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:

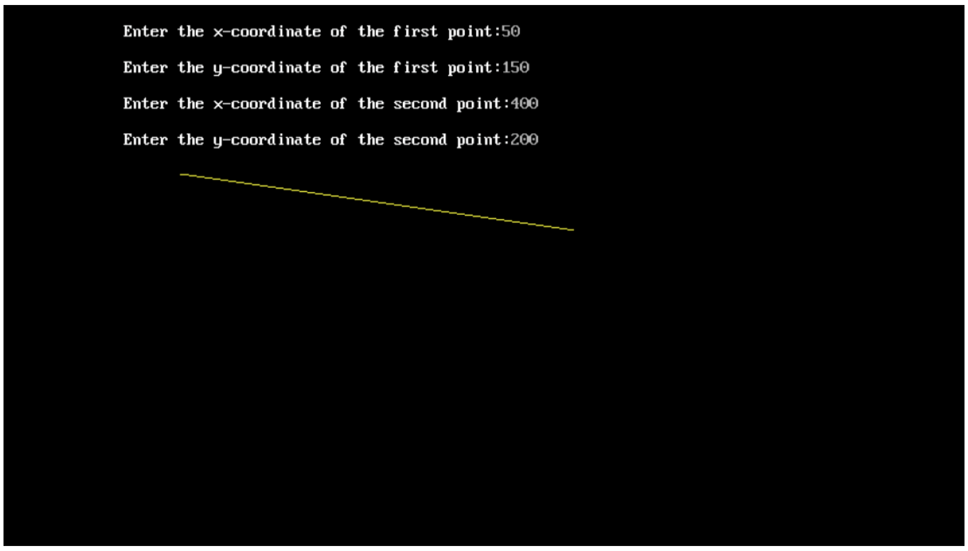
1. Input two endpoints: (x1, y1) and (x2, y2).
2. Calculate the differences in the x and y coordinates:
3. dx = x2 - x1 dy = y2 - y1
4. Determine the number of steps required to draw the line. You can use the maximum difference between dx and dy:
5. steps = max(abs(dx), abs(dy))
6. Calculate the increments for x and y:
7. x_increment = dx / steps y_increment = dy / steps
8. Initialize the current position (x, y) as the starting point (x1, y1):
9. x = x1 y = y1
10. For each step from 1 to steps:
11. a. Round the current coordinates to the nearest integer since pixel positions are discrete. b. Plot the pixel at the current position (x, y). c. Update the current position:
12. x = x + x_increment y = y + y_increment
13. Continue the loop until you have plotted all the necessary pixels to draw the line segment.

Program:

```
#include<graphics.h>

#include<stdio.h>

#include<math.h>

#include<dos.h>
```

```
int main()
{
    float x,y,x1,y1,x2,y2,dx,dy,step;
    int i,gd=DETECT,gm;
    //detectgraph(&gd,&gm);
    initgraph(&gd,&gm,"");
    printf("\nEnter the x-coordinate of the first point:");
    scanf("%f",&x1);
    printf("\nEnter the y-coordinate of the first point:");
    scanf("%f",&y1);
    printf("\nEnter the x-coordinate of the second point:");
    scanf("%f",&x2);
    printf("\nEnter the y-coordinate of the second point:");
    scanf("%f",&y2);
    dx=abs(x2-x1);
    dy=abs(y2-y1);
    if(dx>dy)
    {
        step=dx;
    }
    else
    {
        step=dy;
    }
    dx=dx/step;
    dy=dy/step;
    x=x1;
    y=y1;
    i=1;
    while(i<=step)
    {
        putpixel(x,y,14);
        x=x+dx;
        y=y+dy;
        i=i+1;
        delay(100);
    }
    getch();
    closegraph();
}
```

Output:



```
Enter the x-coordinate of the first point:50
Enter the y-coordinate of the first point:150
Enter the x-coordinate of the second point:400
Enter the y-coordinate of the second point:200
```

Conclusion: Comment on -

1. Pixel- pixel is the smallest unit of an image. **putpixel** is used to color individual pixels on the screen.
2. Equation for line- DDA approximates a line between two points (x1, y1) and (x2, y2) using the equation y = mx + b. It calculates slope and updates coordinates.
3. Need of line drawing algorithm- Computers use pixels on a grid to represent images. To draw continuous lines, you need an algorithm to determine which pixels to color.
4. Slow or fast- DDA is simple but not the fastest.

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

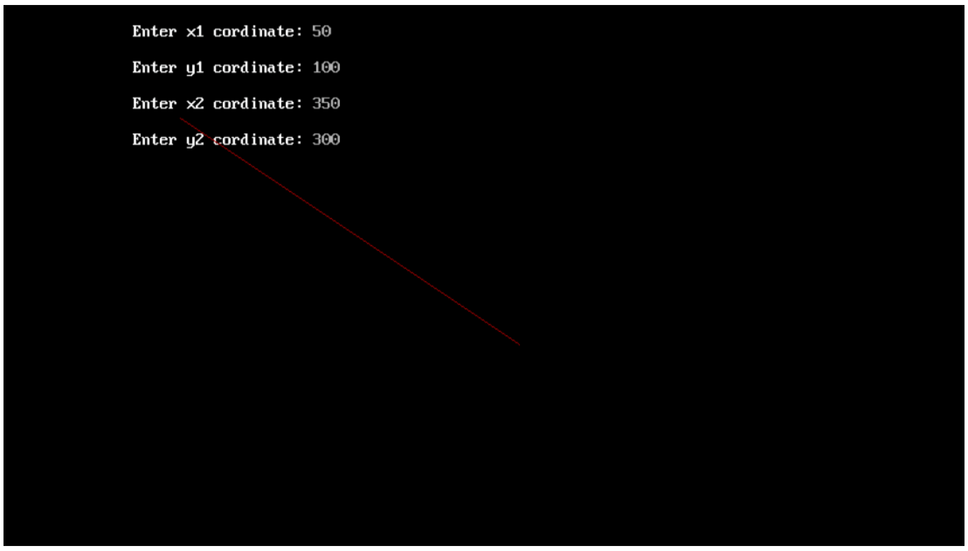
Theory:
In Bresenham's line algorithm pixel positions along the line path are obtained by determining the pixels i.e. nearer the line path at each step.

Algorithm -
1. Input two endpoints: (x1, y1) and (x2, y2).
2. Calculate the differences in the x and y coordinates:
3. dx = x2 - x1 dy = y2 - y1
4. Initialize variables for tracking the current position, decision parameter, and steps:
5. x = x1 y = y1 d = 2 * dy - dx x_increment = 1 y_increment = 1
6. If dx < 0, set x_increment to -1.
7. If dy < 0, set y_increment to -1.
8. Start a loop that runs from 1 to dx (or -dx if dx is negative):
9. a. Plot the pixel at the current position (x, y).
10. b. If the decision parameter is greater than or equal to 0, increment y by y_increment and update the decision parameter:
11. if d >= 0: y = y + y_increment d = d - 2 * dx
12. c. Increment x by x_increment.
13. d. Update the decision parameter:
14. d = d + 2 * dy
15. Repeat the loop until you have plotted all the necessary pixels to draw the line segment.

Program -

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


int main()
{
int x,y,x1,y1,x2,y2,p,dx,dy;
int gd=DETECT,gm=0;
initgraph(&gd,&gm, "");
printf("\n Enter x1 cordinate: ");
scanf("%d",&x1);
printf("\n Enter y1 cordinate: ");
scanf("%d",&y1);
printf("\n Enter x2 cordinate: ");
scanf("%d",&x2);
printf("\n Enter y2 cordinate: ");
scanf("%d",&y2);

x=x1;
y=y1;
dx=x2-x1;
dy=y2-y1;

putpixel (x,y, RED);
p = (2 * dy-dx);

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

}
putpixel (x,y, RED);

}

getch();
closegraph();
}
```

Output –



```
Enter x1 cordinate: 50
Enter y1 cordinate: 100
Enter x2 cordinate: 350
Enter y2 cordinate: 300
```

Conclusion: Comment on -

1. Pixel- The "pixel" is represented by the **putpixel** function. It sets the color of individual pixels on the screen.
2. Equation for line- The algorithm calculates and uses the difference in the x and y coordinates (dx and dy) to determine which pixels to color to approximate the line.
3. Need of line drawing algorithm- The need for a line drawing algorithm arises from the discrete nature of digital screens, which represent images using pixels on a grid. To draw a continuous line on such a grid, an algorithm like Bresenham's is necessary to determine which pixels to color to create the appearance of a smooth line.
4. Slow or fast- Bresenham's algorithm is relatively fast and efficient, especially for drawing lines with integer coordinates. It uses integer arithmetic and avoids floating-point calculations

Experiment No. 3

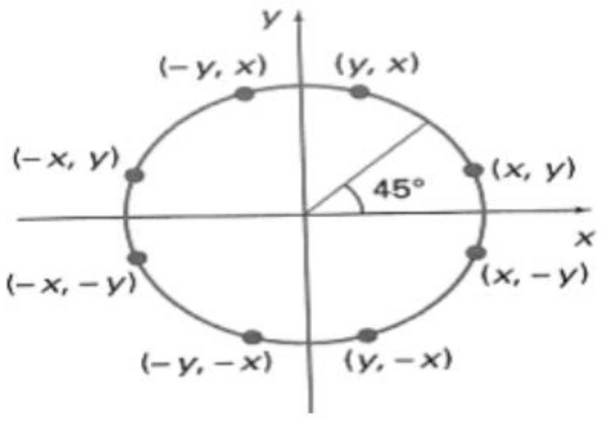
Aim: To implement midpoint circle algorithm.

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

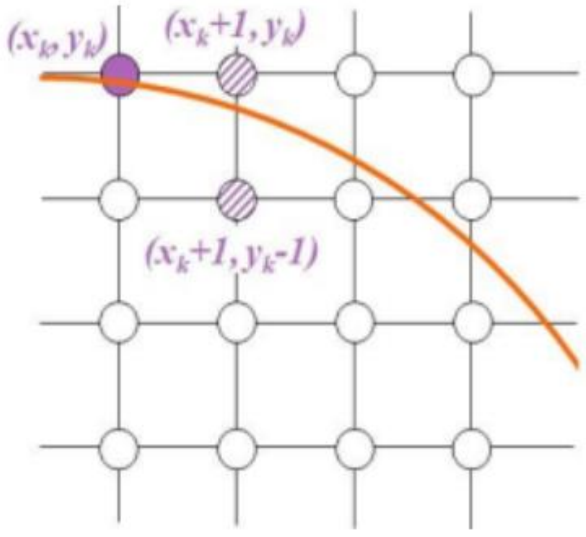
Let the circle function is f circle (x, y) -

 is &lt; 0, if (x, y) is inside circle boundary,

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

 is &gt; 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 &lt; 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 – The Midpoint Circle Algorithm is a simple and efficient method for drawing a circle on a pixel grid in computer graphics. It uses the concept of the "midpoint" to determine which pixels should be part of the circle. Here is the step-by-step derivation of the Midpoint Circle Algorithm:

Assumptions:

1. You have a grid of pixels, and each pixel is identified by its coordinates (x, y), where (0,0) is the center of the grid.

Algorithm:

1. Start with the initial point at (x, y) = (0, r), where r is the radius of the circle.

2. Calculate the initial decision parameter: P = 5/4 - r (i.e., P0 = 5/4 - r).

3. Initialize x = 0 and y = r.

4. At each step, plot the points (x, y), (-x, y), (x, -y), and (-x, -y) to take advantage of the circle's symmetry.

5. Compute the next decision parameter Pk for the next pixel position (xk+1, yk) as follows:

   - If Pk < 0, choose the pixel to the right: xk+1 = xk + 1 and Pk+1 = Pk + 2*xk + 3.

   - If Pk ≥ 0, choose the pixel to the lower-right: xk+1 = xk + 1 and yk+1 = yk - 1, and Pk+1 = Pk + 2*xk - 2*yk + 5.

6. Repeat steps 4 and 5 until x is greater than or equal to y. At this point, you've completed one-eighth of the circle.

7. For each point plotted, reflect it in all eight octants to complete the full circle.

Here's a more detailed explanation:The algorithm starts at the point (0, r), which is chosen because it's on the circle's perimeter, and it's one of the points that minimizes the error when calculating the midpoint. The decision parameter P is initialized as P0 = 5/4 - r.

The algorithm then proceeds by incrementing x and decrementing y while repeatedly calculating the next decision parameter Pk. The choice of the next pixel depends on whether Pk is less than 0 or greater than/equal to 0.

The algorithm continues until x is greater than or equal to y. At this point, one-eighth of the circle is drawn, and the other seven eighths can be generated by reflecting the points in each octant.

The Midpoint Circle Algorithm is efficient because it minimizes the number of calculations and operations needed to draw the circle, making it suitable for use in real-time graphics and situations where performance is important.

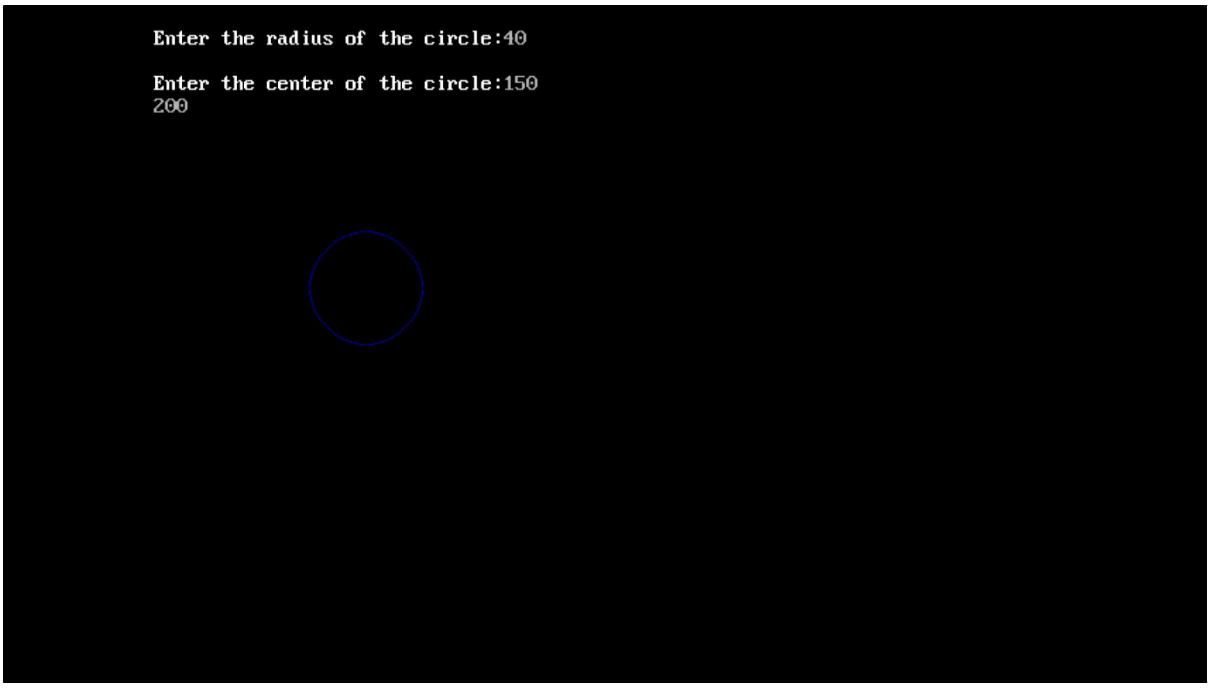
Program –

```
#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,"C:\\TurboC3\\BGI");
```

```
printf("\nEnter the radius of the circle:");
scanf("%d",&r);
printf("\nEnter 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();
return 0;
```

```
}
```

output –



```
Enter the radius of the circle:40

Enter the center of the circle:150
200
```

Conclusion:

The Midpoint Circle Algorithm is a widely used method for drawing circles in computer graphics. It differs from line drawing algorithms in its approach, specifically tailored to circles. Here's a brief comment on the Midpoint Circle Algorithm and a comparison with line drawing algorithms, along with a note on the process and time taken:The Midpoint Circle Algorithm is a fundamental tool in computer graphics, allowing for efficient and precise circle drawing. Unlike line drawing algorithms, it considers the symmetry of circles, optimizing the drawing process.It calculates points on the circle by incrementally moving along the circumference and adjusting the decision parameter.By exploiting symmetry, it reduces computational load compared to line drawing, which needs to account for various angles and slopes.The algorithm proceeds by iteratively determining points at each octant and mirroring them to complete the circle.It is highly time-efficient, with a complexity of O(n) for drawing a circle, where 'n' represents the radius.

Experiment No. 4

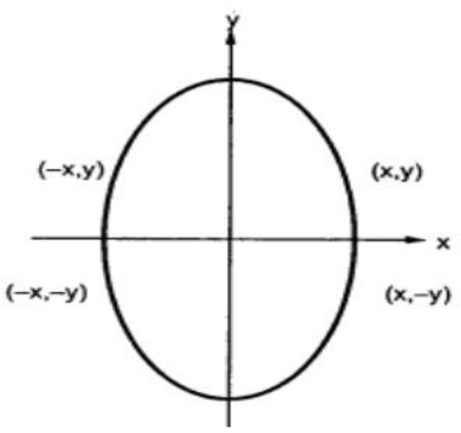
Aim-To implement midpoint Ellipse algorithm

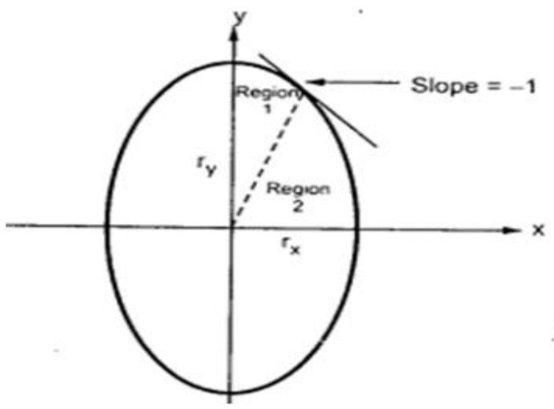
Objective:

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 &lt; 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: int x=0, y=b; [starting point]

int fx=0, fy=$2a^2$ b [initial partial derivatives]

int p = b^2-a^2 b+a^2/4

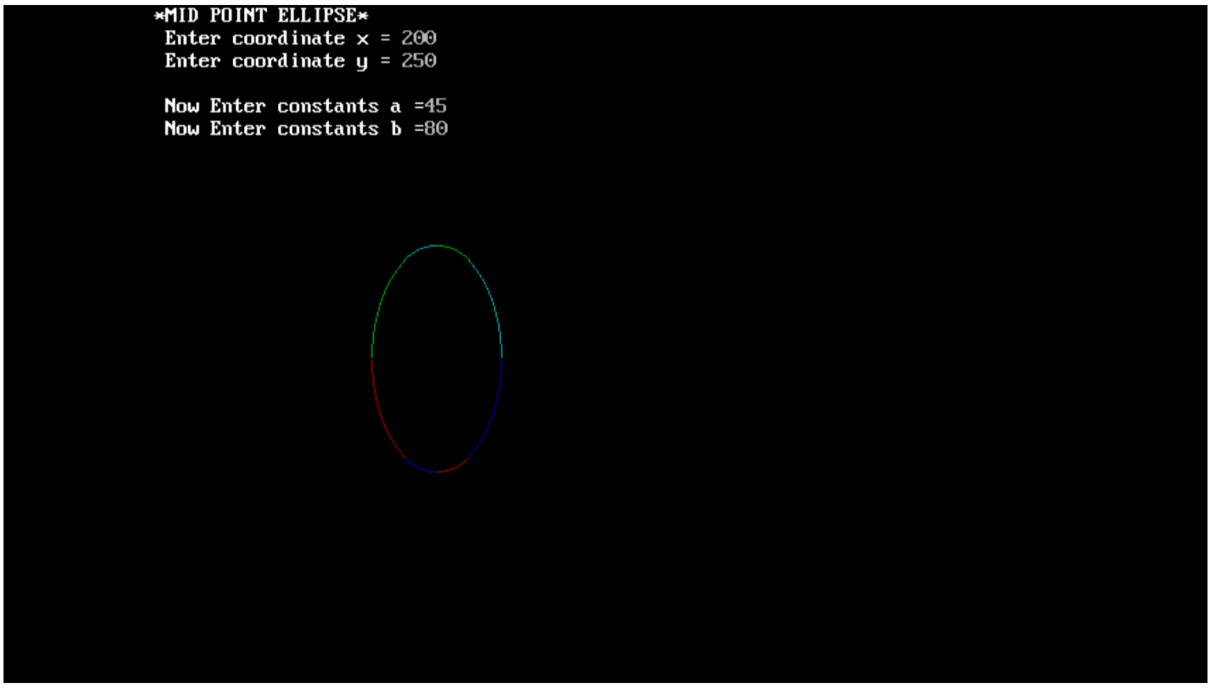
```
while (fx<="" 1="" {="" set="" pixel="" (x,="" y)="" x++;="" fx="fx" +="" 2b^2;
        if (p<0)
        p = p + fx +b2;
        else
        {
                y--;
                fy=fy-2a2
                p = p + fx +b2-fy;
        }
}
Setpixel (x, y);
p=b2(x+0.5)2+ a2 (y-1)2- a2 b2
while (y>0)
{
        y--;
        fy=fy-2a2;
        if (p>=0)
        p=p-fy+a2
      else
        {
                x++;
                fx=fx+2b2
                p=p+fx-fy+a2;
        }
        Setpixel (x,y);
```

```
  }


Program: #include<stdio.h>

#include<graphics.h>

#include<dos.h>

#include<conio.h>

int main()

{

   long x,y,x_center,y_center;

   long a_sqr,b_sqr,fx,fy,d,a,b,tmp1,tmp2;

   int g_driver=DETECT,g_mode;


   initgraph(&g_driver,&g_mode,"C:\\TurboC3\\BGI");

   printf("*MID POINT ELLIPSE*");

   printf("\n Enter coordinate x = ");

   scanf("%ld",&x_center);

   printf(" Enter coordinate y = ");

   scanf("%ld",&y_center);

   printf("\n Now Enter constants a =");

   scanf("%ld",&a,&b);

   printf(" Now Enter constants b =");

   scanf("%ld",&b);

   x=0;

   y=b;

   a_sqr=a*a;

   b_sqr=b*b;
```

```
fx=2*b_sqr*x;

fy=2*a_sqr*y;

d=b_sqr-(a_sqr*b) + (a_sqr*0.25);

do

{

   putpixel(x_center+x,y_center+y,4);

   putpixel(x_center-x,y_center-y,3);

   putpixel(x_center+x,y_center-y,2);

   putpixel(x_center-x,y_center+y,1);


   if(d<0)

   {

      d=d+fx+b_sqr;

   }

   else

   {

      y=y-1;

      d=d+fx+-fy+b_sqr;

      fy=fy-(2*a_sqr);

   }

   x=x+1;

   fx=fx+(2*b_sqr);

   delay(10);

}

while(fx<fy);

tmp1=(x+0.5)*(x+0.5);
```

```
   tmp2=(y-1)*(y-1);
   d=b_sqr*tmp1+a_sqr*tmp2-(a_sqr*b_sqr);

   do
   {
      putpixel(x_center+x,y_center+y,1);
      putpixel(x_center-x,y_center-y,2);
      putpixel(x_center+x,y_center-y,3);
      putpixel(x_center-x,y_center+y,4);

      if(d>=0)
      d=d-fy+a_sqr;
      else
      {
         x=x+1;
         d=d+fx-fy+a_sqr;
         fx=fx+(2*b_sqr);
      }
      y=y-1;
      fy=fy-(2*a_sqr);
   }
   while (y>0);
   getch();
   closegraph();
  return 0;
}
```

Output:



```
*MID POINT ELLIPSE*
 Enter coordinate x = 200
 Enter coordinate y = 250

 Now Enter constants a =45
 Now Enter constants b =80
```

Conclusion: The algorithm used to draw an ellipse is notably different from that of a circle due to the fact that ellipses are not symmetric in the same way that circles are. The primary distinction lies in the process of calculating and plotting the ellipse's points, which involves varying both the horizontal and vertical radii as it moves along the curve. In contrast, circles have a constant radius.

The importance of ellipse drawing algorithms lies in their applicability to various real-world objects. Ellipses are commonly encountered in fields such as engineering, computer graphics, and mathematics. They represent not only simple geometric shapes but also many practical objects like wheels, orbits of celestial bodies, and even the shape of the human eye's cornea. Precisely rendering ellipses is essential for accurately representing these objects in computer graphics, engineering drawings, or scientific simulations. Hence, a robust and efficient algorithm for drawing ellipses is valuable for creating realistic and accurate depictions of objects and phenomena in these domains.

Experiment No. 5

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

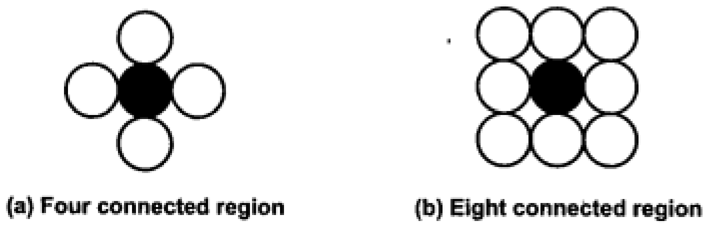
Objective:

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.



Procedure:

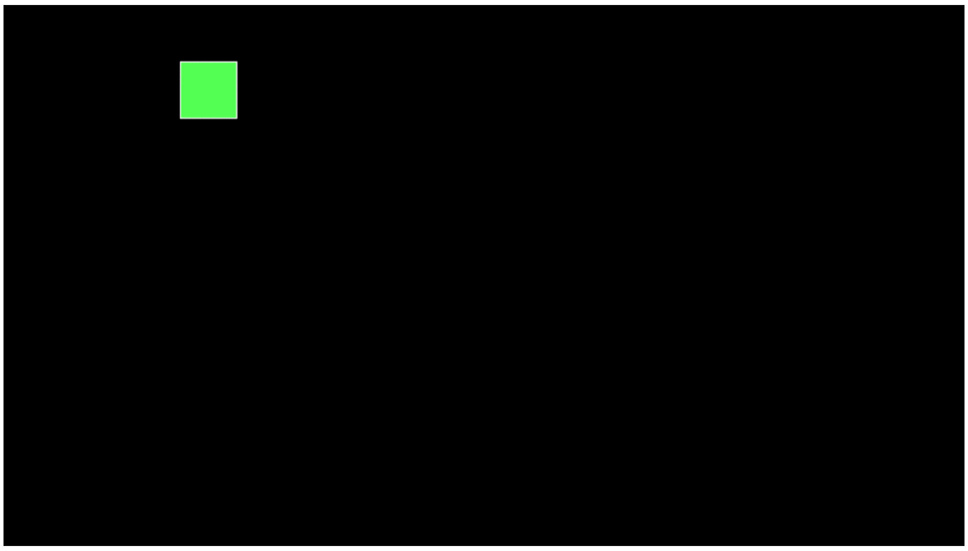
```
boundary_fill (x, y, f_color, b_color)
{
if (getpixel (x, y) != b_colour && getpixel (x, y) != f_colour)
        {
        putpixel (x, y, f_colour)
        boundary_fill (x + 1, y, f_colour, b_colour);
        boundary_fill (x, y + 1, f_colour, b_colour);
        boundary_fill (x - 1, y, f_colour,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

Program:

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

void boundary_fill(int x, int y, int fill_color, int boundary_color) {
   if (getpixel(x, y) != boundary_color && getpixel(x, y) != fill_color) {
      putpixel(x, y, fill_color);
      boundary_fill(x + 1, y, fill_color, boundary_color);
      boundary_fill(x - 1, y, fill_color, boundary_color);
      boundary_fill(x, y + 1, fill_color, boundary_color);
      boundary_fill(x, y - 1, fill_color, boundary_color);
      boundary_fill(x - 1, y - 1, fill_color, boundary_color);
      boundary_fill(x + 1, y - 1, fill_color, boundary_color);
      boundary_fill(x - 1, y + 1, fill_color, boundary_color);
      boundary_fill(x + 1, y + 1, fill_color, boundary_color);
   }
}

int main() {
   int gd = DETECT, gm;
```

```
    initgraph(&gd, &gm, "c:\\turboc3\\bgi");

    rectangle(50, 50, 100, 100);

    boundary_fill(60, 61, 10, 15);

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

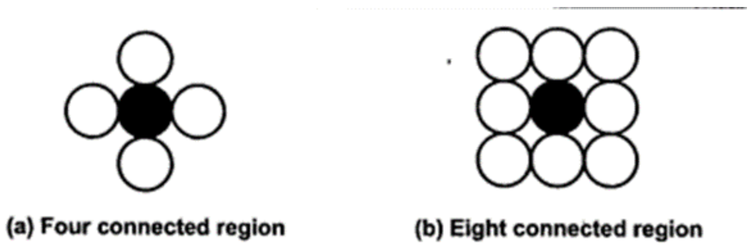
Output:



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.



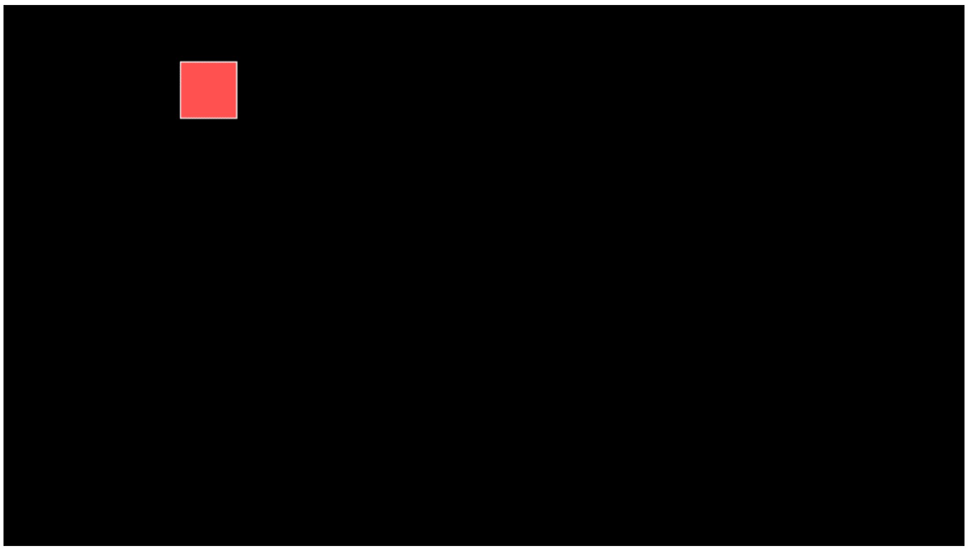
Procedure -

```
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- Flood fill is a vital graphics algorithm used for tasks such as coloring, image editing, and interactive interfaces. It's indispensable in graphic design, game development, and procedural content generation.
2. Limitation of methods- Flood fill struggles with areas containing gaps or complex boundaries. It may also lack fine control, leading to unwanted overspill, and its performance can degrade on large regions.

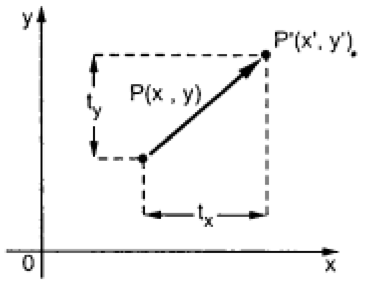
3. Usefulness of method- Flood fill's simplicity makes it an excellent choice for basic filling tasks in graphics, providing quick and interactive region filling. However, for intricate or more controlled operations, alternative techniques may be necessary.

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.

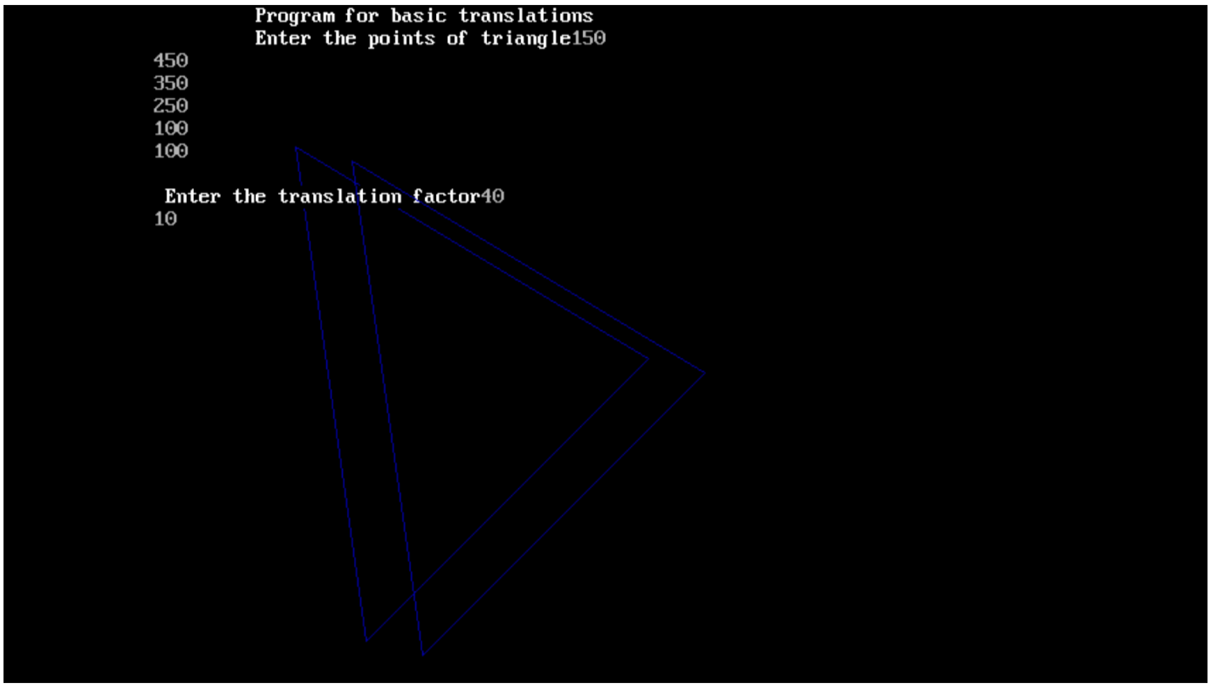


Consider (x,y) are old coordinates of a point. Then the new coordinates of that same point (x',y') can be obtained as follows:
x' = x + tx
y' = y + ty
We denote translation transformation as P. we express above equations in matrix form as:
P' = P + T , where

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

Program:
```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include<math.h>
int main()
{
        int gm;
        int gd=DETECT;
        int x1,x2,x3,y1,y2,y3,nx1,nx2,nx3,ny1,ny2,ny3,c;
        int sx,sy,xt,yt,r;
        float t;
        initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
        printf("\t Program for basic transactions");
        printf("\n\t Enter the points of triangle");
        setcolor(1);
        scanf("%d%d%d%d%d%d",&x1,&y1,&x2,&y2,&x3,&y3);
        line(x1,y1,x2,y2);
        line(x2,y2,x3,y3);
        line(x3,y3,x1,y1);
        printf("\n Enter the translation factor");
        scanf("%d%d",&xt,&yt);
         nx1=x1+xt;
         ny1=y1+yt;
         nx2=x2+xt;
         ny2=y2+yt;
         nx3=x3+xt;
         ny3=y3+yt;
         line(nx1,ny1,nx2,ny2);
         line(nx2,ny2,nx3,ny3);
         line(nx3,ny3,nx1,ny1);
 getch();
 closegraph();
}
```

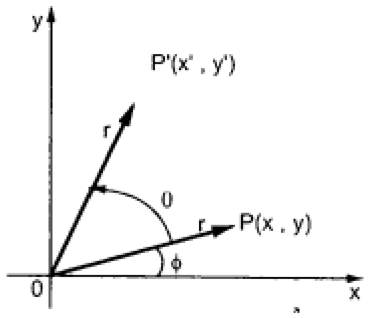
Output –



```
                Program for basic translations
                Enter the points of triangle150
        450
        350
        250
        100
        100

         Enter the translation factor40
        10
```

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

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

$$P' = P \cdot R$$

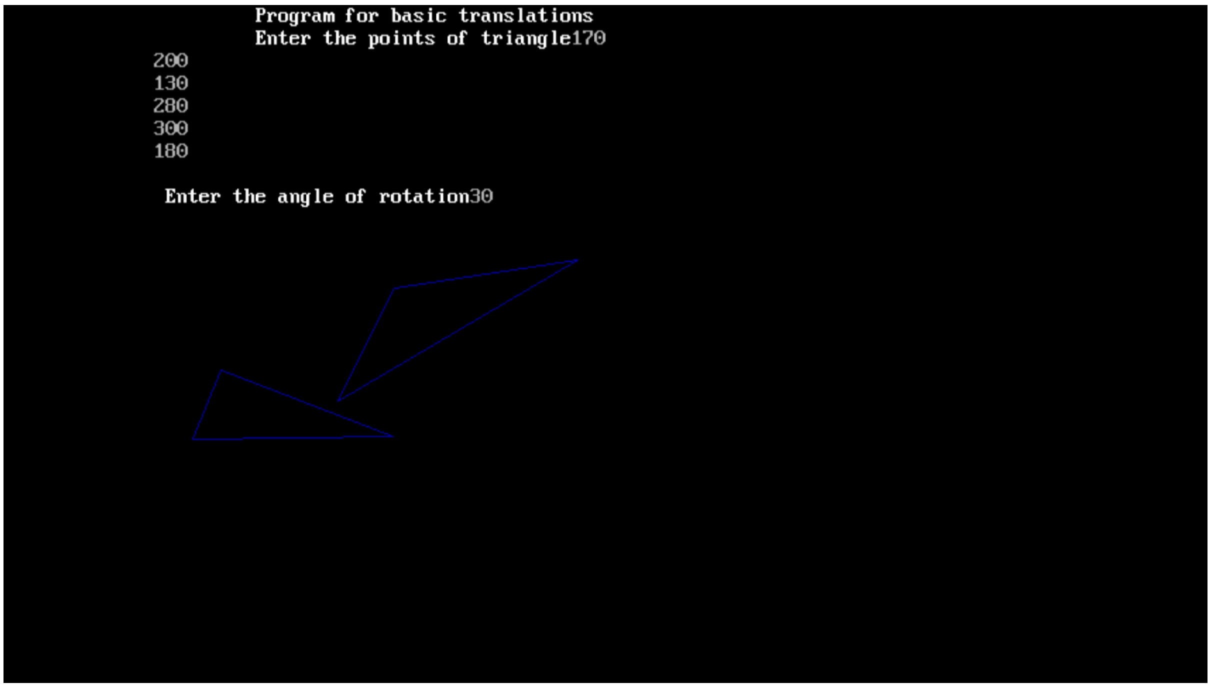
where R is the rotation matrix and it is given as

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

Program: #include <graphics.h>

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include<math.h>
int main()
{
        int gm;
        int gd=DETECT;
        int x1,x2,x3,y1,y2,y3,nx1,nx2,nx3,ny1,ny2,ny3,c;
        int sx,sy,xt,yt,r;
        float t;
        initgraph(&gd,&gm,"C:\\TurboC3\\BGI ");
        printf("\t Program for basic transactions");
        printf("\n\t Enter the points of triangle");
        setcolor(1);
        scanf("%d%d%d%d%d%d",&x1,&y1,&x2,&y2,&x3,&y3);
        line(x1,y1,x2,y2);
        line(x2,y2,x3,y3);
        line(x3,y3,x1,y1);
 printf("\n Enter the angle of rotation");
                        scanf("%d",&r);
                        t=3.14*r/180;
                        nx1=abs(x1*cos(t)-y1*sin(t));
                        ny1=abs(x1*sin(t)+y1*cos(t));
                        nx2=abs(x2*cos(t)-y2*sin(t));
                        ny2=abs(x2*sin(t)+y2*cos(t));
                        nx3=abs(x3*cos(t)-y3*sin(t));
                        ny3=abs(x3*sin(t)+y3*cos(t));
                        line(nx1,ny1,nx2,ny2);
                        line(nx2,ny2,nx3,ny3);
                        line(nx3,ny3,nx1,ny1);
                        getch();
  closegraph();
return 0;
}
```

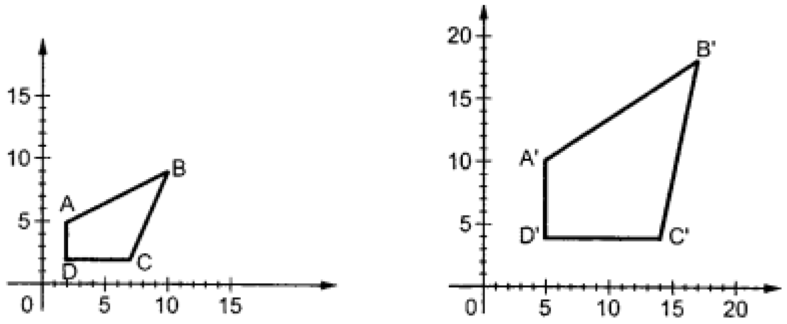
Output:



```
                    Program for basic translations
                    Enter the points of triangle170
            200
            130
            280
            300
            180

             Enter the angle of rotation30
```

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:

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

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

$$= P \cdot S$$

O

Output -

Conclusion: Comment on :

Transformations play a fundamental role in computer graphics, enabling the manipulation and rendering of objects in various ways. They are essential for tasks like scaling, rotation, translation, and skewing. Scaling allows for resizing objects, rotation helps in creating dynamic visual effects, while translation positions objects within a scene. Skewing distorts objects for perspective or artistic effects. Additionally, transformations are vital for 3D rendering, enabling the projection of 3D scenes onto 2D screens. They facilitate animation by smoothly transitioning between different states, making them integral in video games, simulations, and special effects in movies. Overall, transformations are the cornerstone of creating immersive and dynamic visual experiences in the realm of computer graphics.

Matrix-based transformations involve representing transformations using matrices. These matrices enable efficient combination and application of multiple transformations to objects. Geometric transformations, on the other hand, use direct geometric calculations to apply transformations, making them conceptually simpler but less versatile than matrix-based approaches.

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.

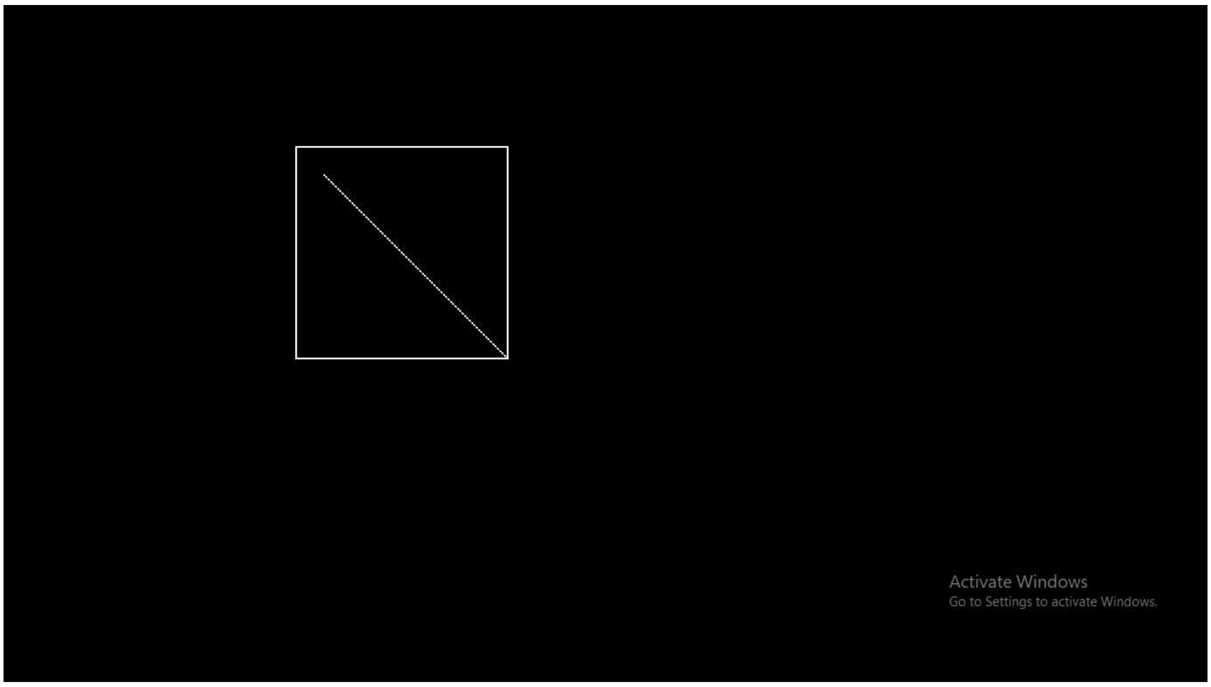
Program:
```
#include<stdio.h>

#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,"C:\\TurboC3\\BGI ");

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[2];
xx2 = x1 + t2 * p[2];
yy1 = y1 + t1 * p[3];
yy2 = y1 + t2 * p[3];
line(xx1,yy1,xx2,yy2);
```

```
}
delay(5000);
closegraph();
return 0;
}
```

Output:



Conclusion: In conclusion, the Liang-Barsky algorithm is a powerful and efficient method for line clipping in computer graphics. It provides a systematic way to determine which portion of a line lies within a specified clipping window, helping optimize rendering and improve the visual representation of objects on the screen. By efficiently discarding portions of lines that are outside the viewing area, it reduces unnecessary computational overhead, making it a valuable tool for real-time graphics applications

Experiment No. 8

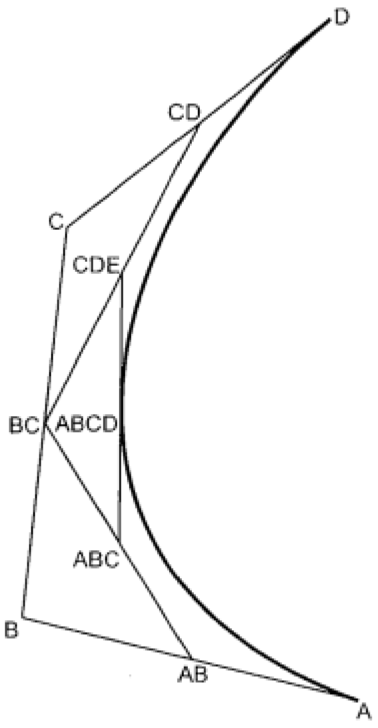
Aim: To implement Bezier curve for n control points. (Midpoint approach)

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.

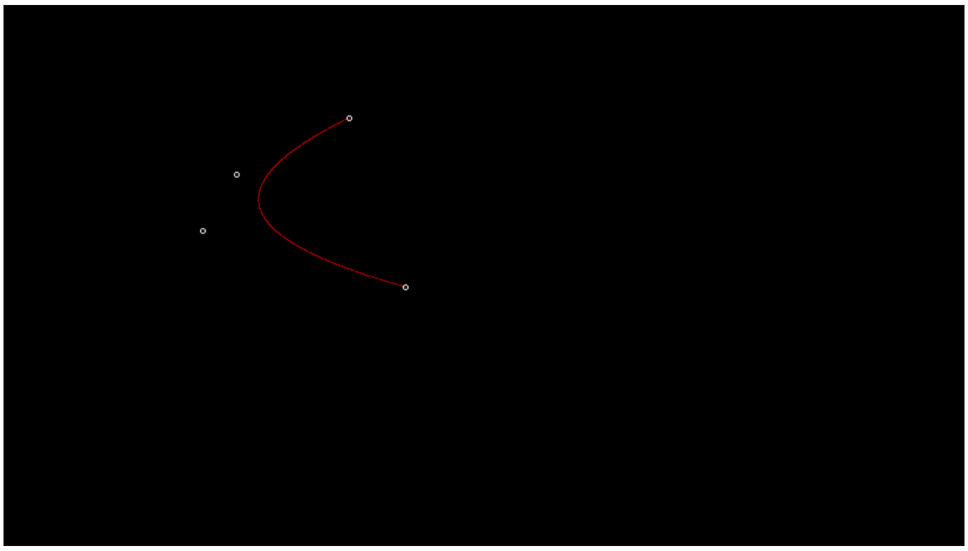
Program:

```
#include<graphics.h>

#include<math.h>

int x[4],y[4];

void bezier(int x[4],int y[4])

{

int gd=DETECT,gm,i;
```

```
double t,xt,yt;
initgraph(&gd,&gm," ");
for(t=0.0;t<1.0;t+=0.0005)
{
xt=pow((1.0-t),3)*x[0]+3*t*pow((1.0-t),2)*x[1]+3*pow(t,2)*(1.0-t)*x[2]+pow(t,3)*x[3];
yt=pow((1.0)-t,3)*y[0]+3*t*pow((1.0)-t,2)*y[1]+3*pow(t,2)*(1.0-t)*y[2]+pow(t,3)*y[3];
putpixel(xt,yt,4);
delay(5);
}
for(i=0;i<4;i++)
{
putpixel(x[i],y[i],5);
circle(x[i],y[i],2);
delay(2);
}
getch();
closegraph();
}
int main()
{
int i,x[4],y[4];
printf("Enter the four control points : ");
for(i=0;i<4;i++)
{
scanf("%d %d",&x[i],&y[i]);
}
```

```
bezier(x,y);

}
```

Output:



Conclusion – Comment on

1. Difference from arc and line- Bezier curves are parametric curves defined by control points and offer more flexibility in defining shapes compared to arcs. Arcs typically represent portions of circles or ellipses and have simpler geometric definitions. Bezier curves can create a wide range of curves with various shapes.

2. Importance of control point- Control points in the Bezier curve determine the shape and behavior of the curve. They provide a high degree of control and allow for precise manipulation of the curve's path. The number and position of control points influence the shape, direction, and curvature of the curve. Control points are essential for defining custom shapes and paths.

3. Applications-

- Graphic Design: Bezier curves are fundamental in graphic design software for creating and editing shapes, paths, and curves.
- Font Design: They are used to define and manipulate the shapes of characters and letters in font design.
- Animation: Bezier curves are employed to create smooth and natural motion paths for animation.
- Engineering and CAD: They are used in computer-aided design (CAD) for modeling curves and surfaces in engineering and architecture.

- Robotics: Bezier curves help define the paths that robots follow for tasks such as welding or assembly.
- Simulation: Bezier curves are used in simulations, such as flight simulations, to define the trajectory of objects.

Experiment No. 9

Aim: To implement Character Generation: Bit Map Method

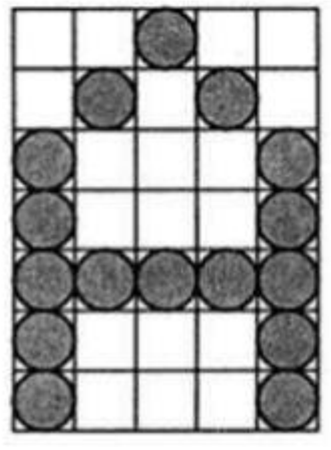
Objective:
Identify the different Methods for Character Generation and generate the character using Stroke

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.

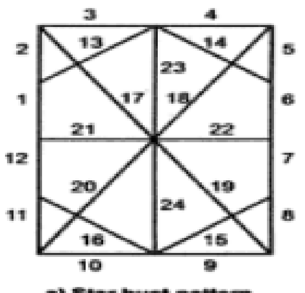
• In bit matrix method when the dots are stored in the form of array the value 1 in array represent the characters i.e. where the dots appear we represent that position with numerical value 1 and the value where dots are not present is represented by 0 in array.

• 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.

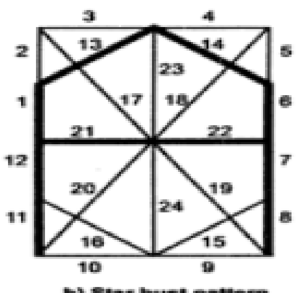
A 5x7 array is commonly used to represent characters. However, 7x9 and 9x13 arrays are also used. Higher resolution devices such as inkjet printer or laser printer may use character arrays that are over 100x100.



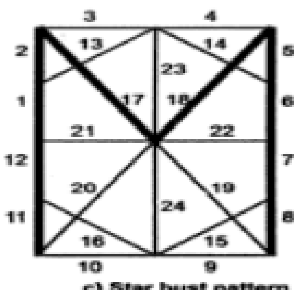
Starburst method –
In this method a fix pattern of line segments is used to generate characters. Out of these 24-line segments, segments required to display for particular character are highlighted. This method of character generation is called starburst method because of its characteristic appearance. 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



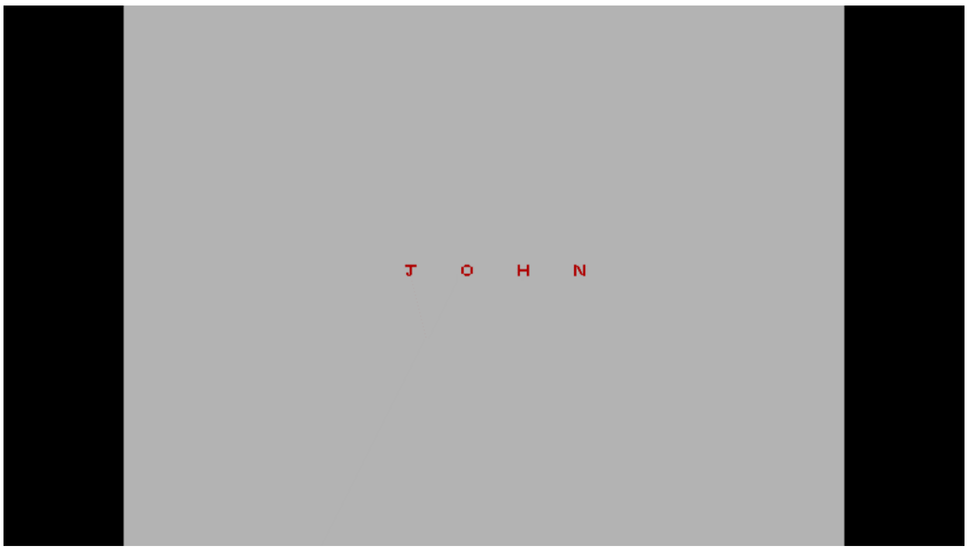
c) Star bust pattern for character M

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," ");//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 -



```
J   O   H   N
```

Conclusion: Comment on

1. different methods- Bitmaps are digital images composed of pixels. Monochrome bitmaps use 1 bit for binary colors like black and white. Grayscale bitmaps employ multiple bits for various gray shades, while color bitmaps use more bits or bytes for a wide color spectrum. Bitmaps are a form of raster graphics with fixed resolutions, making them detailed but challenging to scale without pixelation. They can be compressed losslessly (no quality loss) or lossily (with some quality loss) and are editable pixel by pixel using image software
2. advantage of stroke method- One of the notable advantages of this approach is that it allows for precise control over character design. By manipulating the pixel grid, characters can be customized in terms of shape, size, and style. Each character is represented as a set of 1s (on pixels) and 0s (off pixels), making it easy to create unique character designs.
3. one limitation- While the stroke method provides fine-grained control over character design, it is primarily suitable for displaying characters in a pixelated or bitmapped style. The limitation of this method is that it may not scale well to produce characters with smooth curves or fonts with anti-aliased edges. It is best suited for a retro or pixel art aesthetic but may not be ideal for modern, high-resolution, or anti-aliased text rendering, where more advanced text rendering techniques are preferred.

Experiment No. 10

Aim: To develop programs for making animations such as

Objective:

Draw an object and apply various transformation techniques to this object. Translation, scaling and rotation is applied to object to perform animation.

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:

Output:

Conclusion - Comment on :

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