Experiment No 1.

Aim: To implement DDA algorithms for drawing a line segment between two given end points.

Objective: Draw the line using (vector) generation algorithms which determine the pixels that should be turned ON are called as digital differential analyzer (DDA).It is one of the techniques for obtaining a rasterized straight line. This algorithm can be used to draw the line in all the quadrants.

Theory:
DDA algorithm is an incremental scan conversion method. Here we perform calculations at each step using the results from the preceding step. The characteristic of the DDA algorithm is to take unit steps along one coordinate and compute the corresponding values along the other coordinate. Digital Differential Analyzer (DDA) algorithm is the simple line generation algorithm which is explained step by step here.

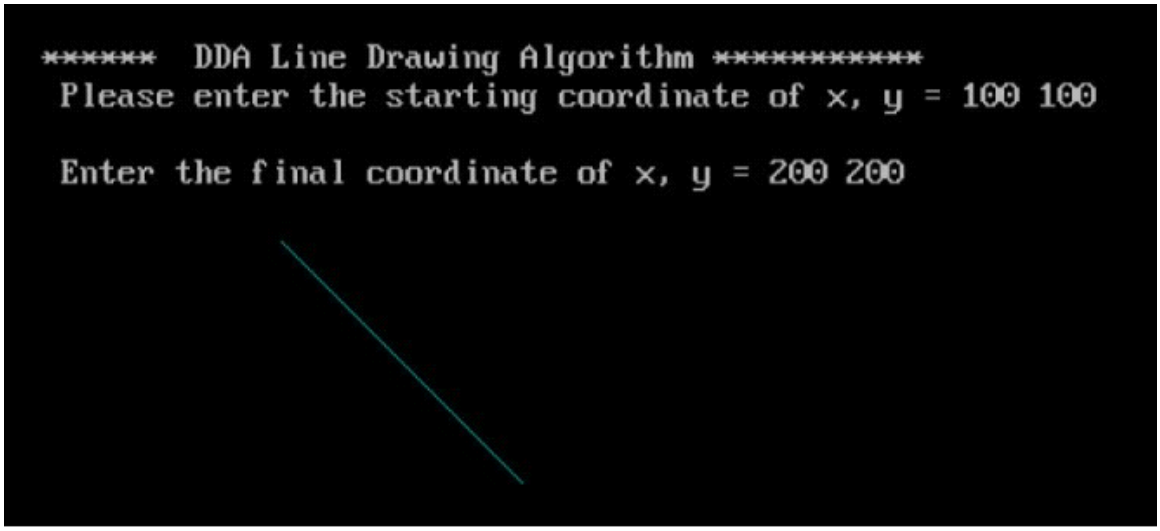
Algorithm:
Input the two endpoints of the line segment, (x1,y1) and (x2,y2).
Calculate the difference between the x-coordinates and y-coordinates of the endpoints as dx and dy respectively.
Calculate the slope of the line as m = dy/dx.
Set the initial point of the line as (x1,y1).
Loop through the x-coordinates of the line, incrementing by one each time, and calculate the corresponding y-coordinate using the equation y = y1 + m(x – x1).
Plot the pixel at the calculated (x,y) coordinate.
Repeat steps 5 and 6 until the endpoint (x2,y2) is reached.

Program:

```
#include<graphics.h>
#include<math.h>
#include<conio.h>
void main()
{
int x0,y0,x1,y1,i=0;
float delx,dely,len,x,y;
int gr=DETECT,gm;
initgraph(&gr,&gm,"C:\\TURBOC3\\BGI");
printf("\n****** DDA Line Drawing Algorithm ***********");
printf("\n Please enter the starting coordinate of x, y = ");
scanf("%d %d",&x0,&y0);
printf("\n Enter the final coordinate of x, y = ");
scanf("%d %d",&x1,&y1);
dely=abs(y1-y0);
```

```
delx=abs(x1-x0);

if(delx<dely)
{
len = dely;
}
else
{
len=delx;
}
delx=(x1-x0)/len;
dely=(y1-y0)/len;
x=x0+0.5;
y=y0+0.5;
do{
putpixel(x,y,3);
x=x+delx;
y=y+dely;
i++;
delay(30);
}while(i<=len);
getch();
closegraph();
}
```

Output:



Conclusion: Comment on -

1. Pixel

2. Equation for line
3. Need of line drawing algorithm
4. Slow or fast

Pixel: In Digital Differential Analyzer (DDA) algorithm, a "pixel" refers to the smallest controllable element on a digital display or raster image. The DDA algorithm is used for generating points on a line between two given points by calculating the intermediate pixel positions.

Equation for line: The DDA algorithm essentially utilizes this line equation and incremental calculations to determine the pixel positions along the line and generates the points to draw the line on a digital display. The equation used is typically the slope-intercept form of a line, which is y = mx + c, where 'm' is the slope of the line and 'c' is the y-intercept.

Need for line drawing algorithm: Line drawing algorithms are fundamental in a wide array of applications, serving as the building blocks for visual representation, aiding in geometric modeling, supporting image processing, and playing a crucial role in both creative and technical domains. Without such algorithms, it would be challenging to accurately represent lines and shapes in computer graphics and various applications that involve graphical representations.

Slow or fast: The DDA algorithm operates by calculating and rounding off incremental values to determine the next pixel's position, which is why it is a little bit slow. Additionally, the DDA algorithm might suffer from precision issues, especially when dealing with lines of high slope or lines that are close to vertical, leading to discrepancies in pixel placement and potentially requiring more iterations to plot the line accurately.

Experiment No. 2

Aim: To implement Bresenham's algorithms for drawing a line segment between two given end points.

Objective:
Draw a line using Bresenham's line algorithm that determines the points of an n-dimensional raster that should be selected to form a close approximation to a straight line between two points

Theory:
In Bresenham's line algorithm pixel positions along the line path are obtained by determining the pixels i.e. nearer the line path at each step.

Algorithm –

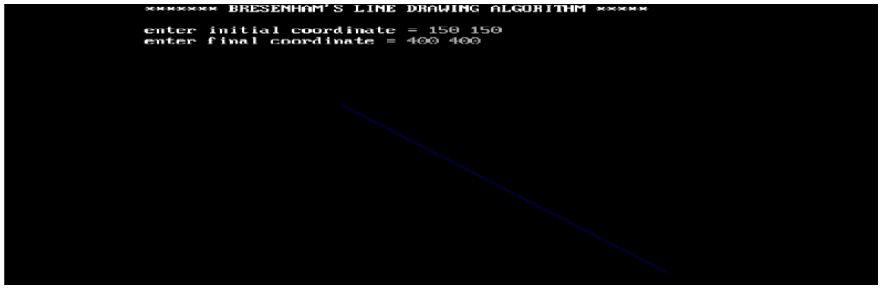
Step1: Start Algorithm
Step2: Declare variable $x_1,x_2,y_1,y_2,d,i_1,i_2,dx,dy$
Step3: Enter value of x_1,y_1,x_2,y_2
Where $x_1,y_1$are coordinates of starting point
And x_2,y_2 are coordinates of Ending point
Step4: Calculate $dx = x_2-x_1$
Calculate $dy = y_2-y_1$
Calculate $i_1=2*dy$
Calculate $i_2=2*(dy-dx)$
Calculate $d=i_1-dx$
Step5: Consider (x, y) as starting point and x_{end} as maximum possible value of x.
If $dx < 0$
Then $x = x_2$
$y = y_2$
$x_{end} = x_1$
If $dx > 0$
Then $x = x_1$
$y = y_1$
$x_{end} = x_2$
Step6: Generate point at (x,y)coordinates.
Step7: Check if whole line is generated.
If $x > = x_{end}$
Stop.
Step8: Calculate co-ordinates of the next pixel
If $d < 0$
Then $d = d + i_1$
If $d \geq 0$
Then $d = d + i_2$
Increment $y = y + 1$

Step9: Increment x = x + 1
Step10: Draw a point of latest (x, y) coordinates
Step11: Go to step 7
Step12: `End of Algorithm`

Program –

```
#include<graphics.h>
#include<stdio.h>
void main()
{
int x,y,x1,y1,delx,dely,m,grtr_d,smlr_d,d;
int gm,gd=DETECT;
initgraph(&gd,&gm,"C:\\TC\\BGI");
printf("******* BRESENHAM'S LINE DRAWING ALGORITHM *****\n\n");
printf("enter initial coordinate = ");
scanf("%d %d",&x,&y);
printf("enter final coordinate = ");
scanf("%d %d",&x1,&y1);
delx=x1-x;
dely=y1-y;
grtr_d=2*dely-2*delx;   // when d > 0
smlr_d=2*dely;          // when d< 0
d=(2*dely)-delx;

do{
putpixel(x,y,1);
if(d<0) {
d=smlr_d+d;
}
else
{
d=grtr_d+d;
y=y+1;
}
x=x+1;
}while(x<x1);
getch();
}
```

OUTPUT:



Conclusion: Comment on -

1. Pixel
2. Equation for line
3. Need of line drawing algorithm
4. Slow or fast

Pixel: The Bresenham's algorithm is utilized to determine the most appropriate pixels to render a line between two given points in a manner that minimizes computational effort. It refers to the smallest controllable element on a digital display or raster image. In the Bresenham line drawing algorithm, a pixel is represented by its coordinates (x, y) on a two-dimensional plane.

Equation for line: The equation for a line in Bresenham's algorithm doesn't rely on a classic y = mx + c form as in the DDA (Digital Differential Analyzer) algorithm. Instead, the algorithm utilizes incremental calculations to decide which pixels to plot in order to create a line between two specified points (x1, y1) and (x2, y2). The algorithm tracks the error at each step and selects the pixel that best fits the line, making efficient use of integer calculations.

Need for line drawing algorithm: Bresenham's line drawing algorithm is crucial for various applications that require efficient, accurate, and optimized rendering of lines on digital displays, making it a fundamental component of many graphical and computational systems.

Slow or fast: Bresenham's line drawing algorithm is generally considered fast and efficient in comparison to other line drawing algorithms, especially when compared to the Digital Differential Analyzer (DDA) algorithm. The algorithm's ability to determine the next best pixel based on the incremental error calculation further contributes to its speed. It chooses the pixels that most closely approximate the line path, minimizing error and ensuring an efficient rendering process.

Experiment No. 3

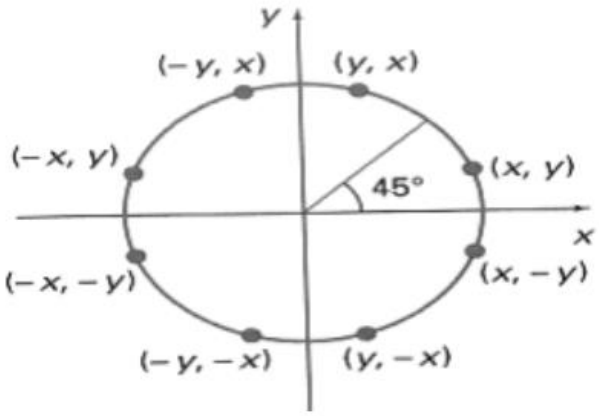
Aim: To implement midpoint circle algorithm.

Objective:

Draw a circle using mid-point circle drawing algorithm by determining the points needed for rasterizing a circle. The mid-point algorithm to calculate all the perimeter points of the circle in the first octant and then print them along with their mirror points in the other octants.

Theory:

The shape of the circle is similar in each quadrant. We can generate the points in one section and the points in other sections can be obtained by considering the symmetry about x-axis and y-axis.



The equation of circle with center at origin is x 2 + y 2 = r 2

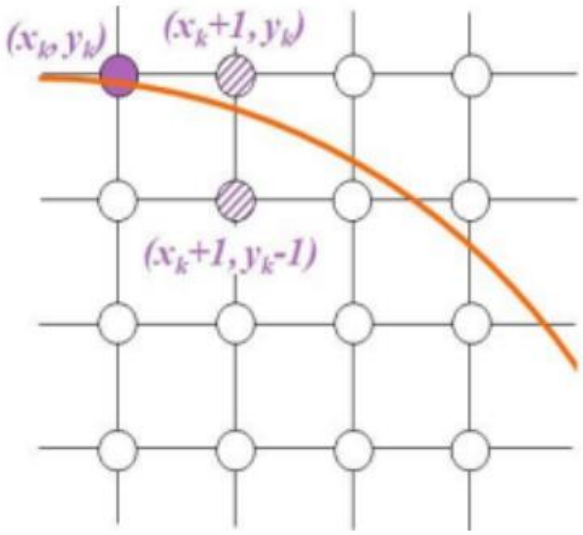
Let the circle function is f circle (x, y) -

 is < 0, if (x, y) is inside circle boundary,

 is = 0, if (x, y) is on circle boundary,

 is > 0, if (x, y) is outside circle boundary.

Consider the pixel at (xk, yk) is plotted,



Now the next pixel along the circumference of the circle will be either (xk + 1, yk) or (xk + 1, yk – 1) whichever is closer the circle boundary.

Let the decision parameter pk is equal to the circle function evaluate at the mid-point between two pixels.

If pk < 0, the midpoint is inside the circle and the pixel at yk is closer to the circle boundary.

Otherwise, the midpoint is outside or on the circle boundary and the pixel at yk – 1 is closer to the circle boundary.

Algorithm –

Step1: Put x =0, y =r in equation 2
We have p=1-r

Step2: Repeat steps while x ≤ y
Plot (x, y)
If (p<0)
Then set p = p + 2x + 3
Else
p = p + 2(x-y)+5
y =y - 1 (end if)
x =x+1 (end loop)

Step3: End

Program –

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

```
void main()
{
int x,y,x_mid,y_mid,radius,dp;
int g_mode,g_driver=DETECT;
clrscr();
initgraph(&g_driver,&g_mode,"C:\\TURBOC3\\BGI");
printf("********** MID POINT Circle drawing algorithm ********\n\n");
printf("\nenter the coordinates= ");
scanf("%d %d",&x_mid,&y_mid);
printf("\n now enter the radius =");
scanf("%d",&radius);
x=0;
y=radius;
dp=1-radius;
do
{
putpixel(x_mid+x,y_mid+y,YELLOW);
putpixel(x_mid+y,y_mid+x,YELLOW);
putpixel(x_mid-y,y_mid+x,YELLOW);
putpixel(x_mid-x,y_mid+y,YELLOW);
putpixel(x_mid-x,y_mid-y,YELLOW);
putpixel(x_mid-y,y_mid-x,YELLOW);
putpixel(x_mid+y,y_mid-x,YELLOW);
putpixel(x_mid+x,y_mid-y,YELLOW);
if(dp<0) {
dp+=(2*x)+1;
}
else{
y=y-1;
dp+=(2*x)-(2*y)+1;
}
x=x+1;
}while(y>x);
getch();
}
```

Output –

Conclusion: Comment on

1. Fast or slow

2. Draw one arc only and repeat the process in 8 quadrants

3. Difference with line drawing method

Fast or slow: The Midpoint Circle Algorithm is fast and particularly efficient when compared to other circle-drawing algorithms due to its simplicity and use of integer calculations. The Midpoint Circle Algorithm provides a relatively quick and efficient way to draw circles, making it suitable for various applications in computer graphics, games, simulations, and other fields where circle rendering is required.

Draw one arc only and repeat the process in 8 quadrants: The Midpoint Circle Algorithm typically draws a circle in one octant (one-eighth of the circle) and reflects it across the x and y axes to complete the circle. However, to draw only an arc in one quadrant and replicate it in all eight quadrants, the process will be slightly modified to draw an arc in the first quadrant.

Difference with line drawing method: The Midpoint Circle Algorithm focuses on plotting points on the circumference of a circle, while line drawing algorithms concentrate on determining pixels to form straight lines between two given points. The Midpoint Circle Algorithm for circles and line drawing algorithms for straight lines. Their underlying

mathematical concepts and computational approaches are distinct due to their respective geometric shapes and applications.

Experiment No. 4

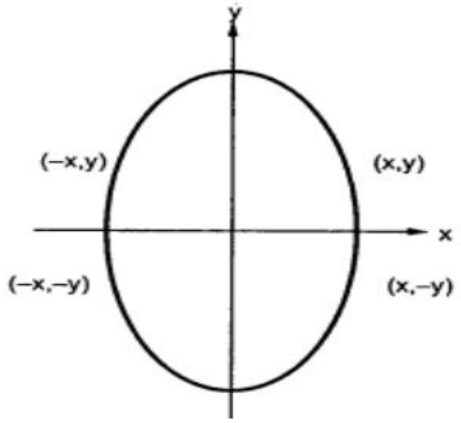
Aim-To implement midpoint Ellipse algorithm

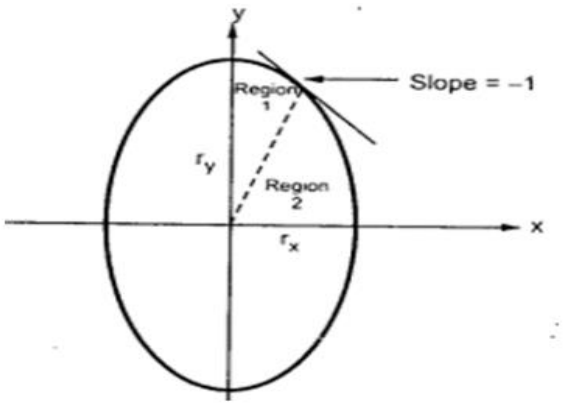
Objective:

Draw the ellipse using Mid-point Ellipse algorithm in computer graphics. Midpoint ellipse algorithm plots (finds) points of an ellipse on the first quadrant by dividing the quadrant into two regions.

Theory:

Midpoint ellipse algorithm uses four way symmetry of the ellipse to generate it. Figure shows the 4-way symmetry of the ellipse.





Here the quadrant of the ellipse is divided into two regions as shown in the fig. Fig. shows the division of first quadrant according to the slope of an ellipse with rx < ry. As ellipse is drawn from 90 0 to 0 0 , x moves in positive direction and y moves in negative direction and ellipse passes through two regions 1 and 2.

The equation of ellipse with center at (xc, yc) is given as -

[(x – xc) / rx] 2 + [(y – yc) / ry] 2 = 1

Therefore, the equation of ellipse with center at origin is given as -

[x / rx] 2 + [y / ry] 2 = 1

i.e. x 2 ry 2 + y 2 rx 2 = rx 2 ry 2

Let, f ellipse (x, y) = x2 ry2 + y2 rx2 - rx2 ry2

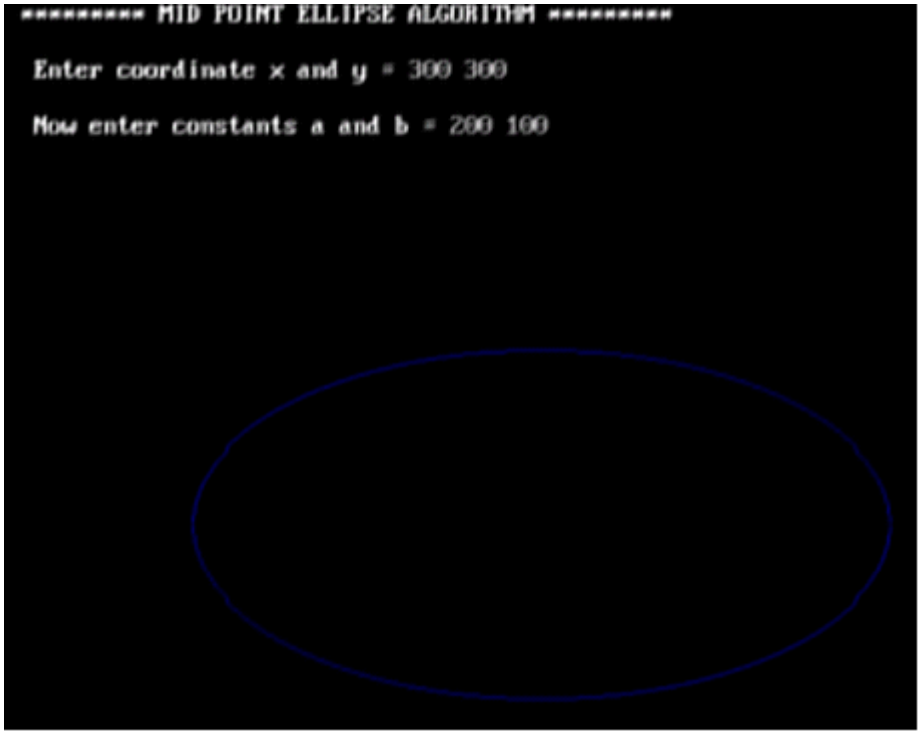
Algorithm:

Program

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void pixel(int x,int y,int xc,int yc)
{
        putpixel(x+xc,y+yc,BLUE);
        putpixel(x+xc,-y+yc,BLUE);
        putpixel(-x+xc,y+yc,BLUE);
        putpixel(-x+xc,-y+yc,BLUE);
        putpixel(y+xc,x+yc,BLUE);
        putpixel(y+xc,-x+yc,BLUE);
        putpixel(-y+xc,x+yc,BLUE);
        putpixel(-y+xc,-x+yc,BLUE);
}
main()
{
        int gd=DETECT,gm=0,r,xc,yc,x,y;
        float p;
        //detectgraph(&gd,&gm);
        initgraph(&gd,&gm," ");
        printf("\n ********* MID POINT ELLIPSE ALGORITHM ********* ")
        printf("\n Enter the radius of the circle:");
        scanf("%d",&r);
        printf("\n Enter the center of the circle:");
        scanf("%d %d",&xc,&yc);
        y=r;
        x=0;
        p=(5/4)-r;
        while(x<y)
        {
                if(p<0)
                {
                        x=x+1;
                        y=y;
                        p=p+2*x+3;
                }
                else
                {
                        x=x+1;
                        y=y-1;
                        p=p+2*x-2*y+5;
                }
                pixel(x,y,xc,yc);
        }
        getch();
```

```
        closegraph();
}
```

Output:



Conclusion: Comment on

1. Slow or fast

2. Difference with circle

3. Importance of object

Speed: The Midpoint Ellipse Algorithm, like its circular counterpart for drawing circles, is also considered a relatively efficient and fast method for drawing ellipses in computer graphics. It uses integer arithmetic and incremental calculations to determine which pixels to plot on the ellipse's perimeter.

Differences with circles: The midpoint ellipse algorithm differs from the midpoint circle algorithm in terms of the decision parameters and the way points are plotted. While both algorithms utilize the concept of symmetry and incremental calculations, the parameters and equations for ellipses are

different from those for circles due to the variation in the geometry of these two shapes. These algorithms use integer arithmetic and incremental calculations to efficiently plot points on the curves they are designed for.

Importance of the object: The importance of an object in Midpoint Ellipse Algorithm often lies in its context and the particular field of study or application being considered. Objects serve as building blocks, subjects, entities, or concepts that contribute to various aspects of our understanding, development, and interaction within different disciplines and industries.

Experiment No. 5

Aim: To implement Area Filling Algorithm: Boundary Fill, Flood Fill.

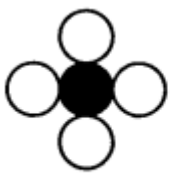
Objective:

Polygon is an ordered list of vertices as shown in the following figure. For filling polygons with particular colors, we need to determine the pixels falling on the border of the polygon and those which fall inside the polygon. Objective is to demonstrate the procedure for filling polygons using different techniques.

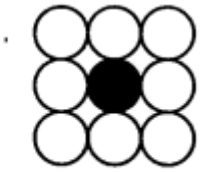
Theory:

1) Boundary Fill algorithm –

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



(a) Four connected region



(b) Eight connected region

Procedure:

```
boundary_fill (x, y, f_color, b_color)
{
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);
        }
}
```

Program:

```
#include<stdio.h>

#include<conio.h>

#include<graphics.h>

#include<doc.h>
```

```
void boundary_fill(int x, int y, int fcolor, int bcolor)

{

if ((getpixel(x, y) != bcolor) && (getpixel(x, y) != fcolor))

{

delay(10);

putpixel(x, y, fcolor);

boundary_fill(x + 1, y, fcolor, bcolor);
boundary_fill(x , y+1, fcolor, bcolor);
boundary_fill(x+1, y + 1, fcolor, bcolor);
boundary_fill(x-1, y - 1, fcolor, bcolor);
boundary_fill(x-1, y, fcolor, bcolor);
boundary_fill(x , y-1, fcolor, bcolor);
boundary_fill(x-1, y + 1, fcolor, bcolor);
boundary_fill(x+1, y - 1, fcolor, bcolor);

}

}

void main()

{

int x, y, fcolor, bcolor;

int gd=DETECT,gm;

initgraph(&gd, &gm, "C:\\TurboC3\\BGI");
printf("Enter the seed point (x,y) : ");
scanf("%d%d", &x, &y);

printf("Enter boundary color : "); scanf("%d", &bcolor); printf("Enter new color : ");

scanf("%d", &fcolor); rectangle(50,50,100,100);
boundary_fill(x,y,fcolor,bcolor); getch();

}
```

Output:

2) Flood Fill algorithm –

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

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

(a) Four connected region (b) Eight connected region

Procedure -

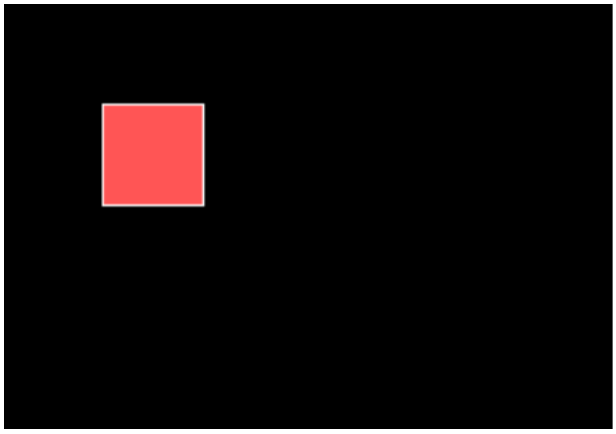
```
flood_fill (x, y, old_color, new_color)
{
if (getpixel (x, y) = old_colour)
        {
        putpixel (x, y, new_colour);
        flood_fill (x + 1, y, old_colour, new_colour);
        flood_fill (x - 1, y, old_colour, new_colour);
```

```
        flood_fill (x, y + 1, old_colour, new_colour);
        flood_fill (x, y - 1, old_colour, new_colour);
        flood_fill (x + 1, y + 1, old_colour, new_colour);
        flood_fill (x - 1, y - 1, old_colour, new_colour);
        flood_fill (x + 1, y - 1, old_colour, new_colour);
        flood_fill (x - 1, y + 1, old_colour, new_colour);
        }
}
```

Program:

```
#include<stdio.h>

#include<graphics.h>
#include<dos.h>

void flood(int,int,int,int);
int main()

{

int gd,gm=DETECT;
//detectgraph(&gd,&gm)
; initgraph(&gd,&gm," ");
rectangle(50,50,100,100)
; flood(55,55,12,0);
closegraph(); return 0;

} void flood(int x,int y, int fill_col, int old_col)

{

if(getpixel(x,y)==old_col)

{ delay(10); putpixel(x,y,fill_col);
flood(x+1,y,fill_col,old_col); flood(x-
1,y,fill_col,old_col);
flood(x,y+1,fill_col,old_col);
flood(x,y-1,fill_col,old_col); flood(x
+ 1, y + 1, fill_col, old_col); flood(x -
1, y - 1, fill_col, old_col); flood(x + 1,
y - 1, fill_col, old_col); flood(x - 1, y
+ 1, fill_col, old_col);

}

}
```

Output:



Conclusion: Comment on

1. Importance of Flood fill
2. Limitation of methods
3. Usefulness of method

Importance of Flood fill: The importance of flood fill lies in its versatility and efficiency in handling various image-based tasks and graphical operations. It provides a method for efficiently filling enclosed regions, making it a valuable tool in numerous fields, from computer graphics and image processing to user interface design and game development.

Limitation of methods: In Area Filling Method to mitigate these limitations, improvements and adaptations have been made in the algorithms, and new techniques have been developed to address specific challenges, such as handling complex shapes, non-enclosed areas, and performance optimizations in graphics software and systems.

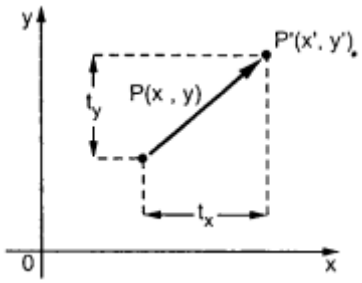
Usefulness of method: The usefulness of Area Filling methods lies in their versatility and efficiency in handling area filling tasks. They provide fundamental tools in various domains, from graphic design and image processing to user interaction in software applications, making them an integral part of computer graphics and digital imaging techniques.

Experiment No. 6

Aim: To implement 2D Transformations: Translation, Scaling, Rotation.

Objective:
To understand the concept of transformation, identify the process of transformation and application of these methods to different object and noting the difference between these transformations.

Theory:
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. tx denotes translation distance along x-axis and ty denotes translation distance along 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 , where

$$P = \begin{bmatrix} x \\ y \end{bmatrix} \quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \quad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

Program:

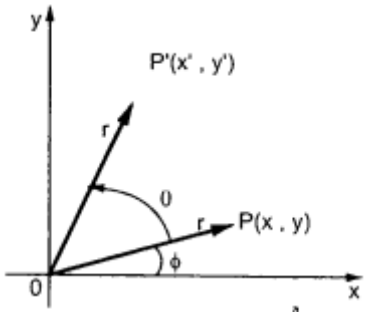
```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
void main()
{
int gd=DETECT,gm;
int x1,y1,x2,y2,tx,ty,x3,y3,x4,y4;
initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
printf("Enter the starting point of line segment:");
```

```
scanf("%d %d",&x1,&y1);
printf("Enter the ending point of line segment:");
scanf("%d %d",&x2,&y2);
printf("Enter translation distances tx,ty:\n");
scanf("%d%d",&tx,&ty);
setcolor(5);
line(x1,y1,x2,y2);
outtextxy(x2+2,y2+2,"Original line");
x3=x1+tx;
y3=y1+ty;
x4=x2+tx;
y4=y2+ty;
setcolor(7);
line(x3,y3,x4,y4);
outtextxy(x4+2,y4+2,"Line after translation");
getch();
}
```

Output –

2) 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.



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

The above equations can be represented in the matrix form as given below

$$[x' \; y'] = [x \; y]\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

$$P' = P \cdot R$$

where R is the rotation matrix and it is given as

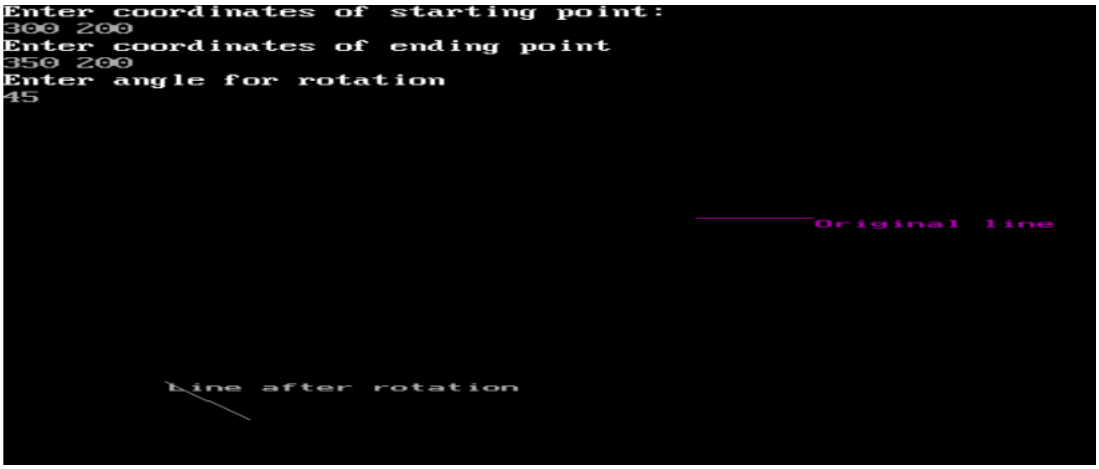
$$R = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

Program:

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
void main()
{
int gd=DETECT,gm;
float x1,y1,x2,y2,x3,y3,x4,y4,a,t;
initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
printf("Enter coordinates of starting point:\n");
scanf("%f%f",&x1,&y1);
printf("Enter coordinates of ending point\n");
scanf("%f%f",&x2,&y2);
printf("Enter angle for rotation\n");
scanf("%f",&a);
setcolor(5);
line(x1,y1,x2,y2);
outtextxy(x2+2,y2+2,"Original line");
t=a*(3.14/180);
x3=(x1*cos(t))-(y1*sin(t));
y3=(x1*sin(t))+(y1*cos(t));
x4=(x2*cos(t))-(y2*sin(t));
y4=(x2*sin(t))+(y2*cos(t));
setcolor(7);
line(x3,y3,x4,y4);
```

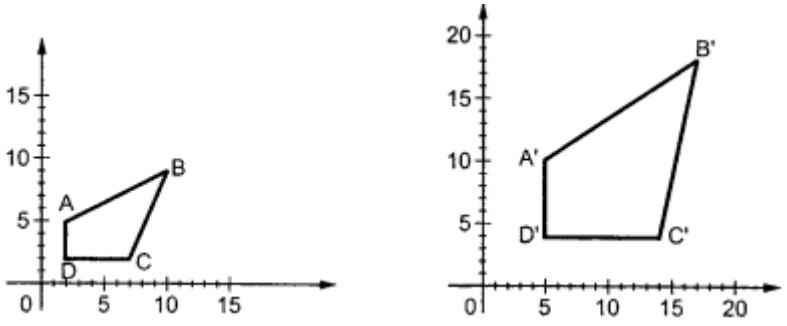
```
outtextxy(x3+2,y3+2,"Line after rotation");
getch();
```

Output:



3) 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:

$$[x' \; y'] = [x \; y] \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$

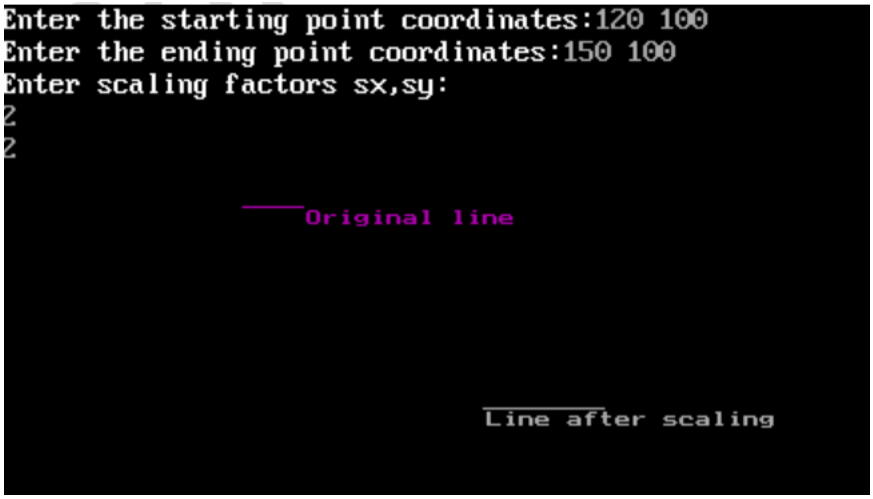
$$= [x \cdot S_x \quad y \cdot Sy]$$

$$= P \cdot S$$

Program:

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
void main()
{
int gd=DETECT,gm;
float x1,y1,x2,y2,sx,sy,x3,y3,x4,y4;
initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
printf("Enter the starting point coordinates:");
scanf("%f %f",&x1,&y1);
printf("Enter the ending point coordinates:");
scanf("%f %f",&x2,&y2);
printf("Enter scaling factors sx,sy:\n");
scanf("%f%f",&sx,&sy);
setcolor(5);
line(x1,y1,x2,y2);
outtextxy(x2+2,y2+2,"Original line");
x3=x1*sx;
y3=y1*sy;
x4=x2*sx;
y4=y2*sy;
setcolor(7);
line(x3,y3,x4,y4);
outtextxy(x3+2,y3+2,"Line after scaling");
getch();
}
```

OUTPUT:



Conclusion: Comment on :

1. Application of transformation
2. Difference noted between methods
3. Application on different object

Application of Transformation: In 2D Transformations, transformations are fundamental operations applied to shapes, objects, or images to alter their position, orientation, size, or appearance. It finds applications in various domains, facilitating the manipulation, positioning, and alteration of shapes, objects, images, and elements within a 2D space for diverse purposes ranging from design and animation to engineering and image processing.

Difference Noted between Translation, Rotation, Scaling: Translation moves objects without changing their size or orientation, rotation changes the orientation of an object around a fixed point, and scaling modifies the size of an object while retaining its shape and orientation. These transformations serve different purposes in manipulating objects within computer graphics applications, enabling various visual effects, animations, and positional changes in designs and interfaces.

Application of Translation, Rotation, Scaling on Different Objects: The application of translation, rotation, and scaling on different objects is crucial in computer graphics for various purposes, including design, animation, and image manipulation. In each case, these transformations (translation, rotation, and scaling) serve specific purposes on different objects, allowing for movement, repositioning, orientation changes, size adjustments, and various visual effects, catering to the requirements of different applications within computer graphics, design, animation, and simulation.

Experiment No. 7

Aim: To implement Line Clipping Algorithm: Liang Barsky

Objective:
To understand the concept of Liang Barsky algorithm to efficiently determine the portion of a line segment that lies within a specified clipping window. This method is particularly effective for lines predominantly inside or outside the window.

Theory:

This Algorithm was developed by Liang and Barsky. It is used for line clipping as it is more efficient because it uses more efficient parametric equations to clip the given line.

These parametric equations are given as:

$x = x1 + tdx$

$y = y1 + tdy, 0 <= t <= 1$

Where $dx = x2 – x1$ & $dy = y2 – y1$

Algorithm

1. Read 2 endpoints of line as p1 (x1, y1) & p2 (x2, y2).

2. Read 2 corners (left-top & right-bottom) of the clipping window as (xwmin, ywmin, xwmax, ywmax).

3. Calculate values of parameters pi and qi for i = 1, 2, 3, 4 such that

$p1 = -dx, q1 = x1 – xwmin$

$p2 = dx, q2 = xwmax – x1$

$p3 = -dy, q3 = y1 – ywmin$

$p4 = dy, q4 = ywmax – y1$

4. if pi = 0 then line is parallel to ith boundary

if qi < 0 then line is completely outside boundary so discard line

else, check whether line is horizontal or vertical and then check the line endpoints with the corresponding boundaries.

5. Initialize t1 & t2 as

t1 = 0 & t2 = 1

6. Calculate values for qi/pi for i = 1, 2, 3, 4.

7. Select values of qi/pi where pi < 0 and assign maximum out of them as t1.

8. Select values of qi/pi where pi > 0 and assign minimum out of them as t2.

9. if (t1 < t2)
{
xx1 = x1 + t1dx

xx2 = x1 + t2dx

yy1 = y1 + t1dy

yy2 = y1 + t2dy

line (xx1, yy1, xx2, yy2)
}

10. Stop.

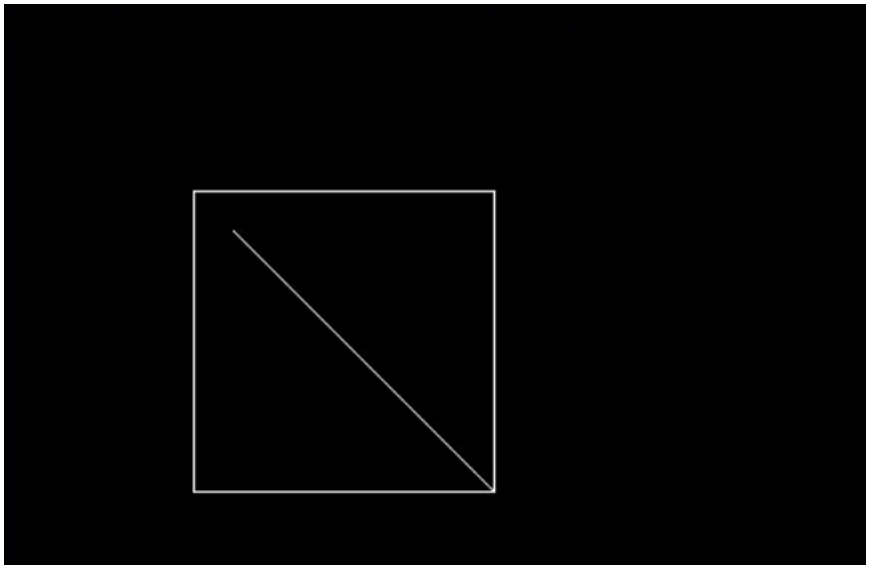
Program:

```
#include<stdio.h>
#include<graphics.h>
#include<math.h>
#include<dos.h>

int main()
{
int i,gd=DETECT,gm;
int x1,y1,x2,y2,xmin,xmax,ymin,ymax,xx1,xx2,yy1,yy2,dx,dy;
```

```
float t1,t2,p[4],q[4],temp;
x1=120;
y1=120;
x2=300;
y2=300;
xmin=100;
ymin=100;
xmax=250;
ymax=250;
initgraph(&gd,&gm," ");
rectangle(xmin,ymin,xmax,ymax);
dx=x2-x1;
dy=y2-y1;
p[0]=-dx;
p[1]=dx;
p[2]=-dy;
p[3]=dy;
q[0]=x1-xmin;
q[1]=xmax-x1;
q[2]=y1-ymin;
q[3]=ymax-y1;
for(i=0;i<4;i++)
{
if(p[i]==0)
{
printf("line is parallel to one of the clipping boundary");
if(q[i]>=0)
{
if(i<2)
{
if(y1<ymin)
{
y1=ymin;
}
if(y2>ymax)
{
y2=ymax;
}
line(x1,y1,x2,y2);
}
if(i>1)
{
if(x1<xmin)
{
x1=xmin;
}
if(x2>xmax)
{
x2=xmax;
```

```
}
line(x1,y1,x2,y2);
}
}
}
}
t1=0;
t2=1;
for(i=0;i<4;i++)
{
temp=q[i]/p[i];
if(p[i]<0)
{
if(t1<=temp)
t1=temp;
}
else
{
if(t2>temp)
t2=temp;
}
}
if(t1<t2)
{
xx1 = x1 + t1 * p[1];
xx2 = x1 + t2 * p[1];
yy1 = y1 + t1 * p[3];
yy2 = y1 + t2 * p[3];
line(xx1,yy1,xx2,yy2);
}
delay(5000);
closegraph();
}
```

Output:



Conclusion:

The Liang-Barsky algorithm, a line-clipping algorithm, is a significant technique used in computer graphics to determine the visible portions of a line segment within a specified window or clipping area. Its primary objective is to efficiently eliminate the portions of a line that are outside the viewing window, enhancing rendering accuracy and performance. It serves as a valuable tool in computer graphics by efficiently determining and rendering only the visible portions of a line segment within a specified window.

Experiment No. 8

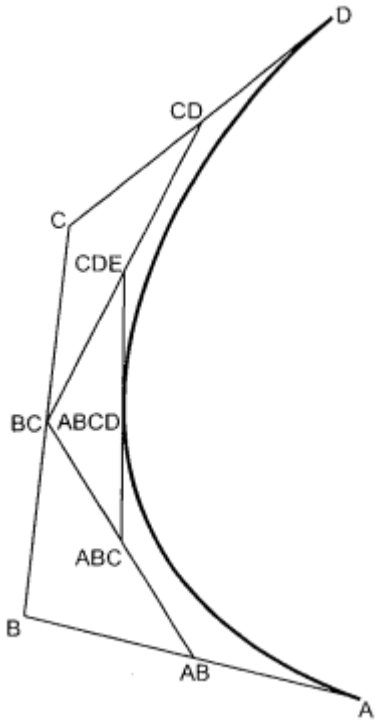
Aim: To implement Bezier curve for n control points. (Midpoint approach)

Objective:

Draw a Bezier curves and surfaces written in Bernstein basis form. The goal of interpolation is to create a smooth curve that passes through an ordered group of points. When used in this fashion, these points are called the control points.

Theory:

In midpoint approach Bezier curve can be constructed simply by taking the midpoints. In this approach midpoints of the line connecting four control points (A, B, C, D) are determined (AB, BC, CD, DA). These midpoints are connected by line segment and their midpoints are ABC and BCD are determined. Finally, these midpoints are connected by line segments and its midpoint ABCD is determined as shown in the figure –



The point ABCD on the Bezier curve divides the original curve in two sections. The original curve gets divided in four different curves. This process can be repeated to split the curve into smaller sections until we have sections so short that they can be replaced by straight lines.

Algorithm:

1) Get four control points say A(xa, ya), B(xb, yb), C(xc, yc), D(xd, yd).

2) Divide the curve represented by points A, B, C, and D in two sections.

xab = (xa + xb) / 2

yab = (ya + yb) / 2

xbc = (xb + xc) / 2

ybc = (yb + yc) / 2

xcd = (xc + xd) / 2

ycd = (yc + yd) / 2

xabc = (xab + xbc) / 2

yabc = (yab + ybc) / 2

xbcd = (xbc + xcd) / 2

ybcd = (ybc + ycd) / 2

xabcd = (xabc + xbcd) / 2

yabcd = (yabc + ybcd) / 2

3) Repeat the step 2 for section A, AB, ABC, ABCD and section ABCD, BCD, CD, D.

4) Repeat step 3 until we have sections so that they can be replaced by straight lines.

5) Repeat small sections by straight lines.

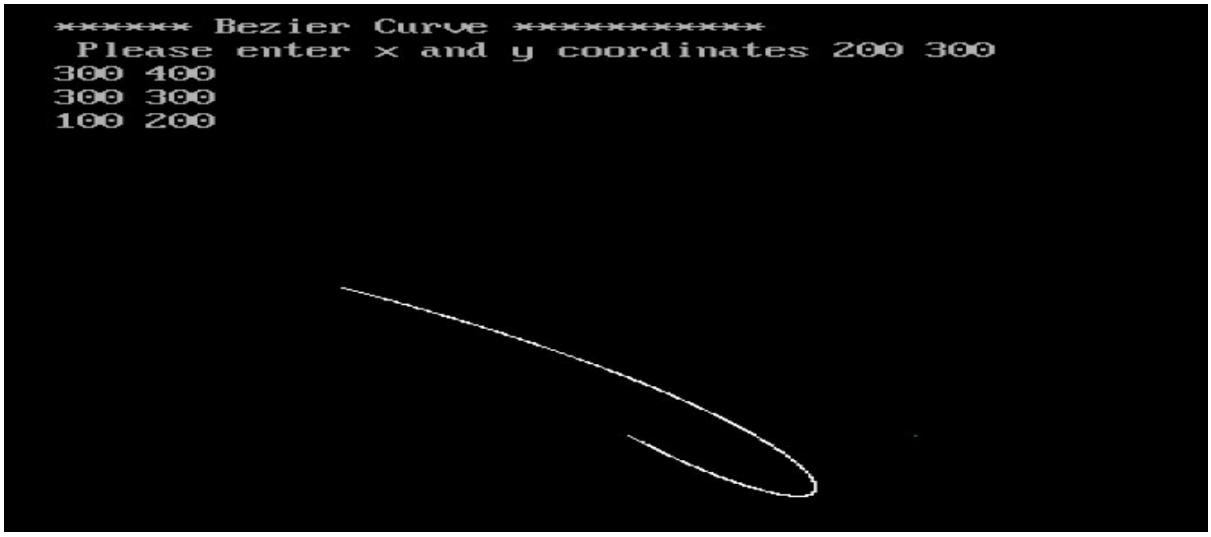
6) Stop.

Program:

```
#include<graphics.h>
#include<math.h>
#include<conio.h>
#include<stdio.h>
void main()
{
int x[4],y[4],i;
double put_x,put_y,t;
int gr=DETECT,gm;
```

```
initgraph(&gr,&gm,"C:\\TURBOC3\\BGI");
printf("\n****** Bezier Curve **********");
printf("\n Please enter x and y coordinates ");
 for(i=0;i<4;i++)
  {
   scanf("%d%d",&x[i],&y[i]);
   putpixel(x[i],y[i],3);
  }

 for(t=0.0;t<=1.0;t=t+0.001)
  {
put_x = pow(1-t,3)*x[0] + 3*t*pow(1-t,2)*x[1] + 3*t*t*(1-t)*x[2] +          pow(t,3)*x[3]; // Formula to draw
curve
put_y =  pow(1-t,3)*y[0] + 3*t*pow(1-t,2)*y[1] + 3*t*t*(1-t)*y[2] +   pow(t,3)*y[3];
putpixel(put_x,put_y, WHITE);
}
 getch();
 closegraph();
}
```

Output:



Conclusion – Comment on

1. Difference from arc and line
2. Importance of control point

3. Applications

Difference from arc and line: In Bezier curves, arcs and lines are distinct mathematical constructs used to represent and define different shapes and paths. The lines are straight paths, arcs represent segments of circles or ellipses, and Bezier curves are flexible parametric curves defined by control points, allowing a wide range of shapes and curves to be represented.

Importance of control points: Control points play a crucial role in defining the shape and characteristics of Bezier curves in computer graphics and design. Control points are essential in Bezier curves as they grant designers and artists control over the curve's shape, smoothness, and trajectory, allowing for versatile and flexible design possibilities in computer graphics, animation, and design applications.

Applications: Bezier curves, due to their versatility and ability to create various shapes and curves, find applications across a wide range of fields in computer graphics, design, engineering, and more. Some of the prominent applications include:

1. Animation and Motion Graphics: In animation software, Bezier curves are used to define motion paths for animated elements, allowing for smooth and controlled movement of objects or characters.
2. Industrial Design: Bezier curves play a significant role in industrial design, helping designers create complex surfaces and shapes for products such as cars, furniture, and consumer goods.
3. 3D Modeling and Rendering: In 3D modeling software, Bezier curves are used to create paths for animation, guide hair and particle systems, and define profiles for shapes and surfaces.

Experiment No. 9

Aim: To implement Character Generation: Bit Map Method

Objective:
Identify the different Methods for Character Generation and generate the character using Stroke

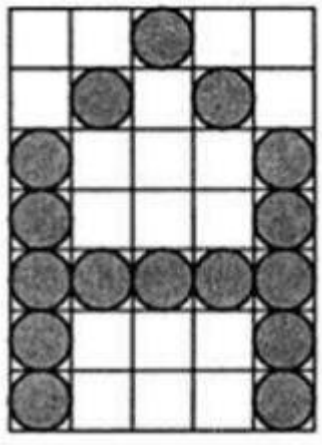
Theory:

Bit map method –

Bitmap method is a called dot-matrix method as the name suggests this method use array of bits for generating a character. These dots are the points for array whose size is fixed.

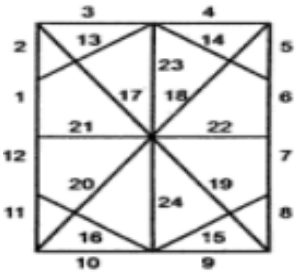
- In bit matrix method when the dots are stored in the form of array the value 1 in array represent the characters i.e. where the dots appear we represent that position with numerical value 1 and the value where dots are not present is represented by 0 in array.
- It is also called dot matrix because in this method characters are represented by an array of dots in the matrix form. It is a two-dimensional array having columns and rows.

A 5x7 array is commonly used to represent characters. However, 7x9 and 9x13 arrays are also used. Higher resolution devices such as inkjet printer or laser printer may use character arrays that are over 100x100.

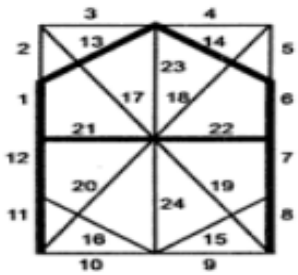


Starburst method –

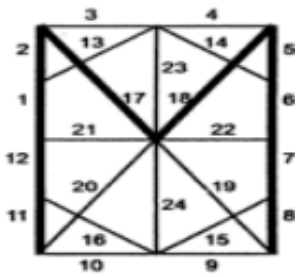
In this method a fix pattern of line segments is used to generate characters. Out of these 24-line segments, segments required to display for particular character are highlighted. This method of character generation is called starburst method because of its characteristic appearance. The starburst patterns for characters A and M. the patterns for particular characters are stored in the form of 24 bit code, each bit representing one line segment. The bit is set to one to highlight the line segment; otherwise, it is set to zero. For example, 24-bit code for Character A is 0011 0000 0011 1100 1110 0001 and for character M is 0000 0011 0000 1100 1111 0011.



a) Star bust pattern of 24 line segments



b) Star bust pattern for character A



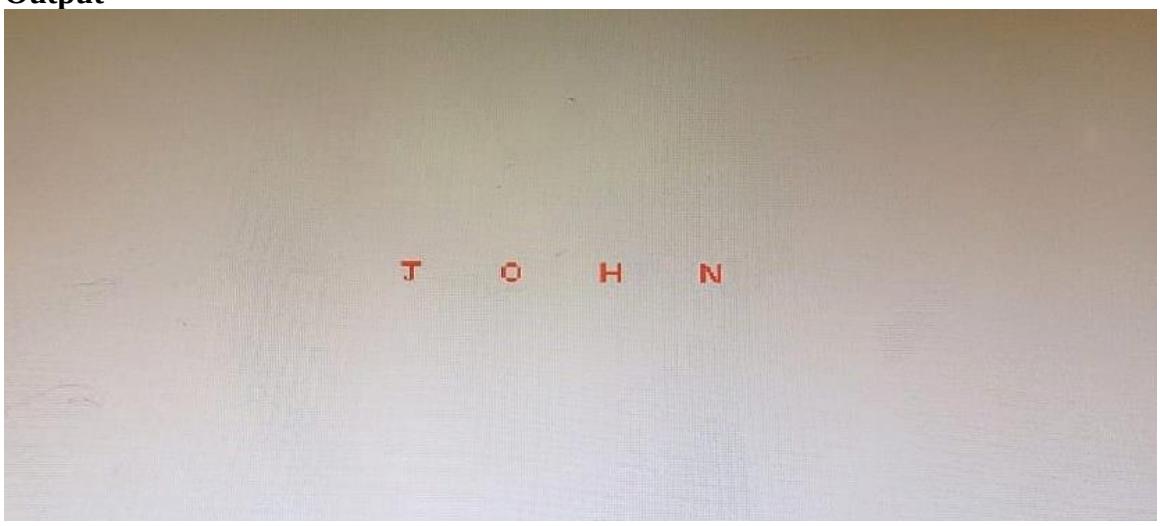
c) Star bust pattern for character M

Program:

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
int main()
{
   int i,j,k,x,y;
   int gd=DETECT,gm;//DETECT is macro defined in graphics.h
   /* ch1 ch2 ch3 ch4 are character arrays that display alphabets */
   int ch1[][10]={ {1,1,1,1,1,1,1,1,1,1},
                   {1,1,1,1,1,1,1,1,1,1},
                   {0,0,0,0,1,1,0,0,0,0},
                   {0,0,0,0,1,1,0,0,0,0},
                   {0,0,0,0,1,1,0,0,0,0},
                   {0,0,0,0,1,1,0,0,0,0},
                   {0,0,0,0,1,1,0,0,0,0},
                   {0,1,1,0,1,1,0,0,0,0},
                   {0,1,1,0,1,1,0,0,0,0},
                   {0,0,1,1,1,0,0,0,0,0}};
   int ch2[][10]={ {0,0,0,1,1,1,1,0,0,0},
                   {0,0,1,1,1,1,1,1,0,0},
                   {1,1,0,0,0,0,0,0,1,1},
                   {1,1,0,0,0,0,0,0,1,1},
                   {1,1,0,0,0,0,0,0,1,1},
                   {1,1,0,0,0,0,0,0,1,1},
                   {1,1,0,0,0,0,0,0,1,1},
                   {1,1,0,0,0,0,0,0,1,1},
                   {0,0,1,1,1,1,1,1,0,0},
                   {0,0,0,1,1,1,1,0,0,0}};
   int ch3[][10]={ {1,1,0,0,0,0,0,0,1,1},
                   {1,1,0,0,0,0,0,0,1,1},
                   {1,1,0,0,0,0,0,0,1,1},
                   {1,1,0,0,0,0,0,0,1,1},
                   {1,1,1,1,1,1,1,1,1,1},
                   {1,1,1,1,1,1,1,1,1,1},
                   {1,1,0,0,0,0,0,0,1,1},
                   {1,1,0,0,0,0,0,0,1,1},
                   {1,1,0,0,0,0,0,0,1,1},
                   {1,1,0,0,0,0,0,0,1,1}};
```

```
   int ch4[][10]={ {1,1,0,0,0,0,0,0,1,1},
                   {1,1,1,1,0,0,0,0,1,1},
                   {1,1,0,1,1,0,0,0,1,1},
                   {1,1,0,1,1,0,0,0,1,1},
                   {1,1,0,0,1,1,0,0,1,1},
                   {1,1,0,0,1,1,0,0,1,1},
                   {1,1,0,0,0,1,1,0,1,1},
                   {1,1,0,0,0,1,1,0,1,1},
                   {1,1,0,0,0,0,1,1,1,1},
                   {1,1,0,0,0,0,0,0,1,1}};
   initgraph(&gd,&gm,"C:\\TURBOC3\\BGI ");//initialize graphic mode
   setbkcolor(LIGHTGRAY);//set color of background to darkgray
   for(k=0;k<4;k++)
   {
      for(i=0;i<10;i++)
      {
         for(j=0;j<10;j++)
         {
            if(k==0)
            {
               if(ch1[i][j]==1)
               putpixel(j+250,i+230,RED);
            }
            if(k==1)
            {
               if(ch2[i][j]==1)
               putpixel(j+300,i+230,RED);
            }
            if(k==2)
            {
               if(ch3[i][j]==1)
               putpixel(j+350,i+230,RED);
            }
            if(k==3)
            {
               if(ch4[i][j]==1)
               putpixel(j+400,i+230,RED);
            }
         }
         delay(200);
      }
   }
   getch();
   closegraph();
}
```

Output –



Conclusion: Comment on

1. Different methods
2. advantage of stroke method
3. one limitation

Different Methods: Bitmaps in computer graphics can be created using various methods, such as the grid-based pixel mapping, which involves assigning each pixel a specific color value, or the vector-based approach that uses mathematical equations to define shapes and lines. Both methods have their own applications and strengths, depending on the specific requirements of the graphics being created.

Advantage of Stroke Method: The stroke method, within the context of bitmap graphics, refers to the technique used to outline or draw the borders of shapes or text with a specified thickness or style. The stroke method in bitmap graphics offers design flexibility, clarity, visual appeal, and emphasis, contributing to the overall aesthetics and organization of elements within an image. It helps create distinction, highlights specific areas, and allows for creative and stylistic expression within bitmap-based design work.

One Limitation: A limitation of the stroke method in bitmap graphics is its potential susceptibility to scaling issues, especially when dealing with raster or bitmap images. To mitigate these issues, vector-based graphics that use mathematical formulas to define

shapes and strokes offer a more scalable solution. Vector graphics retain their quality and smoothness at any scale since the strokes are calculated mathematically and don't rely on fixed pixels. However, in raster-based graphics, scaling limitations of the stroke method can affect image quality and editability, especially when enlarging images.

Experiment No. 10: Mini Project

Theory:

- For moving any object, we incrementally calculate the object coordinates and redraw the picture to give a feel of animation by using for loop.
- Suppose if we want to move a circle from left to right means, we have to shift the position of circle along x-direction continuously in regular intervals.
- The below programs illustrate the movement of objects by using for loop and also using transformations like rotation, translation etc.
- For windmill rotation, we use 2D rotation concept and formulas.

Program:

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include<malloc.h>
#include<dos.h>
#include <conio.h>
int xasp,yasp,gdriver = VGA, gmode=VGAMED, errorcode;
struct pos
{
 int x;
 int y;
};
struct face
{
 int radius;
struct pos posiϴon;
 int mood;
};
typedef struct face face;
face *face1;
void getposiϴon()
{
prinɳ("Enter X Co-ordinate:");
 scanf("%d",&face1->posiϴon.x);
prinɳ("Enter X Co-ordinate:");
 scanf("%d",&face1->posiϴon.y);
}
void drawface()
{
```

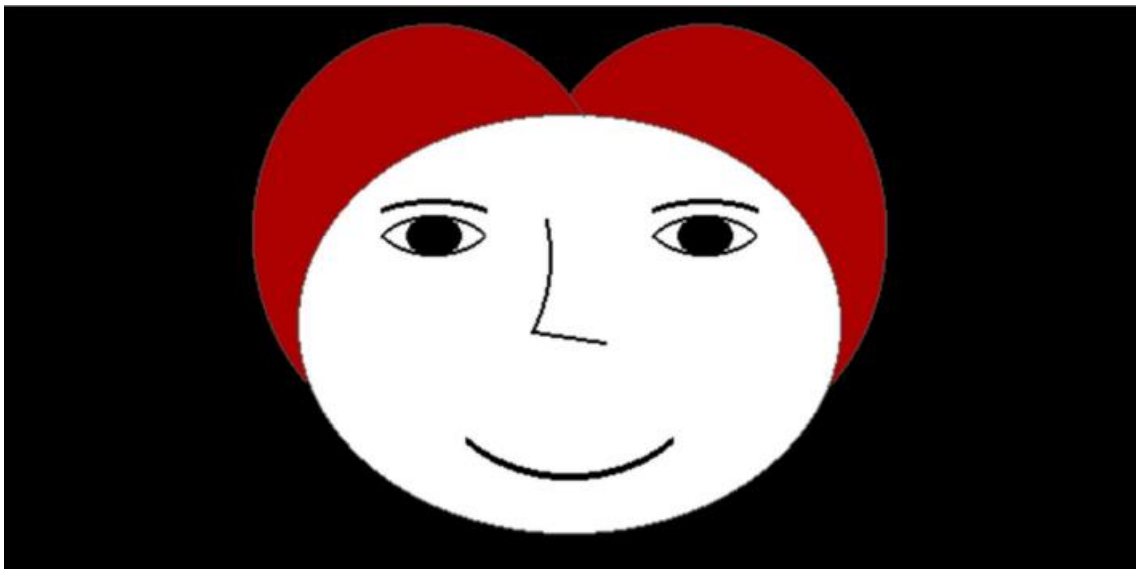
```
 char ch='x';
 int i=0,x,y,color,r,imgsize,dif;
 x=face1->posiϴon.x=320;
 y=face1->posiϴon.y=180;
 face1->radius=150;
 color=15;
 r=face1->radius;
 setbkcolor(0);
getaspectraϴo(&xasp,&yasp);
 setcolor(8);
 circle(x,y,face1->radius);
seɳillstyle(1,color);
 floodfill(x,y,getcolor());
 draweyes(face1);
 drawhair(face1);
 drawmouth(face1);
 drawnose(face1);
}
drawnose()
{
 int i,x,y,r;
 x=face1->posiϴon.x;
 y=face1->posiϴon.y;
 r=face1->radius;
 setcolor(0);
 for(i=0;i<2;i++)
 {
 arc(x-160-i,y-r/4,340-i,10,r);
 line(x-20,y+4+i,x+20,y+10+i);
 }
}
draweyes()
{
 int i,x1,x2,y1,y2,r;
 setcolor(0);
 r=face1->radius;
 x1=face1->posiϴon.x-r/2;
 y1=face1->posiϴon.y-r/4;
 x2=face1->posiϴon.x+r/2;
 y2=face1->posiϴon.y-r/4;
setaspectraϴo(xasp/2,yasp);
```

```
 arc(x1,y1-r/8,40,140,r/4);//leŌ eyebrow
 arc(x1,y1-r/8+1,40,140,r/4);//leŌ eyebrow
 arc(x1,y1-r/8+2,40,140,r/4);//leŌ eyebrow
setaspectraѲo(xasp,yasp);
 for(i=0;i<2;i++)
 {
 arc(x1,y1+i+5,40,140,r/4); //upper leŌ eye
 arc(x1,y1-r/5+i,220,320,r/4); //lower leŌ eye
 }
 circle(x1,y1-r/12,r/10);//leŌ pupul
seɳillstyle(1,0);
 floodfill(x1,y1-r/10,getcolor());
seɳillstyle(1,WHITE);
 floodfill(x1-15,y1-r/6,getcolor());
setaspectraѲo(xasp/2,yasp);
 arc(x2,y2-r/8,40,140,r/4);//right eyebrow
 arc(x2,y2-r/8+1,40,140,r/4);//right eyebrow
 arc(x2,y2-r/8+2,40,140,r/4);//right eyebrow
setaspectraѲo(xasp,yasp);
 for(i=0;i<2;i++)
 {
 arc(x2,y2+i+5,40,140,r/4);//upper right eye
 arc(x2,y2-r/5+i,220,320,r/4);//lower right eye
 }
 circle(x2,y2-r/12,r/10);//right pupil
seɳillstyle(1,0);
 floodfill(x2,y2-r/12,getcolor());
seɳillstyle(1,WHITE);
 floodfill(x2-15,y2-r/6,getcolor());
}
drawmouth()
{
 int x,y,r,i;
 x=face1->posiѲon.x;
 y=face1->posiѲon.y+(face1->radius/1.5);
 r=face1->radius;
 setcolor(BLACK);
 if((face1->mood)==1)
 for(i=0;i<4;i++)
 arc(x,y-r/2+i,220,320,r/2);//make happy
 if((face1->mood)==0)
```

```
 for(i=0;i<4;i++)
 arc(x,y-i,40,140,r/2);//make sad
}
drawhair()
{ int x,y,r;
 setcolor(8);
setaspectraΘo(xasp,yasp/1.5);
 r=face1->radius;
 x=face1->posiΘon.x-r/2;
 y=face1->posiΘon.y-r/3;
 arc(x,y,34,225,100);
 arc(x+r,y,314,138,100);
seηillstyle(1,RED);
 floodfill(x,y-70,getcolor());
 floodfill(x+r,y-70,getcolor());
setaspectraΘo(xasp,yasp);
}
void main(void)
{
 int i=0;
 initgraph(&gdriver, &gmode,"C:\\TC\\BGI");
 while(!kbhit())
 {
 if((i%2)==1)
 {
 setvisualpage(1);
 setacΘvepage(0);
 clearviewport();
 face1->mood=0;
 drawface();
 delay(1000);
 }
 else
 {
 setvisualpage(0);
 setacΘvepage(1);
 clearviewport();
 face1->mood=1;
 drawface();
 delay(300);
 }
```

```
 i++;
 }
getch();
closegraph();
}
```

Output:



Conclusion - Comment on :

1. Importance of story building
2. Defining the basic character of story
3. Apply techniques to these characters