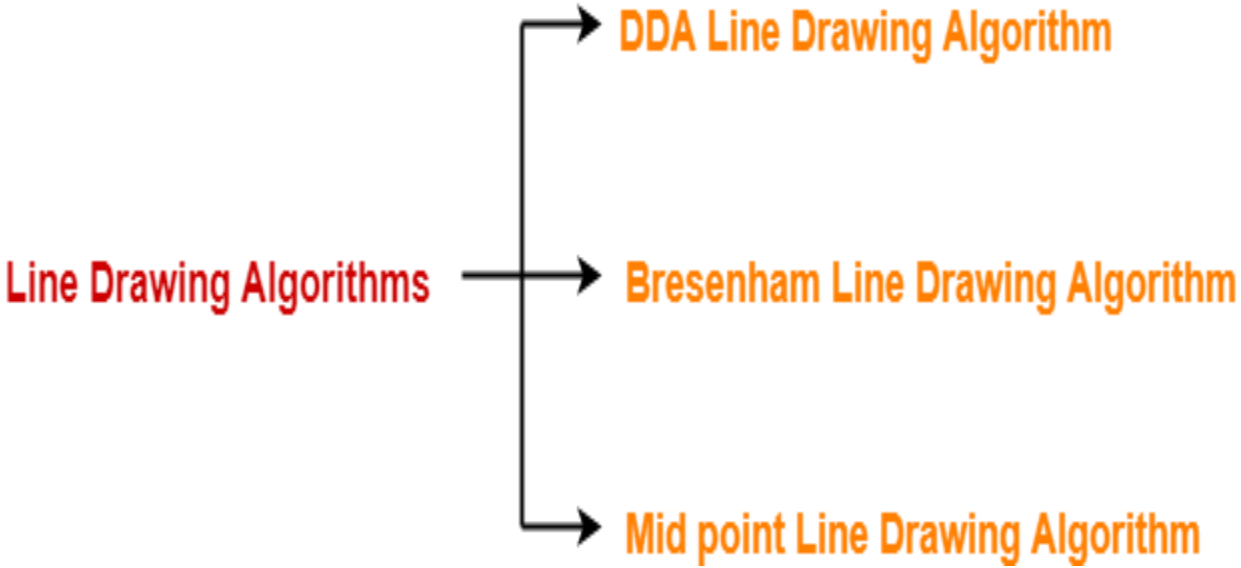
# Chapter-2
# Raster Scan Graphics
## (18 Marks)

# Line Drawing Algorithms-



# DDA Algorithm-

DDA Algorithm is the simplest line drawing algorithm.

**Procedure-**

Given-
Starting coordinates = $(X_0, Y_0)$
Ending coordinates = $(X_n, Y_n)$

The points generation using DDA Algorithm involves the following steps-

**Step-01:**
Calculate ΔX, ΔY and M from the given input.
These parameters are calculated as-
$\Delta X = X_n - X_0$
$\Delta Y = Y_n - Y_0$
$M = \Delta Y / \Delta X$

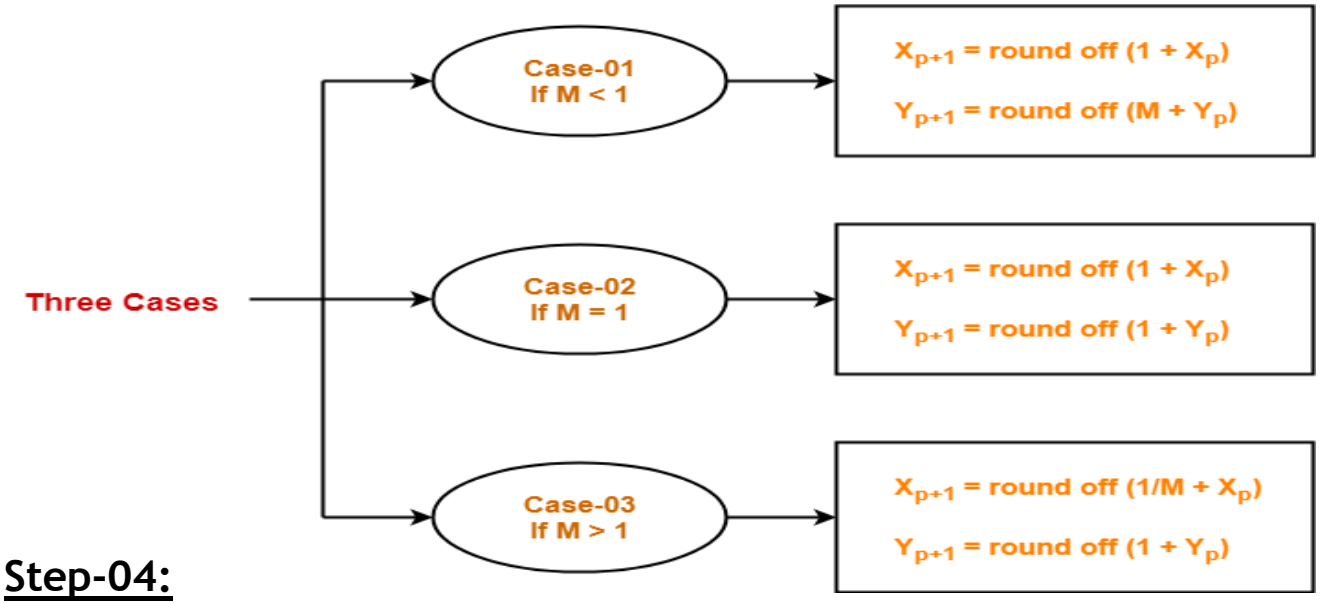
**Step-02:**

Find the number of steps or points in between the starting and ending coordinates.

if (absolute (ΔX) > absolute (ΔY))
Steps = absolute (ΔX);
else
Steps = absolute (ΔY);

**Step-03:**

Suppose the current point is $(X_p, Y_p)$ and the next point is $(X_{p+1}, Y_{p+1})$.




**Step-04:**

Keep repeating Step-03 until the end point is reached or the number of generated new points (including the starting and ending points) equals to the steps count.

## PRACTICE PROBLEMS BASED ON DDA ALGORITHM-

### Problem-01:

Calculate the points between the starting point (5, 6) and ending point (8, 12).

### Solution-

Given-
Starting coordinates = $(X_0, Y_0) = (5, 6)$
Ending coordinates = $(X_n, Y_n) = (8, 12)$

### Step-01:

Calculate ΔX, ΔY and M from the given input.
$\Delta X = X_n - X_0 = 8 - 5 = 3$
$\Delta Y = Y_n - Y_0 = 12 - 6 = 6$
$M = \Delta Y / \Delta X = 6 / 3 = 2$

### Step-02:

Calculate the number of steps.
As $|\Delta X| < |\Delta Y| = 3 < 6$, so number of steps = ΔY = 6

### Step-03:

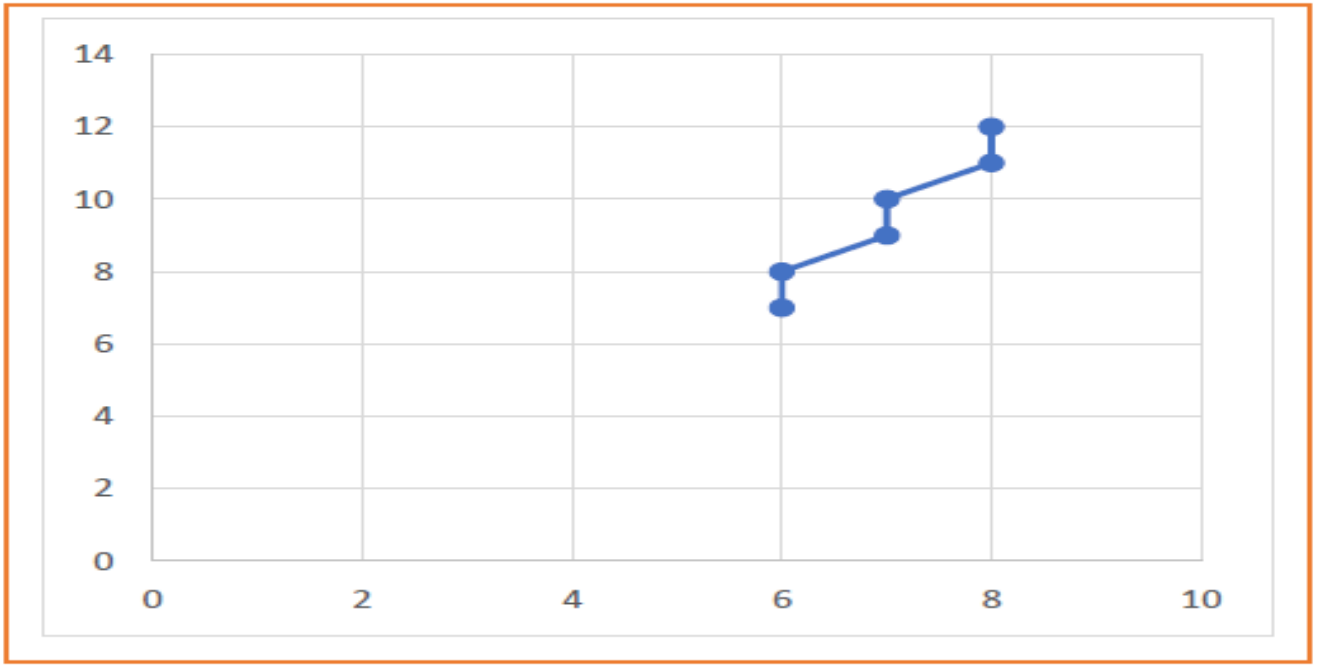
As M > 1, so case-03 is satisfied.
Now, Step-03 is executed until Step-04 is satisfied.

$$X_{p+1} = \text{round off } (1/M + X_p)$$

$$Y_{p+1} = \text{round off } (1 + Y_p)$$

| $X_p$ | $Y_p$ | $X_{p+1}$ | $Y_{p+1}$ | Round off $(X_{p+1}, Y_{p+1})$ |
|---|---|---|---|---|
| 5 | 6 | 5.5 | 7 | (6, 7) |
| | | 6 | 8 | (6, 8) |
| | | 6.5 | 9 | (7, 9) |
| | | 7 | 10 | (7, 10) |
| | | 7.5 | 11 | (8, 11) |
| | | 8 | 12 | (8, 12) |

# DDA LINE OUTPUT



# DDA Line Drawing Algorithm (Digital Differential Analyzer)

Vidyalankar Polytechnic

1. Read line end points(x1,y1) and (x2,y2) such that they are not equal.

2. dy=|y2-y1| and dx=|x2-x1|

3. if(abs(dx)>abs(dy))

    length=abs(dx);

  else

    length=abs(dy);

4.Calculate xincrement and yincrement.

xinc=dx/(float)length;
yinc=dy/(float)length;

5. x=x1

    y=y1

    putpixel(x,y,10);

6.Continue process till I reaches to length as

for(i=0;i<length;i++)

{

putpixel(x,y,10);

x=x+xinc;

y=y+yinc;

delay(10);

}

7.Stop.

# DDA Line Drawing Program

```
#include<stdio.h>
 #include<conio.h>
 #include<graphics.h>
 void main()
{
int x1,y1,x2,y2,dx,dy,length,i;
float x,y,xinc,yinc;
int gd=DETECT,gm;
initgraph(&gd,&gm,"c:\\turboc3\\bgi");
printf("Enter the starting coordinates");
scanf("%d%d",&x1,&y1);
printf("Enter the ending coordinates");
scanf("%d%d",&x2,&y2);
dx=x2-x1;
dy=y2-y1;
if(abs(dx)>abs(dy))
                        length=abs(dx);
                else
                        length=abs(dy);
                xinc=dx/(float)length;
                yinc=dy/(float)length;
                 x=x1;
                 y=y1;
                putpixel(x,y,10);
                 for(i=0;i<length;i++)
                 {
                 x=x+xinc;
                 y=y+yinc;
                 putpixel(x,y,10);
                 delay(10);
                 }
                 getch();
                 closegraph();
                 }
```

# Advantages and Disadvantages

Vidyalankar Polytechnic

Advantages

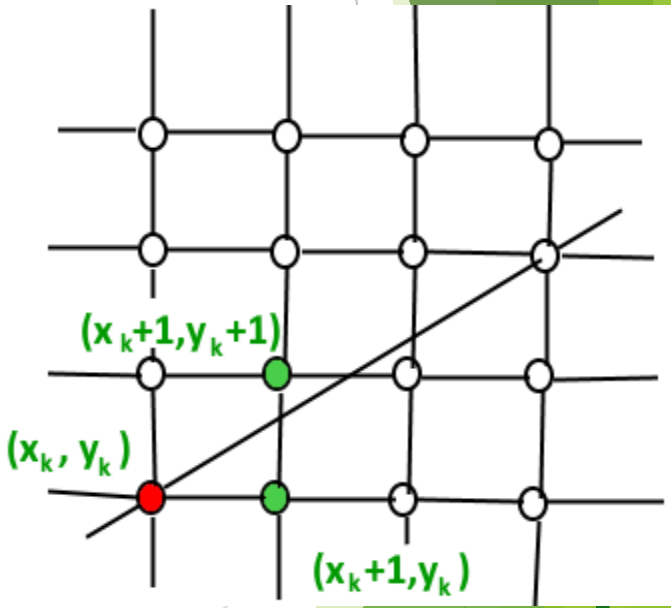
1. It is the simplest algorithm and it does not require special skills for implementation.
2. It is a faster method for calculating pixel positions .

Disadvantages

1. Floating point arithmetic in DDA algorithm is still time-consuming.
2. The algorithm is orientation dependent. Hence end point accuracy is poor.

# Bresenham's Line Drawing Algorithm

- The idea of Bresenham's algorithm is to avoid floating point multiplication and addition to compute mx + c, and then computing round value of (mx + c) in every step. In Bresenham's algorithm, we move across the x-axis in unit intervals.
- We always increase x by 1, and we choose about next y, whether we need to go to y+1 or remain on y. In other words, from any position ($X_k$, $Y_k$) we need to choose between ($X_k$ + 1, $Y_k$) and ($X_k$ + 1, $Y_k$ + 1).
- We would like to pick the y value (among $Y_k$ + 1 and $Y_k$) corresponding to a point that is closer to the original line.

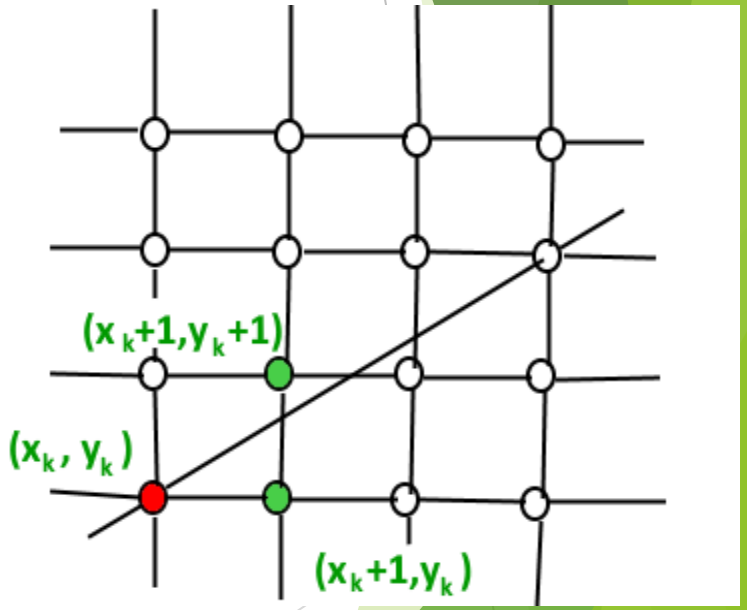


# Bresenham's Line Drawing Algorithm

- We would like to pick the y value (among $Y_k+1$ and $Y_k$) corresponding to a point that is closer to the original line.
- The initial value of slope error or decision parameter is

  2*(y2– y1) – (x2–x1).

  e= 2 ΔY- ΔX



# Bresenham's Line Drawing Algorithm

**Procedure-**

Given-
Starting coordinates = $(X_0, Y_0)$
Ending coordinates = $(X_n, Y_n)$

The points generation using Bresenham's Algorithm involves the following steps-

**Step-01:**

Calculate ΔX, ΔY and M from the given input.
These parameters are calculated as-

$\Delta X = X_n - X_0$
$\Delta Y = Y_n - Y_0$
$e = 2\Delta Y - \Delta X$ (e refers to error or also called as decision parameter $P_k$)

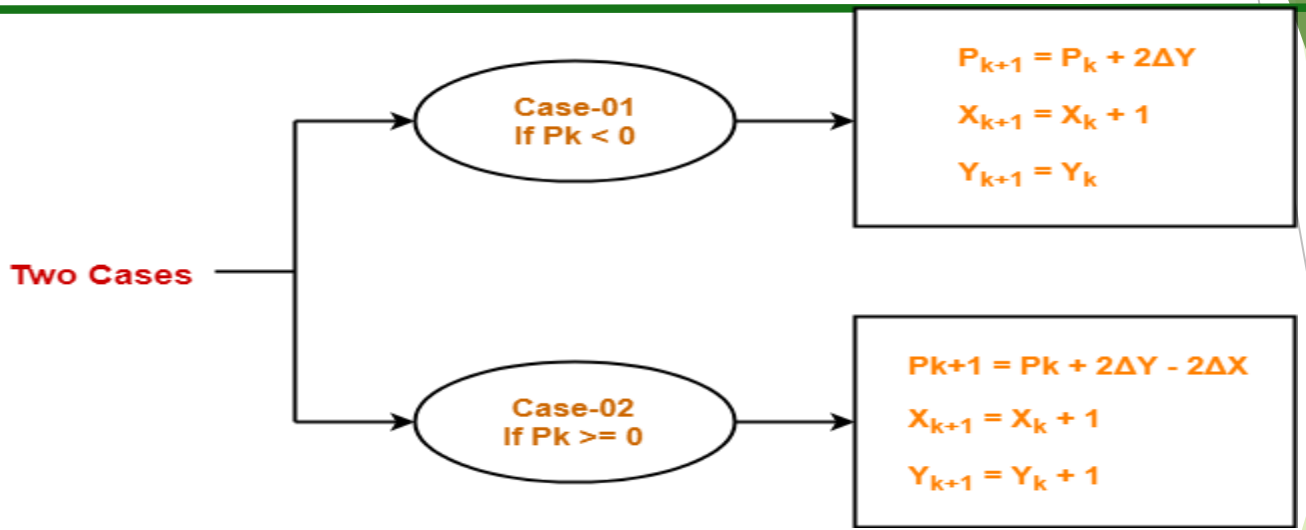
**Step-02:**

Check error or decision parameter.

```
while(x<=x2)
        if(e<0)
          {   x=x+1;
              y=y;
             e=e+2*dy;
           }
              else
              {
               x=x+1;
              y=y+1;
              e=e+2*(dy-dx);
           }
```

**Step-03:**

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




**Step-04:**

Keep repeating Step-03 until the end point is reached.

**Problem-01:**

**Calculate the points between the starting point (9, 18) and ending point (14, 22).**

**Solution:**

Calculate ΔX, ΔY and M from the given input.

$\Delta X = X_n - X_0 = 14 - 9 = 5$

$\Delta Y = Y_n - Y_0 = 22 - 18 = 4$

Calculate the decision parameter.

$P_k = 2\Delta Y - \Delta X$

$P_k = 2 \times 4 - 5$

$P_k = 3$

So, decision parameter Pk = 3

As $P_k >= 0$, so case-02 is satisfied.

Thus,

1st iteration:

$P_{k+1} = P_k + 2\Delta Y - 2\Delta X = 3 + (2 \times 4) - (2 \times 5) = 1$

$X_{k+1} = X_k + 1 = 9 + 1 = 10$

$Y_{k+1} = Y_k + 1 = 18 + 1 = 19$

**Plot(10,19)**

As $P_k >= 0$, so case-02 is satisfied.

2nd iteration:

$P_{k+1} = P_k + 2\Delta Y - 2\Delta X = 1 + (2 \times 4) - (2 \times 5) = -1$

$X_{k+1} = X_k + 1 = 10 + 1 = 11$

$Y_{k+1} = Y_k + 1 = 19 + 1 = 20$

**Plot(11,20)**

As $P_k < 0$, so case-01 is satisfied.

3rd iteration:

$P_{k+1} = P_k + 2\Delta Y = -1 + (2 \times 4) = 7$

$X_{k+1} = X_k + 1 = 11 + 1 = 12$

$Y_{k+1} = Y_k = 20$

**Plot(12,20)**

As $P_k >= 0$, so case-02 is satisfied.

4th iteration:

$P_{k+1} = P_k + 2\Delta Y - 2\Delta X = 7 + (2 \times 4) - (2 \times 5) = 5$

$X_{k+1} = X_k + 1 = 12 + 1 = 13$

$Y_{k+1} = Y_k + 1 = 20 + 1 = 21$

**Plot(13,21)**

As $P_k >= 0$, so case-02 is satisfied.

5th iteration:

$P_{k+1} = P_k + 2\Delta Y - 2\Delta X = 5 + (2 \times 4) - (2 \times 5) = 3$

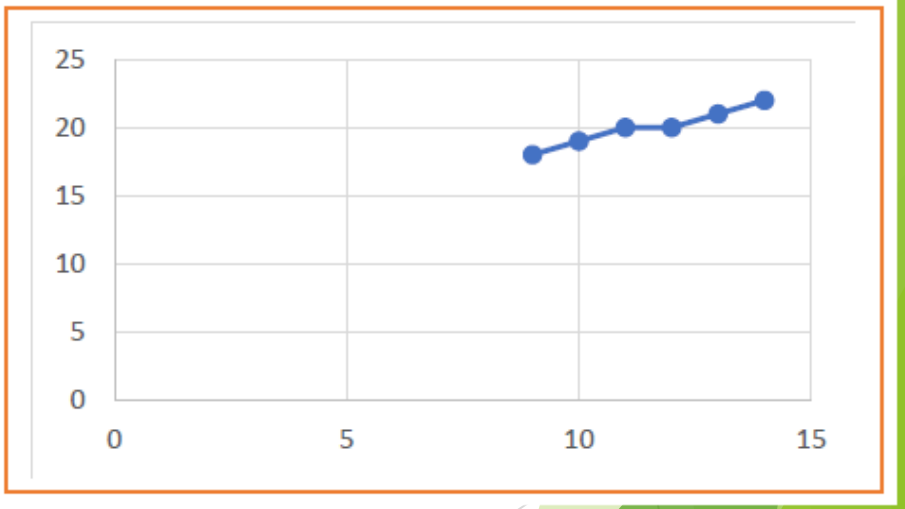
$X_{k+1} = X_k + 1 = 13 + 1 = 14$

$Y_{k+1} = Y_k + 1 = 21 + 1 = 22$

**Plot(14,22)**

# Bresenham's Line Output

| $P_k$ | $P_{k+1}$ | $X_{k+1}$ | $Y_{k+1}$ |
|---|---|---|---|
| | | 9 | 18 |
| 3 | 1 | 10 | 19 |
| 1 | -1 | 11 | 20 |
| -1 | 7 | 12 | 20 |
| 7 | 5 | 13 | 21 |
| 5 | 3 | 14 | 22 |



# Advantages and Disadvantages

## Advantages

- It is easy to implement.
- It is fast and incremental.
- It executes fast but less faster than DDA Algorithm.
- The points generated by this algorithm are more accurate than DDA Algorithm.
- It uses fixed points only.

## Disadvantages

- Though it improves the accuracy of generated points but still the resulted line is not smooth.
- This algorithm is for the basic line drawing.
- It can not handle diminishing jaggies.

# Bresenhams Line Drawing Algorithm

Vidyalankar Polytechnic

1. Read line end points(x1,y1) and (x2,y2) such that they are not equal.

2. dy=|y2-y1| and dx=|x2-x1|

3. e=2*dy-dx;

   x=x1;

   y=y1;

4.while(x<=x2) perform following

   {

5. if(e<0)

   {  x=x+1;

      y=y;

      e=e+2*dy;

   }

6. else

   {

      x=x+1;

      y=y+1;

      e=e+2*(dy-dx);

   }

   putpixel(x,y,15);

}

7.Stop.

# Bresenhams Line Drawing Algorithm



```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
void main()
 {
  int length,dx,dy,x1,y1,x2,y2,e;
  float x,y;
  int gd=DETECT,gm;
  initgraph(&gd,&gm,"c://Turboc3//BGI");
  printf("\n enter coordinates of x1,y1,x2,y2");
  scanf("%d%d%d%d",&x1,&y1,&x2,&y2);
   dx=x2-x1;
   dy=y2-y1;
   e=2*dy-dx;
   x=x1;
   y=y1;
  putpixel(x,y,15);
```

```
  while(x<=x2)
    {
      if(e<0)
      {
            x=x+1;
            y=y;
            e=e+2*dy;
       }
      else
      {
            x=x+1;
            y=y+1;
            e=e+2*(dy-dx);
       }
      putpixel(x,y,15);
  }
  getch();
  closegraph();
  }
```

## Questions:

**Problem-02:**

Rasterize line using Bresenham's line drawing algorithm starting point (5, 5) and ending point (13, 9).

**Problem-03:**

Rasterize line using Bresenham's line drawing algorithm starting point (20, 10) and ending point (30, 18).

| | DDA Line Drawing Algorithm | Bresenhams Line Drawing Algorithm |
|---|---|---|
| Arithmetic | uses floating points | Integer points |
| Operations | Uses multiplication and division in its operations. | Uses only subtraction and addition in its operations. |
| Speed | slower than Bresenham's algorithm | faster than DDA |
| Accuracy & Efficiency | not as accurate and efficient | more efficient &much accurate |
| Drawing | DDA algorithm can draw circles and curves but that are not as accurate as Bresenham's algorithm. | Draw circles and curves with much more accuracy than DDA algorithm. |
| Round Off | DDA algorithm round off the coordinates to integer that is nearest to the line. | Bresenhams' algorithm does not round off but takes the incremental value in its operation. |
| Expensive | Expensive | Less Expensive |

## Problem-01:

Calculate the points between the starting point (6, 5) and ending point (15, 10).

## Solution-

Given-
Starting coordinates = $(X_0, Y_0)$ = (6, 5)
Ending coordinates = $(X_n, Y_n)$ = (15, 10)

## Step-01:

Calculate ΔX, ΔY and M from the given input.
$\Delta X = X_n - X_0 = 15 - 6 = 9$
$\Delta Y = Y_n - Y_0 = 10 - 5 = 5$
Calculate the decision parameter.
$Pk = 2\Delta Y - \Delta X$
$Pk = 2 \times 5 - 9$
$Pk = 1$
So, decision parameter Pk = 1

## Step-02:

As $P_k >= 0$, so case-02 is satisfied.

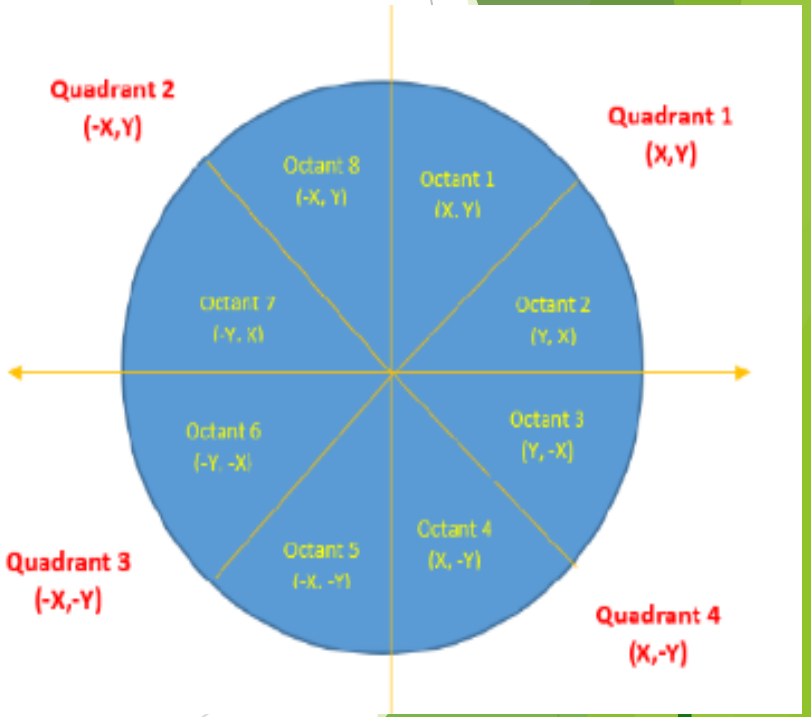
Thus,
$P_{k+1} = P_k + 2\Delta Y - 2\Delta X = 1 + (2 \times 5) - (2 \times 9) = -7$
$X_{k+1} = X_k + 1 = 6 + 1 = 7$
$Y_{k+1} = Y_k + 1 = 5 + 1 = 6$

Similarly, Step-02 is executed until the end point is reached or while(x<=x2)

| $P_k$ | $P_{k+1}$ | $X_{k+1}$ | $Y_{k+1}$ |
|---|---|---|---|
| | | 6 | 5 |
| 1 | -7 | 7 | 6 |
| -7 | 3 | 8 | 6 |
| 3 | -5 | 9 | 7 |
| -5 | 5 | 10 | 7 |
| 5 | -3 | 11 | 8 |
| -3 | 7 | 12 | 8 |
| 7 | -1 | 13 | 9 |
| -1 | 9 | 14 | 9 |
| 9 | 1 | 15 | 10 |

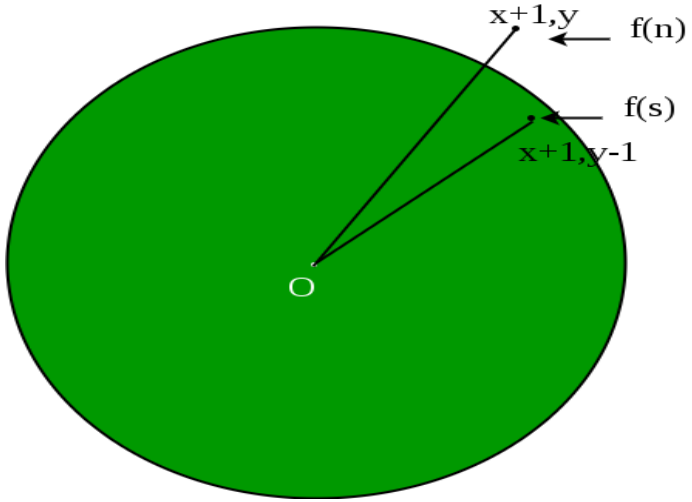
# Bresenham's Circle Drawing Algorithm

- This algorithms uses the key feature of circle that it is highly symmetric. So, for whole 360 degree of circle we will divide it in 8-parts each octant of 45 degree. In order to that we will use Bresenham's Circle Algorithm for calculation of the locations of the pixels in the first octant of 45 degrees. It assumes that the circle is centered on the origin. So for every pixel (x, y) it calculates, we draw a pixel in each of the 8 octants of the circle as shown :



# Bresenham's Circle Drawing Algorithm

- Now, we will see how to calculate the next pixel location from a previously known pixel location (x, y). In Bresenham's algorithm at any point (x, y) we have two option either to choose the next pixel in the east i.e. (x+1, y) or in the south east i.e. (x+1, y-1).



# Bresenham's Circle Drawing Algorithm



**Procedure-**

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

The points generation using Bresenham's Circle Drawing Algorithm involves the following steps-

**Step-01:**

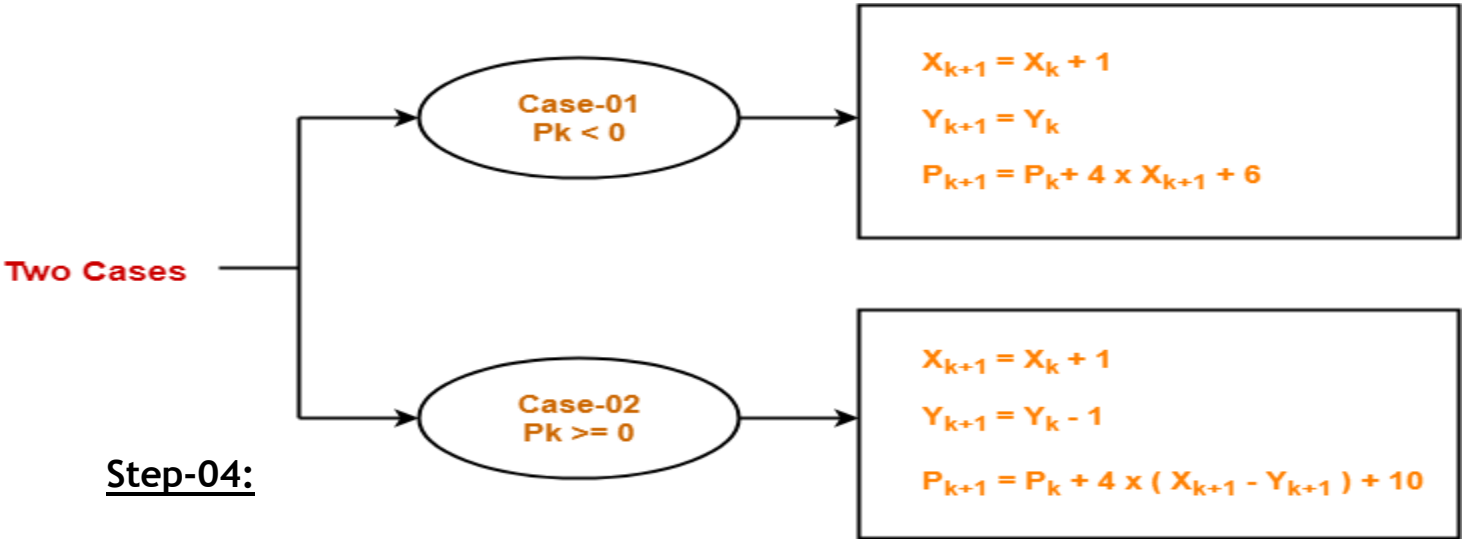
Assign the starting point coordinates $(X_0, Y_0)$ as-
$X_0 = 0$
$Y_0 = R$

## Step-02:

Calculate the value of initial decision parameter $P_0$ as
$P_0 = 3 - 2 \times R$

## 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 $P_k$.




**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:**

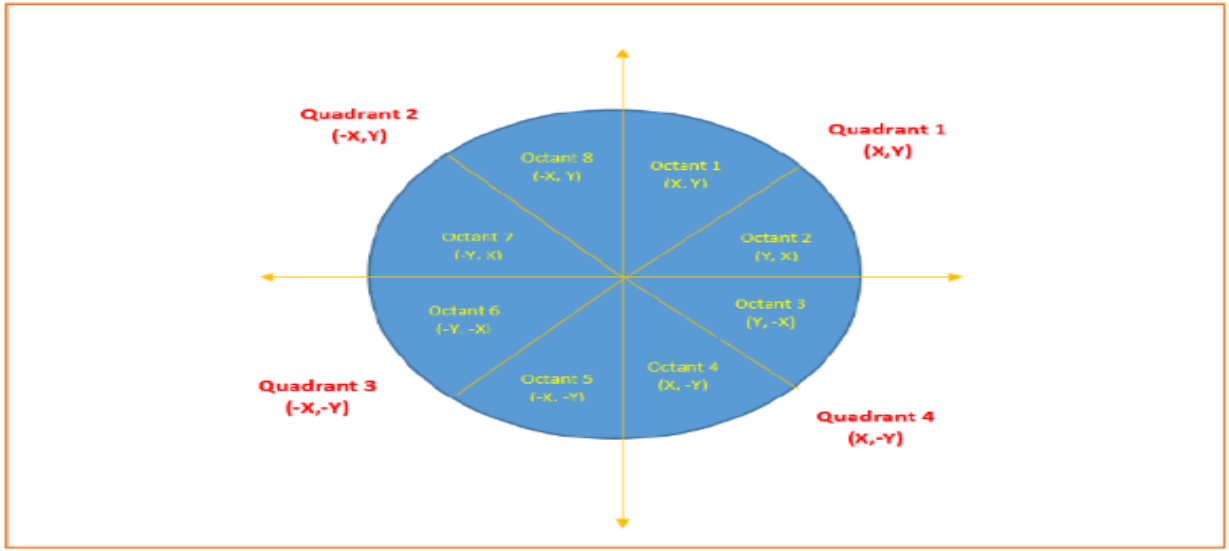
Keep repeating Step-03 and Step-04 until
$X_{plot}$ => $Y_{plot}$.

**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.

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 8, generate all the points to form a circle.

## Solution-

Given-
Centre Coordinates of Circle $(X_0, Y_0)$ = (0, 0)
Radius of Circle = 8

## Step-01:

Assign the starting point coordinates $(X_0, Y_0)$ as-
$X_0 = 0$
$Y_0 = R = 8$

## Step-02:

Calculate the value of initial decision parameter $P_0$ as-
$P_0 = 3 - 2 \times R$
$P_0 = 3 - 2 \times 8$
$P_0 = -13$

## Step-03:

Iteration 1:
As $P_{initial} < 0$, so case-01 is satisfied.

Thus,
$X_{k+1} = X_k + 1 = 0 + 1 = 1$
$Y_{k+1} = Y_k = 8$
$P_{k+1} = P_k + 4 \times X_{k+1} + 6 = -13 + (4 \times 1) + 6 = -3$

**Step-03 continued..**

Iteration 2:

As $P_k < 0$, so case-01 is satisfied.

Thus,

$X_{k+1} = X_k + 1 = 1 + 1 = 2$

$Y_{k+1} = Y_k = 8$

$P_{k+1} = P_k + 4 \times X_{k+1} + 6 = -3 + (4 \times 2) + 6 = 11$

Iteration 3:

As $P_K >= 0$, so case-02 is satisfied.

Thus,

$X_{k+1} = X_k + 1 = 2 + 1 = 3$

$Y_{k+1} = Y_k - 1 = 8-1=7$

$P_{k+1} = P_k + 4 (X_{k+1} - Y_{k+1}) + 10 = 11 + (4 \times -4) + 10 = 5$

Iteration 4:

As $P_K >= 0$, so case-02 is satisfied.

Thus,

$X_{k+1} = X_k + 1 = 3 + 1 = 4$

$Y_{k+1} = Y_k - 1 = 7-1=6$

$P_{k+1} = P_k + 4 (X_{k+1} - Y_{k+1}) + 10 = 5 + (4 \times -2) + 10 = 7$

Iteration 5:

As $P_K > = 0$, so case-02 is satisfied.

Thus,

$X_{k+1} = X_k + 1 = 4 + 1 = 5$

$Y_{k+1} = Y_k - 1 = 6-1 = 5$

$P_{k+1} = P_k + 4 (X_{k+1} - Y_{k+1}) + 10 = 7 + (4 \times 0) + 10 = 17$

**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-

| Pk | Pk+1 | (Xk+1, Yk+1) |
|---|---|---|
| | | (0, 8) |
| -13 | -3 | (1, 8) |
| -3 | 11 | (2, 8) |
| 11 | 5 | (3, 7) |
| 5 | 7 | (4, 6) |
| 7 | 17 | (5, 5) |

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, 8) | (5, 5) |
| (1, 8) | (6, 4) |
| (2, 8) | (7, 3) |
| (3, 7) | (8, 2) |
| (4, 6) | (8, 1) |
| (5, 5) | (8, 0) |

These are all points for Quadrant-1(X,Y).

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, 8) | (0, 8) | (0, -8) | (0, -8) |
| (1, 8) | (-1, 8) | (-1, -8) | (1, -8) |
| (2, 8) | (-2, 8) | (-2, -8) | (2, -8) |
| (3, 7) | (-3, 7) | (-3, -7) | (3, -7) |
| (4, 6) | (-4, 6) | (-4, -6) | (4, -6) |
| (5, 5) | (-5, 5) | (-5, -5) | (5, -5) |
| (6, 4) | (-6, 4) | (-6, -4) | (6, -4) |
| (7, 3) | (-7, 3) | (-7, -3) | (7, -3) |
| (8, 2) | (-8, 2) | (-8, -2) | (8, -2) |
| (8, 1) | (-8, 1) | (-8, -1) | (8, -1) |
| (8, 0) | (-8, 0) | (-8, 0) | (8, 0) |

These are all points of the Circle.

# Bresenham's Circle Drawing Algorithm

1. Read the radius of circle.
2. Calculate initial decision parameter d=3-2r
3. x=0 and y=r [initialize the starting point]
4. while(x<y)

```
{
    Plot(x,y)//plot point(x,y) and other 7 points in all octant
    x=x+1;
    if(d<0)
    {
        d=d+(4*x)+6;
    }
    else
    {
        y=y-1;
        d=d+4*(x-y)+10;
    }
    Plot(x,y)
    }// end of while
```

5. Stop.

# Bresenham's Circle Drawing Program

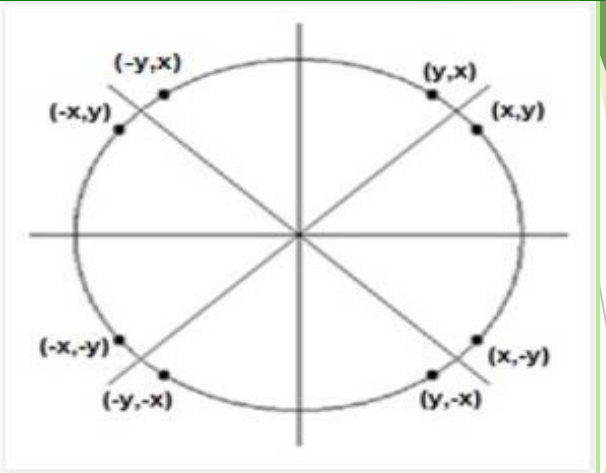
```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<dos.h>
#include<graphics.h>
void bressn(int,int,int);
void plot(int,int,int,int);
void plot(int xc,int yc,int x,int y)
{
      putpixel (xc+x,yc+y,WHITE);
      putpixel(xc-x,yc+y,RED);
      putpixel(xc-x,yc-y,GREEN);
      putpixel(xc+x,yc-y,YELLOW);
      putpixel(xc+y,yc+x,WHITE);
      putpixel(xc-y,yc+x,GREEN);
      putpixel(xc-y,yc-x,YELLOW);
      putpixel(xc+y,yc-x,RED);
}
```

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

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

# Symmetry of circle

- A circle is a geometric figure which is round, and can be divided into 360 degrees. A circle is a symmetrical figure which follows 8-way symmetry.
- 8-Way symmetry: Any circle follows 8-way symmetry. This means that for every point (x,y) 8 points can be plotted. These (x,y), (y,x), (-y,x), (-x,y), (-x,-y), (-y,-x), (y,-x), (x,-y).




Drawing a circle: To draw a circle we need two things, the coordinates of the Centre and the radius of the circle.

Here r is the radius of the circle. If the circle has origin (0,0) as its Centre then the above equation can be reduced to $x^2 + y^2 = r^2$

# Types of Polygon

- Convex Polygon
- Concave Polygon

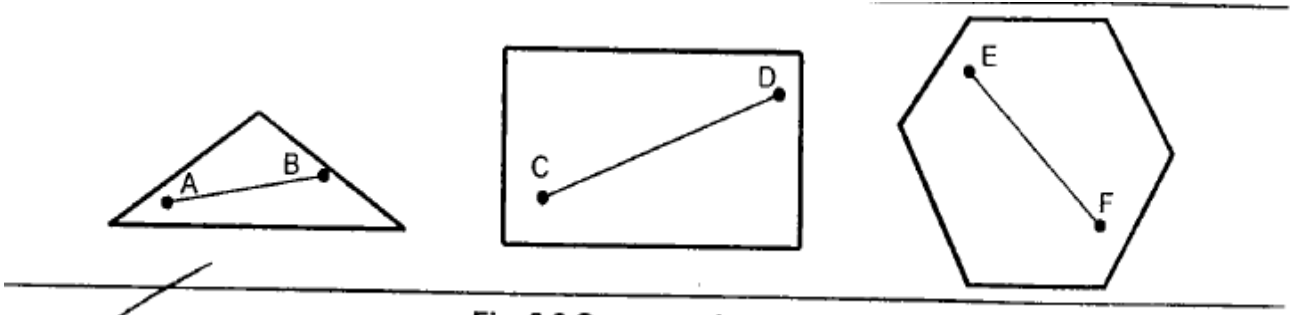



Fig. 3.2 Convex polygons

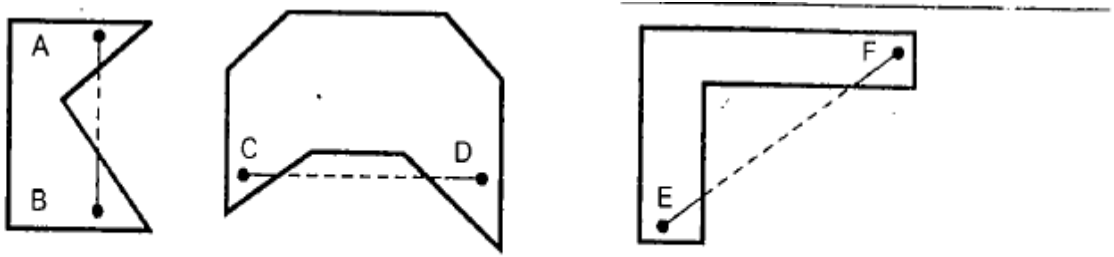



Fig. 3.3 Concave polygons

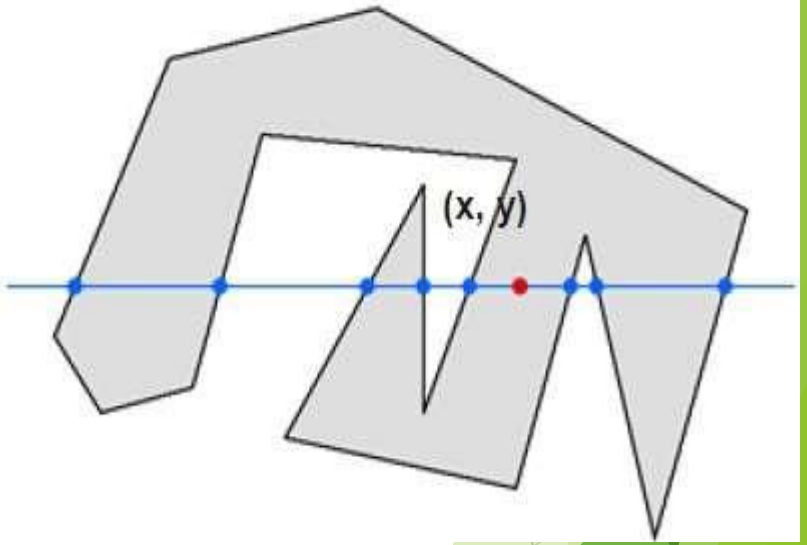
# Polygon Inside-Outside Test

This method is also known as **counting number method**. While filling an object, we often need to identify whether particular point is inside the object or outside it. There are two methods by which we can identify whether particular point is inside an object or outside.

- Odd-Even Rule
- Nonzero winding number rule

# Odd-Even Rule

In this technique, we will count the edge crossing along the line from any point (x,y) to infinity. If the number of interactions is odd, then the point (x,y) is an interior point; and if the number of interactions is even, then the point (x,y) is an exterior point. The following example depicts this concept.

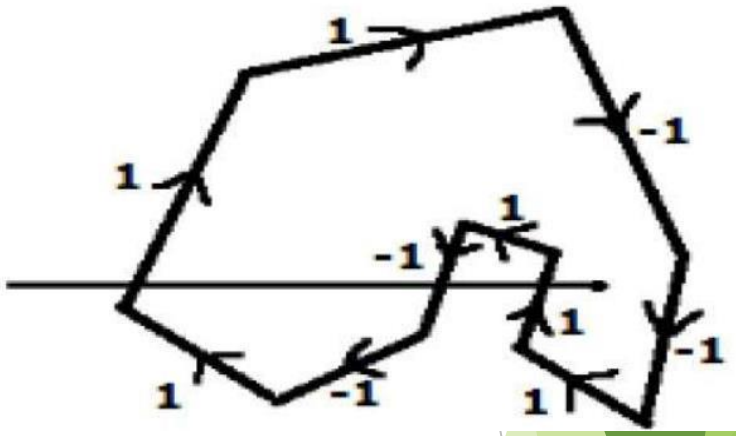



From the above figure, we can see that from the point (x,y), the number of interactions point on the left side is 5 and on the right side is 3. From both ends, the number of interaction points is odd, so the point is considered within the object.

# Nonzero Winding Number Rule

This method is also used with the simple polygons to test the given point is interior or not.

- Give the value 1 to all the edges which are going to upward direction and all other -1 as direction values.
- Check the edge direction values from which the scan line is passing and sum up them.
- If the total sum of this direction value is non-zero, then this point to be tested is an **interior point,** otherwise it is an **exterior point**.




In the above figure, we sum up the direction values from which the scan line is passing then the total is 1 – 1 + 1 = 1; which is non-zero. So the point is said to be an interior point.

# Polygon Filling



- One way to fill polygon is to start from given **“seed”**, point known to be inside the polygon and highlight outward from this point i.e. neighboring pixels until we encounter the boundary pixels. This approach is called **seed fill.**
- Seed fill algorithm further classified as flood fill and boundary fill algorithm.
- Algorithms the fill interior defined regions are flood fill algorithms; those that fill boundary defined regions are called boundary fill algorithm.

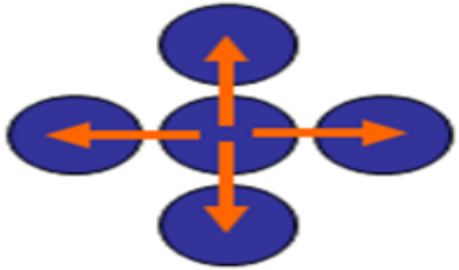
Another approach to fill the polygon is to apply the inside test i.e. to check whether the pixel is inside the polygon or outside the polygon and then highlight pixels which lie inside the polygon. This approach is known as scan-line algorithm.

# Flood Fill

Vidyalankar Polytechnic

Flood fill colors an entire area in an enclosed figure using a single color. The algorithm works in a manner so as to give color to all the pixels inside the boundary the same color leaving the boundary and the pixels outside.

- Flood fill algorithm is used for filling the interior of a polygon.
- Used when an area defined with multiple color boundaries.
- Start at a point inside a region− Replace a specified interior color (old color) with fill color
- Fill the 4−connected or 8−connected region until all interior points being replaced.



**4–Connected Regions**

From a given pixel, the region that you can get to by a series of 4 way moves (north, south, east, west)

```
void fillcolor(int x,int y,int old_color,int new_color)
{
if(getpixel(x,y)==old_color)
{
delay(5);
putpixel(x,y,new_color);
fillcolor(x+1,y,old_color,new_color);
fillcolor(x-1,y,old_color,new_color);
fillcolor(x,y+1,old_color,new_color);
fillcolor(x,y-1,old_color,new_color);
}
}
```

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<dos.h>
void fillcolor(int,int,int,int);
void main()
{
 int gd,gm;
detectgraph(&gd,&gm);
initgraph(&gd,&gm,"c:\\turboc3\\bgi");
rectangle(50,50,100,100);
fillcolor(55,55,0,11);
getch();
closegraph();
 }
```

```
void fillcolor(int x,int y,int old_color,int
new_color)
 {
 if(getpixel(x,y)==old_color)
{
delay(5);
 putpixel(x,y,new_color);
 fillcolor(x+1,y,old_color,new_color);
 fillcolor(x-1,y,old_color,new_color);
fillcolor(x,y+1,old_color,new_color);
fillcolor(x,y-1,old_color,new_color);
 }
}
```

## 8−Connected Regions

From a given pixel, the region that you can get to by a series of 8-way moves (north, south, east, west, NE, NW, SE, SW)

```
void fillcolor(int x,int y,int old_color,int new_color)
{
if(getpixel(x,y)==old_color)
{
delay(5);
putpixel(x,y,new_color);
fillcolor(x+1,y,old_color,new_color);
fillcolor(x-1,y,old_color,new_color);
fillcolor(x,y+1,old_color,new_color);
fillcolor(x,y-1,old_color,new_color);
fillcolor(x+1,y+1,old_color,new_color);
fillcolor(x+1,y-1,old_color,new_color);
fillcolor(x-1,y-1,old_color,new_color);
fillcolor(x-1,y+1,old_color,new_color);
}
}
```

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
 #include<dos.h>
void fillcolor(int,int,int,int);
void main()
{
int gd,gm;
detectgraph(&gd,&gm);
initgraph(&gd,&gm,"c:\\turboc3\\bgi");
rectangle(50,50,100,100);
fillcolor(55,55,0,11);
getch();
closegraph();
}
```

```
void fillcolor(int x,int y,int old_color,int new_color)
 {
 if(getpixel(x,y)==old_color)
{
delay(5);
putpixel(x,y,new_color);
fillcolor(x+1,y,old_color,new_color);
fillcolor(x-1,y,old_color,new_color);
fillcolor(x,y+1,old_color,new_color);
fillcolor(x,y-1,old_color,new_color);
fillcolor(x+1,y+1,old_color,new_color);
fillcolor(x+1,y-1,old_color,new_color);
 fillcolor(x-1,y-1,old_color,new_color);
fillcolor(x-1,y+1,old_color,new_color);
 }
}
```

# Boundary Fill

Start at a point inside the figure and paint with a particular color. Filling continues until a boundary color is encountered.

- In boundary fill algorithm Recursive method is used to fill the whole boundary.
- Start at a point inside a region.
- Paint the interior outward toward the boundary.
- The boundary is specified in a single color.

**4-connected boundary fill Algorithm:-**

```
boundary_fill(x, y, f_colour, b_colour)
{
if(getpixel(x, y)! = b_colour && getpixel(x, y)! = f_colour)
{
putpixel(x, y, f_colour);
boundary_fill(x + 1, y, f_colour, b_colour);
boundary_fill(x, y + 1, f_colour, b_colour);
boundary_fill(x -1, y, f_colour, b_colour);
boundary_fill(x, y – 1, f_colour, b_colour);
}
}
```

```
#include<stdio.h>
 #include<conio.h>
 #include<graphics.h>
 #include<dos.h>
void boundary_fill (int,int,int,int);
void main()
{
int gd,gm;
detectgraph(&gd,&gm);
initgraph(&gd,&gm,"c:\\turboc3\\bgi");
 rectangle(50,50,100,100);
boundary_fill(55,55,6,15);
getch();
closegraph();
}
```

```
void boundary_fill(int x, int y, int
fcolor, int bcolor)
{
if((getpixel(x,y)!=bcolor) &&
(getpixel(x,y)!=fcolor))
{
delay(5);
putpixel(x,y,fcolor);
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);
 }
}
```

**8-connected boundary fill Algorithm:-**

```
boundary_fill(x, y, f_colour, b_colour)
{
if(getpixel(x, y)! = b_colour && getpixel(x, y)! = f_colour)
{
putpixel(x, y, f_colour);
boundary_fill(x + 1, y, f_colour, b_colour);
boundary_fill(x - 1, y, f_colour, b_colour);
boundary_fill(x, y + 1, f_colour, b_colour);
boundary_fill(x, y - 1, f_colour, b_colour);
boundary_fill(x + 1, y + 1, f_colour, b_colour);
boundary_fill (x - 1, y - 1, f_colour, b_colour);
boundary_fill(x + 1, y - 1, f_colour, b_colour);
boundary_fill(x - 1, y + 1, f_colour, b_colour);
}
}
```

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<dos.h>
void boundary_fill(int,int,int,int);
 void main()
 {
 int gd,gm;
detectgraph(&gd,&gm);
initgraph(&gd,&gm,"c:\\turboc3\\bgi");
 rectangle(50,50,100,100);
boundary_fill(55,55,6,15);
getch();
 closegraph();
 }
```

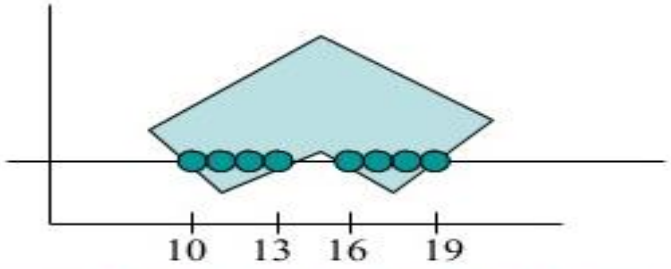
```
void boundary_fill(int x, int y, int fcolor, int
bcolor)
{
if((getpixel(x,y)!=bcolor)&&(getpixel(x,y)!=fco
lor))
{
 delay(5);
putpixel(x,y,fcolor);
boundary_fill(x+1,y,fcolor,bcolor);
boundary_fill(x-1,y,fcolor,bcolor);
boundary_fill(x,y+1,fcolor,bcolor);
boundary_fill(x,y-1,fcolor,bcolor);
boundary_fill(x+1,y+1,fcolor,bcolor);
boundary_fill(x-1,y+1,fcolor,bcolor);
boundary_fill(x+1,y-1,fcolor,bcolor);
boundary_fill(x-1,y-1,fcolor,bcolor);
}
}
```

| Flood Fill Algorithm | Boundary Fill Algorithm |
|---|---|
| 1.In flood fill algorithm the area is defined with multiple colour | 1 In Boundary fill algorithm the area is defined with single colour. |
| 2. The interior is coloured with any colour. | 2. The interior point is replaced with new colour. |
| 3. The old colour is replaced with new colour. | 3. The interior points are replaced with new colour. |
| 4. A flood fill may use an unpredictable amount of memory to finish because it isn't known how many sub-fills will be spawned. | 4. Boundary fill is usually more complicated but it is a linear algorithm and doesn't require recursion. |
| 5. It is Time Consuming. | 5. It is less time Consuming. |

# Scan Line Polygon Fill

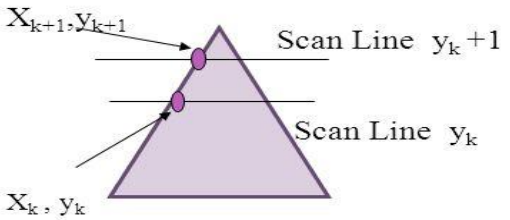
Determine overlap Intervals for scan lines that cross that area.

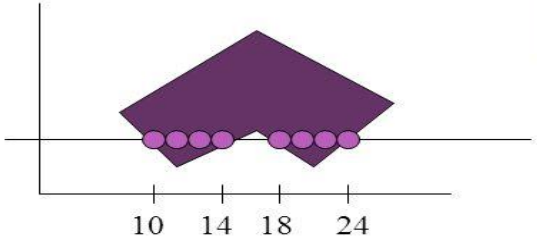
- Requires determining intersection positions of the edges of the polygon with the scan lines
- Fill colors are applied to each section of the scanline that lies within the interior of the region.
- Interior regions determined as in odd-even test




Interior pixels along a scan line passing through a polygon area

# Scan Line Polygon Fill Algorithm




Interior pixels along a scan line passing through a polygon area

- For each scan line crossing a polygon are then sorted from left to right, and the corresponding frame buffer positions between each intersection pair are set to the specified color.
- These intersection points are then sorted from left to right , and the corresponding frame buffer positions between each intersection pair are set to specified color
- In the example, four pixel intersections define stretches from x=10 to x=14 and x=18 to x=24

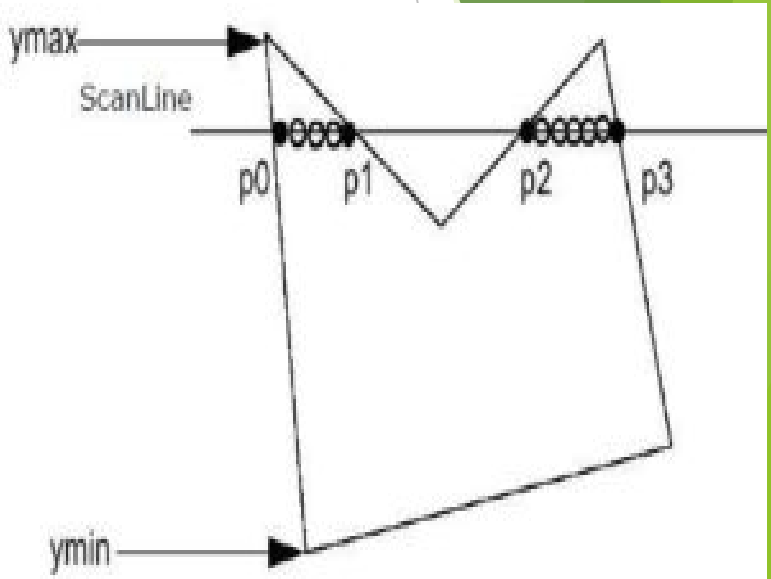
Scanline Fill Algorithm

- Intersect scanline with polygon edges
- Fill between pairs of intersections
- Basic algorithm:

For y = ymin to ymax

1) intersect scanline y with each edge
2) sort inter sections by increasing x [p0,p1,p2,p3]
3) fill pairwise (p0 –> p1, p2–> p3, ....)

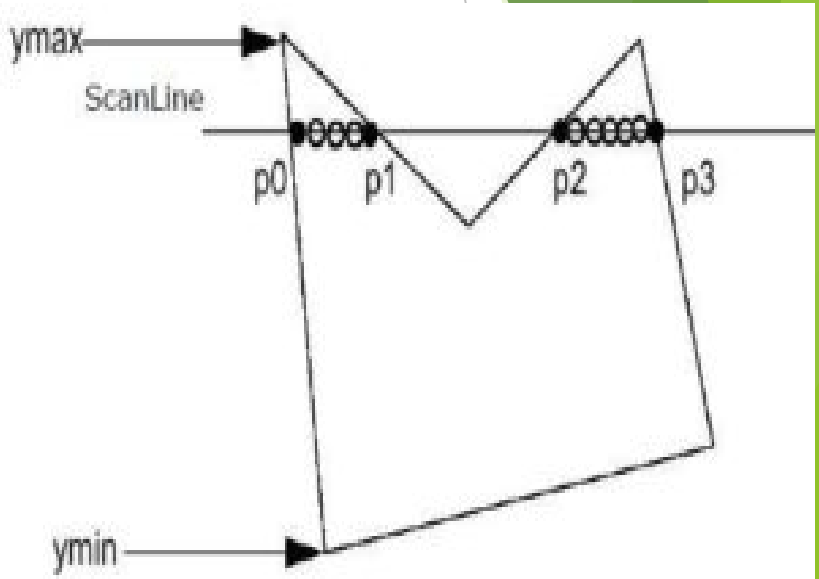
p0 p1 p2 p3



Special handling:

a) Make sure we only fill the interior pixels
Define interior: For a given pair of intersecting points (Xi, Y), (Xj, Y) -> Fill ceiling(Xi) to floor(Xj) important when we have polygons adjacent to each other

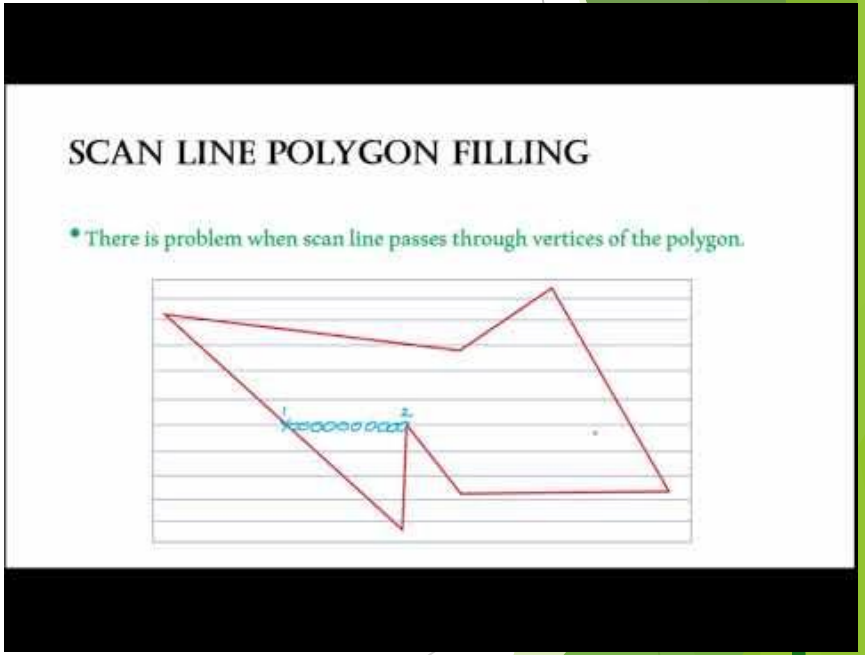
b) Intersection has an integer X coordinate -> if Xi is integer, we define it to be interior -> if Xj is integer, we define it to be exterior (so don't fill)



Special handling (cont'd)

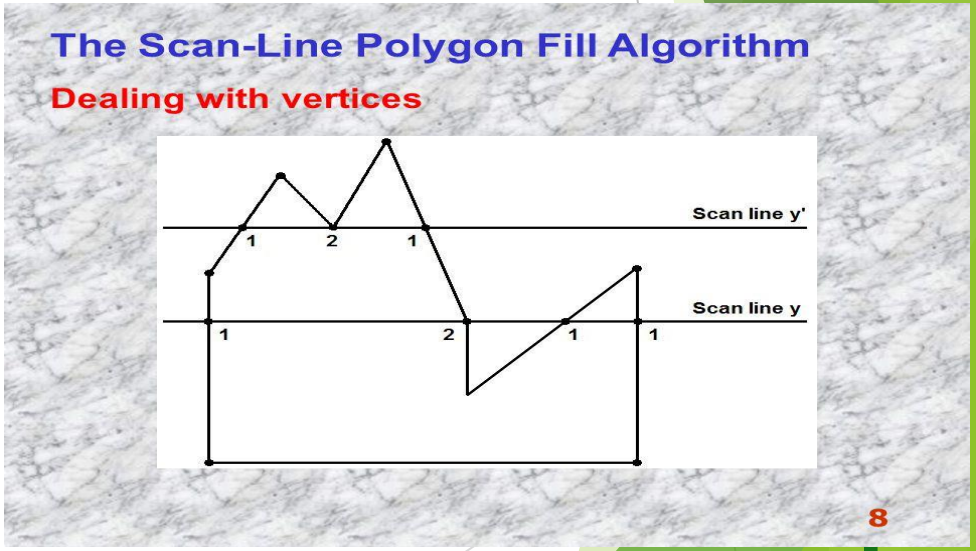
c) Intersection is an edge end point ymin ymax p0 p1 p2 Intersection points: (p0, p1, p2) ??? –> (p0,p1,p1,p2) so we can still fill pairwise –> In fact,

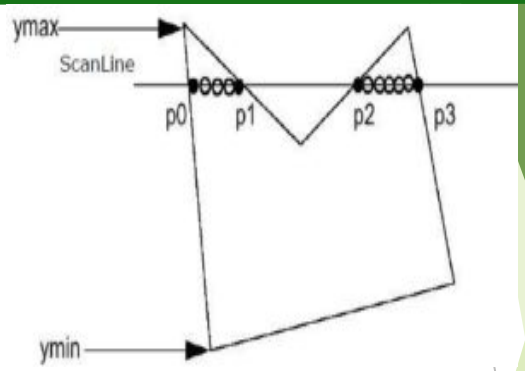
if we compute the intersection of the scanline with edge e1 and e2 separately, we will get the intersection point p1 twice. Keep both of the p1.



Special handling (cont'd)

c) Intersection is an edge end point (cont'd) However, in this case we don't want to count p1 twice (p0,p1,p1,p2,p3), otherwise we will fill pixels between p1 and p2, which is wrong






Special handling (cont'd)

c) Intersection is an edge end point (cont'd)

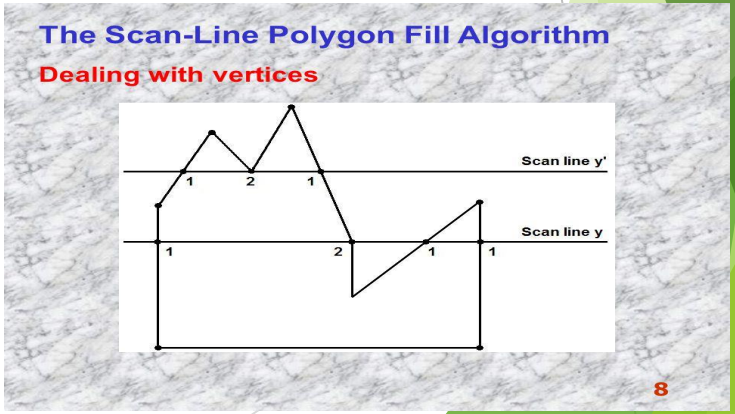
Rule: If the intersection is the ymin of the edge's endpoint, count it. Otherwise, don't.

- p0 p1 p2 scanline and edge e1 e2

Yes, count p1 for both e1 and e2

- p0 p1 p2 p3 scanline and edge e1 e2

  No, don't count p1 for the edge e2



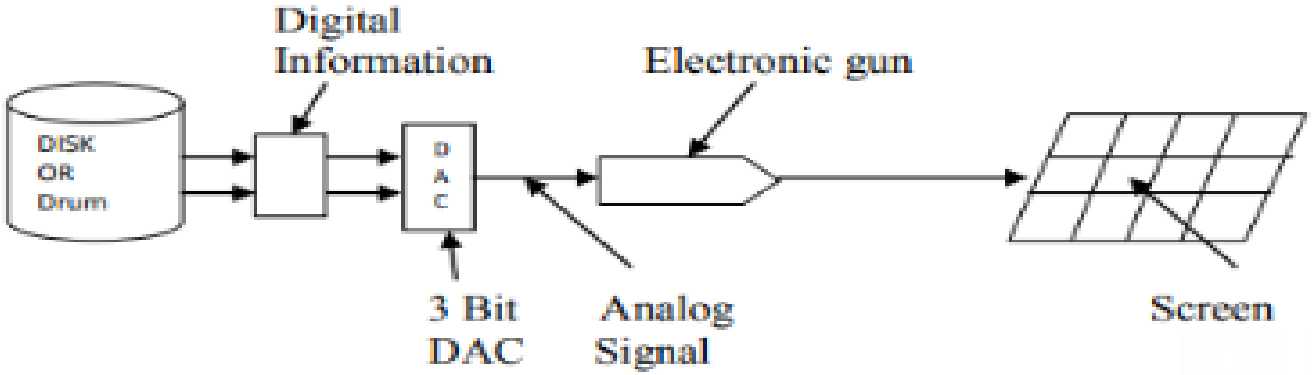
# Scan Conversion

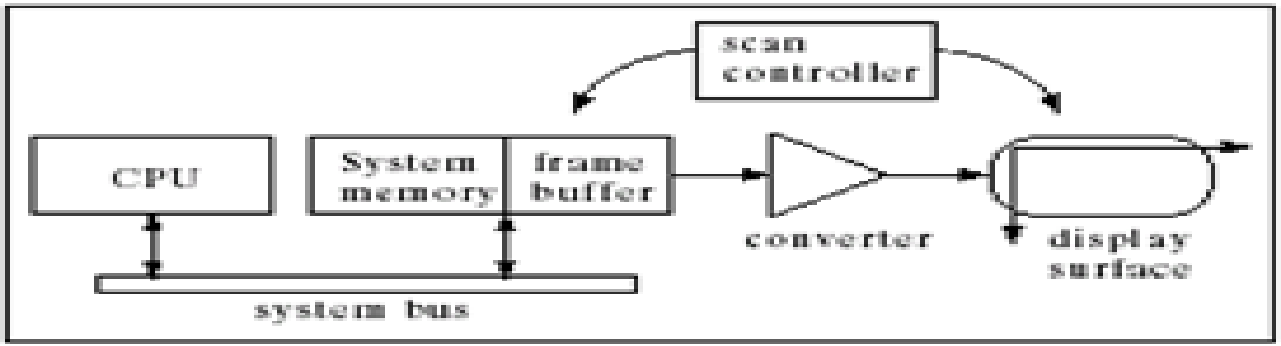
- The process of representing continuous graphics objects as a collection of discrete pixels is called scan conversion. Scan conversion serves as a bridge between TV and computer graphics technology.
- Scan conversion or scan converting rate is a video processing technique for changing the vertical / horizontal scan frequency of video signal for different purposes and applications.
- The device which performs this conversion is called a scan converter.
- The application of scan conversion: video projectors, cinema equipment, TV and video capture cards, standard and HDTV televisions, LCD monitors, radar displays and many different aspects of picture processing.

# Frame Buffer

Vidyalankar Polytechnic

- Each screen pixel corresponds to a particular entry in a 2D array residing in memory. This memory is called a frame buffer or a bit map.
- The number of rows in the frame buffer equals to the number of raster lines on the display screen. The number of columns in this array equals to the number of pixels on each raster line.
- Frame buffer is a large part of computer memory used to store display image. Different kind of memory can be used for frame buffers like drums, disk or IC – shift registers.
- To generate a pixel of desired intensity to read the disk or drum. The information stored in disk or drum is in digital for, hence it is necessary to convert it into analog from using DAC and then this analog signal is used to generate the pixel.




The Scanning Process

# Character Generation

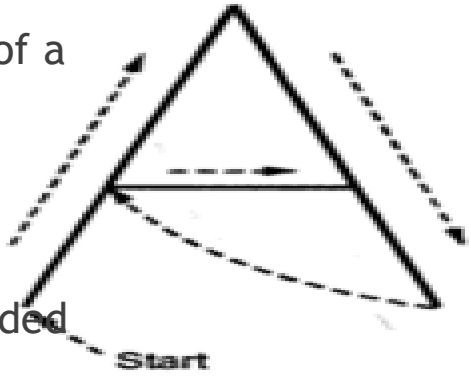
- Most of the times characters are built into the graphics display devices, usually as hardware but sometimes through software.

- There are basic three methods:
  - Stroke method
  - Starburst method
  - Bitmap method

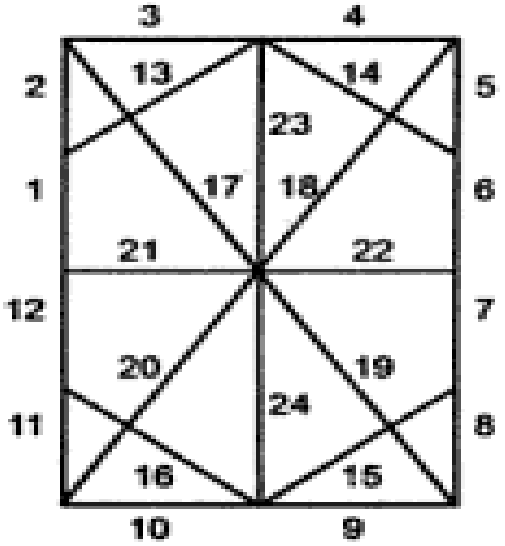
# Stroke Method

This method uses small line segments to generate a character.

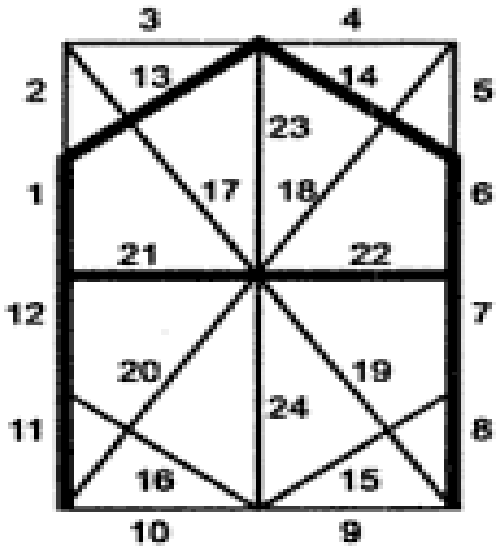
- The small series of line segments are drawn like a strokes of a pen to form a character as shown in figure.
- We can build our own stroke method.
- By calling a line drawing algorithm.
- Here it is necessary to decide which line segments are needed for each character and
- Then drawing these segments using line drawing algorithm.
- This method supports scaling of the character.
- It does this by changing the length of the line segments used for character drawing.



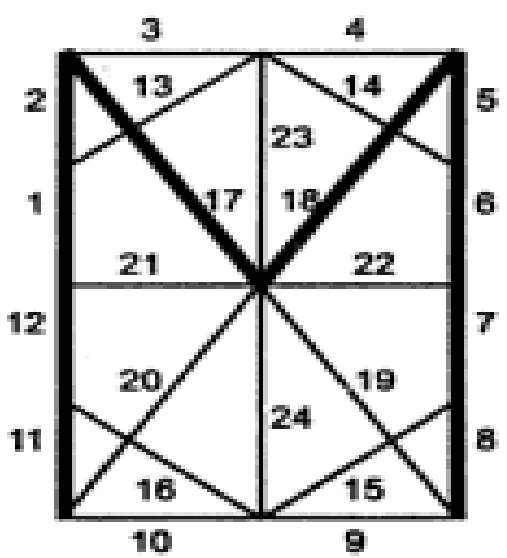
# Starburst method




a) Star bust pattern of 24 line segments




b) Star bust pattern for character A




c) Star bust pattern for character M

In this method a fix pattern of line segments are used to generate characters.

- As shown in figure, there are 24 line segments.
- Out of 24 line segments, segments required to display for particular character, are highlighted
- This method is called starburst method because of its characteristic appearance.

This method of character generation has some disadvantages. They are

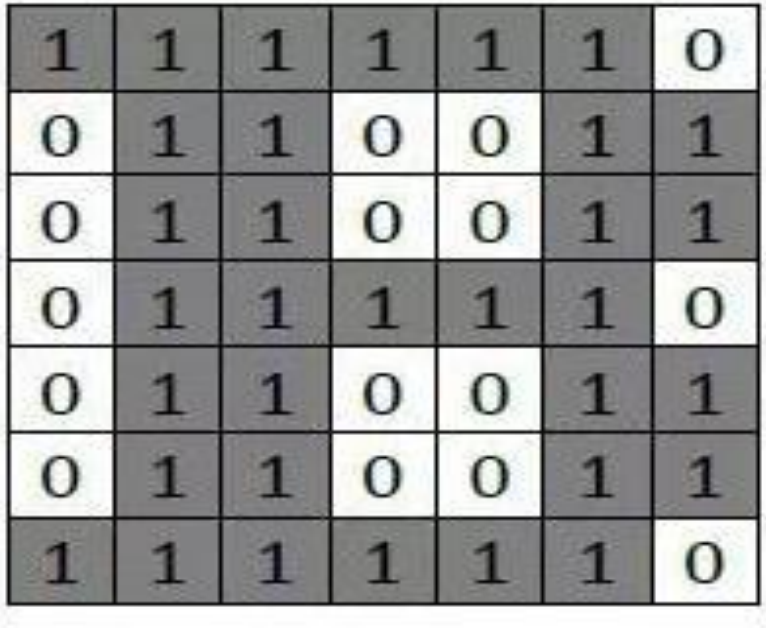
1. The 24-bits are required to represent a character. Hence more memory is required
2. Requires code conversion software to display character from its 24-bit code
3. Character quality is poor. It is worst for curve shaped characters.

# Bitmap Method

Vidyalankar Polytechnic

Also known as dot matrix because in this method characters are represented by an array of dots in the matrix form.

- It's a two dimensional array having columns and rows : 7 X 7 as shown in figure.
- 7 X 9 and 9 X 13 arrays are also used.
- Higher resolution devices may use character array 100 X 100.



| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 |