



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:

Step1: Start Algorithm

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

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

Step4: Calculate $dx = x_2 - x_1$

Step5: Calculate $dy = y_2 - y_1$

Step6: If $ABS(dx) > ABS(dy)$

Then $step = abs(dx)$

Else

Step7: $x_{inc} = dx / step$

$y_{inc} = dy / step$

assign $x = x_1$

assign $y = y_1$

Step8: Set pixel (x, y)

Step9: $x = x + x_{inc}$

$y = y + y_{inc}$

Set pixels $(Round(x), Round(y))$



Step10: Repeat step 9 until $x = x_2$

Step11: End Algorithm

Program:

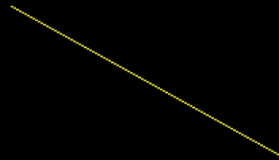
```
#include<graphics.h>
#include<stdio.h>
#include<math.h>
#include<dos.h>
int main()
{
    float x,y,x1,y1,x2,y2,dx,dy,step;
    int i,gd=DETECT,gm;
    //detectgraph(&gd,&gm);
    initgraph(&gd,&gm,"");
    printf("\nEnter the x-coordinate of the first point:");
    scanf("%f",&x1);
    printf("\nEnter 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:200  
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:250
```



Conclusion: Comment on -

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



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 $x \geq x_{end}$

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>
```

```
#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 coordinate: ");
```

```
    scanf("%d",&x1);
```

```
    printf("\n Enter y1 coordinate: ");
```

```
    scanf("%d",&y1);
```

```
    printf("\n Enter x2 coordinate: ");
```

```
    scanf("%d",&x2);
```

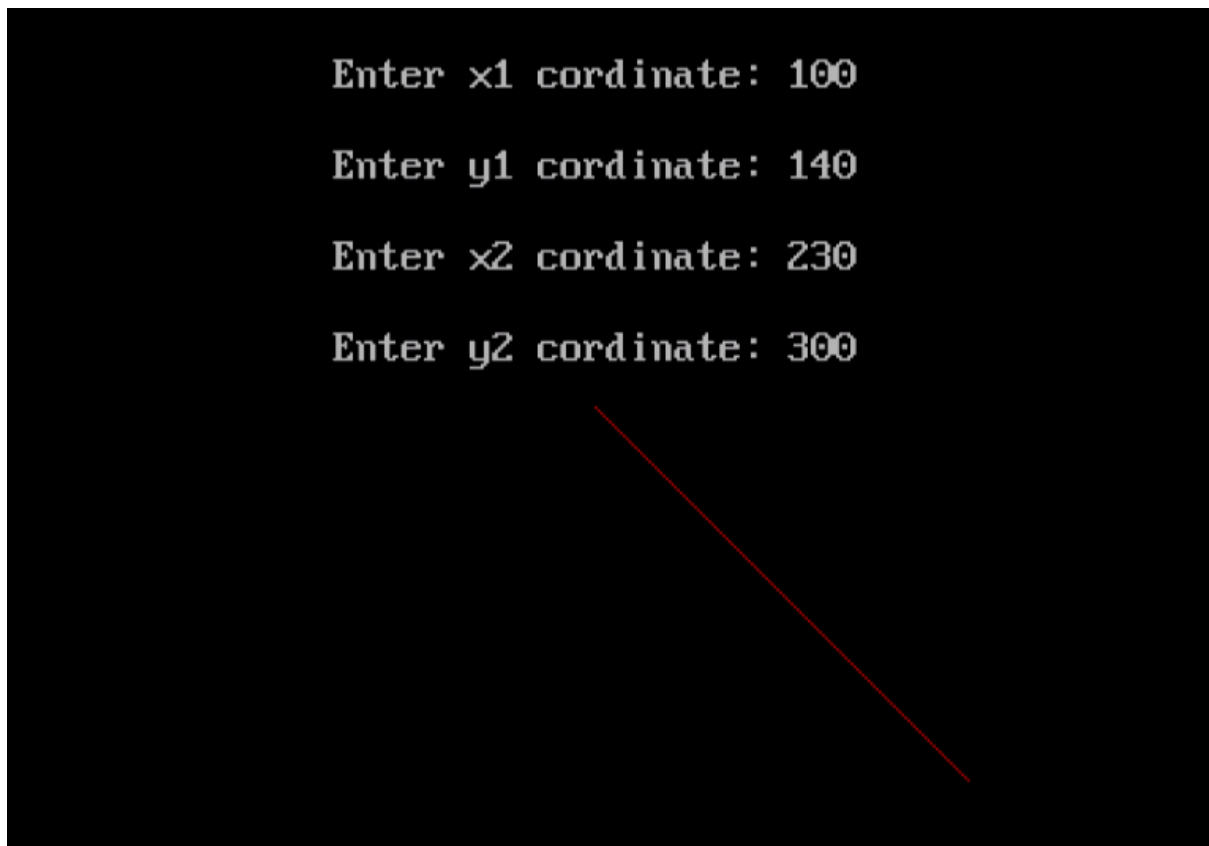


```
printf("\n Enter y2 coordinate: ");  
scanf("%d",&y2);  
  
x=x1;  
y=y1;  
dx=x2-x1;  
dy=y2-y1;  
  
putpixel (x,y, RED);  
p = (2 * dy-dx);  
  
while(x <= x2)  
{  
    if(p<0)  
    {  
        x = x+1;  
        p = p + 2*dy;  
    }  
    else  
    {  
        x = x + 1;  
        y = y + 1;  
        p = p + (2 * dy) - (2 * dx);  
    }  
    putpixel (x,y, RED);  
}
```



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

Output –



Conclusion: Comment on -

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



Experiment No. 3

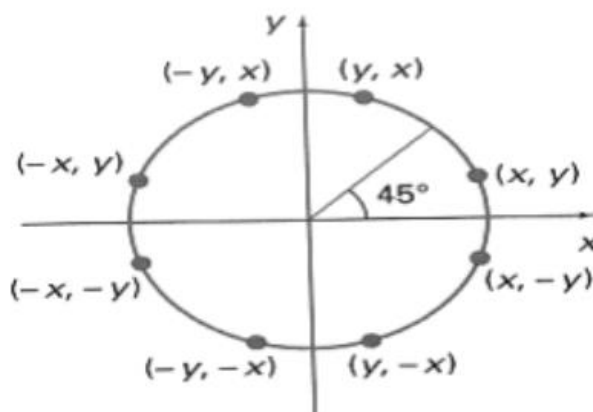
Aim: To implement midpoint circle algorithm.

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 $x^2 + y^2 = r^2$

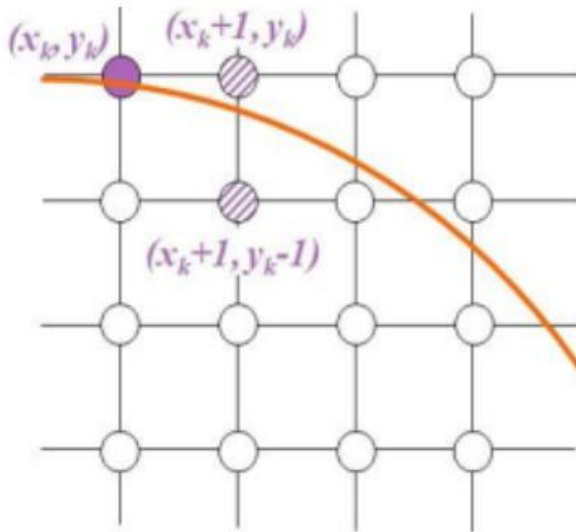
Let the circle function is $f_{\text{circle}}(x, y)$ -

$f_{\text{circle}} < 0$, if (x, y) is inside circle boundary,

$f_{\text{circle}} = 0$, if (x, y) is on circle boundary,

$f_{\text{circle}} > 0$, if (x, y) is outside circle boundary.

Consider the pixel at (x_k, y_k) is plotted,



Now the next pixel along the circumference of the circle will be either $(x_k + 1, y_k)$ or $(x_k + 1, y_k - 1)$ whichever is closer the circle boundary.

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

If $p_k < 0$, the midpoint is inside the circle and the pixel at y_k is closer to the circle boundary.

Otherwise, the midpoint is outside or on the circle boundary and the pixel at $y_k - 1$ 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 $x \leq y$

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<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 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);

}

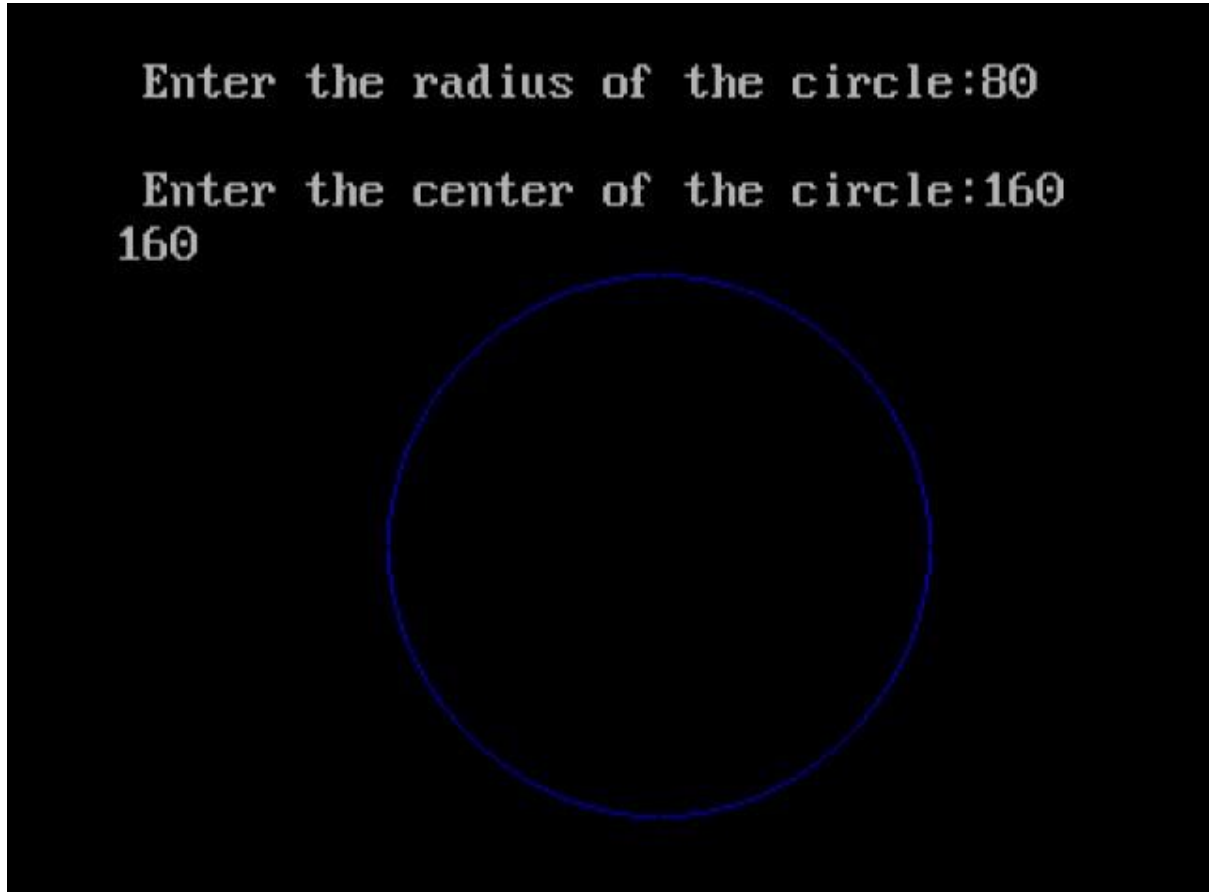
getch();

closegraph();

}
```



output –



Conclusion: Comment on

1. Fast or slow
2. Draw one arc only and repeat the process in 8 quadrants
3. Difference with line drawing method



Experiment No. 4

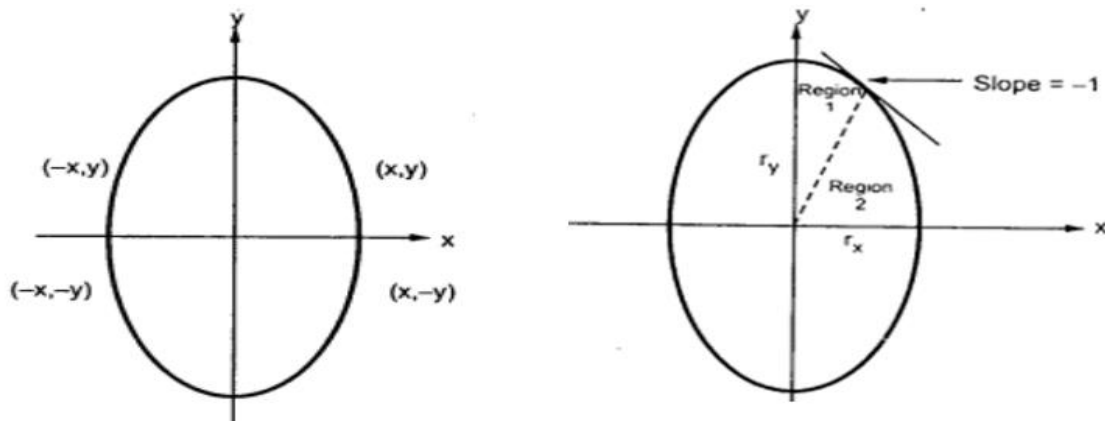
Aim-To implement midpoint Ellipse algorithm

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 $r_x \leq r_y$. As ellipse is drawn from 90° to 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 (x_c, y_c) is given as -

$$\left[\frac{(x - x_c)}{r_x}\right]^2 + \left[\frac{(y - y_c)}{r_y}\right]^2 = 1$$

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

$$\left[\frac{x}{r_x}\right]^2 + \left[\frac{y}{r_y}\right]^2 = 1$$

$$\text{i.e. } x^2 r_y^2 + y^2 r_x^2 = r_x^2 r_y^2$$

$$\text{Let, } f_{\text{ellipse}}(x, y) = x^2 r_y^2 + y^2 r_x^2 - r_x^2 r_y^2$$



Algorithm:

Step 1: Start

Step 2: Declare r_x , r_y , x , y , m , dx , dy , P , $P2$.

Step 3: Initialize initial point of region1 as

$$x=0, \quad y=r_y$$

Step 4: Calculate $P = r_y^2 + r_x^2 / 4 - r_y r_x^2$

$$dx = 2 r_y^2 x$$

$$dy = 2 r_x^2 y$$

Step 5: Update values of dx and dy after each iteration.

Step 6: Repeat steps while ($dx < dy$):

Plot (x, y)

if ($P < 0$)

Update $x = x + 1$;

$P += r_y^2 [2x + 3]$

Else

Update $x = x + 1$

$$y = y - 1$$

Step 7: When $dx \geq dy$, plot region 2:

Step 8: Calculate $P2 = r_y^2 (x+1 / 2)^2 + r_x^2 (y-1)^2 - r_x^2 r_y^2$

Step 9: Repeat till ($y > 0$)

If ($P2 > 0$)

Update $y = y - 1$ (x will remain same)

$$P2 = P2 - 2 y r_x^2 + r_x^2$$

else

$$x = x + 1$$



$$y = y-1$$

$$P2 = P2 + 2 r_y^2 [2x] - 2 y r_x^2 + r_x^2$$

Step 10: End

Program:

```
#include<stdio.h>

#include<graphics.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,"");

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

    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);

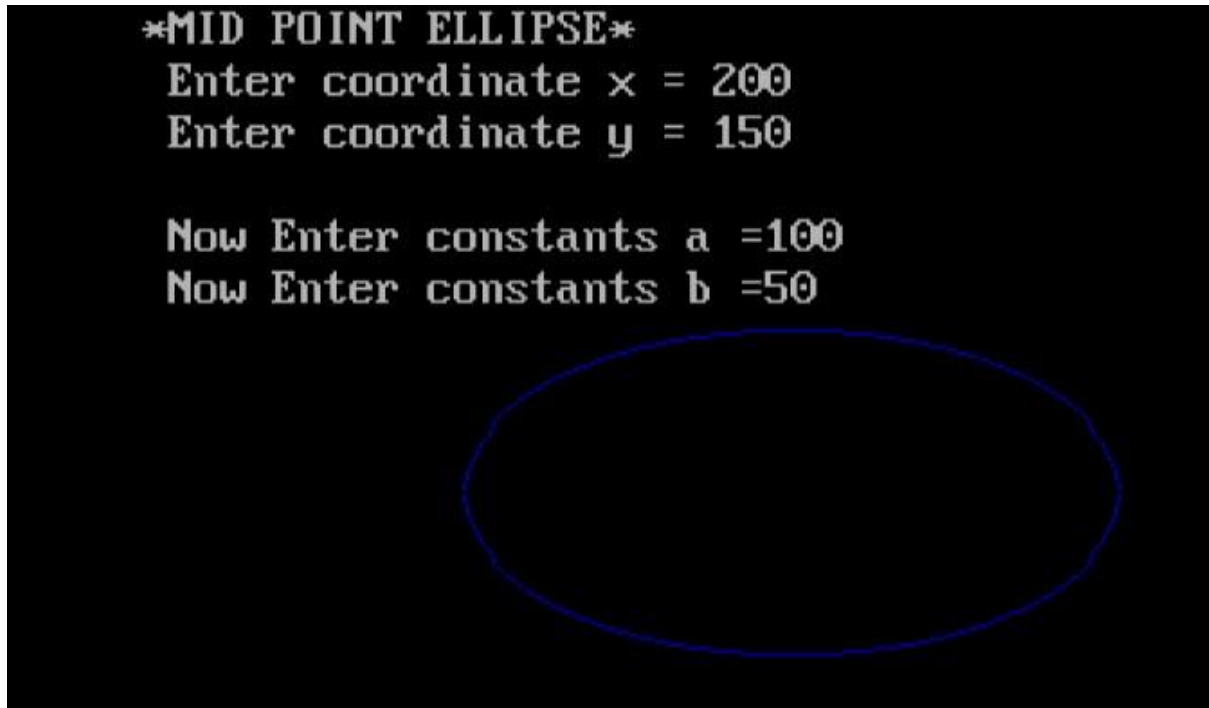
getch();

closegraph();

}
```



Output:



Conclusion: Comment on

1. Slow or fast
2. Difference with circle
3. Importance of object



Experiment No. 5

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

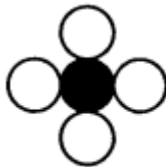
Objective:

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.

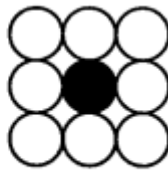
Theory:

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 <graphics.h>

#include <dos.h>
```

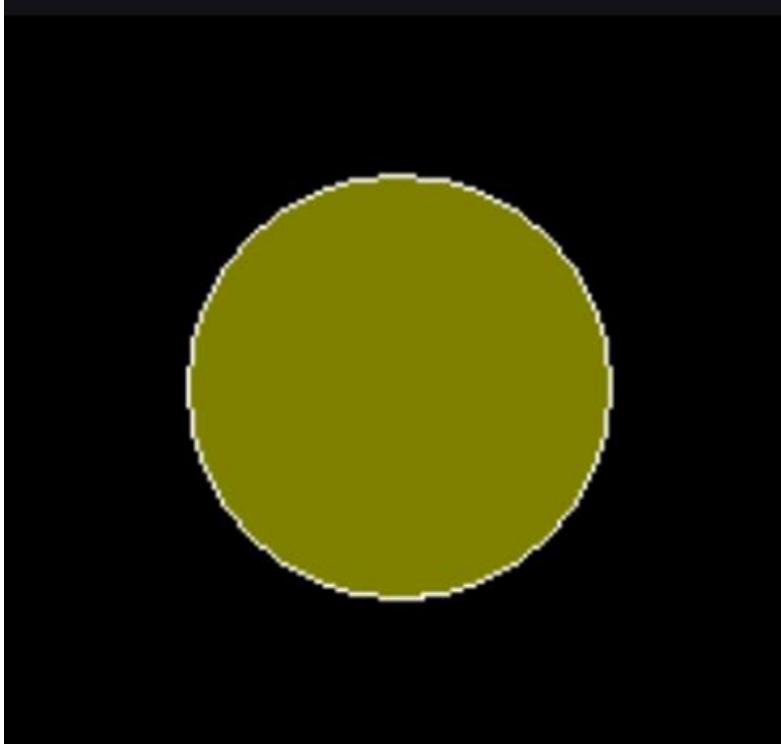


```
void boundaryfill(int x,int y,int f_c,int b_c)
{
    if (getpixel(x,y)!=b_c && getpixel(x,y)!=f_c)
    {
        putpixel(x,y,f_c);
        boundaryfill(x+1,y,f_c,b_c);
        boundaryfill(x,y+1,f_c,b_c);
        boundaryfill(x-1,y,f_c,b_c);
        boundaryfill(x,y-1,f_c,b_c);
    }
}

int main()
{
    int gm,gd=DETECT,radius,x,y;
    printf("Enter x and y co-ordinates for circle : ");
    scanf("%d %d",&x,&y);
    printf("Enter radius of the circle : ");
    scanf("%d",&radius);
    initgraph(&gd,&gm," ");
    circle(x,y,radius);
    rectangle(100,100,200,200);
    printf("Enter the value of x and y : ");
    scanf("%d %d",&x,&y);
    boundaryfill(x,y,5,15);
    delay(5000);
    closegraph();
    return 0;
}
```



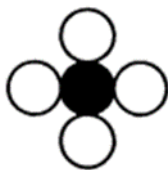
Output:



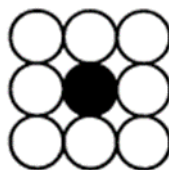
2) Flood Fill algorithm –

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.



(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;
}

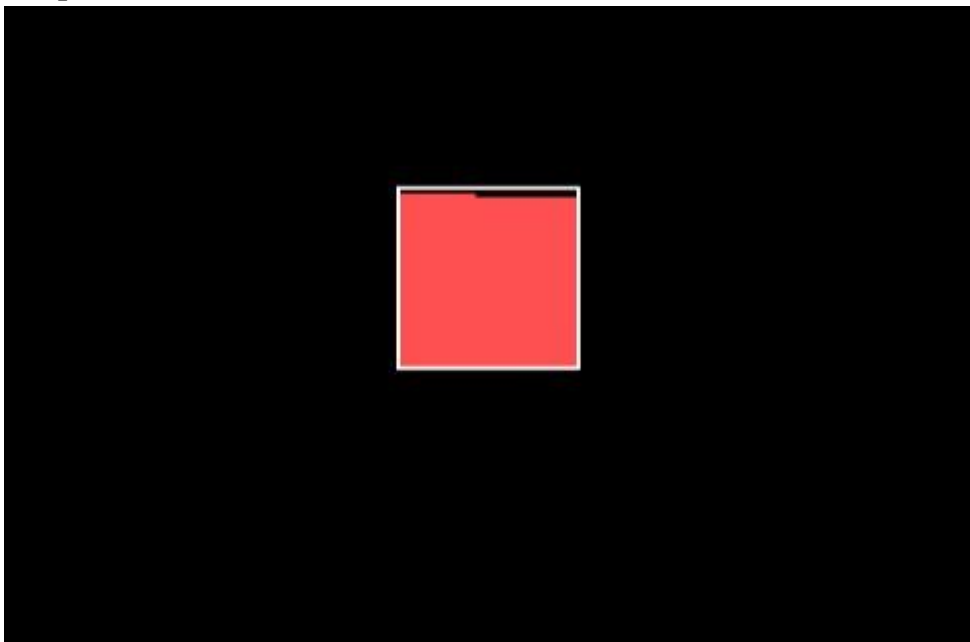
void flood(int x,int y, int fill_col, int old_col)
{

```



```
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



Experiment No. 6

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

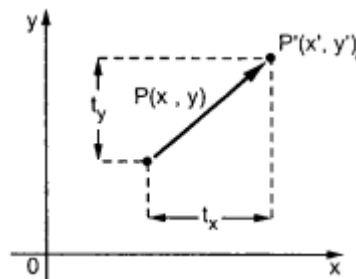
Objective:

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. t_x denotes translation distance along x-axis and t_y 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:

$$x' = x + t_x$$

$$y' = y + t_y$$

We denote translation transformation as T . 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," ");  
    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 –

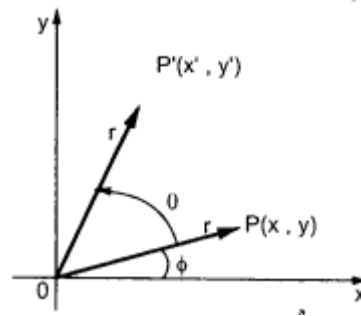
```
Program for basic transactions
Enter the points of triangle200 300
300 200
100 250

Enter the translation factor100 150
```

The image shows two identical triangles drawn in blue lines on a black background. The top triangle has vertices at approximately (100, 250), (200, 300), and (300, 200). The bottom triangle is a translated version of the top one, with vertices at approximately (200, 400), (300, 450), and (400, 350). This visualizes the translation of the triangle by a factor of (100, 150).

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 θ . 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

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

$$P' = P \cdot R$$

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," ");

    printf("\t Program for basic transactions");
```



```
printf("\n\t Enter the points of triangle");

setcolor(1);

scanf("%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();

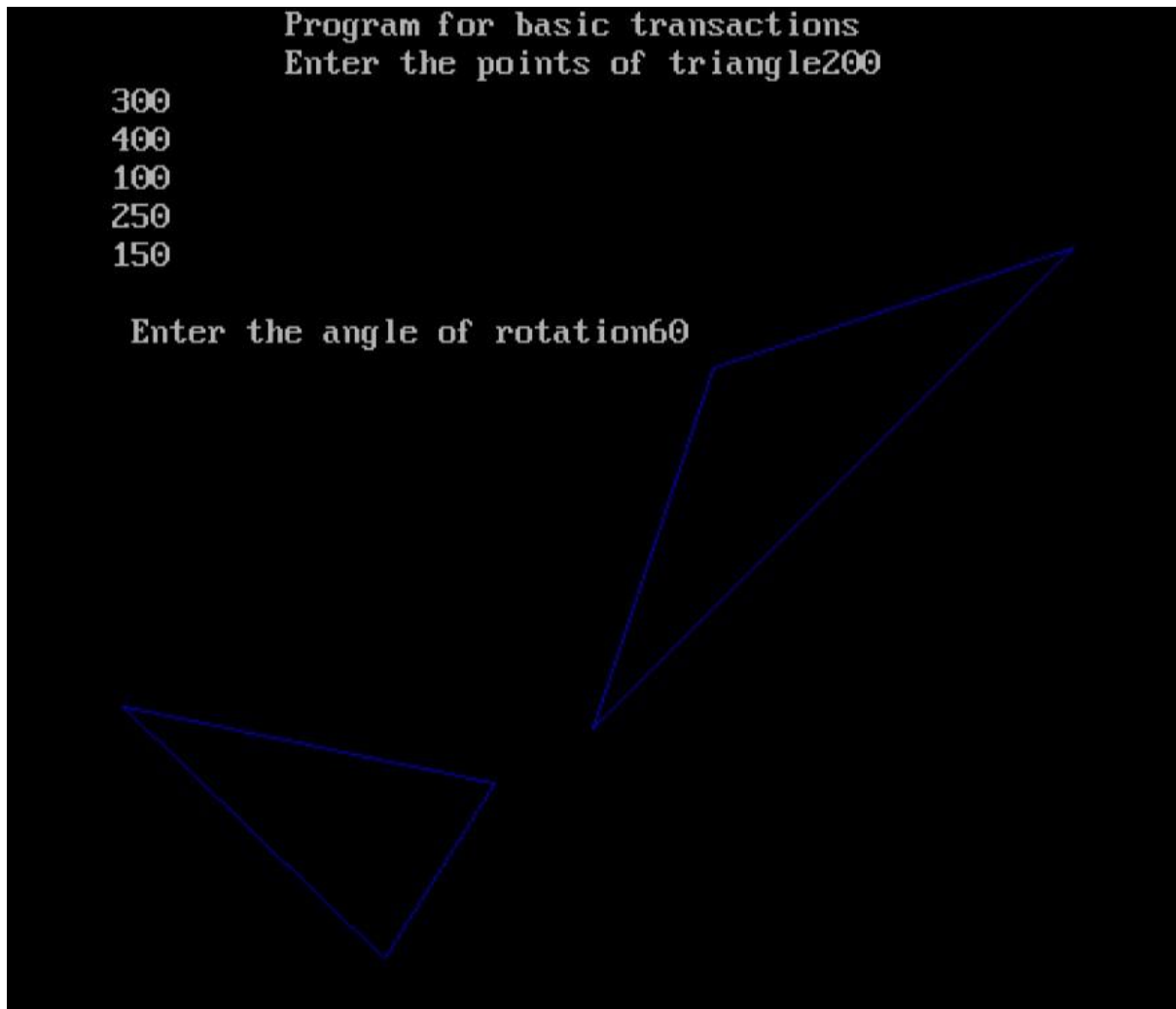
closegraph();

return 0;

}
```

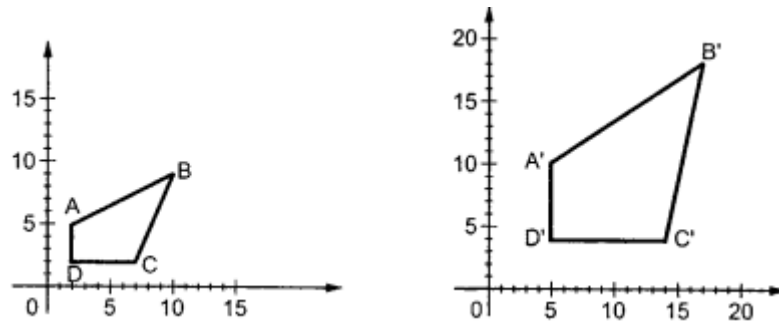


Output:



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:

$$x' = x \cdot S_x$$

$$y' = y \cdot S_y$$

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

$$\begin{aligned} [x' \ y'] &= [x \ y] \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \\ &= [x \cdot S_x \quad y \cdot S_y] \\ &= P \cdot S \end{aligned}$$

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," ");

    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 scalling factor");

scanf("%d%d",&sx,&sy);

nx1=x1*sx;

ny1=y1*sy;

nx2=x2*sx;

ny2=y2*sy;

nx3=x3*sx;

ny3=y3*sy;

line(nx1,ny1,nx2,ny2);

line(nx2,ny2,nx3,ny3);

line(nx3,ny3,nx1,ny1);

getch();

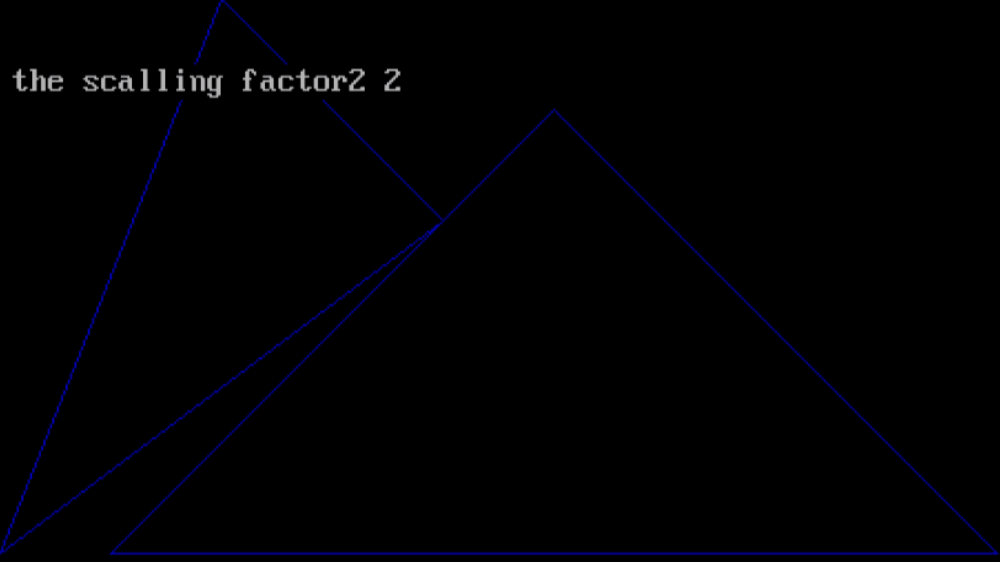
closegraph();

}
```




Output –

```
Program for basic transactions
Enter the points of triangle50 300
250 150
150 50
Enter the scalling factor2 2
```



Conclusion: Comment on :

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



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:

$$x = x_1 + tdx$$

$$y = y_1 + tdy, 0 \leq t \leq 1$$

Where $dx = x_2 - x_1$ & $dy = y_2 - y_1$

Algorithm

1. Read 2 endpoints of line as $p_1 (x_1, y_1)$ & $p_2 (x_2, y_2)$.
2. Read 2 corners (left-top & right-bottom) of the clipping window as $(x_{wmin}, y_{wmin}, x_{wmax}, y_{wmax})$.

3. Calculate values of parameters p_i and q_i for $i = 1, 2, 3, 4$ such that

$$p_1 = -dx, q_1 = x_1 - x_{wmin}$$

$$p_2 = dx, q_2 = x_{wmax} - x_1$$

$$p_3 = -dy, q_3 = y_1 - y_{wmin}$$

$$p_4 = dy, q_4 = y_{wmax} - y_1$$

4. if $p_i = 0$ then line is parallel to i th boundary

if $q_i < 0$ 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 q_i/p_i for $i = 1, 2, 3, 4$.

7. Select values of q_i/p_i where $p_i < 0$ and assign maximum out of them as $t1$.

8. Select values of q_i/p_i where $p_i > 0$ 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);  
}  
}
```



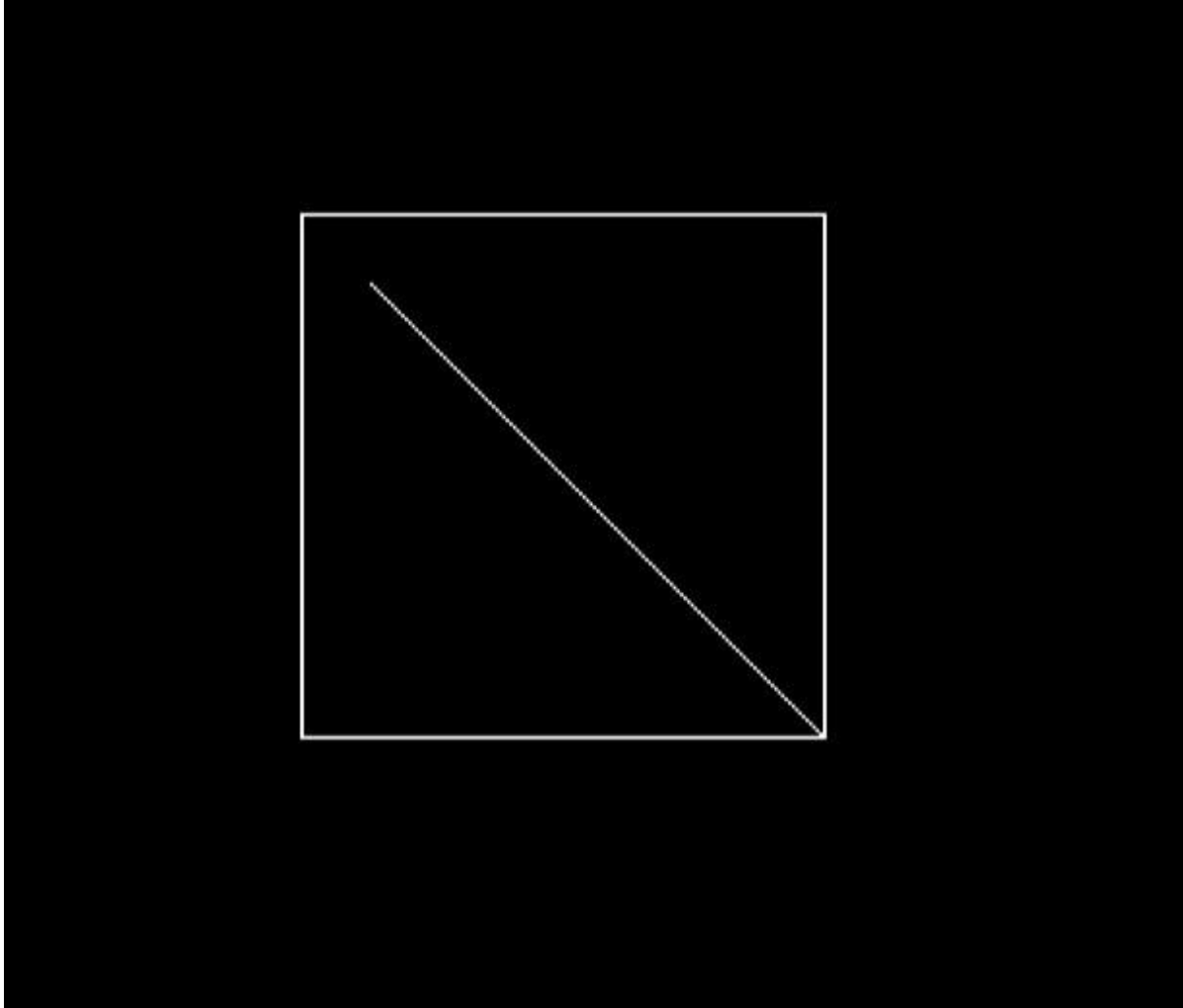
```
}  
}  
}  
t1=0;  
t2=1;  
for(i=0;i<4;i++)  
{  
temp=q[i]/p[i];  
if(p[i]<0)  
{  
if(t1<=temp)  
t1=temp;  
}  
else  
{  
if(t2>temp)  
t2=temp;  
}  
}  
if(t1<t2)  
{  
xx1 = x1 + t1 * p[1];  
xx2 = x1 + t2 * p[1];  
yy1 = y1 + t1 * p[3];  
yy2 = y1 + t2 * p[3];  
line(xx1,yy1,xx2,yy2);  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



```
closegraph();
```

```
}
```

Output:



Conclusion:



Experiment No. 8

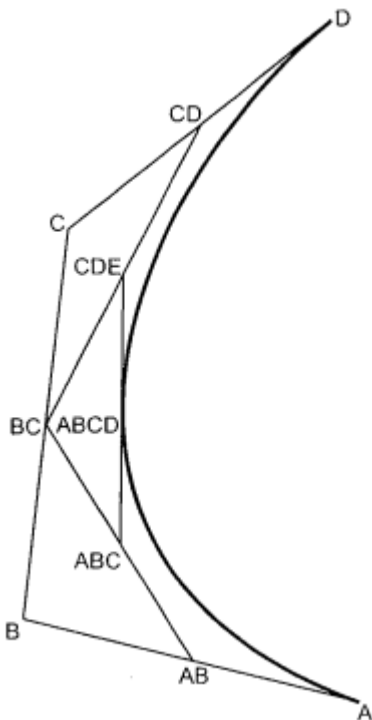
Aim: To implement Bezier curve for n control points. (Midpoint approach)

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.

$$x_{ab} = (x_a + x_b) / 2$$

$$y_{ab} = (y_a + y_b) / 2$$

$$x_{bc} = (x_b + x_c) / 2$$

$$y_{bc} = (y_b + y_c) / 2$$

$$x_{cd} = (x_c + x_d) / 2$$

$$y_{cd} = (y_c + y_d) / 2$$

$$x_{abc} = (x_{ab} + x_{bc}) / 2$$

$$y_{abc} = (y_{ab} + y_{bc}) / 2$$

$$x_{bcd} = (x_{bc} + x_{cd}) / 2$$

$$y_{bcd} = (y_{bc} + y_{cd}) / 2$$

$$x_{abcd} = (x_{abc} + x_{bcd}) / 2$$

$$y_{abcd} = (y_{abc} + y_{bcd}) / 2$$

- 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++)

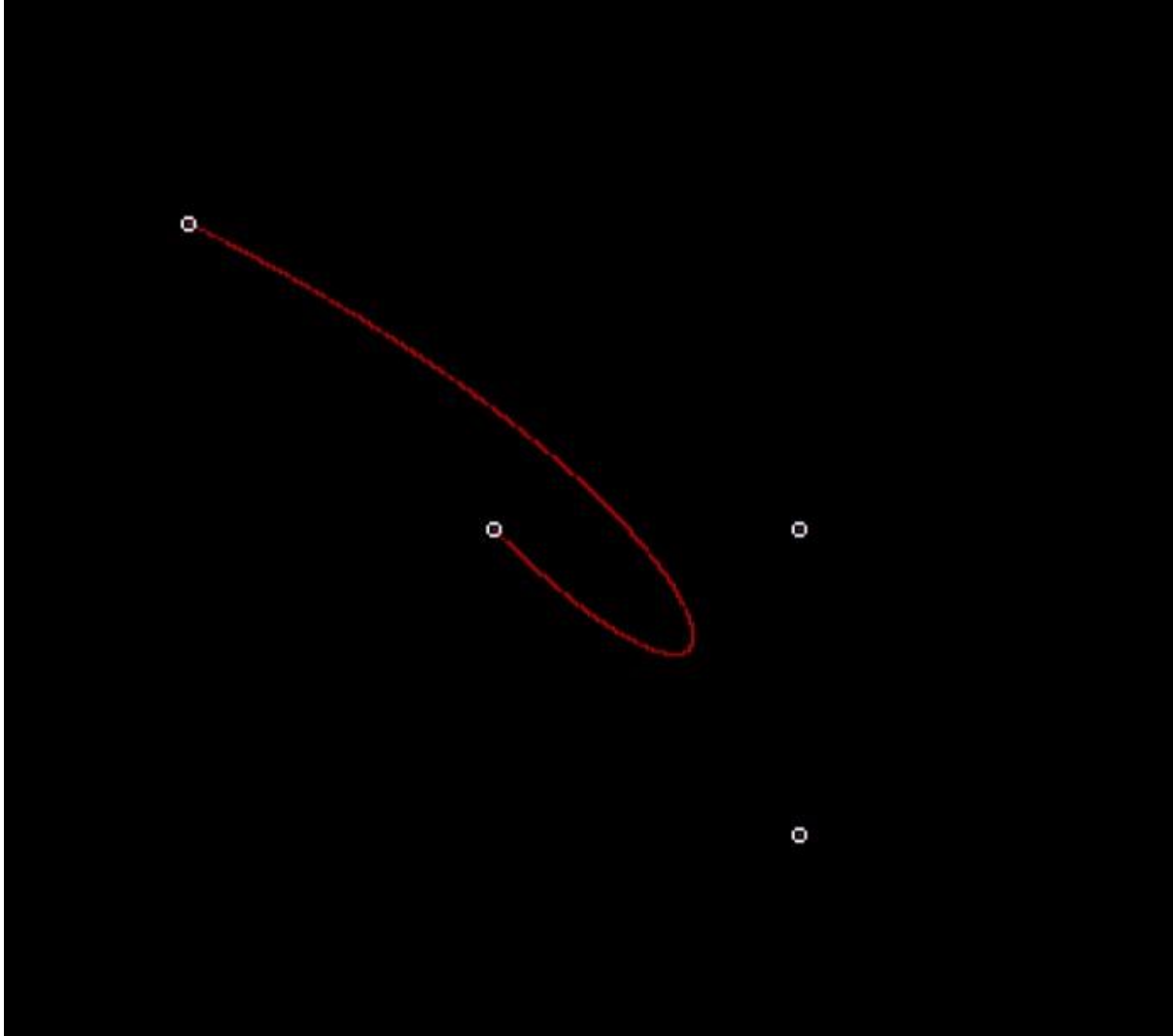
    {
        scanf("%d %d",&x[i],&y[i]);
    }
}
```



```
bezier(x,y);
```

```
}
```

Output:



Conclusion – Comment on

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



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

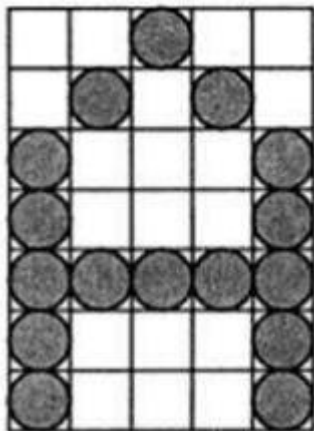
Theory:

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.

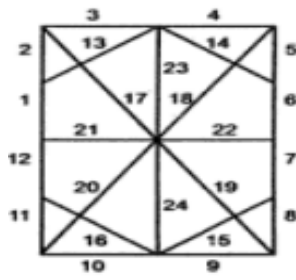
- In bit matrix method when the dots are stored in the form of array the value 1 in array represent the characters i.e. where the dots appear we represent that position with numerical value 1 and the value where dots are not present is represented by 0 in array.
- It is also called dot matrix because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

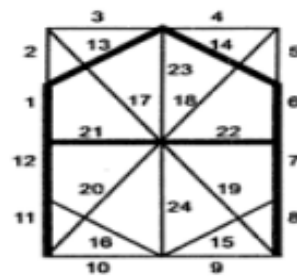


Starburst method –

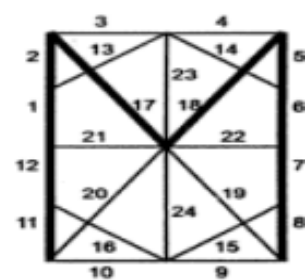
In this method a fix pattern of line segments is used to generate characters. Out of these 24-line segments, segments required to display for particular character are highlighted. This method of character generation is called starburst method because of its characteristic appearance. The starburst patterns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

```
    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},  
                {1,1,1,1,1,1,1,1,1,1},  
                {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
2. advantage of stroke method
3. one limitation



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:

```
#include <stdio.h>

#include <conio.h>

#include <graphics.h>

#include <dos.h>

void main()

{

clrscr

{

int gd=DETECT,gm,i;

initgraph(&gd,&gm,"C:\\TURBOC3\\BGI");

for(i=0;i<=100;i++)

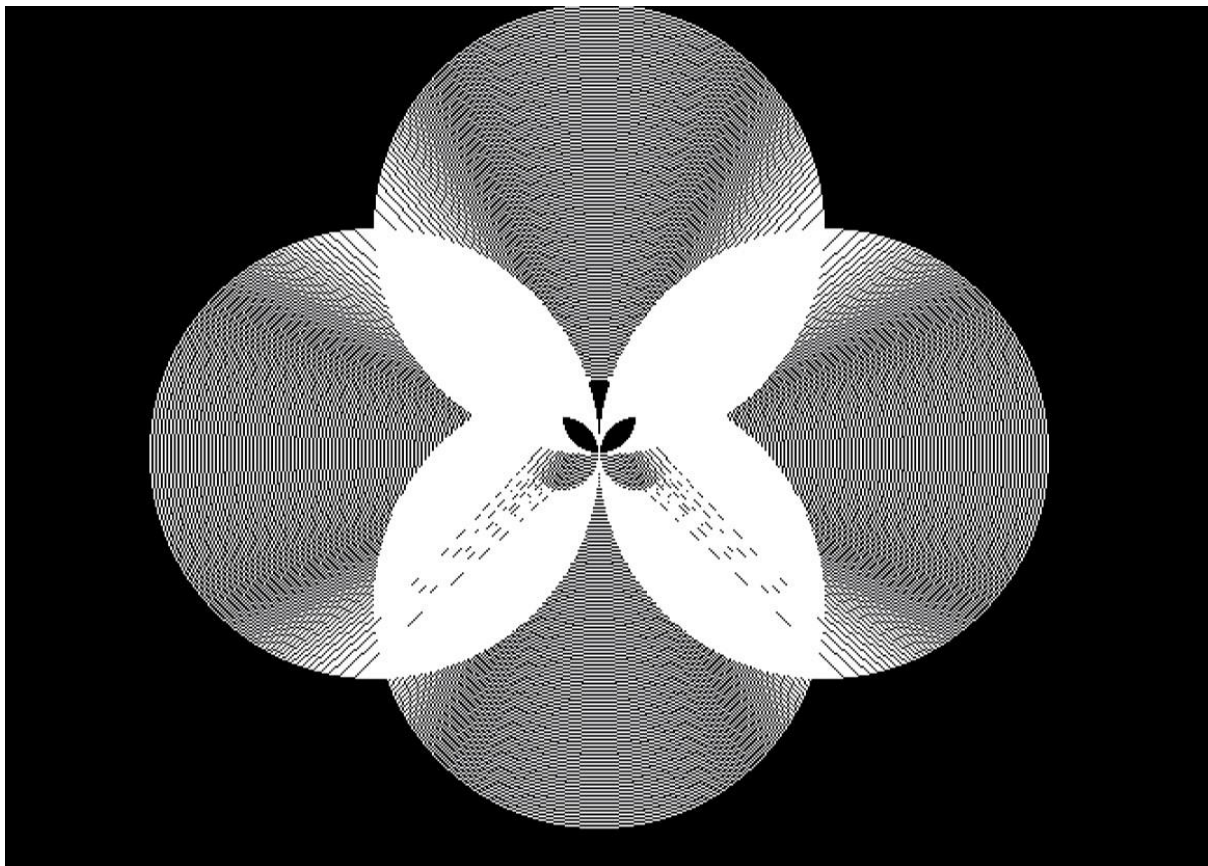
{

circle(319,219-i,20+i);
```



```
circle(319,219+i,20+i);  
circle(299-i,239,20+i);  
circle(339+i,239,20+i);  
delay(100);  
}  
getch();  
}  
}
```

Output:



Conclusion - Comment on :

1. Importance of story building
2. Defining the basic character of story



Vidyavardhini's College of Engineering & Technology

Department of Artificial Intelligence and Data Science

3. Apply techniques to these characters