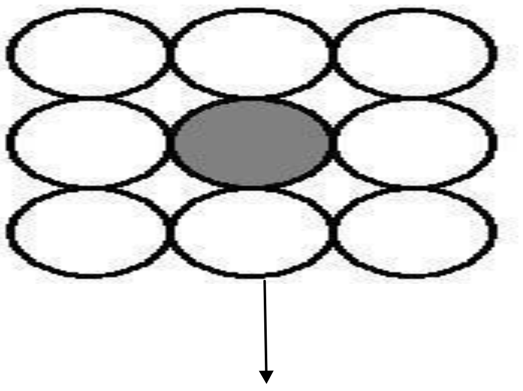
Disadvantages:

- In the 4-connected approach, it does not color corners.

Flood Fill Algorithm:

Sometimes we want to fill in (recolor) an area that is not defined within a single color boundary. We paint such areas by replacing a specified interior color instead of searching for a boundary color value. This approach is called a flood-fill algorithm.

1. We start from a specified interior pixel (x, y) and reassign all pixel values that are currently set to a given interior color with the desired fill color.
2. If the area has more than one interior color, we can first reassign pixel values so that all interior pixels have the same color.
3. Using either 4-connected or 8-connected approach, we then step through pixel positions until all interior pixels have been repainted.

4-connected approach 8-connected approach

4-connected Flood Fill approach:

1. We can implement flood fill algorithm by using recursion.
2. First all the pixels should be reassigned to common color. here common color is black.
3. Start with a point inside given object, check the following condition: if(getpixel(x,y)==old_col)---old_col is common color
4. If above condition is satisfied, then following 4 steps are followed in filling the object.

```
//Recursive 4-way floodfill.

Step 1> Flood(Int x,Int y,New color,Old color)

Step 2> If (getpixel(Int x,Int y)==Old color)

Step 3> putpixel(x,y,New_color);

Step 4> flood(x+1,y,New_col,old_col);

            flood(x-1,y,New_col,old_col);

            flood(x,y+1,New_col,old_col);

           flood(x,y-1,New_col,old_col);
```

8-connected Flood fill approach:

1. We can implement flood fill algorithm by using recursion.
2. First all the pixels should be reassigned to common color. here common color is black.
3. Start with a point inside given object, check the following condition: if(getpixel(x,y)==old_col)---old_col is common color
4. If above condition is satisfied, then following 8 steps are followed in filling the object.

```
//Recursive 4-way floodfill.

Step 1> Flood(Int x,Int y,New color,Old color)

Step 2> If (getpixel(Int x,Int y)==Old color)

Step 3> putpixel(x,y,New_color);

Step 4> flood(x+1,y,new_col,old_col);

           flood(x-1,y,new_col,old_col);

           flood(x,y+1,new_col,old_col);

           flood(x,y-1,new_col,old_col);

           flood(x + 1, y - 1, new_col, old_col);

           flood(x + 1, y + 1, new_col, old_col);

           flood(x - 1, y - 1, new_col, old_col);

           flood(x - 1, y + 1, new_col, old_col);
```

Advantages:

- This method is easy to fill colors in computer graphics.
- It fills the same color inside the boundary.

Disadvantages:

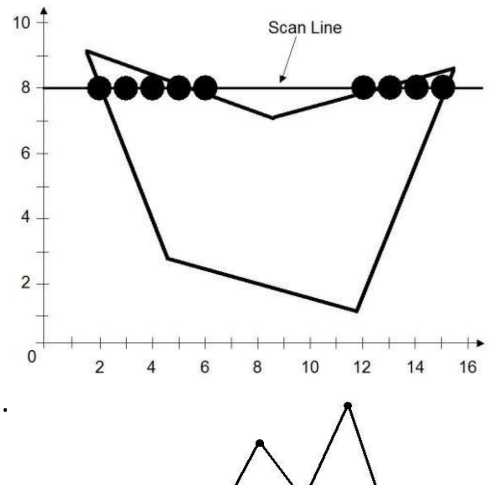
- Fails with large area polygons.
- It is a slow method to fill the area.

Scan Line Polygon Fill Algorithm:

- The basic scan-line algorithm is as follows:
  - Find the intersections of the scan line with all edges of the polygon
  - Sort the intersections by increasing x coordinate
  - Fill in all pixels between pairs of intersections that lie interior to the polygon

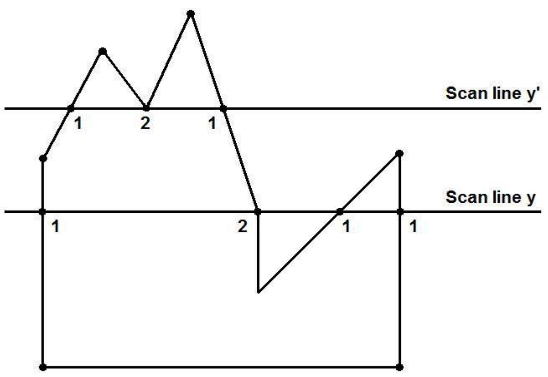
The scan-line polygon-filling algorithm involves

- the **horizontal scanning** of the polygon from its **lowermost** to its **topmost** vertex,
- identifying which edges intersect the scan-line,
- and finally drawing the interior horizontal lines with the specified fill color.

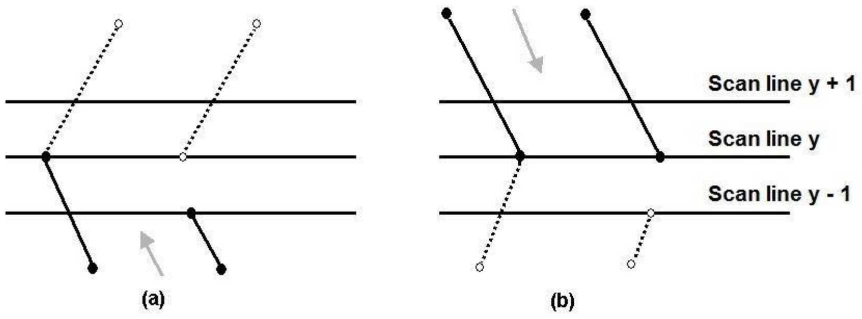


process.

Dealing with vertices:



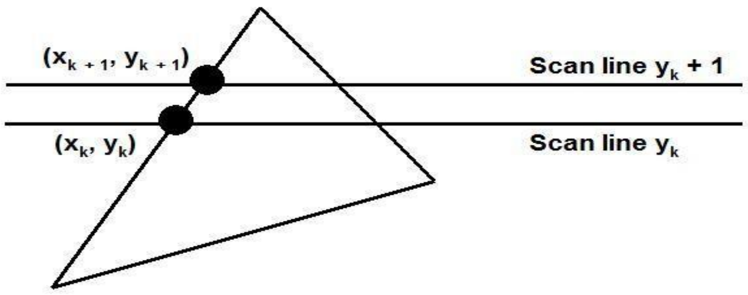
- When the endpoint **y** coordinates of the two edges are **increasing**, the **y** value of the upper endpoint for the **current edge** is decreased by one (a)
- When the endpoint **y** values are **decreasing**, the **y** value of the **next edge** is decreased by one (b)



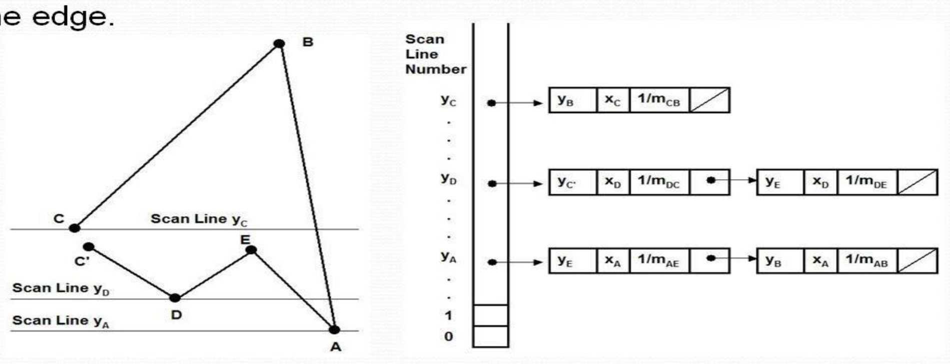
- **Determining Edge Intersections**

$m = (y_{k+1} – y_k) / (x_{k+1} – x_k)$ $y_{k+1} – y_k = 1$

$x_{k+1} = x_k + 1/m$



- Each **entry** in the table for a particular scan line contains the **maximum y** value for that edge, the **x-intercept** value (**at the lower vertex**) for the edge, and the **inverse slope** of the edge.


B
C
Scan Line y_C
E
C'
Scan Line y_D
D
Scan Line y_A
A
Scan Line Number
y_C
y_D
y_A
1
0
y_B | x_C | $1/m_{CB}$
$y_{C'}$ | x_D | $1/m_{DC}$
y_E | x_D | $1/m_{DE}$
y_E | x_A | $1/m_{AE}$
y_B | x_A | $1/m_{AB}$

Steps in algorithm:

1. Find the minimum enclosed rectangle
2. Here the number of scanlines are equal to (ymax-ymin+1)
3. For each scanline, do
    a. Obtain the intersection points of scanline with polygon edges
    b. Sort the intersection edges from left to right
    c. Form the pairs of intersection points from the obtained list
    d. Fill with colors with in each pair if intersection points
    e. Repeat above procedure until ymax

Algorithm Steps:

1. the horizontal scanning of the polygon from its lowermost to its topmost vertex
2. identify the edge intersections of scan line with polygon
3. Build the edge table
    a. Each entry in the table for a particular scan line contains the maximum y value for that edge, the x-intercept value (at the lower vertex) for the edge, and the inverse slope of the edge.
4. Determine whether any edges need to be splitted or not. If there is need to split, split the edges.
5. Add new edges and build modified edge table.

6. Build Active edge table for each scan line and fill the polygon based on intersection of scanline with polygon edges.

Advantages:

- Scan fill algorithm fills the polygon in the same order as rendering.
- It takes advantage of coherence thus, a fast algorithm.

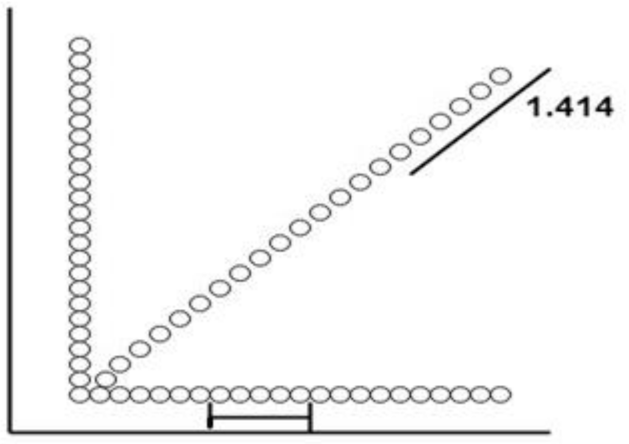
Disadvantages:

- The intensity of the pixels is unequal.
- Staircase-like appearance when scan line converted to circles.

Side effects of Scan Conversion:

1. Staircase or Jagged: Staircase like appearance is seen while the scan was converting line or circle.

2. Unequal Intensity: It deals with unequal appearance of the brightness of different lines. An inclined line appears less bright as compared to the horizontal and vertical line.



Pixels along with horizontal line are 1 unit apart and vertical.
Pixels along diagonal line are 1.414 units.