

## Assignment No: 01

### Title/ Problem Statement:

Write C++/Java program to draw line using DDA and Bresenham's algorithm. Inherit pixel class and Use function overloading.

**Prerequisites:** Coordinates System, Mathematics

**Objectives:** To learn use of slope to draw line.

**Theory :**

### DDA (Digital Differential Analyzer):

Scan conversion line algorithm based on  $\Delta x$  or  $\Delta y$ . Sample the line at unit interval in one coordinate and determine corresponding integer value. Faster method than  $y = mx + b$  using but may specifies inaccurate pixel section. Rounding off operations and floating point arithmetic operations are time consuming. DDA is used in drawing straight line to form a line, triangle or polygon in computer graphics. DDA analyzes samples along the line at regular interval of one coordinate as the integer and for the other coordinate it rounds off the integer that is nearest to the line. Therefore as the line progresses it scan the first integer coordinate and round the second to nearest integer. Therefore a line drawn using DDA for x coordinate it will be  $x_0$  to  $x_1$  but for y coordinate it will be  $y = ax + b$  and to draw function it will be  $fn(x, y \text{ rounded off})$ .

	i Plot	x	y	e
		5	5	0
	1 (5, 5)	6	6	-8
	2 (6, 6)	7	6	0
3 (7, 6)		8	7	-8
4 (8, 7)		9	7	0
5 (9, 7)		10	8	-8
		11	8	0
	6 (10, 8)	12	9	-8
	7 (11, 8)	13	9	0
	8 (12, 9)			

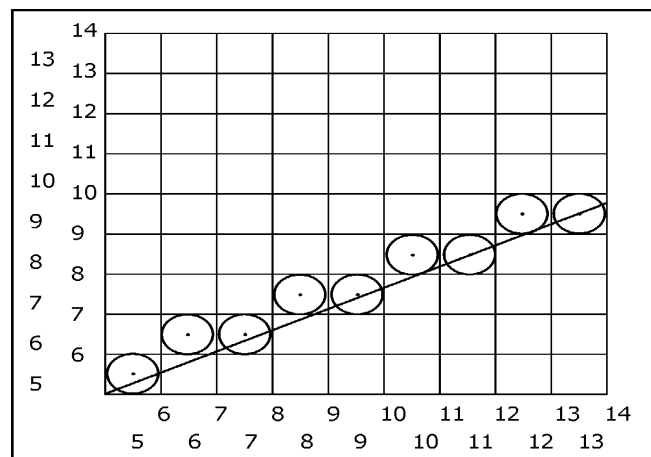


Fig DDA Line Algorithm

### Advantages & disadvantages of DDA:-

- Faster than the direct use of the line equation and it does not do any floating point multiplication.
- Floating point Addition is still needed.
- Precision loss because of rounding off.
- Pixels drift farther apart if line is relatively larger.

### Bresenham line algorithm:

This was developed by J.E.Bresenham in 1962 and it is much accurate and much more efficient than DDA. An algorithm which determines which order to form a close approximation to a straight line between two given points. It is commonly used to draw lines on a computer screen, as it uses only integer addition, subtraction and bit shifting, all of which are very cheap operations in standard computer architectures. It scans the coordinates but instead of rounding them off it takes the incremental value in account by adding or subtracting and therefore can be used for drawing circle and curves. Therefore if a line is to be drawn between two points x and y then next coordinates will be(  $x_a+1$ ,  $y_a$ ) and ( $x_a+1$ ,  $y_a+1$ ) where a is the incremental value of the next coordinates and difference between these two will be calculated by subtracting or adding the equations formed by them.

	i Plot	x	y	e
		2	5	2
1.	(2, 5)	3	6	-4
2.	(3, 6)	4	6	6
3.	(4, 6)	5	7	4
4.	(5, 7)	6	8	2
5.	(6, 7)	7	8	+8
6.	(7, 8)	8	9	2
7.	(8, 9)	9	10	-4
8.	(9, 10)	10	10	

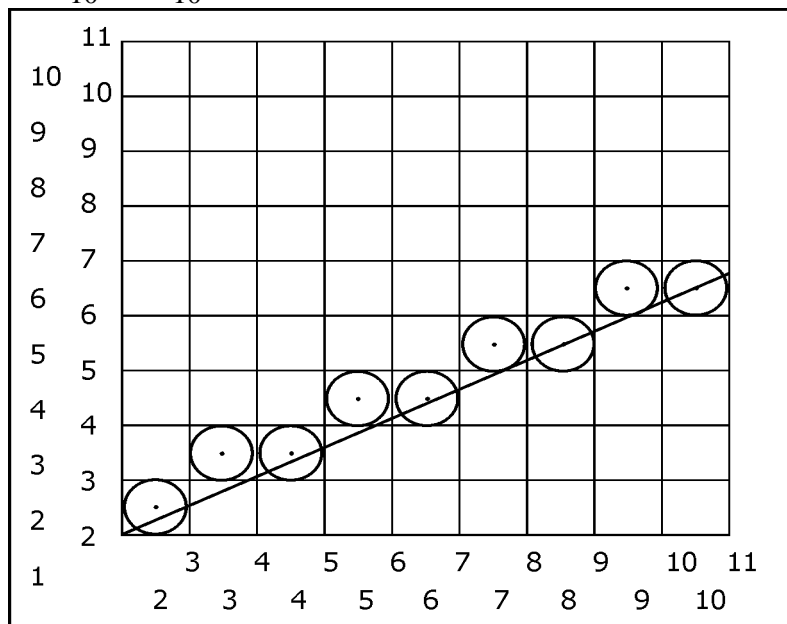


Fig Bresenham's Line Algorithm

## Algorithm :

### ***DDA LINE DRAWING ALGORITHM***

1. Start.
2. Declare variables  $x, y, x_1, y_1, x_2, y_2, k, dx, dy, s, x_i, y_i$  and also declare  $gdriver=DETECT, gmode$ .
3. Initialise the graphic mode with the path location in TC folder.
4. Input the two line end-points and store the left end-points in  $(x_1, y_1)$ .
5. Load  $(x_1, y_1)$  into the frame buffer; that is, plot the first point. put  $x=x_1, y=y_1$ .
6. Calculate  $dx=x_2-x_1$  and  $dy=y_2-y_1$ .
7. If  $\text{abs}(dx) > \text{abs}(dy)$ , then  $\text{steps}=\text{abs}(dx)$ .
8. Otherwise  $\text{steps}=\text{abs}(dy)$ .
9. Then  $x_i=dx/\text{steps}$  and  $y_i=dy/\text{steps}$ .
10. Start from  $k=0$  and continuing till  $k<\text{steps}$ , the points will be
  - i.  $x=x+x_i$ .
  - ii.  $y=y+y_i$ .
11. Plot pixels using `putpixel` at points  $(x, y)$  in specified colour.
12. Close Graph.
13. Stop.

### ***BRESENHAM'S LINE DRAWING ALGORITHM WITH SLOPE < 1***

1. Start.
2. Declare variables  $x, y, x_1, y_1, x_2, y_2, p, dx, dy$  and also declare  $gdriver=DETECT, gmode$ .
3. Initialize the graphic mode with the path location in TC folder.
4. Input the two line end-points and store the left end-points in  $(x_1, y_1)$ .
5. Load  $(x_1, y_1)$  into the frame buffer; that is, plot the first point put  $x=x_1, y=y_1$ .
6. Calculate  $dx=x_2-x_1$  and  $dy=y_2-y_1$ , and obtain the initial value of decision parameter  $p$  as: a.  $p=(2dy-dx)$ .
7. Starting from first point  $(x, y)$  perform the following test:
8. Repeat step 9 while  $(x \leq x_2)$ .
9. If  $p < 0$ , then
  - next point is  $(x+1, y)$
  - and calculate  $p=(p+2dy)$ .
- Otherwise,
  - the next point to plot is  $(x+1, y+1)$

and calculate  $p=(p+2dy-2dx)$ .

10. Plot pixels using putpixel at points (x,y) in specified colour.

11. Close Graph.

12. Stop.

## ***Write GENERALIZED BRESENHAM'S LINE DRAWING ALGORITHM***

**Input :**

//Assignment No.1

```
#include<iostream>
#include<graphics.h>
using namespace std;
```

```
class Pixel
{
public:
    void plotPixel(int x,int y)
    {
        putpixel(x,y,WHITE);
    }
};
```

```
class Line : public Pixel
{
public:
    void drawLine(int x1,int y1,int x2,int y2,int d)//dda
    {
        int steps,i;
        float dx,dy,xinc,yinc,x,y;
        dx = abs(x2-x1);
        dy = abs(y2-y1);
        if(dx>dy) //gentle
        {
            steps = dx;
            xinc = dx/steps;
            yinc = dy/steps;
            plotPixel(x1,y1);
            x = x1;
            y = y1;
            for(i=1 ;i <= steps; i++)
            {
                if(x1<x2) //forward
                {
                    x = x + xinc;
                    if(y1>y2)
                        y = y - yinc;
```

```

        else
            y = y + yinc;
            plotPixel(x,y+0.5);
        }
        else //backward
        {
            x = x - xinc;
            if(y1>y2)
                y = y - yinc;
            else
                y = y + yinc;
            plotPixel(x,y+0.5);
        }
        delay(50);
    }
}
else //steep
{
    steps = dy;
    xinc = dx/steps;
    yinc = dy/steps;
    plotPixel(x1,y1);
    x = x1;
    y = y1;
    for(i=1 ;i <= steps; i++)
    {
        if(y1<y2) //forward
        {
            if(x1<x2)
                x = x + xinc;
            else
                x = x - xinc;
            y = y + yinc;
            plotPixel(x+0.5,y);
        }
        else //backward
        {
            if(x1<x2)
                x = x + xinc;
            else
                x = x - xinc;
            y = y - yinc;
            plotPixel(x+0.5,y);
        }
        delay(50);
    }
}
}
}

//end
void drawLine(int x1,int y1,int x2,int y2,char d)//bresenham
{
    int x,y,i;
    float dx,dy,g;
    x = x1;
    y = y1;
    plotPixel(x,y);
    dx = abs(x2-x1);
    dy = abs(y2-y1);

```

```

if(dx>dy) //gentle
{
    g = 2*dy-dx;
    for(i = 1; i<= dx ; i++)
    {
        if(x1<x2) //forward
        {
            x = x + 1;
            if(g>0)
            {
                if(y1<y2)
                    y = y + 1;
                else
                    y = y - 1;
                g = g + 2 * dy - 2 * dx;
            }
            else
            {
                g = g + 2 * dy;
            }
            plotPixel(x,y);
            delay(50);
        }
        else //backward
        {
            x = x - 1;
            if(g>0)
            {
                if(y1<y2)
                    y = y + 1;
                else
                    y = y - 1;
                g = g + 2 * dy - 2 * dx;
            }
            else
            {
                g = g + 2 * dy;
            }
            plotPixel(x,y);
            delay(50);
        }
    }
}

else //steep
{
    g = 2*dx-dy;
    for(i = 1; i<= dy ; i++)
    {
        if(y1<y2) //forward
        {
            y = y + 1;
            if(g>0)
            {
                if(x1<x2)
                    x = x + 1;
            }
        }
    }
}

```

```

        else
            x = x - 1;
            g = g + 2 * dx - 2 * dy;
        }
        else
        {
            g = g + 2 * dx;
        }
        plotPixel(x,y);
        delay(50);
    }
    else //backward
    {
        y = y - 1;
        if(g>0)
        {
            if(x1<x2)
                x = x + 1;
            else
                x = x - 1;
            g = g + 2 * dx - 2 * dy;
        }
        else
        {
            g = g + 2 * dx;
        }
        plotPixel(x,y);
        delay(50);
    }
}

}

}

} //end

};

int main()
{
    int gd = DETECT, gm,x1,x2,y1,y2;
    initgraph(&gd,&gm,NULL);
    Line l;
    cout<<"Enter line coordinates";
    cin>>x1>>y1>>x2>>y2;
    l.drawLine(x1,y1,x2,y2,1); //dda
    l.drawLine(x1+10,y1+10,x2+10,y2+10,'b'); //bre
    getch();
    closegraph();
    return 0;
}

```

### Output : Line Plotted

### Conclusion:

Thus we have implemented program to draw Line using DDA and Bresenham's circle generation algorithms.

## Assignment No: 02

### Title/ Problem Statement:

Write C++/Java program to draw circle using Bresenham's algorithm. Inherit pixel class.

### Prerequisites:

### OBJECTIVE:

1. To understand Bresenham's circle drawing algorithms used for computer graphics.

### THEORY:

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 Midpoint circle drawing algorithm works on the same midpoint concept as the Bresenham's line algorithm. In the Midpoint circle drawing algorithm, you determine the next pixel to be plotted based on the position of the midpoint between the current and next consecutive pixel.

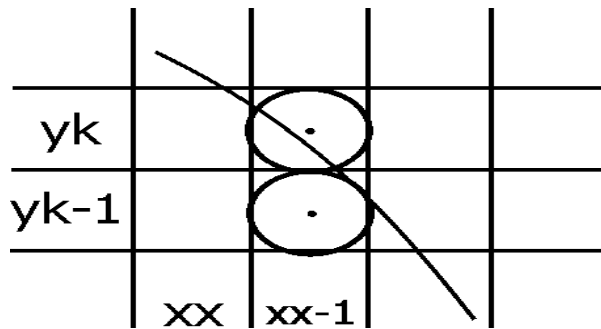


Fig 6.1 Midpoint circle Algorithm

### PLATFORM REQUIRED:

- Microsoft Windows 7/ Windows 8 Operating System onwards or 64-bit Open source Linux or its derivative.

### Algorithm :

#### ***BRESENHAM'S ALGORITHM TO DRAW A CIRCLE.***

1. Start.
2. Declare variables x,y,p and also declare gdriver=DETECT,gmode.
3. Initialise the graphic mode with the path location in TC folder.



4. Input the radius of the circle  $r$ .
5. Load  $x=0, y=r$ , initial decision parameter  $p=1-r$ . so the first point is  $(0, r)$ .
6. Repeat Step 7 while  $(x < y)$  and increment  $x$ -value simultaneously.
7. If  $(p > 0)$ ,
  - do  $p = p + 2 * (x - y) + 1$ .
8. Otherwise
  - $p = p + 2 * x + 1$  and
  - $y$  is decremented simultaneously.
9. Then calculate the value of the function `circlepoints()` with parameters  $(x, y)$ .
10. Place pixels using `putpixel` at points  $(x+300, y+300)$  in specified colour in `circlepoints()` function shifting the origin to 300 on both  $x$ -axis and  $y$ -axis.
11. Close Graph.
12. Stop.

## BRESENHAM'S CIRCLE ALGORITHM

### Bresenham Circle ( $X_c, Y_c, R$ ):

**Description:** Here  $X_c$  and  $Y_c$  denote the  $x$  – coordinate and  $y$  – coordinate of the center of the circle.  $R$  is the radius.

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

$$P_0 = 3 - 2R$$

3. If  $p_k < 0$ , the next point along the circle is  $(x_k + 1, y_k)$  and

$$P = P + 4X + 6$$

Otherwise, the next point along the circle is  $(x_k + 1, y_k - 1)$  and

$$P = P + 4(X - Y) + 10$$

4. Determine symmetry points in the other seven octants.
5. Move each calculated pixel position  $(x, y)$  onto the circular path centered on  $(x_c, y_c)$  and plot the coordinate values :

$$x = x + x_c$$

$$y = y + y_c$$

6. Repeat steps 3 through 5 while  $x < y$ .

### **Draw Symmetry points ( $X_c, Y_c, X, Y$ ):**

1. Call PutPixel( $X_c + X, Y_c, + Y$ )
2. Call PutPixel( $X_c - X, Y_c, + Y$ )
3. Call PutPixel( $X_c + X, Y_c, - Y$ )
4. Call PutPixel( $X_c - X, Y_c, - Y$ )
5. Call PutPixel( $X_c + Y, Y_c, + X$ )
6. Call PutPixel( $X_c - Y, Y_c, + X$ )
7. Call PutPixel( $X_c + Y, Y_c, - X$ )
8. Call PutPixel( $X_c - Y, Y_c, - X$ )

## ***WAP TO DRAW A CIRCLE USING BRESENHAM'S ALGORITHM.***

```
#include<iostream>
#include<graphics.h>
using namespace std;

class Pixel
{
public:
    void plotPixel(int x,int y)
    {
        putpixel(x,y,WHITE);
    }
};

class MyCircle : public Pixel
{
public:
    void drawCircle(int xc,int yc,int r)
    {
        int x,y;
        float s;
        x = 0;
        y = r;
        s = 3-2*r;
        while(x < y)
        {
            if(s<=0)
            {
                s = s + 4 * x + 6;
                x = x + 1;
            }
        }
    }
};
```

```

        }
        else
        {
            s = s + 4 * (x-y) + 10;
            x = x + 1;
            y = y - 1;
        }
        display(xc,yc,x,y);
        delay(100);
    }

}

void display(int xc,int yc,int x,int y)
{
    plotPixel(xc+x,yc+y);
    plotPixel(xc+y,yc+x);
    plotPixel(xc+y,yc-x);
    plotPixel(xc+x,yc-y);
    plotPixel(xc-x,yc-y);
    plotPixel(xc-y,yc-x);
    plotPixel(xc-y,yc+x);
    plotPixel(xc-x,yc+y);
}

};

int main()
{
    int gd = DETECT, gm,xc,yc,r;
    initgraph(&gd,&gm,NULL);
    MyCircle c;
    cout<<"Enter center coordinates and radius";
    cin>>xc>>yc>>r;
    c.drawCircle(xc,yc,r);
    getch();
    closegraph();
    return 0;
}

```

**Input :** Coordinates of Center and radius of Circle

**Output :** Circle plotted

**Conclusion:** Thus we have implemented program to draw circle using Bresenham's circle generation algorithms.

## Assignment No: 03

### **Title/ Problem Statement:**

Write C++/Java program to draw 2-D object and perform following basic transformations,

- a) Scaling
- b) Translation
- c) Rotation

Use operator overloading

**Prerequisites:** Matrix operations, such as addition, subtraction, multiplication

**Objectives:** To study basic object transformations using matrix operations.

### **Theory :**

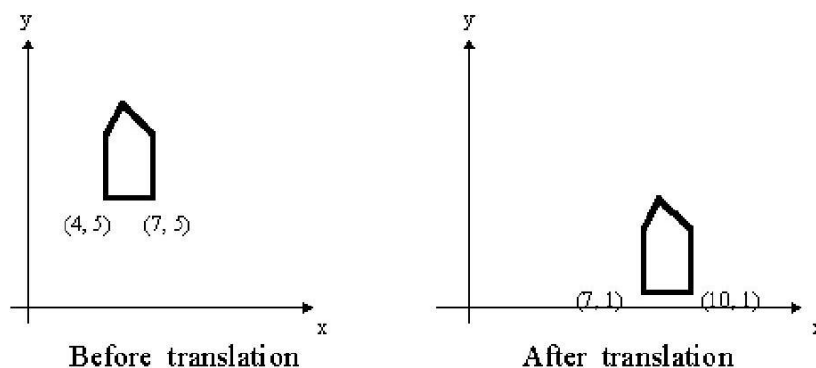
**Aim:** To apply the basic 2D transformations such as translation, Scaling, Rotation, shearing and reflection for a given 2D object.

**Description:** We have to perform 2D transformations on 2D objects. Here we perform transformations on a line segment.

The 2D transformations are:

1. Translation
2. Scaling
3. Rotation
4. Reflection
5. Shear

**1. Translation:** Translation is defined as moving the object from one position to another position along straight line path.



We can move the objects based on translation distances along x and y axis.  $t_x$  denotes translation distance along x-axis and  $t_y$  denotes translation distance along y axis.

**Translation Distance:** It is nothing but by how much units we should shift the object from one location to another along x, y-axis.

Consider  $(x,y)$  are old coordinates of a point. Then the new coordinates of that same point  $(x',y')$  can be obtained as follows:

$$X'=x+tx$$

$$Y'=y+ty$$

We denote translation transformation as P. we express above equations in matrix form as:

$$P' = P + T$$

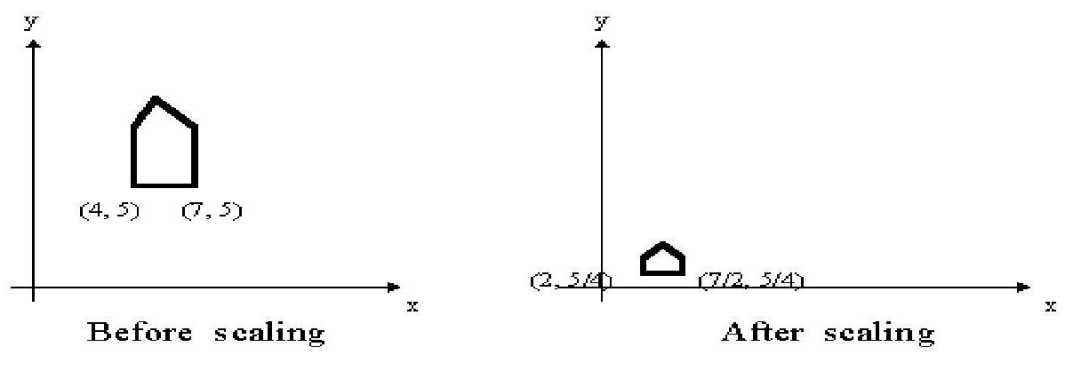
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} tx \\ ty \end{bmatrix}$$

$x,y$ ---old coordinates

$x',y'$ ---new coordinates after translation

$tx,ty$ ---translation distances, T is

**2. Scaling:** scaling refers to changing the size of the object either by increasing or decreasing. We will increase or decrease the size of the object based on scaling factors along x and y-axis.



If  $(x, y)$  are old coordinates of object, then new coordinates of object after applying scaling transformation are obtained as:

$$x' = x * sx$$

$$y' = y * sy.$$

$sx$  and  $sy$  are scaling factors along x-axis and y-axis. we express the above equations in matrix form as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} sx & 0 \\ 0 & sy \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Scaling Matrix

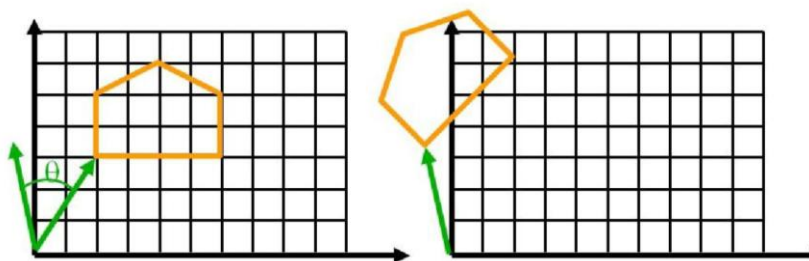
**3. Rotation:** A rotation repositions all points in an object along a circular path in the plane centered at the pivot point. We rotate an object by an angle theta.

New coordinates after rotation depend on *both* x and y

$$x' = x \cos \theta - y \sin \theta$$

$$y' = x \sin \theta + y \cos \theta$$

or in matrix form:



$$\mathbf{P}' = \mathbf{R} \bullet \mathbf{P},$$

$$\mathbf{R} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

R-rotation matrix.

$$\begin{bmatrix} \sin \theta & \cos \theta \end{bmatrix}$$

### Algorithm:

```
#include<iostream>
#include<stdlib.h>
#include<graphics.h> //header file for switching text to graphics mode
#include<math.h>
```

```
using namespace std;
class transform
{
    private:
        int edges,sx,sy,tx,ty;
        int input[20][2],scalm[2][2];
        float rot[2][2],theta,resmr[20][2];
        int resm[20][2];
        int trans[2][2],clockw;

    public:
        void menu();
        void accept();
        void scale();
        void translate();
        void rotate();
        void multiply(int a[20][2],int b[2][2],int c[20][2]);
        void multiply1(int a[20][2],float b[2][2],float c[20][2]);
        void plot(int mat[20][2]);
        void plot1(float mat[20][2]);
};
```

```
void transform::accept()
{
    int i,j;
    cout<<"\n Enter no. of edges in figure:";
```

```

cin>>edges;
cout<<"Enter co-ordinates in matrix form:";
for(i=1;i<=edges;i++)
{
    for(j=1;j<=2;j++)
    {
        cout<<"\nA["<<i<<"]["<<j<<"]="";
        cin>>input[i][j];
    }
}
plot(input);
}

void transform::plot(int mat[20][2])
{
    int i;
    line(0,240,640,240);
    line(320,0,320,480);
    for(i=1;i < edges;i++)
        line(320+mat[i][1],240-mat[i][2],320+mat[i+1][1],240-mat[i+1][2]);
    line(320+mat[1][1],240-mat[1][2],320+mat[i][1],240-mat[i][2]);
}

void transform::plot1(float mat[20][2])
{
    int i;
    line(0,240,640,240);
    line(320,0,320,480);
    for(i=1;i < edges;i++)
        line(320+mat[i][1],240-mat[i][2],320+mat[i+1][1],240-mat[i+1][2]);
    line(320+mat[1][1],240-mat[1][2],320+mat[i][1],240-mat[i][2]);
}

int main(void)
{
    char ch;
    int gd=DETECT,gm,i;
    initgraph(&gd,&gm,NULL);
    do
    {

        transform t;
        t.menu();
        cout<<"\n Do you want to continue:";
        cin>>ch;
    }while(ch=='y' | | ch=='Y');
    return 0;
}

void transform::menu()
{

```

```

int ch;
cout<<"_____";
cout<<"\n1. Scaling::";
cout<<"\n2. Translation::";
cout<<"\n3. Rotation::\n";
cout<<"_____";
cout<<"\nEnter ur choice::\t";
cin>>ch;

switch(ch)
{
    case 1:
        scale();
        break;

    case 2:
        translate();
        break;

    case 3:
        rotate();
        break;

    default: exit(0);
}
}

void transform::rotate()
{
    int i;
    float theta1;
    cleardevice();
    accept(); //accept the original polygon
    cout<<"\n Enter the angle for rotation:";
    cin>>theta; //accept angle for rotation
    cout<<"\n Enter 1 for clockwise rotation 0 for anti clockwise rotation:";
    cin>>clockw;
    theta1=(3.14*theta)/180; //conversion of degree to radian
    if(clockw==1)
    {
        rot[1][1]=rot[2][2]=cos(theta1);
        rot[1][2]=-sin(theta1);
        rot[2][1]=sin(theta1);
    }
    else
    {
        rot[1][1]=rot[2][2]=cos(theta1);
        rot[1][2]=sin(theta1);
        rot[2][1]=-sin(theta1);
    }
    multiply1(input,rot,resmr);
    plot1(resmr);
}

```



```

void transform::scale()
{
    int i;
    cleardevice();
    accept();
    cout<<"\n Enter the scale X factor:";
    cin>>sx;
    cout<<"\n Enter the scale Y factor:";
    cin>>sy;
    scalm[1][1]=sx;
    scalm[1][2]=scalm[2][1]=0;
    scalm[2][2]=sy;
    multiply(input,scalm,resm);
    plot(resm);
}

```

```

void transform::translate()
{
    int i,j;
    cleardevice();
    accept();
    cout<<"\n Enter tx factor";
    cin>>tx;
    cout<<"\n Enter ty factor";
    cin>>ty;
    for(i=1;i<=edges;i++)
    {
        input[i][1]= input[i][1]+tx;
        input[i][2]= input[i][2]+ty;
    }
    plot(input);
}

```

```

void transform::multiply(int a[20][2],int b[2][2],int c[20][2])
{
    int i;
    for(i=1;i<=edges;i++)
    {
        c[i][1]=(a[i][1]*b[1][1])+(a[i][2]*b[2][1]);
        c[i][2]=(a[i][1]*b[1][2])+(a[i][2]*b[2][2]);
    }
}

```

```

void transform::multiply1(int a[20][2],float b[2][2],float c[20][2])
{
    int i;
    for(i=1;i<=edges;i++)
    {
        c[i][1]=(a[i][1]*b[1][1])+(a[i][2]*b[2][1]);
        c[i][2]=(a[i][1]*b[1][2])+(a[i][2]*b[2][2]);
    }
}

```

```
}  
}
```

### Input/Output-

```
isbm@isbm-ThinkCentre-M72e:~/cg$ g++ ass3.cpp -lgraph  
isbm@isbm-ThinkCentre-M72e:~/cg$ ./a.out
```

- 
1. Scaling::
  2. Translation::
  3. Rotation::

---

Enter ur choice:: 2

Enter no. of edges in figure:3  
Enter co-ordinates in matrix form:  
A[1][1]=20

A[1][2]=50

A[2][1]=70

A[2][2]=100

A[3][1]=100

A[3][2]=20  
/n Enter tx factor3  
/n Enter ty factor3

Do you want to continue::y

- 
1. Scaling::
  2. Translation::
  3. Rotation::

---

Enter ur choice:: 1

Enter no. of edges in figure:3  
Enter co-ordinates in matrix form:  
A[1][1]=20

A[1][2]=50

A[2][1]=70

A[2][2]=100

A[3][1]=100

A[3][2]=20

Enter the scale X factor:2

Enter the scale Y factor:2

Do you want to continue::y

- 
1. Scaling::
  2. Translation::
  3. Rotation::

---

Enter ur choice:: 3

Enter no. of edges in figure:3

Enter co-ordinates in matrix form:

A[1][1]=20

A[1][2]=50

A[2][1]=70

A[2][2]=100

A[3][1]=100

A[3][2]=20

Enter the angle for rotation:45

Enter 1 for clockwise rotation 0 for anti clockwise rotation:1

Do you want to continue::n

**Conclusion:** Thus we have implemented program to perform scalling, translation and rotation operation.

## Assignment No: 4

### **Title/ Problem Statement:**

Write C++/Java program to fill polygon using scan line algorithm. Use mouse interfacing to draw polygon.

### **Objectives:**

1. To understand Scan Line Polygon filling algorithms used for computer graphics.
2. To understand Scan Line Concept

**Aim:** To implement the Scan line polygon fill algorithm for coloring a given object.

### **Theory :**

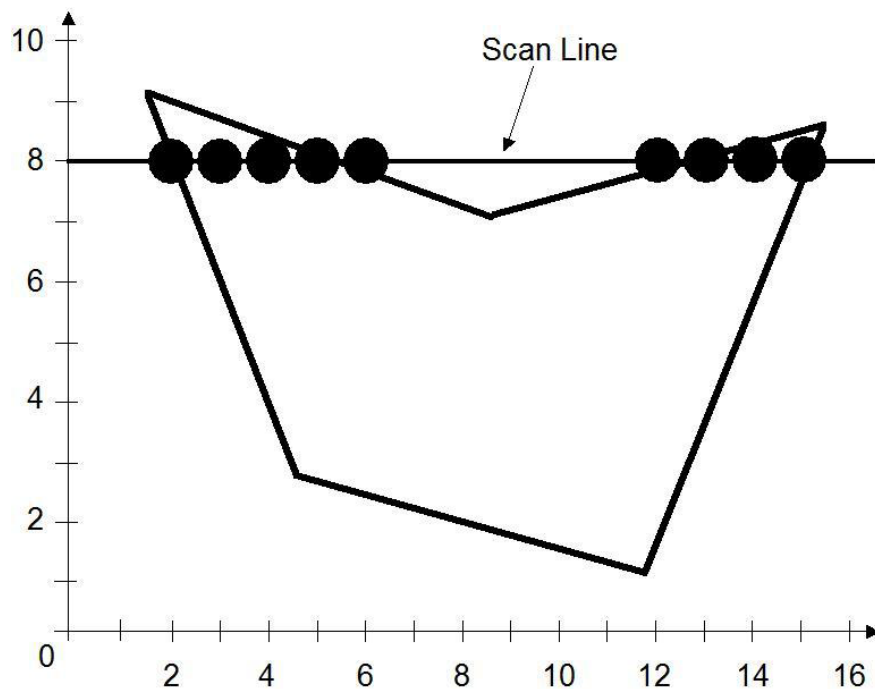
The basic scan-line algorithm is as follows:

- ☐ 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

Process involved:

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.



**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 spitted 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 scan line with polygon edges.

## Program:

// Program to Fill a Polygon Using Scan Line Fill Algorithm in C++.

```
#include <conio.h>
#include <iostream>
#include <graphics.h>
#include <stdlib.h>
using namespace std;

//Declaration of class point
class point
{
    public:
        int x,y;
};

class poly
{
    private:
        point p[20];
        int inter[20],x,y;
        int v,xmin,ymin,xmax,ymax;
    public:
        int c;
        void read();
        void sort();
        void display();
        void calculateX(float);
        void fillPoly(int);
        void drawPoly();
};

void poly::read()
{
    int i;
    cout<<"\n\tSCAN_FILL ALGORITHM";
    cout<<"\n Enter the no of vertices of polygon:";
    cin>>v;
    if(v>2)
    {
        for(i=0;i<v; i++) //ACCEPT THE VERTICES
        {
            cout<<"\nEnter the co-ordinate no.- "<<i+1<<" : ";
            cout<<"\n\tx"<<(i+1)<<"=";
            cin>>p[i].x;
            cout<<"\n\ty"<<(i+1)<<"=";
            cin>>p[i].y;
        }
    }
}
```

```

        p[i].x=p[0].x;
        p[i].y=p[0].y;
        xmin=xmax=p[0].x;
        ymin=ymax=p[0].y;
    }
    else
        cout<<"\n Enter valid no. of vertices.";
}
//FUNCTION FOR FINDING
void poly::sort()
{ //MAX,MIN
    for(int i=0;i<v;i++)
    {
        if(xmin>p[i].x)
            xmin=p[i].x;
        if(xmax<p[i].x)
            xmax=p[i].x;
        if(ymin>p[i].y)
            ymin=p[i].y;
        if(ymax<p[i].y)
            ymax=p[i].y;
    }
}
//DISPLAY FUNCTION
void poly::display()
{
    int ch1;
    char ch='y';
    float s,s2;
    do
    {
        cout<<"\n\nMENU:";
        cout<<"\n\n\t1 . Scan line Fill ";
        cout<<"\n\n\t2 . Exit ";
        cout<<"\n\nEnter your choice:";
        cin>>ch1;
        switch(ch1)
        {
            case 1:
                s=ymin+0.01;
                delay(100);
                drawPoly();
                cleardevice();
                while(s<=ymax)
                {
                    calculateX(s);
                    fillPoly(s);
                    s++;
                }
                break;
            case 2:
                exit(0);

```

```

    }

    cout<<"Do you want to continue?: ";
    cin>>ch;
}while(ch=='y' || ch=='Y');
}

void poly::calculateX(float z) //calculateX FUNCTION INTS
{
    int x1,x2,y1,y2,temp;
    c=0;
    for(int i=0;i<v;i++)
    {
        x1=p[i].x;
        y1=p[i].y;
        x2=p[i+1].x;
        y2=p[i+1].y;
        if(y2<y1)
        {
            temp=x1;
            x1=x2;
            x2=temp;
            temp=y1;
            y1=y2;
            y2=temp;
        }
        if(z<=y2&& z>=y1)
        {
            if((y1-y2)==0)
                x=x1;
            else // used to make changes in x. so that we can fill our polygon after cerain distance
            {
                 $x = ((x2 - x1) * (z - y1)) / (y2 - y1);$ 
                x=x+x1;
            }
            if(x<=xmax && x>=xmin)
                inter[c++] = x;
        }
    }
}

void poly::drawPoly() //Draw Poly FUNCTION
{
    int i;
    for(i=0;i<v;i++)
    {
        line(p[i].x,p[i].y,p[i+1].x,p[i+1].y); // used to make hollow outlines of a polygon
    }
}

void poly::fillPoly(int z) //fill Poly FUNCTION
{
    int i;

```



```

        for(i=0; i<c;i+=2)
        {
            delay(100);
            line(inter[i],z,inter[i+1],z); // Used to fill the polygon ....
        }
    }

int main() //START OF MAIN
{
    int cl;
    initwindow(500,600);
    cleardevice();
    poly x;
    x.read();
    x.sort();
    cleardevice();
    cout<<"\n\tEnter the colour u want:(0-15)->"; //Selecting colour
    cin>>cl;
    setcolor(cl);
    x.display();
    closegraph(); //CLOSE OF GRAPH
    getch();
    return 0;
}

```

### Output:

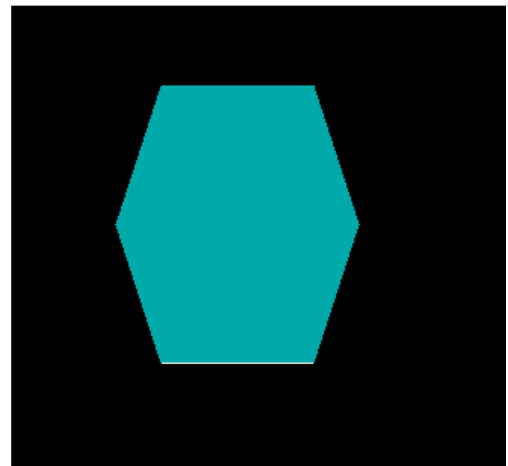
```

enter no. of edges of polygon:6

enter cordinatesof polygon :

x0 y0:100 200
x1 y1:200 200
x2 y2:230 300
x3 y3:200 400
x4 y4:100 400
x5 y5:70 300_

```

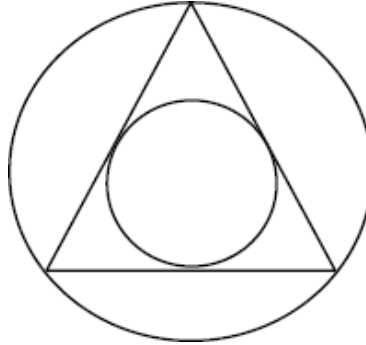


**Conclusion:** Thus we have studied Scan Line Polygon Filling Algorithm.

## Assignment No:5

### **Title/ Problem Statement:**

Write C++/Java program to draw inscribed and Circumscribed circles in the triangle as shown as an example below. (Use any Circle drawing and Line drawing algorithms)



### **Objectives:**

1. To understand Bresenham's circle drawing algorithms used for computer graphics.
2. To understand Equilateral Triangle concept

### **Theory :**

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 Midpoint circle drawing algorithm works on the same midpoint concept as the Bresenham's line algorithm. In the Midpoint circle drawing algorithm, you determine the next pixel to be plotted based on the position of the midpoint between the current and next consecutive pixel.

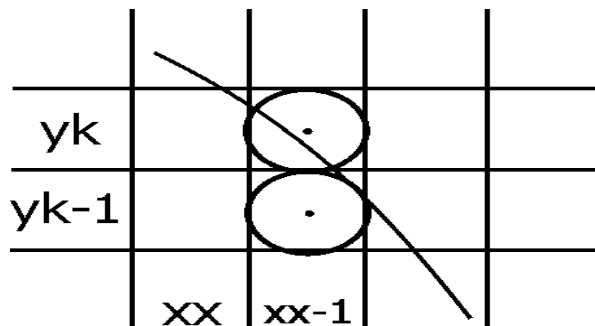


Fig 6.1 Midpoint circle Algorithm

### **Platform Required:**

- Microsoft Windows 7/ Windows 8 Operating System onwards or 64-bit Open source Linux or its derivative.

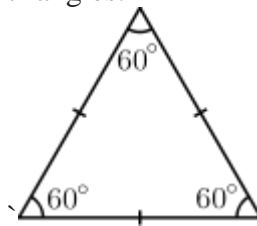
### **Algorithm :**

### **Bresenham's Algorithm to Draw a Circle.**

1. Start.
2. Declare variables  $x, y, p$  and also declare  $gdriver=DETECT, gmode$ .
3. Initialise the graphic mode with the path location in TC folder.
4. Input the radius of the circle  $r$ .
5. Load  $x=0, y=r$ , initial decision parameter  $p=1-r$ . so the first point is  $(0, r)$ .
6. Repeat Step 7 while  $(x < y)$  and increment  $x$ -value simultaneously.
7. If  $(p > 0)$ ,  
  
do  $p = p + 2*(x - y) + 1$ .
8. Otherwise  
 $p = p + 2*x + 1$  and  
 $y$  is decremented simultaneously.
9. Then calculate the value of the function  $circlepoints()$  with  $p$ . parameters  $(x, y)$ .
10. Place pixels using  $putpixel$  at points  $(x+300, y+300)$  in specified colour in  
 $circlepoints()$  function shifting the origin to 300 on both  $x$ -axis and  $y$ -axis.
11. Close Graph.
12. Stop.

### **Equilateral triangle**

In geometry, an equilateral triangle is a triangle in which all three sides are equal. In the familiar Euclidean geometry, equilateral triangles are also equiangular; that is, all three internal angles are also congruent to each other and are each  $60^\circ$ . They are regular polygons, and can therefore also be referred to as regular triangles.



### **Equal cevians**

Three kinds of cevians are equal for (and only for) equilateral triangles:

- The three altitudes have equal lengths.
- The three medians have equal lengths.
- The three angle bisectors have equal lengths.

### **Coincident triangle centers**

Every triangle center of an equilateral triangle coincides with its centroid, which implies that the equilateral triangle is the only triangle with no Euler line connecting some of the centers. For some pairs of triangle centers, the fact that they coincide is enough to ensure that the triangle is equilateral. In particular:

- A triangle is equilateral if any two of the circumcenter, incenter, centroid, or orthocenter coincide.
- It is also equilateral if its circumcenter coincides with the Nagel point, or if its incenter coincides with its nine-point center.

### **Six triangles formed by partitioning by the medians**

For any triangle, the three medians partition the triangle into six smaller triangles.

- A triangle is equilateral if and only if any three of the smaller triangles have either the same perimeter or the same inradius.
- A triangle is equilateral if and only if the circumcenters of any three of the smaller triangles have the same distance from the centroid.

### **Algorithm:**

```
#include<iostream>
#include<graphics.h>
using namespace std;

class Pixel
{
public:
    void plotPixel(int x,int y)
    {
        putpixel(x,y,WHITE);
    }
};

class Line : public Pixel
{
public:
    void drawLine(int x1,int y1,int x2,int y2,int d)//dda
    {
        int steps,i;
        float dx,dy,xinc,yinc,x,y;
        dx = abs(x2-x1);
        dy = abs(y2-y1);
        if(dx>dy) //gentle
        {
            steps = dx;
            xinc = dx/steps;
            yinc = dy/steps;
            plotPixel(x1,y1);
            x = x1;
            y = y1;
            for(i=1 ;i <= steps; i++)
```

```

{
    if(x1<x2)    //forward
    {
        x = x + xinc;
        if(y1>y2)
            y = y - yinc;
        else
            y = y + yinc;
        plotPixel(x,y+0.5);
    }
    else    //backward
    {
        x = x - xinc;
        if(y1>y2)
            y = y - yinc;
        else
            y = y + yinc;
        plotPixel(x,y+0.5);
    }
    delay(50);
}

else    //steep
{
    steps = dy;
    xinc = dx/steps;
    yinc = dy/steps;
    plotPixel(x1,y1);
    x = x1;
    y = y1;
    for(i=1 ;i <= steps; i++)
    {
        if(y1<y2)    //forward
        {
            if(x1<x2)
                x = x + xinc;
            else
                x = x - xinc;
            y = y + yinc;
            plotPixel(x+0.5,y);
        }
        else    //backward
        {
            if(x1<x2)
                x = x + xinc;
            else
                x = x - xinc;
            y = y - yinc;
            plotPixel(x+0.5,y);
        }
        delay(50);
    }
}

}
//end
void drawLine(int x1,int y1,int x2,int y2,char d)//bresenham
{

```

```

int x,y,i;
float dx,dy,g;
x = x1;
y = y1;
plotPixel(x,y);
dx = abs(x2-x1);
dy = abs(y2-y1);

if(dx>dy) //gentle
{
    g = 2*dy-dx;
    for(i = 1; i<= dx ; i++)
    {
        if(x1<x2) //forward
        {
            x = x + 1;

            if(g>0)
            {
                if(y1<y2)
                    y = y + 1;
                else
                    y = y - 1;
                g = g + 2 * dy - 2 * dx;
            }
            else
            {
                g = g + 2 * dy;
            }
            plotPixel(x,y);
            delay(50);
        }
        else //backward
        {
            x = x - 1;
            if(g>0)
            {
                y = y - 1;
                g = g + 2 * dy - 2 * dx;
            }
            else
            {
                g = g + 2 * dy;
            }
            plotPixel(x,y);
            delay(50);
        }
    }
}

else //steep
{
    g = 2*dx-dy;
    for(i = 1; i<= dy ; i++)
    {
        if(y1<y2) //forward
        {

```

```

        y = y + 1;
    if(g>0)
    {
        if(x1<x2)
            x = x + 1;
        else
            x = x - 1;
        g = g + 2 * dx - 2 * dy;
    }
    else
    {
        g = g + 2 * dx;
    }
    plotPixel(x,y);
    delay(50);
}
else //backward
{
    y = y - 1;
    if(g>0)
    {
        if(x1<x2)
            x = x + 1;
        else
            x = x - 1;
        g = g + 2 * dx - 2 * dy;
    }
    else
    {
        g = g + 2 * dx;
    }
    plotPixel(x,y);
    delay(50);
}

}

}

} //end

};

class MyCircle : public Pixel
{
public:
    void drawCircle(int xc,int yc,int r)
    {
        int x,y;
        float s;
        x = 0;
        y = r;
        s = 3-2*r;
        while(x < y)
        {
            if(s<=0)
            {
                s = s + 4 * x + 6;

```

```

        x = x + 1;
    }
    else
    {
        s = s + 4 * (x-y) + 10;
        x = x + 1;
        y = y - 1;
    }
    display(xc,yc,x,y);
    delay(100);
}

}

void display(int xc,int yc,int x,int y)
{
    plotPixel(xc+x,yc+y);
    plotPixel(xc+y,yc+x);
    plotPixel(xc+y,yc-x);
    plotPixel(xc+x,yc-y);
    plotPixel(xc-x,yc-y);
    plotPixel(xc-y,yc-x);
    plotPixel(xc-y,yc+x);
    plotPixel(xc-x,yc+y);
}

};

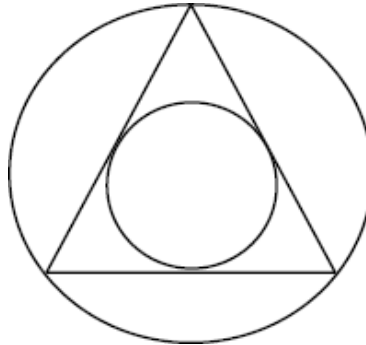
int main()
{
    int gd = DETECT, gm,x1,x2,y1,y2,xc,yc,r;
    float x3,y3,xm,ym,t1,t2,t,d;
    initgraph(&gd,&gm,NULL);
    Line l;
    MyCircle c;
    x1=200;y1=200;d=200;
    t=(sqrt(3)/2)*d;
    xm=x1;
    ym=y1+t;
    x2=xm-(d/2);
    y2=ym;
    x3=xm+(d/2);
    y3=ym;
    xc=(x1+x2+x3)/3;
    yc=(y1+y2+y3)/3;
    t1=sqrt(pow(xm-xc,2)+pow(ym-yc,2));
    t2=t-t1;
    l.drawLine(x2,y2,x1,y1,1);
    l.drawLine(x2,y2,x3,y3,1);
    l.drawLine(x1,y1,x3,y3,1);
    c.drawCircle(xc,yc,t2);
    c.drawCircle(xc,yc,t1);
    getch();
    closegraph();
    return 0;
}

```



**Input:** length of edge of triangle and starting point of line.

**Output:**



**Conclusion:** Thus we have Studied inscribed and Circumscribed circles in the triangle.

## Assignment No:

### **Title/ Problem Statement:**

Write C++/Java program for line drawing using DDA or Bresenham's algorithm with patterns such as solid, dotted, dashed and thick.

### **Objectives:**

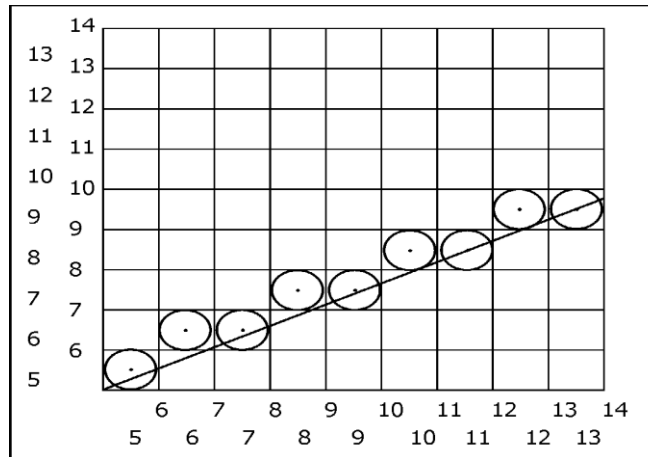
1. To understand Bresenham's or DDA Line drawing algorithms used for computer graphics.
2. To understand concept of different line styles using line algorithms

### **Theory :**

#### **DDA (Digital Differential Analyzer):**

Scan conversion line algorithm based on  $\Delta x$  or  $\Delta y$ . Sample the line at unit interval in one coordinate and determine corresponding integer value. Faster method than  $y = mx + b$  using but may specify inaccurate pixel section. Rounding off operations and floating point arithmetic operations are time consuming. DDA is used in drawing straight line to form a line, triangle or polygon in computer graphics. DDA analyzes samples along the line at regular interval of one coordinate as the integer and for the other coordinate it rounds off the integer that is nearest to the line. Therefore as the line progresses it scan the first integer coordinate and round the second to nearest integer. Therefore a line drawn using DDA for x coordinate it will be  $x_0$  to  $x_1$  but for y coordinate it will be  $y = ax + b$  and to draw function it will be  $fn(x, y \text{ rounded off})$ .

i Plot	x	y	e
	5	5	0
1 (5, 5)	6	6	-8
2 (6, 6)	7	6	0
3 (7, 6)	8	7	-8
4 (8, 7)	9	7	0
5 (9, 7)	10	8	-8
6 (10, 8)	11	8	0
7 (11, 8)	12	9	-8
8 (12, 9)	13	9	



**Fig DDA Line Algorithm**

### **Advantages & disadvantages of DDA:-**

- Faster than the direct use of the line equation and it does not do any floating point multiplication.
- Floating point Addition is still needed.
- Precision loss because of rounding off.
- Pixels drift farther apart if line is relatively larger.

```
#include<iostream>
#include<graphics.h>
using namespace std;
class linep
{
public:
    int x1,y1,x2,y2;
    void drawl(int,int,int,int);
    void thick(int,int,int,int,int);
    void dotted(int,int,int,int);
    void dash(int ,int ,int,int);
};

void linep::thick(int x1,int y1,int x2,int y2,int w)
{
    int i,dx,dy,step,xinc,yinc;
    dx=x2-x1;
    dy=y2-y1;
    if(dy>=dx)
    {
        step=dy;
    }
    else
    {
        step=dx;
    }
    xinc=dx/step;
```

```

        yinc=dy/step;
        while(x1<=x2)
        {
            x1=x1+xinc+0.5;
            y1=y1+yinc+0.5;
            for(i=0;i<w;i++)
            {
                putpixel(x1+i,y1,BLUE);
            }
        }
    }
void linep::drawl(int x1,int y1,int x2,int y2)
{
    int dx,dy,step,xinc,yinc;
    dx=x2-x1;
    dy=y2-y1;
    if(dy>=dx)
    {
        step=dy;
    }
    else
    {
        step=dx;
    }
    xinc=dx/step;
    yinc=dy/step;
    putpixel(x1,y1,BLUE);
    while(x1<=x2)
    {
        x1=x1+xinc+0.5;
        y1=y1+yinc+0.5;
        putpixel(x1,y1,BLUE);
    }
}
void linep::dotted(int x1,int y1,int x2,int y2)
{
    int i=0,dx,dy,step,xinc,yinc;
    dx=x2-x1;
    dy=y2-y1;
    if(dy>=dx)
    {
        step=dy;
    }
    else
    {
        step=dx;
    }
    xinc=dx/step;
    yinc=dy/step;
    putpixel(x1,y1,BLUE);
    while(x1<=x2)
    {

```

```

        if(i%2==0)
        {
            putpixel(x1,y1,BLUE);
        }
        x1=x1+xinc+0.5;
        y1=y1+yinc+0.5;
        i++;
    }
}

void linep::dash(int x1,int y1,int x2,int y2)
{
    int i=0,dx,dy,step,xinc,yinc;
    dx=x2-x1;
    dy=y2-y1;
    if(dy>=dx)
    {
        step=dy;
    }
    else
    {
        step=dx;
    }
    xinc=dx/step;
    yinc=dy/step;
    putpixel(x1,y1,BLUE);
    while(x1<=x2)
    {
        if(i%5!=0)
        {
            putpixel(x1,y1,BLUE);
        }
        x1=x1+xinc+0.5;
        y1=y1+yinc+0.5;
        i++;
    }
}

int main()
{
    class linep p;
    int gd=DETECT ,gm,n,i,w;
    initgraph(&gd,&gm,NULL);
    cout<<"Enter co-ordinates of line x1,y1,x2,y2";
    cin>>p.x1>>p.y1>>p.x2>>p.y2;
    cout<<" \n1.thin line \n2.thick line \n3.dotted line \n4.dash
line \nenter your choice:";
    cin>>i;
    switch(i)
    {
        case 1:
            p.drawl(p.x1,p.y1,p.x2,p.y2);
            break;
        case 2:

```

```

        cout<<"enter width of line in pixel";
        cin>>w;
        p.thick(p.x1,p.y1,p.x2,p.y2,w);
        break;
    case 3:
        p.dotted(p.x1,p.y1,p.x2,p.y2);
        break;
    case 4:p.dash(p.x1,p.y1,p.x2,p.y2);
        break;
        default:
        cout<<"Enter proper choice";
    }
    getch();
    return 0;
}

```

**Input:**

**Output:**

Line Displayed in Different styles like thin, thick, dotted, dash.

**Conclusion:** Thus we have Studied inscribed and Circumscribed circles in the triangle.

## Assignment No:

### **Title/ Problem Statement:**

Write C++/Java program to draw a convex polygon and fill it with desired color using Seed fill algorithm.

### **Prerequisites:**

### **Objectives:**

1. To understand Seed Fill algorithms used for computer graphics.

### **Theory :**

Fill Algorithms •

Given the edges defining a polygon, and a color for the polygon, we need to fill all the pixels inside the polygon.

- Three different algorithms: –
  1. Scan-line fill
  2. Boundary fill
  3. Flood fill

### **Filled Area Primitives**

Two basic approaches to area filling on raster systems:

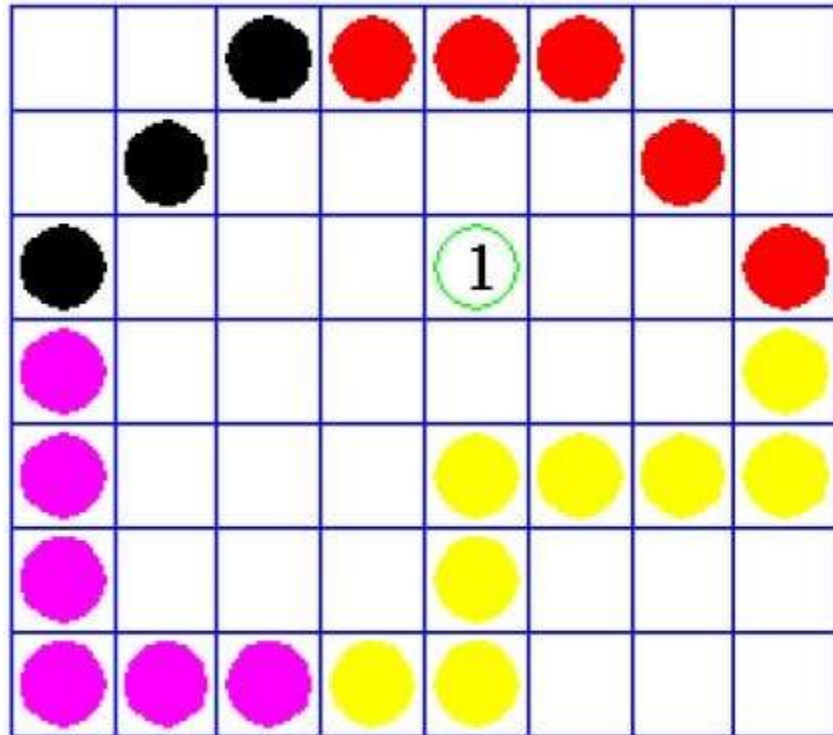
- Determine the overlap intervals for scan lines that cross the area (scan-line)
- Start from an interior position and point outward from this point until the boundary condition reached (fill method)
  - Scan-line: simple objects, polygons, circles,
  - Fill-method: complex objects, interactive fill.

### **Flood Fill Algorithm**

Sometimes we come across an object where we want to fill the area and its boundary with different colors. We can paint such objects with a specified interior color instead of searching for particular boundary color as in boundary filling algorithm.

Instead of relying on the boundary of the object, it relies on the fill color. In other words, it replaces the interior color of the object with the fill color. When no more pixels of the original interior color exist, the algorithm is completed.

Once again, this algorithm relies on the Four-connect or Eight-connect method of filling in the pixels. But instead of looking for the boundary color, it is looking for all adjacent pixels that are a part of the interior.



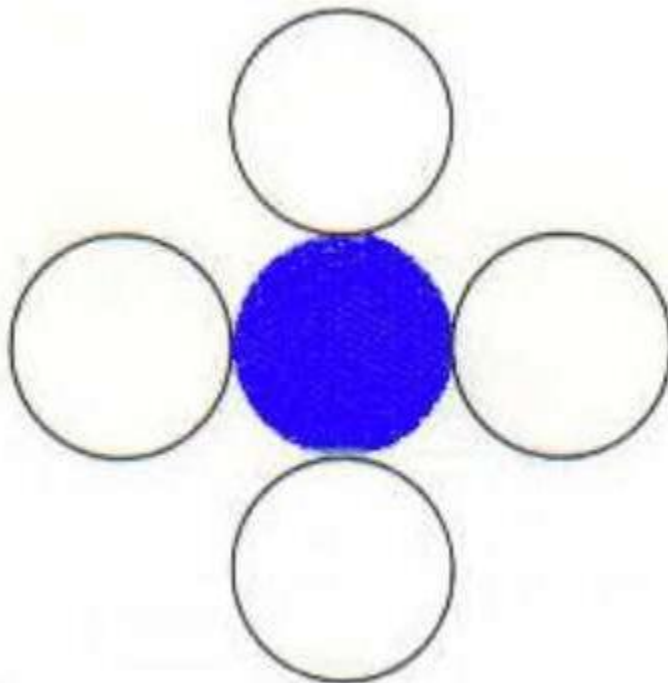
### Pixel-Defined Regions

**Definition:** Region R is the set of all pixels having color C that are connected to a given pixel S.

**4-adjacent:** pixels that lie next to each other horizontally or vertically, NOT diagonally.

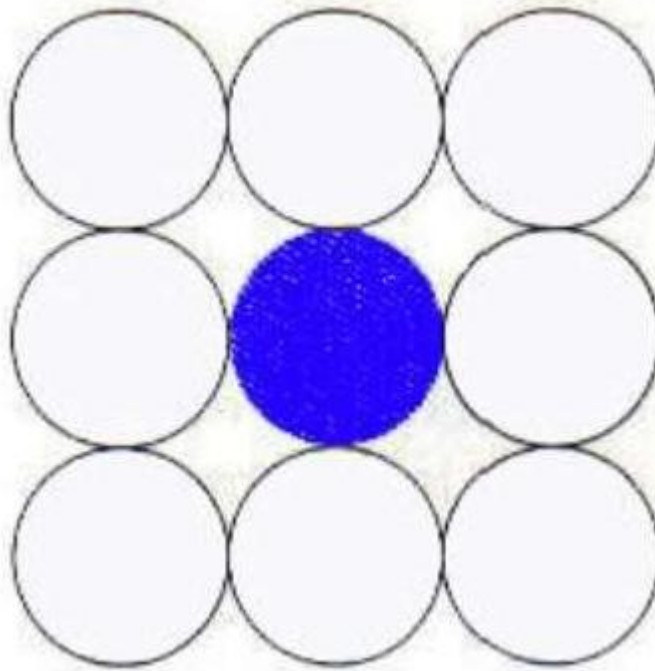
**8-adjacent:** pixels that lie next to each other horizontally, vertically OR diagonally.

**4-connected:** if there is unbroken path of 4-adjacent pixels connecting them.





**8-connected:** unbroken path of 8-adjacent pixels connecting them.



### Recursive Flood-Fill Algorithm

- Recursive algorithm
  - Starts from initial pixel of color, intColor
  - Recursively set 4-connected neighbors to newColor
  - **Flood-Fill:** floods region with newColor
- 
- **Basic idea:**
  - start at “seed” pixel (x, y)
  - If (x, y) has color intColor, change it to newColor
  - Do same recursively for all 4 neighbors

### Recursive Flood-Fill Algorithm

Note: getPixel(x,y) used to interrogate pixel color at (x, y)

```
void floodFill(short x, short y, short intColor)
{
    if(getPixel(x, y) == intColor)
    {
        setPixel(x, y);
        floodFill(x - 1, y, intColor); // left pixel
        floodFill(x + 1, y, intColor); // right pixel
        floodFill(x, y + 1, intColor); // down pixel
        floodFill(x, y - 1, intColor); // fill up
    }
}
```

}

### **Recursive Flood-Fill Algorithm**

- This version defines region using intColor
- Can also have version defining region by boundary
- Recursive flood-fill is somewhat blind and some pixels may be retested several times before algorithm terminates
- Region coherence is likelihood that an interior pixel mostly likely adjacent to another interior pixel Coherence can be used to improve algorithm performance
- A run is a group of adjacent pixels lying on same
- Exploit runs(adjacent, on same scan line) of pixels

**Conclusion:** Thus we have studied inscribed and Circumscribed circles in the triangle.

## Assignment No: 08

### **Title/ Problem Statement:**

Write C++/Java program to draw a concave polygon and fill it with desired pattern using scan line algorithm.

### **Prerequisites:**

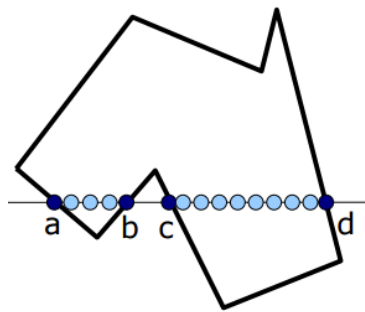
### **Objectives:**

1. To understand Scan Line drawing algorithms used for polygon filling computer graphics.
2. To understand concept of different line styles using line algorithms for filling polygon with desired pattern.

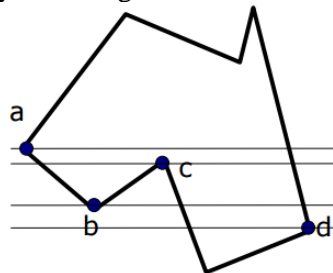
### **Theory :**

#### **Scan-line Polygon Fill**

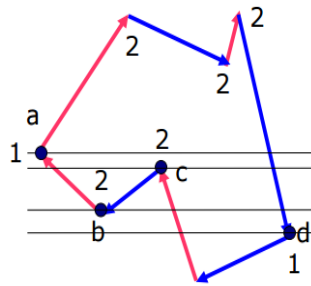
- For each scan-line:
  - Locate the intersection of the scan line with the edges ( $y=y_s$ )
  - Sort the intersection points from left to right.
  - Draw the interiors intersection points pairwise. (a-b), (c-d)
- Problem with corners. Same point counted twice or not?



- a , b , c and d are intersected by 2 line segments each.



- Count b,c twice but a and d once. Why?

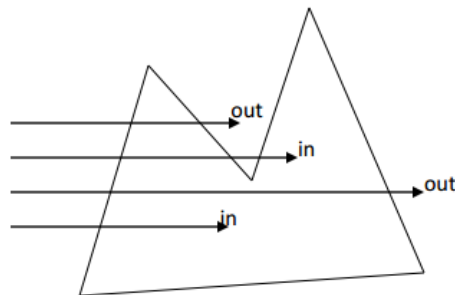


Solution: Make a clockwise or counter-clockwise traversal on edges. Check if y is monotonically increasing or decreasing. If direction changes, double intersection, otherwise single intersection.

### Basic algorithm:

- Assume scan line start from the left and is outside the polygon.
- When intersect an edge of polygon, start to color each pixel (because now we're inside the polygon), when intersect another edge, stop coloring ...
- Odd number of edges: inside
- Even number of edges: outside

- **Advantage of scan-line fill:** It does fill in the same order as rendering, and so can be pipelined.



### Scan-Line Polygon Fill Algorithm

#### • Odd-parity rule

Calculate span extrema (intersections) on each scan line

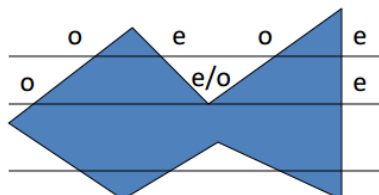
Using parity rule to determine whether or not a point is inside a region

Parity is initially even

◊each intersection encountered thus inverts the parity bit

parity is odd -> interior point (draw)

parity is even-> exterior point (do not draw)



# setlinestyle

## Syntax

```
#include <graphics.h>
```

```
void setlinestyle(int linestyle, unsigned upattern, int thickness);
```

## Description

setlinestyle sets the style for all lines drawn by line, lineto, rectangle, drawpoly, and so on. The linesettingstype structure is defined in graphics.h as follows:

```
struct linesettingstype
{
    int linestyle;

    unsigned upattern;

    int thickness;
};
```

linestyle specifies in which of several styles subsequent lines will be drawn (such as solid, dotted, centered, dashed). The enumeration line\_styles, which is defined in graphics.h, gives names to these operators:

Name	Value	Description
SOLID_LINE	0	Solid line
DOTTED_LINE	1	Dotted line
CENTER_LINE	2	Centered line
DASHED_LINE	3	Dashed line
USERBIT_LINE	4	User-defined line style

thickness specifies whether the width of subsequent lines drawn will be normal or thick.

Name	Value	Description
NORM_WIDTH	1	1 pixel wide
THICK_WIDTH	3	3 pixels wide

upattern is a 16-bit pattern that applies only if linestyle is USERBIT\_LINE (4). In that case, whenever a bit in the pattern word is 1, the corresponding pixel in the line is drawn in the current drawing color. For example, a solid line corresponds to a upattern of 0xFFFF (all pixels drawn), and a dashed line can correspond to a upattern of 0x3333 or 0x0F0F. If the linestyle parameter to setlinestyle is not USERBIT\_LINE (in other words, if it is not equal to 4), you must still provide the upattern parameter, but it will be ignored.

**Note:** The linestyle parameter does not affect arcs, circles, ellipses, or pie slices. Only the thickness parameter is used.

## Return Value

If invalid input is passed to setlinestyle, graphresult returns -11, and the current line style remains unchanged.

**Output:**

Polygon is filled with pattern.

**Conclusion:** Thus we have studied C++/Java program to draw a concave polygon and fill it with desired pattern using scan line algorithm.

## Experiment No: 11

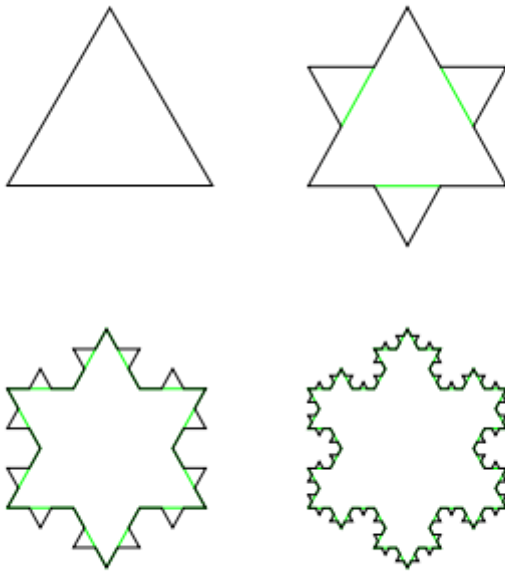
**Title:** Write C++/Java program to generate fractal patterns by using Koch curves.

**Software Required:** Ubuntu, GCC

### Theory:

The Koch curve (also known as the Koch star, or Koch island) is a mathematical curve and one of the earliest fractal curves to have been described. It is based on the Koch curve, which appeared in a 1904 paper titled "On a continuous curve without tangents, constructible from elementary geometry".

The progression for the area of the snowflake converges to  $\frac{8}{5}$  times the area of the original triangle, while the progression for the snowflake's perimeter diverges to infinity. Consequently, the snowflake has a finite area bounded by an infinitely long line.



The Koch snowflake can be constructed by starting with an equilateral triangle, then recursively altering each line segment as follows:

1. divide the line segment into three segments of equal length.
2. draw an equilateral triangle that has the middle segment from step 1 as its base and points outward.
3. remove the line segment that is the base of the triangle from step 2.

After one iteration of this process, the resulting shape is the outline of a hexagram. The Koch snowflake is the limit approached as the above steps are followed over and over again. The Koch curve originally described by Helge von Koch is constructed with only one of the three sides of the original triangle. In other words, three Koch curves make a Koch snowflake.

### Algorithm:

```
#include<graphics.h>
#include<iostream>
#include<math.h>

using namespace std;

void koch(int x1, int y1, int x2, int y2, int it)
{
    float angle = 60/180;
    int x3 = (2*x1+x2)/3;
    int y3 = (2*y1+y2)/3;

    int x4 = (x1+2*x2)/3;
    int y4 = (y1+2*y2)/3;

    int x = x3 + (x4-x3)*cos(angle)+(y4-y3)*sin(angle);
    int y = y3 - (x4-x3)*sin(angle)+(y4-y3)*cos(angle);

    if(it > 0)
    {
        koch(x1, y1, x3, y3, it-1);
        koch(x3, y3, x, y, it-1);
        koch(x, y, x4, y4, it-1);
        koch(x4, y4, x2, y2, it-1);
    }
    else
    {
        line(x1, y1, x3, y3);
    }
}
```



```

        line(x3, y3, x, y);
        line(x, y, x4, y4);
        line(x4, y4, x2, y2);
        // delay(50);
    }
}

int main(void)
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    int x1,x2,y1,y2,x3,y3,it;

    cout<<"Enter coordinates of initial line(x1,y1,x2,y2,x3,y3):";
    cin>>x1>>y1>>x2>>y2>>x3>>y3;

    //int x1 = 100, y1 = 100, x2 = 400, y2 = 400;

    cout<<"Enter number of iterations:";
    cin>>it;

    //it=1,2,3,4,....

    koch(x1, y1, x2, y2, it);
    koch(x2, y2, x3, y3, it);
    koch(x3, y3, x1, y1, it);

    delay(500000);
    closegraph();
    return 0;
}

```

**Conclusion:** We implemented fractal patterns by using Koch curves.

## Questions:

1. What is the meaning of fractal?
2. What are the different types of fractals?
3. What is the logic to generate Koch curve?
4. What is cups and joins?
5. List out the examples of fractals.

6. Why fractals are very important in computer graphics?
7. What is the difference between fractals and curves?
8. What are the different types of curves?
9. What is Bezier curve?
10. What are the different applications of fractals?

## **Assignment: 12**

**Title :** Animation : Implement any one of the following animation assignments,

- i) Clock with pendulum
- ii) National Flag hoisting
- iii) Vehicle/boat locomotion
- iv) Falling Water drop into the water and generated waves after impact
- v) Kaleidoscope views generation (at least 3 colorful patterns)

**Aim:**

To implement Animation sequences

**Objective:**

To understand Animation Assignment

**Theory:**

Animation means giving life to any object in computer graphics. It has the power of injecting energy and emotions into the most seemingly inanimate objects. Computer-assisted animation and computer-generated animation are two categories of computer animation. It can be presented via film or video.

The basic idea behind animation is to play back the recorded images at the rates fast enough to fool the human eye into interpreting them as continuous motion. Animation can make a series of dead images come alive. Animation can be used in many areas like entertainment, computer aided-design, scientific visualization, training, education, e-commerce, and computer art.

### Animation Techniques

Animators have invented and used a variety of different animation techniques. Basically there are six animation techniques which we would discuss one by one in this section.

## **Traditional Animation (frame by frame)**

Traditionally most of the animation was done by hand. All the frames in an animation had to be drawn by hand. Since each second of animation requires 24 frames (film), the amount of efforts required to create even the shortest of movies can be tremendous.

## **Key framing**

In this technique, a storyboard is laid out and then the artists draw the major frames of the animation. Major frames are the ones in which prominent changes take place. They are the key points of animation. Keyframing requires that the animator specifies critical or key positions for the objects. The computer then automatically fills in the missing frames by smoothly interpolating between those positions.

## **Procedural**

In a procedural animation, the objects are animated by a procedure – a set of rules – not by keyframing. The animator specifies rules and initial conditions and runs simulation. Rules are often based on physical rules of the real world expressed by mathematical equations.

## **Behavioral**

In behavioral animation, an autonomous character determines its own actions, at least to a certain extent. This gives the character some ability to improvise, and frees the animator from the need to specify each detail of every character's motion.

## **Performance Based (Motion Capture)**

Another technique is Motion Capture, in which magnetic or vision-based sensors record the actions of a human or animal object in three dimensions. A computer then uses these data to animate the object.

This technology has enabled a number of famous athletes to supply the actions for characters in sports video games. Motion capture is pretty popular with the animators mainly because some of

the commonplace human actions can be captured with relative ease. However, there can be serious discrepancies between the shapes or dimensions of the subject and the graphical character and this may lead to problems of exact execution.

### **Physically Based (Dynamics)**

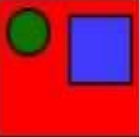

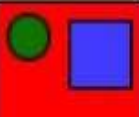
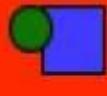
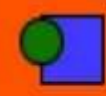





Unlike key framing and motion picture, simulation uses the laws of physics to generate motion of pictures and other objects. Simulations can be easily used to produce slightly different sequences while maintaining physical realism. Secondly, real-time simulations allow a higher degree of interactivity where the real person can maneuver the actions of the simulated character.

In contrast the applications based on key-framing and motion select and modify motions from a pre-computed library of motions. One drawback that simulation suffers from is the expertise and time required to handcraft the appropriate controls systems.

### **Key Framing**

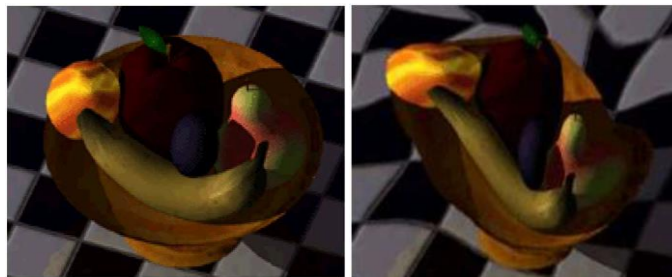
A key frame is a frame where we define changes in animation. Every frame is a keyframe when we create frame by frame animation. When someone creates a 3D animation on a computer, they usually don't specify the exact position of any given object on every single frame. They create keyframes.

Key frames are important frames during which an object changes its size, direction, shape or other properties. The computer then figures out all the in-between frames and saves an extreme amount of time for the animator. The following illustrations depict the frames drawn by user and the frames generated by computer.

1	2	3	4	5	6	7	8
							
1	2	3	4	5	6	7	8
							

## ***Morphing***

The transformation of object shapes from one form to another form is called morphing. It is one of the most complicated transformations.



A morph looks as if two images melt into each other with a very fluid motion. In technical terms, two images are distorted and a fade occurs between them.

**Conclusion :** Implement the following animation assignment.

Questions:

1. Which company publish MAYA 3D ?
2. What is the workspace called in here ?
3. What is Navier Stokes Equation?

4. What is the Full form of NURBS?
5. Difference Between Bump map and Normal Map?
6. What is Motion capture?
7. What is the first Computer Animated character?
8. What is the first fully computer generated film?
9. What is the meaning of animation sequences?
10. Define Miniature Effect.