Computer Graphics Lab (MC318)

Delhi Technological University

Submitted By:

Sidharth
2K18/MC/114

Program 1

Aim: Write a program to draw/create a hut using DDA line algorithm.

Theory:

A digital differential analyzer (DDA) is hardware or software used
for interpolation of variables over an interval between start and end point. DDAs are used for rasterization of lines, triangles and polygons. They can be extended to nonlinear functions, such as perspective correct texture mapping, quadratic curves, and traversing voxels.

Algorithm:

A linear DDA starts by calculating the smaller of dy or dx for a unit increment of the other. A line is then sampled at unit intervals in one coordinate and corresponding integer values nearest the line path are determined for the other coordinate.

Considering a line with positive slope, if the slope is less than or equal to 1, we sample at unit x intervals (dx=1) and compute successive y values as

$$y_{k+1} = y_k + m$$

$$x_{k+1} = x_k + 1$$

Subscript k takes integer values starting from 0, for the 1st point and increases by 1 until endpoint is reached. y value is rounded off to nearest integer to correspond to a screen pixel. For lines with slope greater than 1, we reverse the role of x and y i.e. we sample at dy=1 and calculate consecutive x values as

$$x_{k+1} = x_k + \frac{1}{m}$$

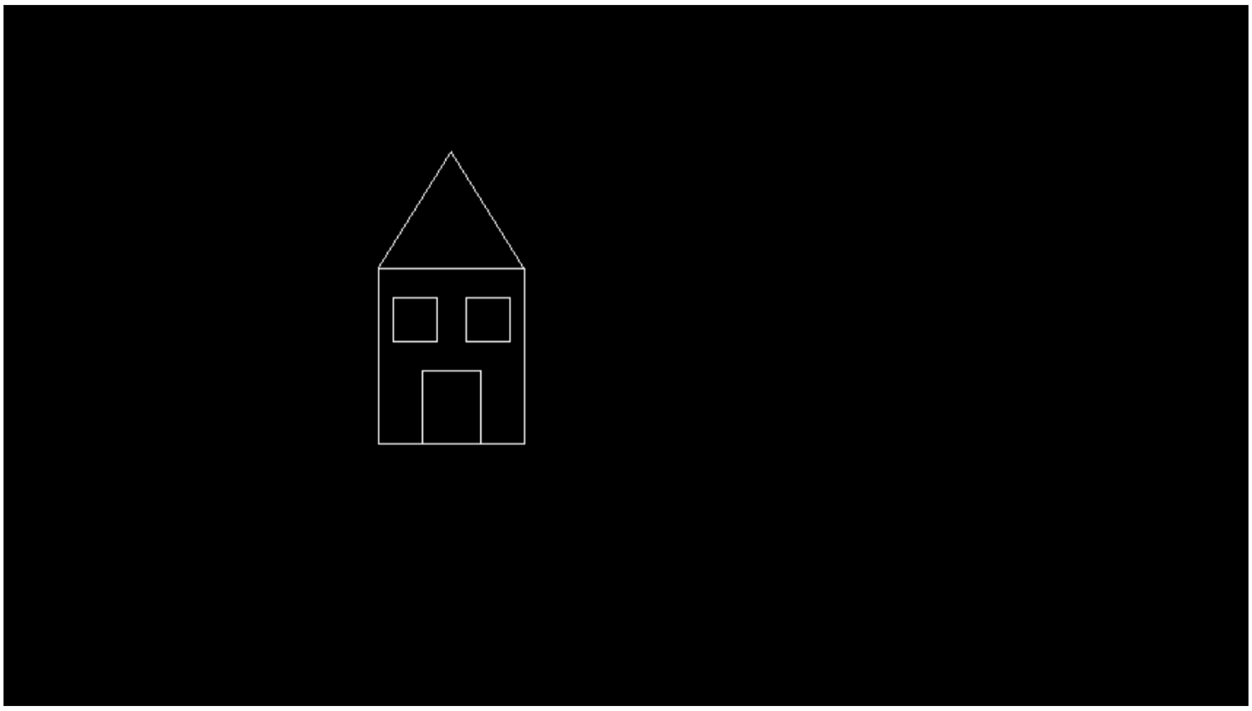
$$y_{k+1} = y_k + 1$$

Code:

```
#include<graphics.h>

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

int abs(int n){
   return ((n>0) ? n : ( n *(-1)));
}

void DDA(int X0, int Y0, int X1, int Y1){

   int dx = X1 - X0;
   int dy = Y1 - Y0;

   int steps = abs(dx) > abs(dy) ? abs(dx) : abs(dy);

   float x_inc = dx / (float) steps;
   float y_inc = dy / (float) steps;

   float X = X0;
   float Y = Y0;
   for(int i = 0; i <= steps; i++){
      putpixel (X,Y,WHITE);
      X += x_inc;
      Y += y_inc;
   }
}

int main(){

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

      DDA(150,300,250,300);
      DDA(150,180,250,180);
      DDA(150,300,150,180);
      DDA(250,300,250,180);

      DDA(180,300,220,300);
      DDA(180,250,220,250);
      DDA(180,300,180,250);
      DDA(220,300,220,250);
```

```
        DDA(160,230,190,230);
        DDA(160,200,190,200);
        DDA(160,230,160,200);
        DDA(190,230,190,200);

        DDA(210,230,240,230);
        DDA(210,200,240,200);
        DDA(210,230,210,200);
        DDA(240,230,240,200);

        DDA(200,100,150,180);
        DDA(200,100,250,180);

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

Output:



Program 2

Aim: Write a program to draw/create a car (without tyres) using Mid-Point Algorithm/Bresenham Algorithm.

Theory:

Bresenham's line algorithm is an algorithm that determines the points of an n-dimensional raster that should be selected in order to form a close approximation to a straight line between two points. It is commonly used to draw line primitives in a bitmap image, as it uses only integer addition, subtraction and bit shifting, all of which are very cheap operations in standard computer architectures. It is an incremental error algorithm. It is one of the earliest algorithms developed in the field of computer graphics.

Algorithm:

Step 1 − Input the two end-points of line, storing the left end-point in (X0,Y0).

Step 2 − Plot the point (X0,Y0).

Step 3 − Calculate the constants dx, dy, 2dy, and (2dy – 2dx) and get the first value for the decision parameter as −

$$\mathbf{p0 = 2dy - dx}$$

Step 4 − At each Xk along the line, starting at k = 0, perform the following -

If pkpk < 0, the next point to plot is (Xk+1,Yk) and

$$\mathbf{Pk + 1 = Pk + 2dy}$$

Otherwise, (Xk,Yk+1)

$$\mathbf{Pk + 1 = Pk + 2dy - 2dx}$$

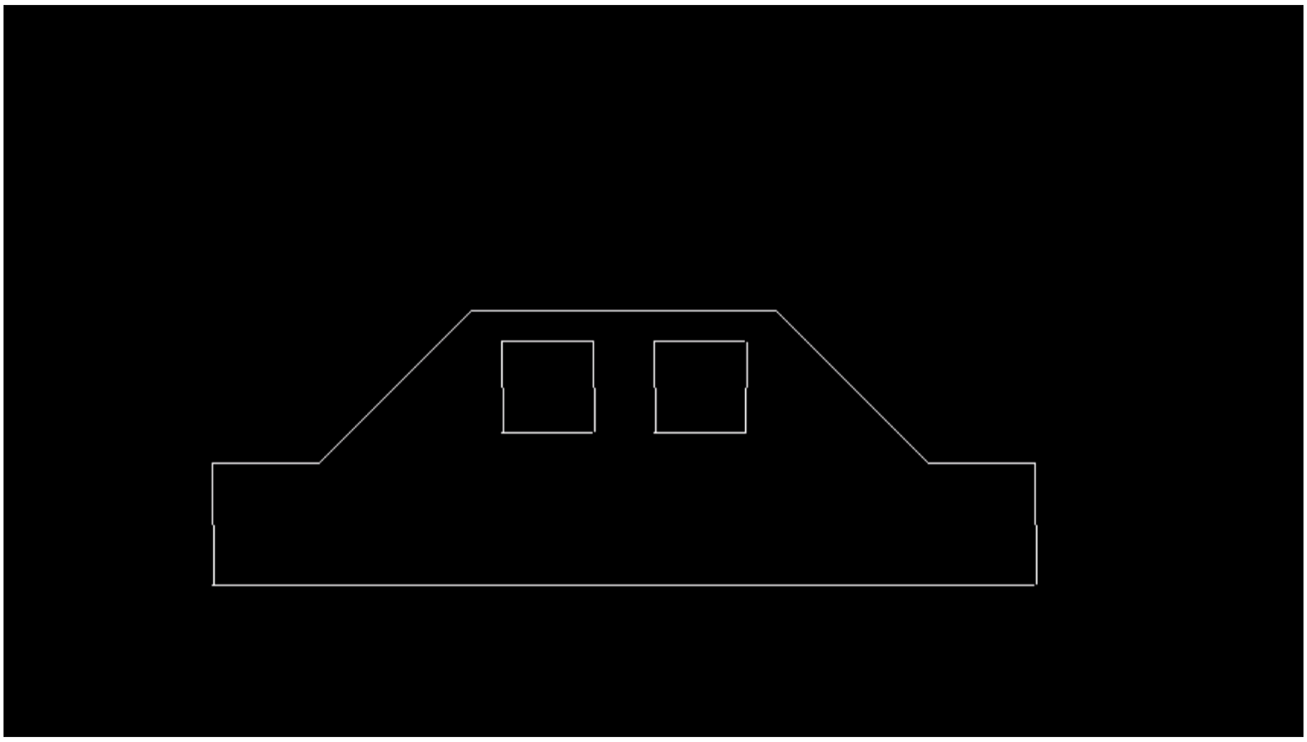
Step 5 − Repeat Step 4 (dx – 1) times.

Code:

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

void midPointLineAlgo(int X1, int Y1, int X2, int Y2){

    int dx, dy, lastX, lastY, i, nextX, nextY;
    int p, x, y, t, d;
    if(X2<X1){
        t = X1;
        X1 = X2;
        X2 = t;
        t = Y1;
        Y1 = Y2;
        Y2 = t;
    }
    dx = X2 - X1;
    dy = Y2 - Y1;

    if(dx == 0)
        dx = 1;

    lastX = X1;
    lastY = Y1;

    if(abs(dy)>abs(dx)){
        d = 2 * abs(dx) - abs(dy);
        if(dy>0 && dx>0){
            for (i=1; i<=abs(dy); i++){
                nextY = lastY + 1;
                if (d>0){
                    nextX = lastX + 1;
                    d = d + 2 * abs(dx) - 2 * abs(dy);
                }else{
                    nextX = lastX;
                    d = d + 2 * abs(dx);
                }
                putpixel(lastX, lastY, WHITE);
                lastX = nextX;
                lastY = nextY;
            }
        }else{
            for(i=1; i<=abs(dy); i++){
                nextY = lastY - 1;
                if(d<0){
```

```c
                    nextX = lastX;
                    d = d + 2 * abs(dx);
                }else{
                    nextX = lastX + 1;
                    d = d + 2 * abs(dx) - 2 * abs(dy);
                }
                putpixel(lastX, lastY, WHITE);
                lastX = nextX;
                lastY = nextY;
            }
        }
    }else{
        d = 2 * abs(dy) - abs(dx);
        if(dy>0 && dx>0){
            for(i=1; i<=abs(dx); i++){
                nextX = lastX + 1;
                if(d<0){
                    nextY = lastY;
                    d = d + 2 * abs(dy);
                }else{
                    nextY = lastY + 1;
                    d = d + 2 * abs(dy) - 2 * abs(dx);
                }
                putpixel(lastX, lastY, WHITE);
                lastX = nextX;
                lastY = nextY;
            }
        }else{
            for(i=1; i<=abs(dx); i++){
                nextX = lastX + 1;
                if(d>=0){
                    nextY = lastY - 1;
                    d = d + 2 * abs(dy) - 2 * abs(dx);
                }else{
                    nextY = lastY;
                    d = d + 2 * abs(dy);
                }
                putpixel(lastX, lastY, WHITE);
                lastX = nextX;
                lastY = nextY;
            }
        }
    }
}

int main(){
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");
    midPointLineAlgo(200,200,400,200);
    midPointLineAlgo(400,200,500,300);
    midPointLineAlgo(200,200,100,300);

    midPointLineAlgo(500,300,570,300);
    midPointLineAlgo(570,300,570,380);
    midPointLineAlgo(100,300,30,300);
    midPointLineAlgo(30,380,570,380);
    midPointLineAlgo(30,300,30,380);

    midPointLineAlgo(220,220,280,220);
    midPointLineAlgo(280,220,280,280);
    midPointLineAlgo(220,220,220,280);
    midPointLineAlgo(280,280,220,280);

    midPointLineAlgo(320,220,380,220);
    midPointLineAlgo(320,220,320,280);
    midPointLineAlgo(320,280,380,280);
    midPointLineAlgo(380,280,380,220);

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

Output:



Program 3

Aim: Write a program to draw/create a teddy bear (circular face tummy eyes and mouth) using Midpoint Circle drawing Algorithm.

Theory:

As in the raster line algorithm, we sample at unit intervals and determine the closest pixel position to the specified circle path at each step. For a given radius r and screen center position (xc, yc), we can first set up our algorithm to calculate pixel positions around a circle path centered at the coordinate origin (0, 0). Then each calculated position (x, y) is moved to its proper screen position by adding x to xc and yto yc.

Along the circle section from x = 0 to x = y in the first quadrant, the slope of the curve varies from 0 to -1. Therefore, we can take unit steps in the positive x direction over this octant and use a decision parameter to determine which of the two possible y positions is closer to the circle path.

Algorithm:

1. Calculate the initial value of the decision parameter as

p0=5/4-r

2. At each xk position, starting at k = 0, perform the following test: If pk< 0, the next point along the circle centered on (0, 0) is (xk+ 1, yk) and

Pk+1 = Pk + 2xk+1+ 1

Otherwise, the next point along the circle is (xk + 1, yk - I) and where

2xk+1 = 2xk+ 2 and 2yk+1 == 2yk - 2.

3. Determine symmetry points in the other seven octants.

4. Move each calculated pixel position (x, y) onto the circular path centered on (xc, yc) and plot the coordinate values:

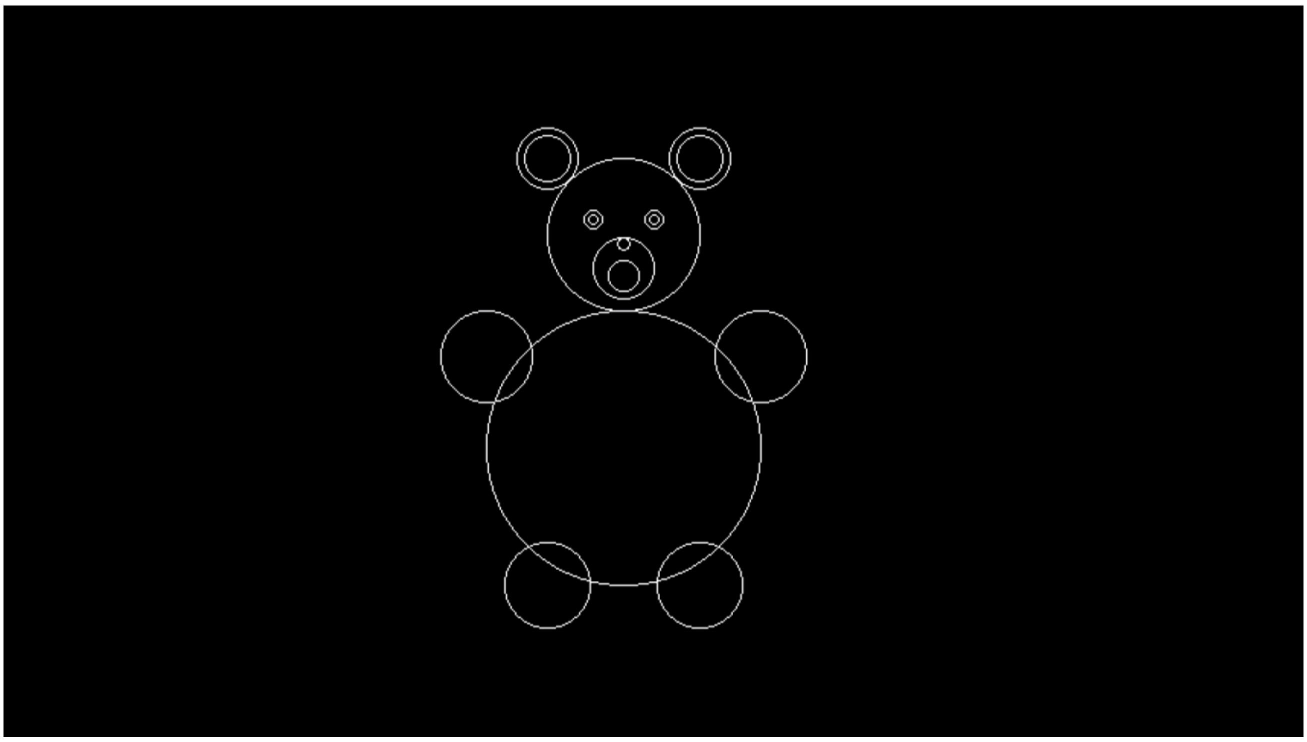
y=y+yc and x =x+xc

5. Repeat steps 3 through 5 until x > y.

Code:

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

void setpixel(int xc, int yc, int x, int y){
    putpixel(xc + x, yc + y, 15);
    putpixel(xc + x, yc - y, 15);
    putpixel(xc - x, yc + y, 15);
    putpixel(xc - x, yc - y, 15);
    putpixel(xc + y, yc + x, 15);
    putpixel(xc + y, yc - x, 15);
    putpixel(xc - y, yc + x, 15);
    putpixel(xc - y, yc - x, 15);
}

void midptcircle(int xc, int yc, int r){
    int p = 1 - r;
    int x = 0, y = r;
    setpixel(xc, yc, x, y);
    while(x<y){
        x++;
        if(p<0){
            p += 2 * x + 1;
        }else{
            y--;
            p += 2 * (x - y) + 1;
        }
        setpixel(xc, yc, x, y);
    }
}

int main(){

    int g_mode,g_driver=DETECT;
    initgraph(&g_driver,&g_mode,"C:\\TURBOC3\\BGI");
    //body
    midptcircle(300, 290, 90);

    //face
    midptcircle(300, 150, 50);

    //legs
    midptcircle(250, 380, 28);
    midptcircle(350, 380, 28);

    //hands
    midptcircle(210, 230, 30);
    midptcircle(390, 230, 30);

    //ears
    midptcircle(250, 100, 20);
    midptcircle(250, 100, 15);
    midptcircle(350, 100, 20);
    midptcircle(350, 100, 15);

    //eyes
    midptcircle(280, 140, 6);
    midptcircle(280, 140, 3);
    midptcircle(320, 140, 6);
    midptcircle(320, 140, 3);

    //mouth
    midptcircle(300, 172, 20);
    midptcircle(300, 156, 4);
    midptcircle(300, 177, 10);

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

Output:



Program 4

Aim: Write a program to draw/create a teddy bear (elliptical face, eyes and mouth and circular tummy) using Midpoint Ellipse drawing Algorithm.

Theory:

Mid-Point Ellipse Drawing Algorithm uses symmetry of ellipse to draw an ellipse in computer graphics. This algorithm is implemented for one quadrant only. We divide the quadrant into two regions and the boundary of two regions is the point at which the curve has a slope of -1. We process by taking unit steps in the x direction to the point P (where curve has a slope of -1), then taking unit steps in the y direction and applying midpoint algorithm at every step.

Algorithm:

1. Input rx’ ry’, and ellipse center (xc’, yc’), and obtain the first point on an ellipse centered on the origin as

(x0, y0) = (0, ry)

2. Calculate the initial value of the decision parameter in region 1 as

p10 = r2y – r2xry + ¼ r2x

3. At each xk position in region 1, starting at k = 0 perform the following test: If plk < 0, the next point along the ellipse centered on (0, 0) is (xk + 1 ,yk) and

P1k+1 = p1k + 2r2yxk+1 + r2y

Otherwise, the next point along the circle is (xk +1 , yk -1) and

P1k+1 = p1k + 2r2yxk+1 - 2r2xyk+1 + r2y

with

2r2yxk+1 = 2r2yxk + 2r2y 2r2xyk+1 = 2r2xyk - 2r2x

And continue until 2r2yx >= 2r2xy.

4. Calculate the initial value of the decision parameter in region 2 using the last point (x0, y0) calculated in region 1 as

p20 = r2y(x0 + ½)2 + r2x(y0 – 1)2 – r2xr2y

5. At each yk position in region 2, starting at k = 0, perform the following test: If p2k> 0, the next point along the ellipse centered on (0, 0) is (xk , yk - 1) and

P2k+1 = p2k - 2r2xyk+1 + r2x

Otherwise, the next point along the circle is (xk + 1, yk - 1) and

P2k+1 = p2k + 2r2yxk+1 - 2r2xyk+1 + r2x

using the same incremental calculations for x and y as in region 1.

6. Determine symmetry points in the other three quadrants.

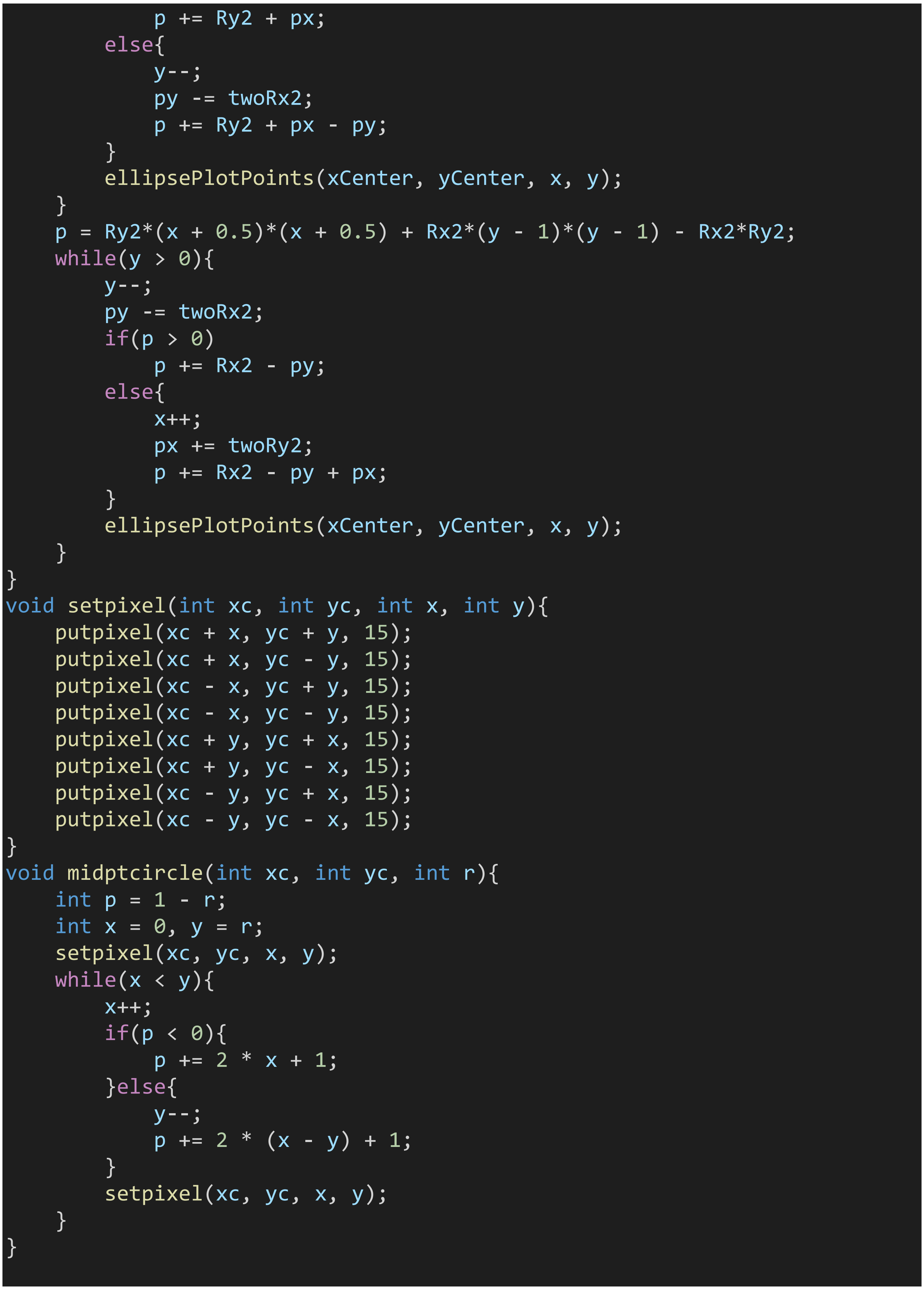
7. Move each calculated pixel position (x, y) onto the elliptical path centered on (xc , yc) and plot the coordinate values:

x = x + xc, y = y + yc

8. Repeat the steps for region 1 until 2r2yx >= 2r2xy.

Code:

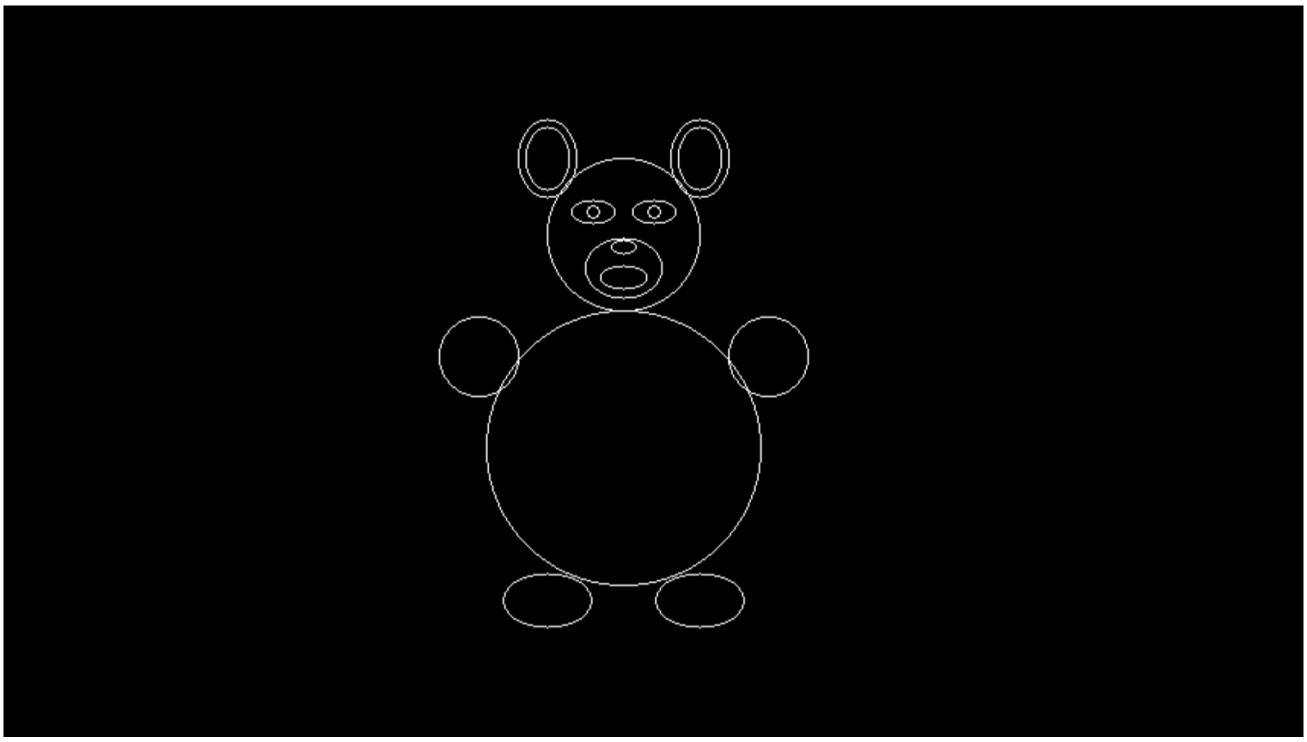
```c
#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
#include <conio.h>

void ellipsePlotPoints(int xCenter, int yCenter, int x, int y){
    putpixel(xCenter + x, yCenter + y, WHITE);
    putpixel(xCenter - x, yCenter + y, WHITE);
    putpixel(xCenter + x, yCenter - y, WHITE);
    putpixel(xCenter - x, yCenter - y, WHITE);
}
void midPointEllipse(int xCenter, int yCenter, int Rx, int Ry){
    int Rx2 = Rx * Rx;
    int Ry2 = Ry * Ry;
    int twoRx2 = 2 * Rx2;
    int twoRy2 = 2 * Ry2;
    int p, x = 0, y = Ry, px = 0;
    int py = twoRx2 * y;
    ellipsePlotPoints(xCenter, yCenter, x, y);
    p = Ry2 - (Rx2 * Ry) + (0.25 * Rx2);
    while(px < py){
        x++;
        px += twoRy2;
        if (p < 0)
```



```
            p += Ry2 + px;
        else{
            y--;
            py -= twoRx2;
            p += Ry2 + px - py;
        }
        ellipsePlotPoints(xCenter, yCenter, x, y);
    }
    p = Ry2*(x + 0.5)*(x + 0.5) + Rx2*(y - 1)*(y - 1) - Rx2*Ry2;
    while(y > 0){
        y--;
        py -= twoRx2;
        if(p > 0)
            p += Rx2 - py;
        else{
            x++;
            px += twoRy2;
            p += Rx2 - py + px;
        }
        ellipsePlotPoints(xCenter, yCenter, x, y);
    }
}
void setpixel(int xc, int yc, int x, int y){
    putpixel(xc + x, yc + y, 15);
    putpixel(xc + x, yc - y, 15);
    putpixel(xc - x, yc + y, 15);
    putpixel(xc - x, yc - y, 15);
    putpixel(xc + y, yc + x, 15);
    putpixel(xc + y, yc - x, 15);
    putpixel(xc - y, yc + x, 15);
    putpixel(xc - y, yc - x, 15);
}
void midptcircle(int xc, int yc, int r){
    int p = 1 - r;
    int x = 0, y = r;
    setpixel(xc, yc, x, y);
    while(x < y){
        x++;
        if(p < 0){
            p += 2 * x + 1;
        }else{
            y--;
            p += 2 * (x - y) + 1;
        }
        setpixel(xc, yc, x, y);
    }
}
```

```c
int main(){

    int g_mode, g_driver = DETECT;
    initgraph(&g_driver, &g_mode, "C:\\TURBOC3\\BGI");

    //body
    midptcircle(300, 290, 90);

    //face
    midptcircle(300, 150, 50);

    //legs
    midPointEllipse(250, 390, 30, 18);
    midPointEllipse(350, 390, 30, 18);

    //hands
    midptcircle(205, 230, 26);
    midptcircle(395, 230, 26);

    //ears
    midPointEllipse(250, 100, 20, 26);
    midPointEllipse(250, 100, 15, 21);

    midPointEllipse(350, 100, 20, 26);
    midPointEllipse(350, 100, 15, 21);

    //eyes
    midptcircle(280, 135, 4);
    midptcircle(320, 135, 4);
    midPointEllipse(280, 135, 15, 8);
    midPointEllipse(320, 135, 15, 8);

    //mouth
    midPointEllipse(300, 158, 8, 5);
    midPointEllipse(300, 178, 16, 8);
    midPointEllipse(300, 172, 26, 20);

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

Output:



Program 5

Aim: Write a program to clip line using Cohen Sutherland Algorithm.

Theory:

The **Cohen–Sutherland algorithm** is a computer-graphics algorithm used for line clipping. The algorithm divides a two-dimensional space into 9 regions and then efficiently determines the lines and portions of lines that are visible in the central region of interest (the viewport).

The algorithm was developed in 1967 during flight-simulator work by Danny Cohen and Ivan Sutherland.

Algorithm:

NEW

```
1. Calculate positions of both endpoints of the line.

2. Perform OR operation on both of these end-points.

3. If the OR operation gives 0000
      Then
            line is considered to be visible
      else
         Perform AND operation on both endpoints
     If And ≠ 0000
         then the line is invisible
       else
     And=0000
   Line is considered the clipped case.

4. If a line is clipped case, find an intersection with boundaries of the window
           $m=(y_2-y_1)(x_2-x_1)$

(a) If bit 1 is "1" line intersects with left boundary of rectangle window
            $y_3=y_1+m(x-X_1)$
            where $X = X_{wmin}$
            where $X_{wmin}$is the minimum value of X co-ordinate of window

(b) If bit 2 is "1" line intersect with right boundary
            $y_3=y_1+m(X-X_1)$
            where $X = X_{wmax}$
            where X more is maximum value of X co-ordinate of the window
```

(c) If bit 3 is "1" line intersects with bottom boundary

$X_3=X_1+(y-y_1)/m$

where $y = y_{wmin}$

y_{wmin} is the minimum value of Y co-ordinate of the window

(d) If bit 4 is "1" line intersects with the top boundary

$X_{3=x}1+(y-y_1)/m$

where $y = y_{wmax}$

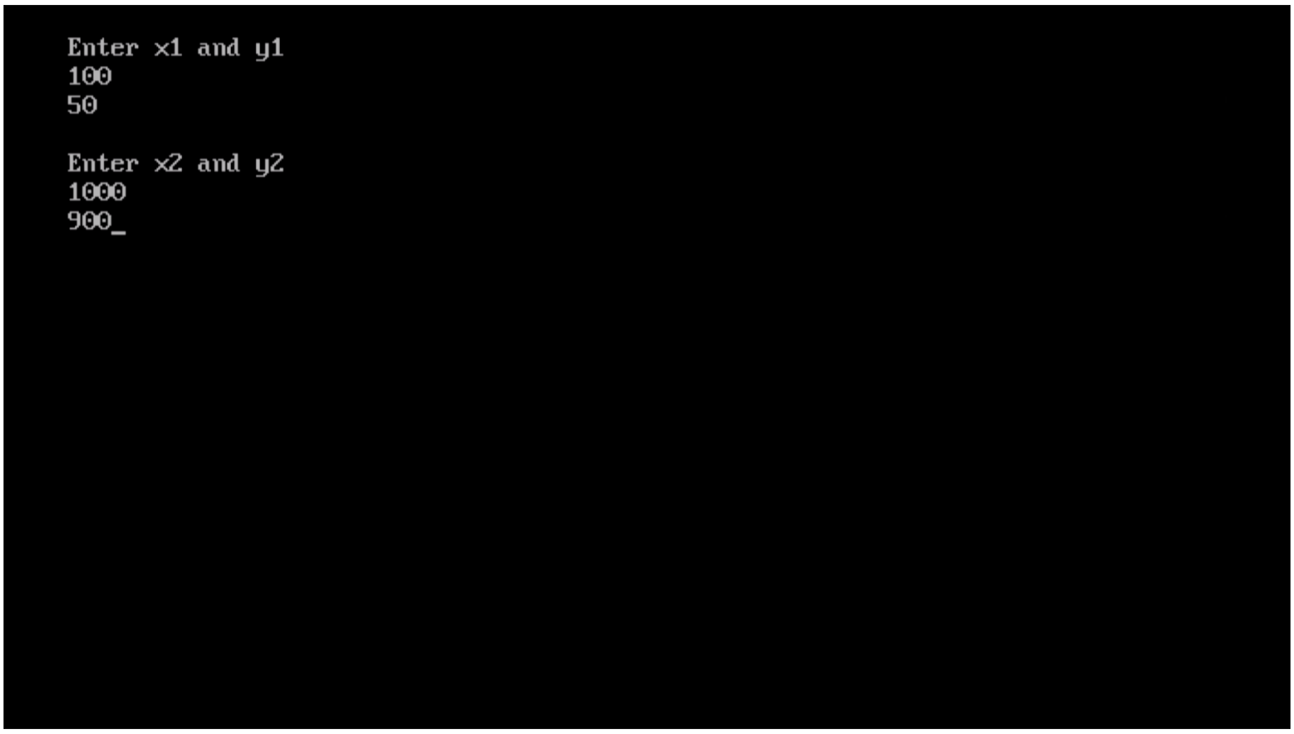
y_{wmax} is the maximum value of Y co-ordinate of the window

Code:

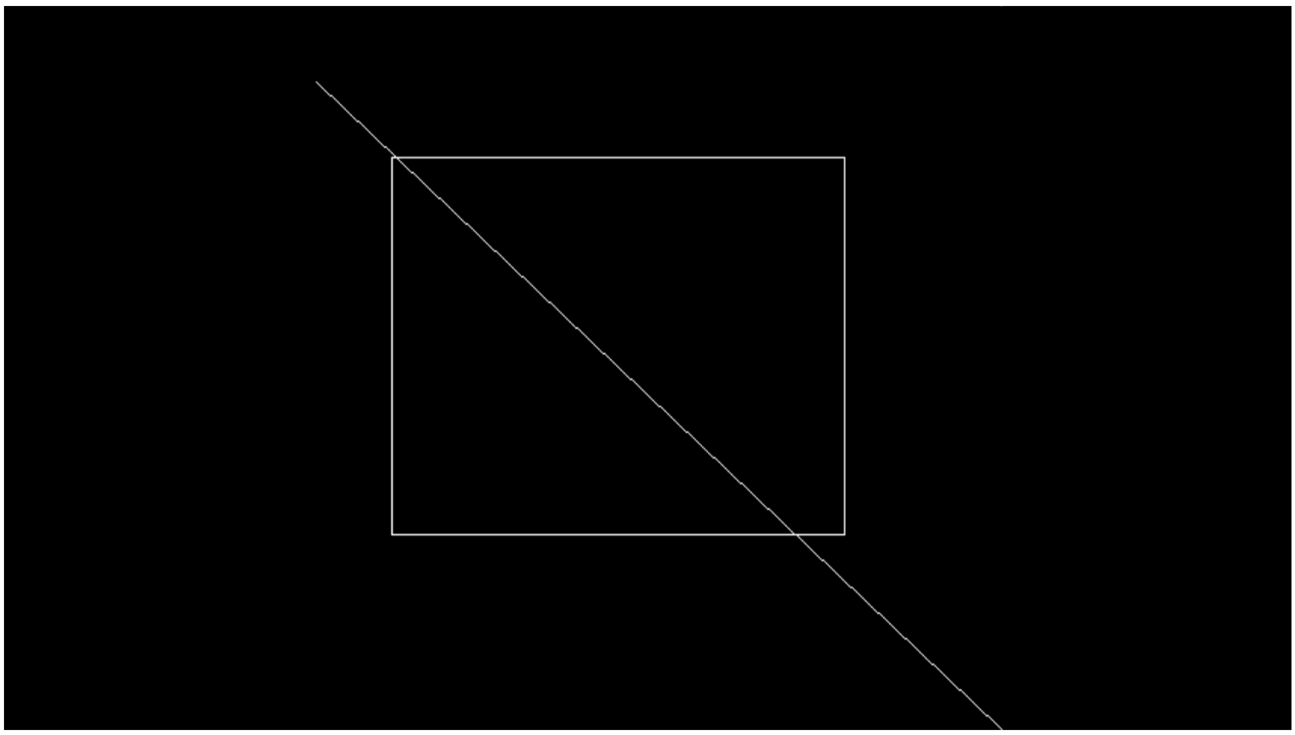
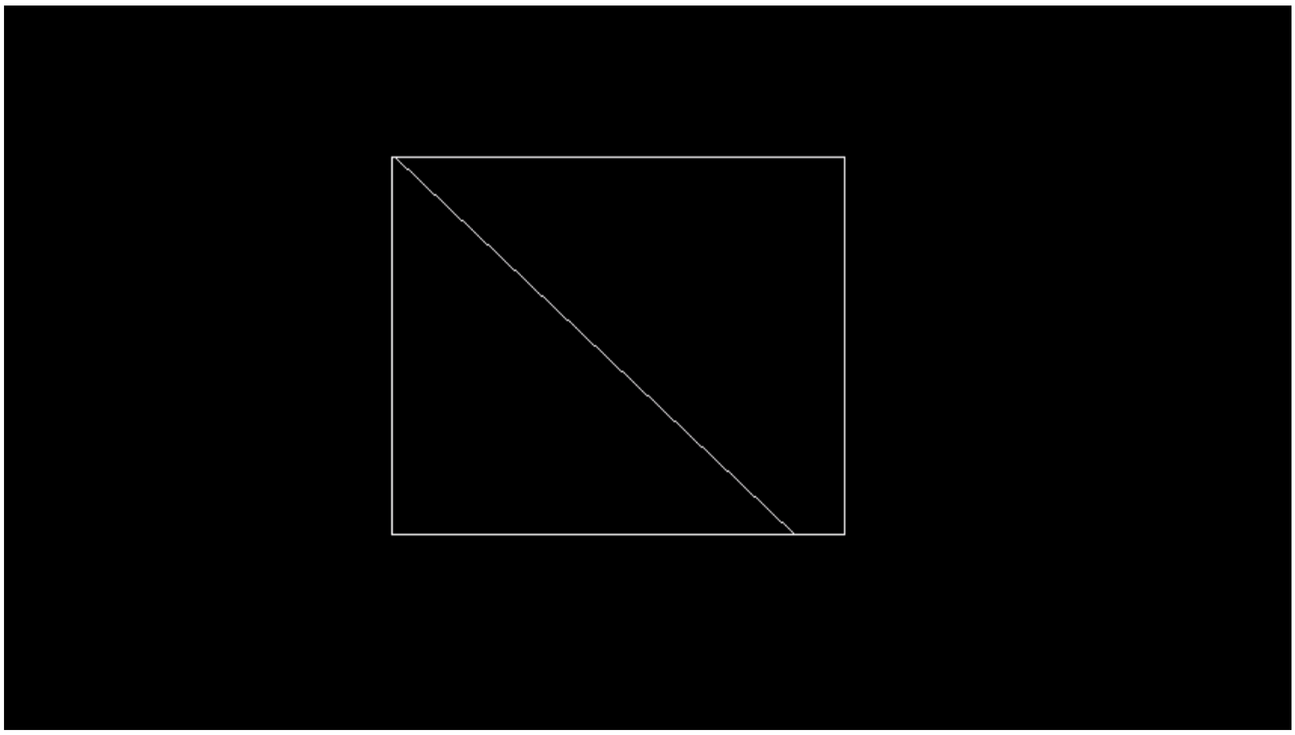
```
#include<iostream.h>
#include<stdlib.h>
#include<math.h>
#include<graphics.h>
#include<dos.h>

typedef struct coordinates{
    int x,y;
    char code[4];
}PT;

void drawwindow();
void drawline(PT p1,PT p2);
PT setcode(PT p);
int visibility(PT p1,PT p2);
PT resetendpt(PT p1,PT p2);

int main(){
    int v;
    PT p1,p2,p3,p4,ptemp;
    cout<<"\nEnter x1 and y1\n";
    cin>>p1.x>>p1.y;
    cout<<"\nEnter x2 and y2\n";
    cin>>p2.x>>p2.y;

    int g_mode, g_driver = DETECT;
    initgraph(&g_driver, &g_mode, "C:\\TURBOC3\\BGI");

    drawwindow();
    delay(500);

    drawline(p1,p2);
    delay(500);
    cleardevice();

    delay(500);
    p1=setcode(p1);
    p2=setcode(p2);
    v=visibility(p1,p2);
    delay(500);

    switch(v){
    case 0: drawwindow();
            delay(500);
            drawline(p1,p2);
            break;
    case 1: drawwindow();
            delay(500);
            break;
    case 2: p3=resetendpt(p1,p2);
            p4=resetendpt(p2,p1);
            drawwindow();
            delay(500);
            drawline(p3,p4);
            break;
    }

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

void drawwindow(){
    line(150,100,450,100);
    line(450,100,450,350);
    line(450,350,150,350);
    line(150,350,150,100);
}

void drawline(PT p1,PT p2){
    line(p1.x,p1.y,p2.x,p2.y);
```

```
}

PT setcode(PT p){
    PT ptemp;

    if(p.y<100)
        ptemp.code[0]='1';
    else
        ptemp.code[0]='0';

    if(p.y>350)
        ptemp.code[1]='1';
    else
        ptemp.code[1]='0';

    if(p.x>450)
        ptemp.code[2]='1';
    else
        ptemp.code[2]='0';

    if(p.x<150)
        ptemp.code[3]='1';
    else
        ptemp.code[3]='0';

    ptemp.x=p.x;
    ptemp.y=p.y;

    return(ptemp);
}

int visibility(PT p1,PT p2){
    int i,flag=0;

    for(i=0;i<4;i++){
        if((p1.code[i]!='0')||(p2.code[i]!='0'))
            flag=1;
    }

    if(flag==0)
        return(0);
    for(i=0;i<4;i++){
        if((p1.code[i]==p2.code[i])&&(p1.code[i]=='1'))
            flag='0';
    }

    if(flag==0)
        return(1);

    return(2);
}

PT resetendpt(PT p1,PT p2){
    PT temp;
    int x,y,i;
    float m,k;

    if(p1.code[3]=='1')
        x=150;

    if(p1.code[2]=='1')
        x=450;

    if((p1.code[3]=='1') || (p1.code[2]=='1')){
        m=(float)(p2.y-p1.y)/(p2.x-p1.x);
        k=(p1.y+(m*(x-p1.x)));
        temp.y=k;
        temp.x=x;

        for(i=0;i<4;i++)
            temp.code[i]=p1.code[i];

        if(temp.y<=350 && temp.y>=100)
            return (temp);
    }

    if(p1.code[0]=='1')
        y=100;

    if(p1.code[1]=='1')
        y=350;

    if((p1.code[0]=='1') || (p1.code[1]=='1')){
        m=(float)(p2.y-p1.y)/(p2.x-p1.x);
        k=(float)p1.x+(float)(y-p1.y)/m;
        temp.x=k;
        temp.y=y;

        for(i=0;i<4;i++)
            temp.code[i]=p1.code[i];

        return(temp);
    }else
        return(p1);
}
```

Output:



```
Enter x1 and y1
100
50

Enter x2 and y2
1000
900_
```

Program 6

Aim:

Write a program to clip line using Cyrus beck Algorithm. All the cases discussed in class has to be demonstrated.

Theory:

The Cyrus–Beck algorithm is a generalized line clipping algorithm. It was designed to be more efficient than the Cohen–Sutherland algorithm, which uses repetitive clipping.[1] Cyrus–Beck is a general algorithm and can be used with a convex polygon clipping window, unlike Sutherland–Cohen, which can be used only on a rectangular clipping area.

Here the parametric equation of a line in the view plane is

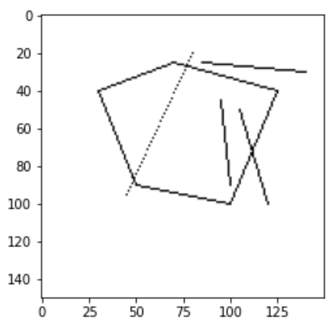
$$\mathbf{p}(t) = t\mathbf{p}_1 + (1 - t)\mathbf{p}_0$$

Algorithm:

- Normals of every edge is calculated.
- Vector for the clipping line is calculated.
- Dot product between the difference of one vertex per edge and one selected end point of the clipping line and the normal of the edge is calculated (for all edges).
- Dot product between the vector of the clipping line and the normal of edge (for all edges) is calculated.
- The former dot product is divided by the latter dot product and multiplied by -1. This is 't'.
- The values of 't' are classified as entering or exiting (from all edges) by observing their denominators (latter dot product).
- One value of 't' is chosen from each group, and put into the parametric form of a line to calculate the coordinates.
- If the entering 't' value is greater than the exiting 't' value, then the clipping line is rejected.

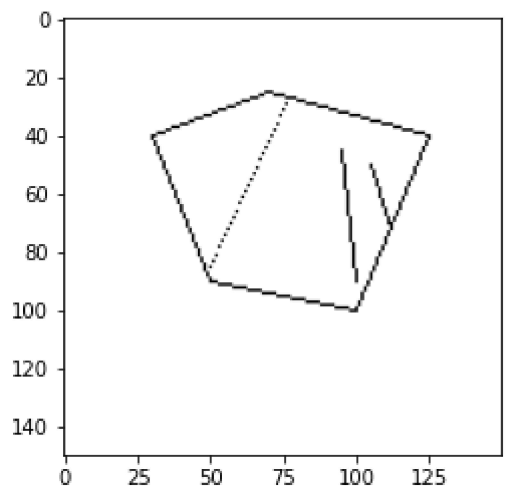
```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

def ROUND(a):

  return int(a + 0.5)
```

```
def drawDDA(x1,y1,x2,y2,img):

  x,y = x1,y1
  length = (x2-x1) if (x2-x1) > (y2-y1) else (y2-y1)

  dx = (x2-x1)/float(length)

  dy = (y2-y1)/float(length)

  img.putpixel((ROUND(x),ROUND(y)),1)

  for i in range(length):

    x += dx

    y += dy

    img.putpixel((ROUND(x),ROUND(y)),1)

img = Image.fromarray(np.zeros((150, 150), dtype=np.float32), mode= '

drawDDA(30,40,70,25,img)
drawDDA(70, 25,125, 40,img)
drawDDA(125, 40, 100, 100,img)
drawDDA(50,90,100,100,img)
drawDDA(30,40,50,90,img)

drawDDA(95,45,100,90,img)
drawDDA(45,95,80,20,img)
drawDDA(85,25,140,30,img)
drawDDA(105,50,120,100,img)
plt.imshow(np.array(img),cmap='binary')
plt.savefig('OP1')
```



```
Points = np.array([[30 , 40],
                   [70 , 25],
                   [125, 40],
                    [100,100],
                    [50 , 90],
                    [30 , 40]])

def perpendicular( a ) :
    b = np.empty_like(a)
    b[0] = -a[1]
    b[1] = a[0]
    return b

def normalize(a):
    a = np.array(a)
    return a/np.linalg.norm(a)


normals =[]

for i in range(5):
  vec= Points[i]-Points[i+1]
  temp=perpendicular(vec)
  temp=normalize(temp)
  normals.append(temp)




img2 = Image.fromarray(np.zeros((150, 150), dtype=np.float32), mode=

drawDDA(30,40,70,25,img2)
drawDDA(70, 25,125, 40,img2)
drawDDA(125, 40, 100, 100,img2)
drawDDA(50,90,100,100,img2)
drawDDA(30,40,50,90,img2)


def CyrusBeck(x1,y1,x2,y2,normal,points):

  P1=np.array([x2,y2])
  P0=np.array([x1,y1])

  tE_arr = [0]
  tL_arr  = [1]
  for i in range(len(normals)):

    Pe=points[i]
    n=normals[i]
    try:
      t_temp= -1*np.dot(n,P0-Pe)/(np.dot(n,P1-P0))
      denom= np.dot(n,P1-P0)

      if denom<0:
        tE_arr.append(t_temp)

      else:
        tL_arr.append(t_temp)

    except:
      continue

  tE=max(tE_arr)

  tL=min(tL_arr)

  print('tE= {} \ntL={} \n'.format(tE,tL) )
  if tL<tE:
    print("Line outside\n")

  else:

    x1_=ROUND(x1+(x2-x1)*tE)
    x2_=ROUND(x1+(x2-x1)*tL)
    y1_=ROUND(y1+(y2-y1)*tE)
    y2_=ROUND(y1+(y2-y1)*tL)

    drawDDA(x1_,y1_,x2_,y2_,img2)



CyrusBeck(95,45,100,90,normals,Points)
CyrusBeck(45,95,80,20,normals,Points)
CyrusBeck(85,25,140,30,normals,Points)
CyrusBeck(105,50,120,100,normals,Points)

plt.imshow(np.array(img2),cmap='binary')
plt.savefig('OP2')
```

```
tE= 0
tL=1

tE= 0.10769230769230767
tL=0.9086021505376345

tE= 0
tL=-0.4090909090909091

Line outside

tE= 0
tL=0.4418604651162791
```



Program 7

Aim: Write a program to fill the polygon using

a) Boundary Fill

b) Flood Fill

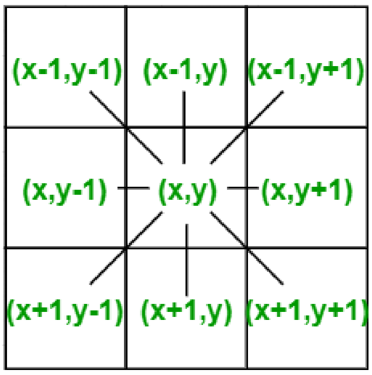
c) Scan Fill methods.

Theory:

Boundary Fill

The boundary fill algorithm works as its name. This algorithm picks a point inside an object and starts to fill until it hits the boundary of the object. The color of the boundary and the color that we fill should be different for this algorithm to work.

In this algorithm, we assume that color of the boundary is same for the entire object. The boundary fill algorithm can be implemented by 4-connected pixels or 8-connected pixels.

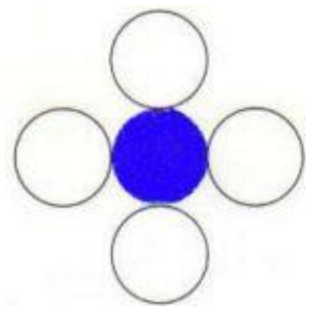


Flood Fill

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

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

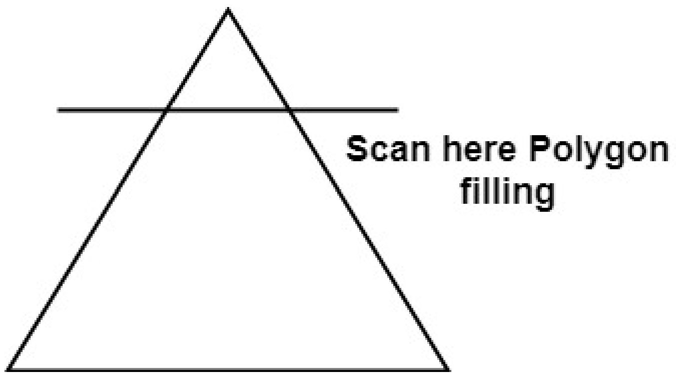
In Flood Fill algorithm we start with some seed and examine the neighboring pixels, however pixels are checked for a specified interior color instead of boundary color and is replaced by a new color. It can be done using 4 connected or 8 connected region method.


4 Connected Region

Scan Fill

This algorithm lines interior points of a polygon on the scan line and these points are done on or off according to requirement. The polygon is filled with various colors by coloring various pixels.

In above figure polygon and a line cutting polygon in shown. First of all, scanning is done. Scanning is done using raster scanning concept on display device. The beam starts scanning from the top left corner of the screen and goes toward the bottom right corner as the endpoint. The algorithms find points of intersection of the line with polygon while moving from left to right and top to bottom. The various points of intersection are stored in the frame buffer. The intensities of such points is keep high. Concept of coherence property is used. According to this property if a pixel is inside the polygon, then its next pixel will be inside the polygon.



Algorithm:

Boundary Fill

```
Procedure fill (x, y, color, color1: integer)
int c;
c=getpixel (x, y);
if (c!=color) (c!=color1) {
    setpixel (x, y, color);
    fill (x+1, y, color, color 1);
    fill (x-1, y, color, color 1);
    fill (x, y+1, color, color 1);
    fill (x, y-1, color, color 1);
}
```

Flood Fill

Step 1 − Initialize the value of seed point (seedx, seedy), fcolor and dcol.

Step 2 − Define the boundary values of the polygon.

Step 3 − Check if the current seed point is of default color, then repeat the steps 4 and 5 till the boundary pixels reached.

If getpixel(x, y) = dcol then repeat step 4 and 5

Step 4 − Change the default color with the fill color at the seed point.

setPixel(seedx, seedy, fcol)

Step 5 − Recursively follow the procedure with four neighborhood points.

FloodFill (seedx – 1, seedy, fcol, dcol)

FloodFill (seedx + 1, seedy, fcol, dcol)

FloodFill (seedx, seedy - 1, fcol, dcol)

FloodFill (seedx – 1, seedy + 1, fcol, dcol)

Step 6 – Exit

Scan Fill

Step 1 − Find out the Y_{min} and Y_{max} from the given polygon.

Step 2 – Scan Line intersects with each edge of the polygon from Y_{min} to Y_{max}. Name each intersection point of the polygon. As per the figure shown above, they are named as p0, p1, p2, p3.

Step 3 − Sort the intersection point in the increasing order of X coordinate i.e. (p0, p1), (p1, p2), and (p2, p3).

Step 4 − Fill all those pair of coordinates that are inside polygons and ignore the alternate pairs.

Step 5 − Exit

Code:

Boundary Fill

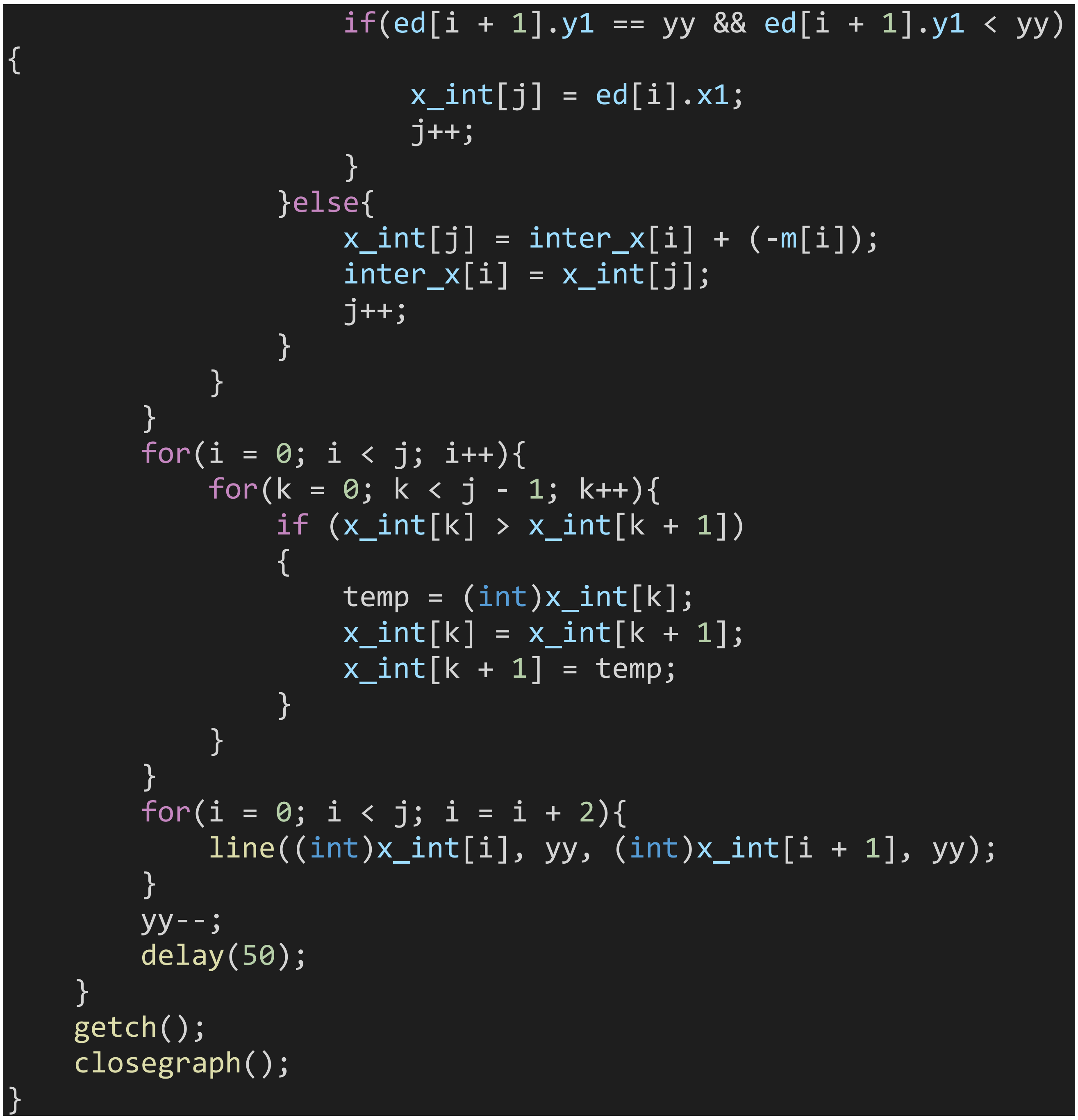
```
#include <conio.h>
#include <stdio.h>
#include <graphics.h>
#include <dos.h>

void fill_right(int x, int y);
void fill_left(int x, int y);

int main(){
    int n, i, x, y;
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");

    line(50, 50, 200, 50);
    line(200, 50, 200, 300);
    line(200, 300, 50, 300);
    line(50, 300, 50, 50);

    x = 100;
    y = 100;
    fill_right(x, y);
    fill_left(x - 1, y);
    getch();
    return 0;
}

void fill_right(int x, int y){
    if((getpixel(x, y) != WHITE) && (getpixel(x, y) != RED)){
        putpixel(x, y, RED);
        fill_right(++x, y);
        x = x - 1;
            fill_right(x, y - 1);
            fill_right(x, y + 1);
    }
    delay(0);
}

void fill_left(int x, int y){
    if((getpixel(x, y) != WHITE) && (getpixel(x, y) != RED)){
        putpixel(x, y, RED);
        fill_left(--x, y);
        x = x + 1;
            fill_left(x, y - 1);
            fill_left(x, y + 1);
    }
    delay(0);
}
```

Flood Fill

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

void floodFill(int x, int y, int oldcolor, int newcolor){
    if(getpixel(x, y) == oldcolor){
        putpixel(x, y, newcolor);
        floodFill(x + 1, y, oldcolor, newcolor);
        floodFill(x, y + 1, oldcolor, newcolor);
        floodFill(x - 1, y, oldcolor, newcolor);
        floodFill(x, y - 1, oldcolor, newcolor);
    }
}

int main(){
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");
    int x, y, radius;

    printf("Enter x and y positions for circle\n");
    scanf("%d%d", &x, &y);
    printf("Enter radius of circle\n");
    scanf("%d", &radius);
    circle(x, y, radius);
    floodFill(x, y, 0, 15);
    delay(3000);
    getch();
    closegraph();
    return 0;
}
```

Scan Fill

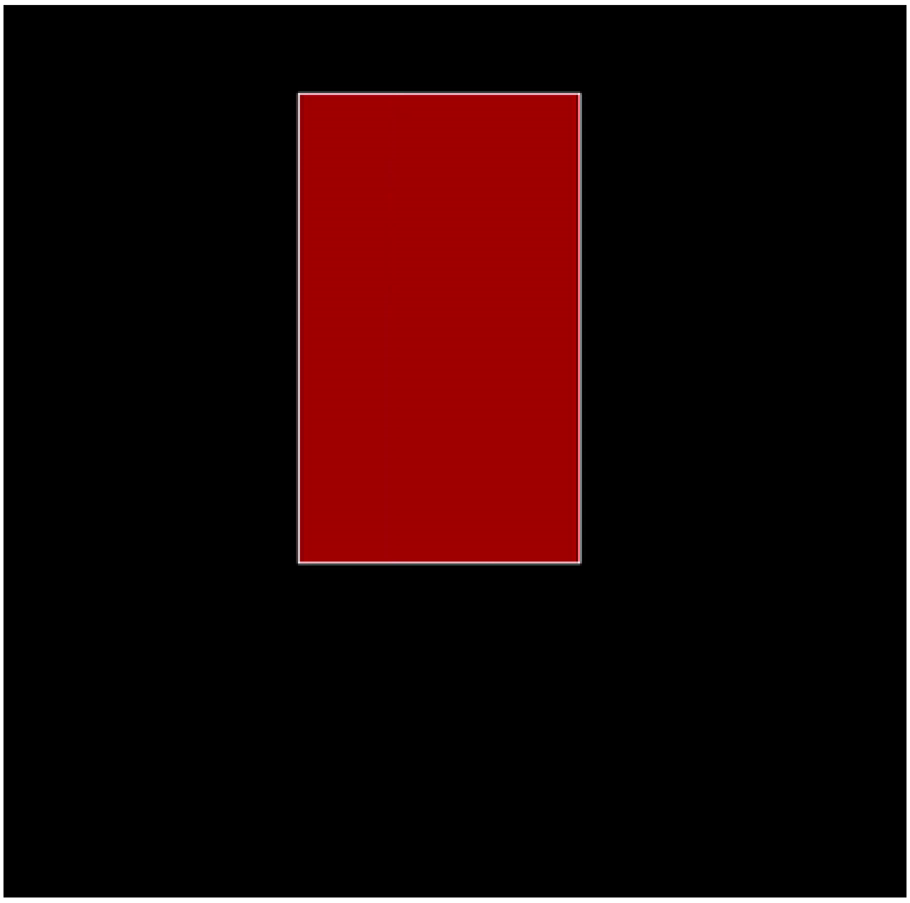
```
#include <iostream.h>
#include <conio.h>
#include <graphics.h>
#include <dos.h>

struct edge{
    int x1, y1, x2, y2, flag;
};

int main(){
    int n, i, j, k;
    struct edge ed[10], temped;
    float dx, dy, m[10], x_int[10], inter_x[10];
    int x[10], y[10], ymax = 0, ymin = 480, yy, temp;
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");

    cout <<"Enter the no.of vertices of the graph: ";
    cin >> n;
    cout <<"Enter the vertices";
    for(i = 0; i < n; i++){
        cin >> x[i];
        cin >> y[i];
        if (y[i] > ymax)
            ymax = y[i];
        if (y[i] < ymin)
            ymin = y[i];
        ed[i].x1 = x[i];
        ed[i].y1 = y[i];
    }
    for(i = 0; i < n - 1; i++){
        ed[i].x2 = ed[i + 1].x1;
        ed[i].y2 = ed[i + 1].y1;
        ed[i].flag = 0;
    }
    ed[i].x2 = ed[0].x1;
    ed[i].y2 = ed[0].y1;
    ed[i].flag = 0;
    for(i = 0; i < n; i++){
        if(ed[i].y1 < ed[i].y2){
            temp = ed[i].x1;
            ed[i].x1 = ed[i].x2;
            ed[i].x2 = temp;
            temp = ed[i].y1;
            ed[i].y1 = ed[i].y2;
            ed[i].y2 = temp;
        }
    }
    for(i = 0; i < n; i++){
        line(ed[i].x1, ed[i].y1, ed[i].x2, ed[i].y2);
    }
    for(i = 0; i < n - 1; i++){
        for(j = 0; j < n - 1; j++){
            if (ed[j].y1 < ed[j + 1].y1){
                temped = ed[j];
                ed[j] = ed[j + 1];
                ed[j + 1] = temped;
            }
            if(ed[j].y1 == ed[j + 1].y1){
                if (ed[j].y2 < ed[j + 1].y2){
                    temped = ed[j];
                    ed[j] = ed[j + 1];
                    ed[j + 1] = temped;
                }
                if(ed[j].y2 == ed[j + 1].y2){
                    if (ed[j].x1 < ed[j + 1].x1){
                        temped = ed[j];
                        ed[j] = ed[j + 1];
                        ed[j + 1] = temped;
                    }
                }
            }
        }
    }
    for(i = 0; i < n; i++){
        dx = ed[i].x2 - ed[i].x1;
        dy = ed[i].y2 - ed[i].y1;
        if(dy == 0){
            m[i] = 0;
        }else{
            m[i] = dx / dy;
        }
        inter_x[i] = ed[i].x1;
    }
    yy = ymax;
    while(yy > ymin){
        for(i = 0; i < n; i++){
            if(yy > ed[i].y2 && yy <= ed[i].y1){
                ed[i].flag = 1;
            }else
                ed[i].flag = 0;
        }
        j = 0;
        for(i = 0; i < n; i++){
            if(ed[i].flag == 1){
                if(yy == ed[i].y1){
                    j++;
                    if(ed[i - 1].y1 == yy && ed[i - 1].y1 < yy)
{
                        x_int[j] = ed[i].x1;
                        j++;
                    }
```



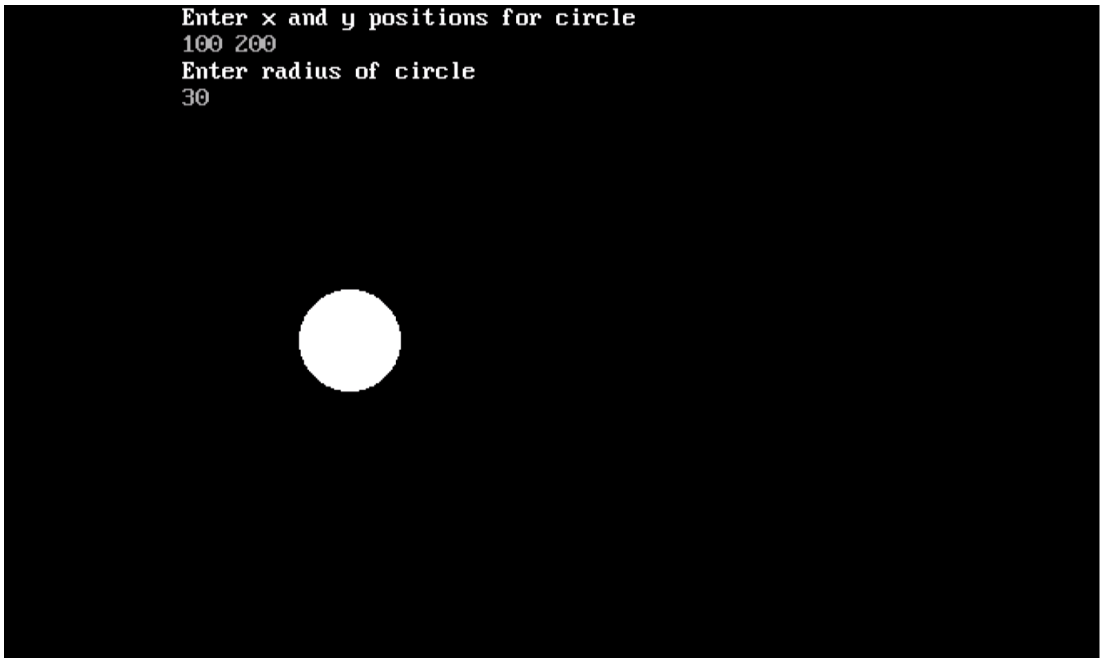
```
                    if(ed[i + 1].y1 == yy && ed[i + 1].y1 < yy)
{
                        x_int[j] = ed[i].x1;
                        j++;
                    }
                }else{
                    x_int[j] = inter_x[i] + (-m[i]);
                    inter_x[i] = x_int[j];
                    j++;
                }
            }
        }
        for(i = 0; i < j; i++){
            for(k = 0; k < j - 1; k++){
                if (x_int[k] > x_int[k + 1])
                {
                    temp = (int)x_int[k];
                    x_int[k] = x_int[k + 1];
                    x_int[k + 1] = temp;
                }
            }
        }
        for(i = 0; i < j; i = i + 2){
            line((int)x_int[i], yy, (int)x_int[i + 1], yy);
        }
        yy--;
        delay(50);
    }
    getch();
    closegraph();
}
```

Output:

Boundary Fill

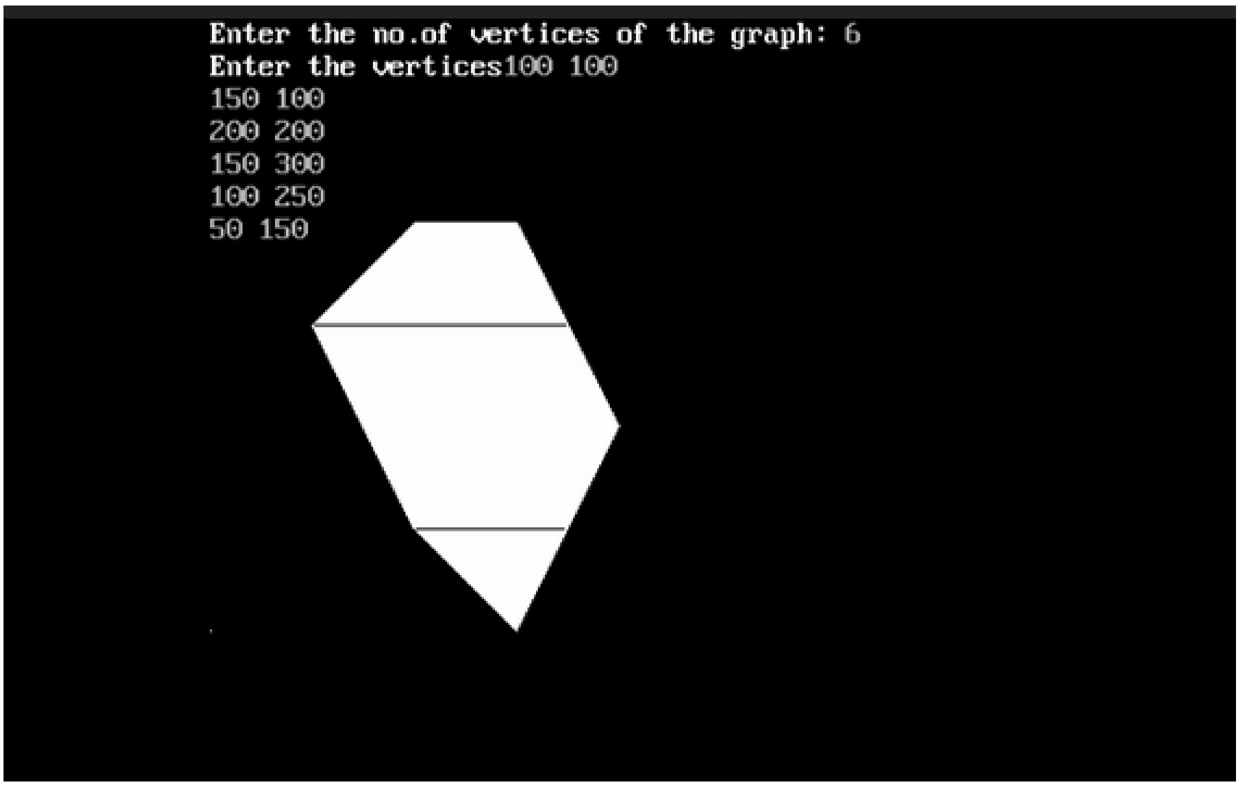


Flood Fill



```
Enter x and y positions for circle
100 200
Enter radius of circle
30
```

Scan Fill



```
Enter the no.of vertices of the graph: 6
Enter the vertices100 100
150 100
200 200
150 300
100 250
50 150
```

Program 8

Aim: Perform the following 2D Transformation operation:

a) Translation

b) Rotation

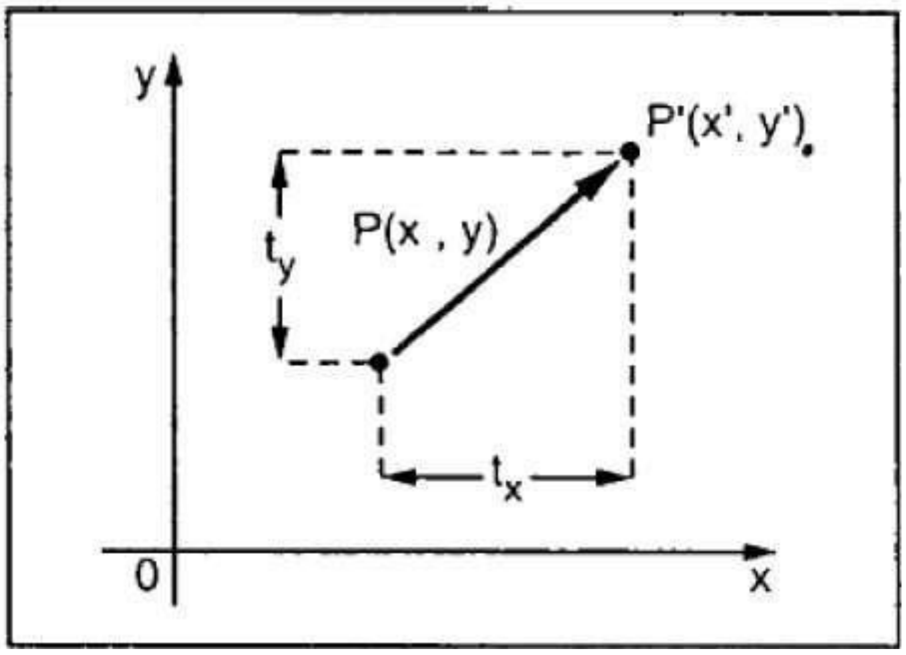
c) Scaling

d) Sheering

Theory:

Translation

A translation moves an object to a different position on the screen. A point in 2D can be translated by adding translation coordinate (tx, ty) to the original coordinate (X, Y) to get the new coordinate (X', Y').



From the above figure,

X' = X + tx

Y' = Y + ty

The pair (tx, ty) is called the translation vector or shift vector. The above equations can also be represented using a translation matrix.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

In this experiment, the 3 coordinated of a triangle are given along with the translation factor. We need to find the new coordinates and plot the translated triangle with respect to the original triangle.

Rotation

It is a process of changing the angle of the object. Rotation can be clockwise or anticlockwise. For rotation, we have to specify the angle of rotation and rotation point. Rotation point is also called a pivot point. It is print about which object is rotated. The positive value of the pivot point (rotation angle) rotates an object in a counter-clockwise (anti-clockwise) direction. The negative value of the pivot point (rotation angle) rotates an object in a clockwise direction. When the object is rotated, then every point of the object is rotated by the same angle.

Scaling

Transformation is a process of converting the original picture coordinates into a different set of picture coordinates by applying certain rules to change their position and/or structure. When a transformation takes place on a 2D plane, it is called 2D transformation. Scaling is a type of linear transformation which is used to change the size of an object. In the scaling process, the dimensions of the object are either expanded or compressed. Scaling can be achieved by multiplying the original coordinates of the object with the scaling factor to get the desired result. When (Sx, Sy) <1, the image is compressed, else it is enlarged.

Let us assume that in a two-dimensional system, the original coordinates are (X, Y), the scaling factors are (Sx, Sy), and the produced coordinates are (X’, Y’).

This can be mathematically represented as shown below:

X' = X. Sx and Y' = Y. Sy

The scaling factor Sx, Sy scales the object in X and Y direction respectively. The above equations can also be represented in matrix form as below –

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

Shearing

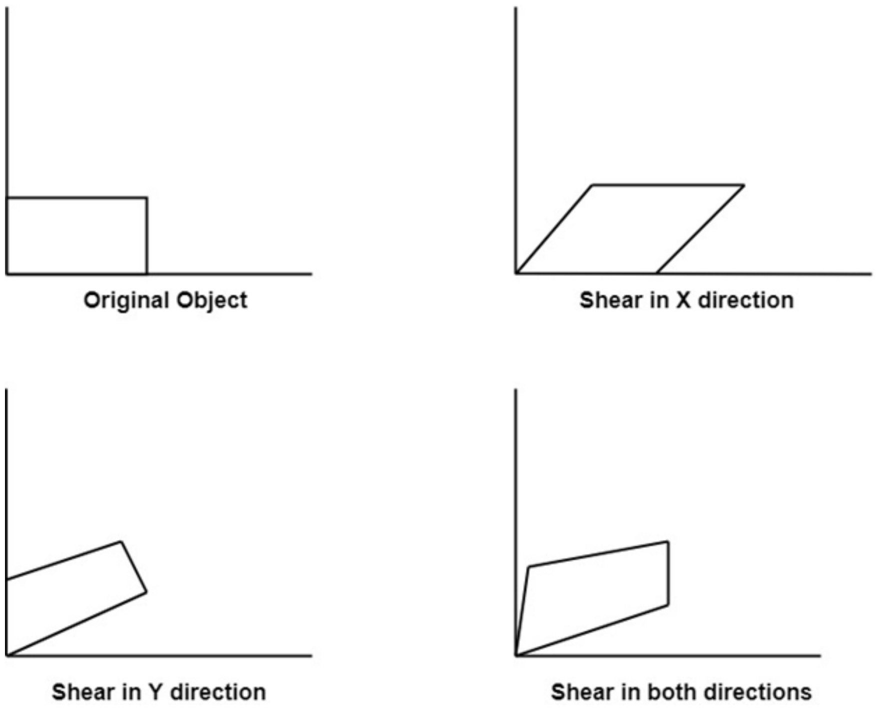
It is transformation which changes the shape of object. The sliding of layers of object occurs. The shear can be in one direction or in two directions.

Shearing in the X-direction: In this horizontal shearing sliding of layers occur. The homogeneous matrix for shearing in the x-direction is shown below:

$$\begin{bmatrix} 1 & 0 & 0 \\ Sh_x & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Shearing in the Y-direction: Here shearing is done by sliding along vertical or y-axis.

$$\begin{bmatrix} 1 & Sh_y & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Algorithm:

Translation

Step 1 - Start

Step 2 - Initialize the graphics mode.

Step 3 - Construct the Triangle using the three coordinates to draw three lines.

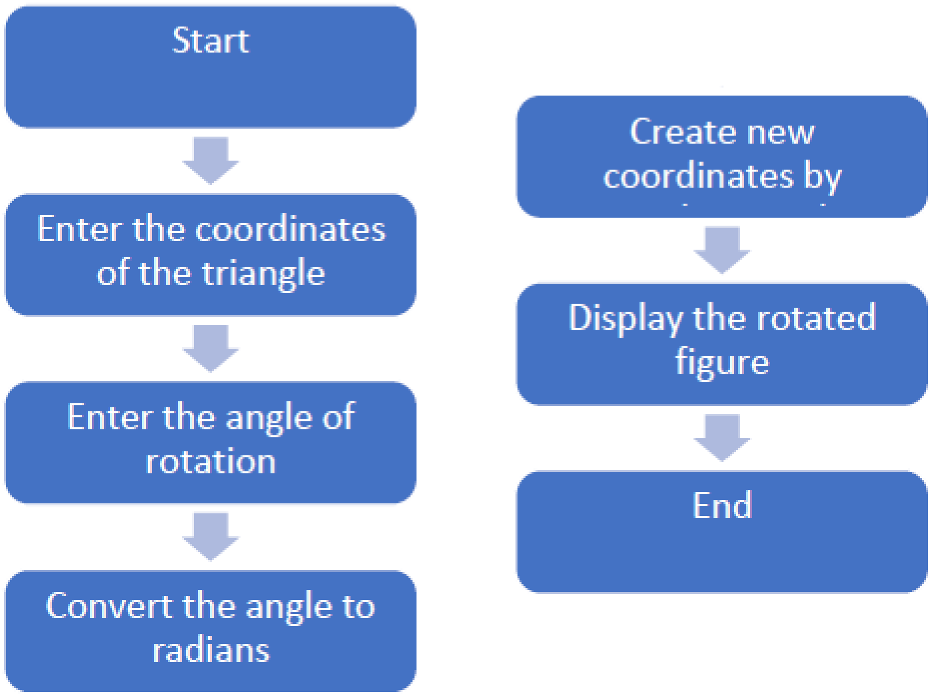
Step 4 - Translation

a) Get the translation value tx, ty

b) Move the 2d object with tx, ty (x'=x+tx,y'=y+ty for all the three points).

c)Plot the new three points, i.e. (x',y')

Rotation


Start
Enter the coordinates of the triangle
Enter the angle of rotation
Convert the angle to radians
Create new coordinates by
Display the rotated figure
End

Scaling

Step 1 - Start

Step 2 - Initialize the graphics mode.

Step 3 - Plot a 2D object using given coordinates (x,y)

Step 4 - Get the scaling value Sx,Sy

Step 5 - Resize the object with Sx,Sy (x'=x*Sx,y'=y*Sy)

Step 6 - Plot the new object using new coordinates (x',y')

Code:

Translation

```c
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>

int main(){
    int gm, gd = DETECT;
    initgraph(&gd, &gm, "C:\\TurboC3\\BGI");

    int x1, x2, x3, y1, y2, y3;
    printf("2D Transformation operation\nTranslation\nEnter the
 points of triangle\n");
    scanf("%d%d%d%d%d%d", &x1, &y1, &x2, &y2, &x3, &y3);
    printf("Enter translation factor\n");
    int tx, ty;
    scanf("%d %d", &tx, &ty);

    line(x1, y1, x2, y2);
    line(x2, y2, x3, y3);
    line(x1, y1, x3, y3);

    int nx1, nx2, nx3, ny1, ny2, ny3;
    nx1 = x1 + tx;
    nx2 = x2 + tx;
    nx3 = x3 + tx;
    ny1 = y1 + ty;
    ny2 = y2 + ty;
    ny3 = y3 + ty;
    line(nx1, ny1, nx2, ny2);
    line(nx2, ny2, nx3, ny3);
    line(nx3, ny3, nx1, ny1);

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

Rotation

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

int main(){
    int gm, gd = DETECT;
    initgraph(&gd, &gm, "C:\\TurboC3\\BGI");

    int x1, x2, x3, y1, y2, y3;
    printf("2D Transformation operation\nRotation\nEnter the po
ints of triangle\n");
    scanf("%d%d%d%d%d%d", &x1, &y1, &x2, &y2, &x3, &y3);
    line(x1, y1, x2, y2);
    line(x2, y2, x3, y3);
    line(x1, y1, x3, y3);

    printf("Enter the angle of rotation:\n");
    int rq;
    scanf("%d", &rq);

    rq = 3.14 * rq / 180;
    int nx1, nx2, nx3, ny1, ny2, ny3;
    nx1 = abs(x1 * cos(rq) + y1 * sin(rq));
    ny1 = abs(x1 * sin(rq) + y1 * cos(rq));
    nx3 = abs(x3 * cos(rq) + y3 * sin(rq));
    ny2 = abs(x2 * sin(rq) + y2 * cos(rq));
    nx2 = abs(x2 * cos(rq) + y2 * sin(rq));
    ny3 = abs(x3 * sin(rq) + y3 * cos(rq));

    line(nx1, ny1, nx2, ny2);
    line(nx2, ny2, nx3, ny3);
    line(nx3, ny3, nx1, ny1);

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

Scaling

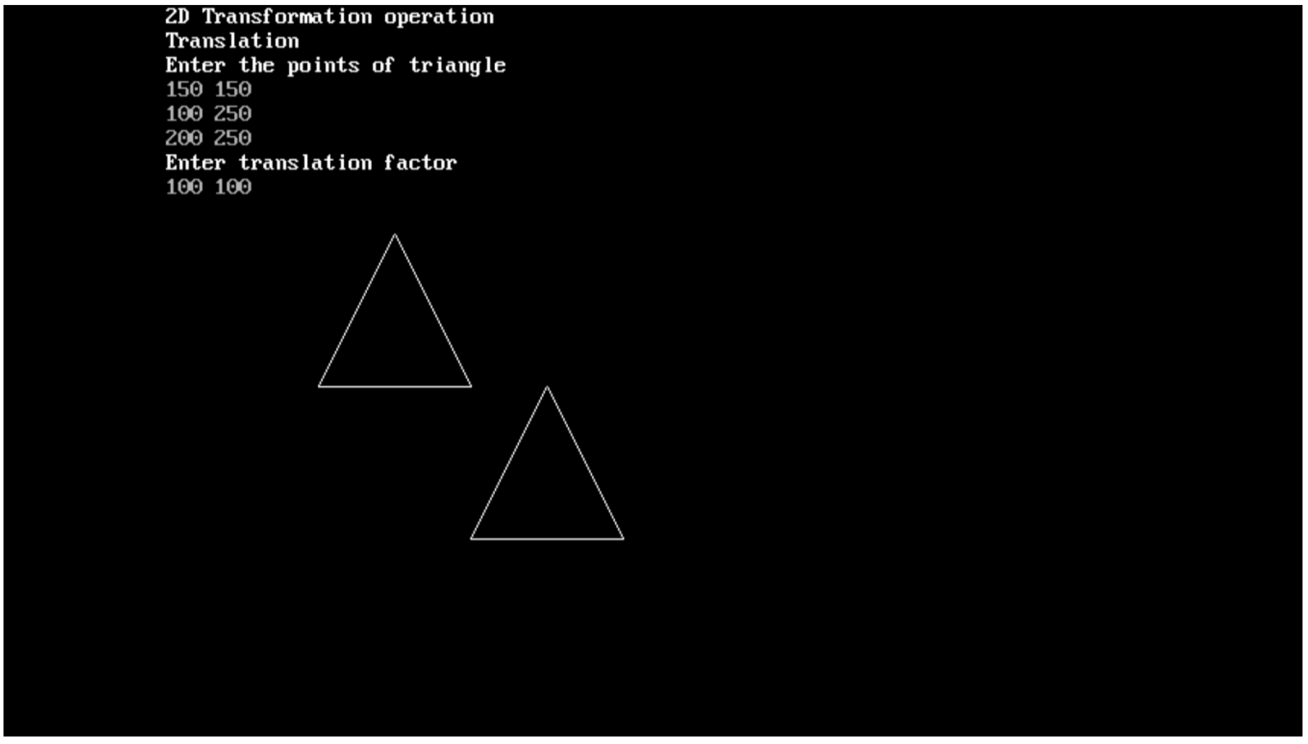
```
#include <iostream.h>
#include <conio.h>
#include <graphics.h>
#include <math.h>

int PolygonPoints[4][2] = {{10, 10}, {10, 100}, {100, 100}, {10
0, 10}};
float Sx = 0.5;
float Sy = 2.0;

void PolyLine(){
    cleardevice();
    line(0, 240, 640, 240);
    line(320, 0, 320, 480);
    for(int itr=0; itr<4; itr++){
        line(320 + PolygonPoints[itr][0], 240 - PolygonPoints[i
tr][1], 320 + PolygonPoints[(itr + 1) % 4][0], 240 - PolygonPoi
nts[(itr + 1) % 4][1]);
    }
}
void Scale(){
    int itr;
    int Tx, Ty;
    cout<<endl;
    for(itr=0; itr<4; itr++){
        PolygonPoints[itr][0] *= Sx;
        PolygonPoints[itr][1] *= Sy;
    }
}

void main(){
    int gDriver = DETECT, gMode;
    int itr;
    initgraph(&gDriver, &gMode, "C:\\TURBOC3\\BGI");
    cout<<"2D Transformation operation\nScaling\n";
    PolyLine();
    getch();
    Scale();
    PolyLine();
    getch();
}
```

Shearing

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

void mul(int mat[3][3],int points[10][3],int n);
void shear(int points[10][3],int n);
void init(int points[10][3],int n);

int main(){
    int i,x,y;
    int points[10][3],n;
    clrscr();
    cout<<"2D Transformation operation\nShearing\nEnter the no.
 of points : ";
    cin>>n;
    for(i=0;i<n;i++){
        cin>>x>>y;
        points[i][0]=x;
        points[i][1]=y;
        points[i][2]=1;
    }
    shear(points,n);
    getch();
    return 0;
}

void init(int points[10][3],int n){
    int gd=DETECT,gm,i;
    initgraph(&gd, &gm, "C:\\TurboC3\\BGI");
    setcolor(10);
    line(0,240,640,240);
    line(320,0,320,480);
    setcolor(3);
    for(i=0;i<n-1;i++){
        line(320+points[i][0],240-
points[i][1],320+points[i+1][0],240-points[i+1][1]);
    }
    line(320+points[n-1][0],240-points[n-
1][1],320+points[0][0],240-points[0][1]);
}

void mul(int mat[3][3],int points[10][3],int n){
    int i,j,k;
    int res[10][3];
    for(i=0;i<n;i++){
        for(j=0;j<3;j++){
            res[i][j]=0;
            for(k=0;k<3;k++){
                res[i][j] = res[i][j] + points[i][k]*mat[k][j];
            }
        }
    }
    setcolor(15);
    for(i=0;i<n-1;i++){
       line(320+res[i][0],240-res[i][1],320+res[i+1][0],240-
res[i+1][1]);
    }
    line(320+res[n-1][0],240-res[n-1][1],320+res[0][0],240-
res[0][1]);
}
void shear(int points[10][3],int n){
    int opt, arr_shear[3][3];
    cout<<"\n1.X-shear\n2.Y-shear\nYour Choice: ";
    cin>>opt;
    switch(opt){
        case 1:
            int xsh;
            cout<<"\nEnter the x shear : ";
            cin>>xsh;
            arr_shear[0][0]=1;
            arr_shear[1][0]=xsh;
            arr_shear[2][0]=0;
            arr_shear[0][1]=0;
            arr_shear[1][1]=1;
            arr_shear[2][1]=0;
            arr_shear[0][2]=0;
            arr_shear[1][2]=0;
            arr_shear[2][2]=1;
            init(points,n);
            mul(arr_shear,points,n);
        break;
        case 2:
            int ysh;
            cout<<"\nEnter the y shear : ";
            cin>>ysh;
            arr_shear[0][0]=1;
            arr_shear[1][0]=0;
            arr_shear[2][0]=0;
            arr_shear[0][1]=ysh;
            arr_shear[1][1]=1;
            arr_shear[2][1]=0;
            arr_shear[0][2]=0;
            arr_shear[1][2]=0;
            arr_shear[2][2]=1;
            init(points,n);
            mul(arr_shear,points,n);
        break;
    }
}
```

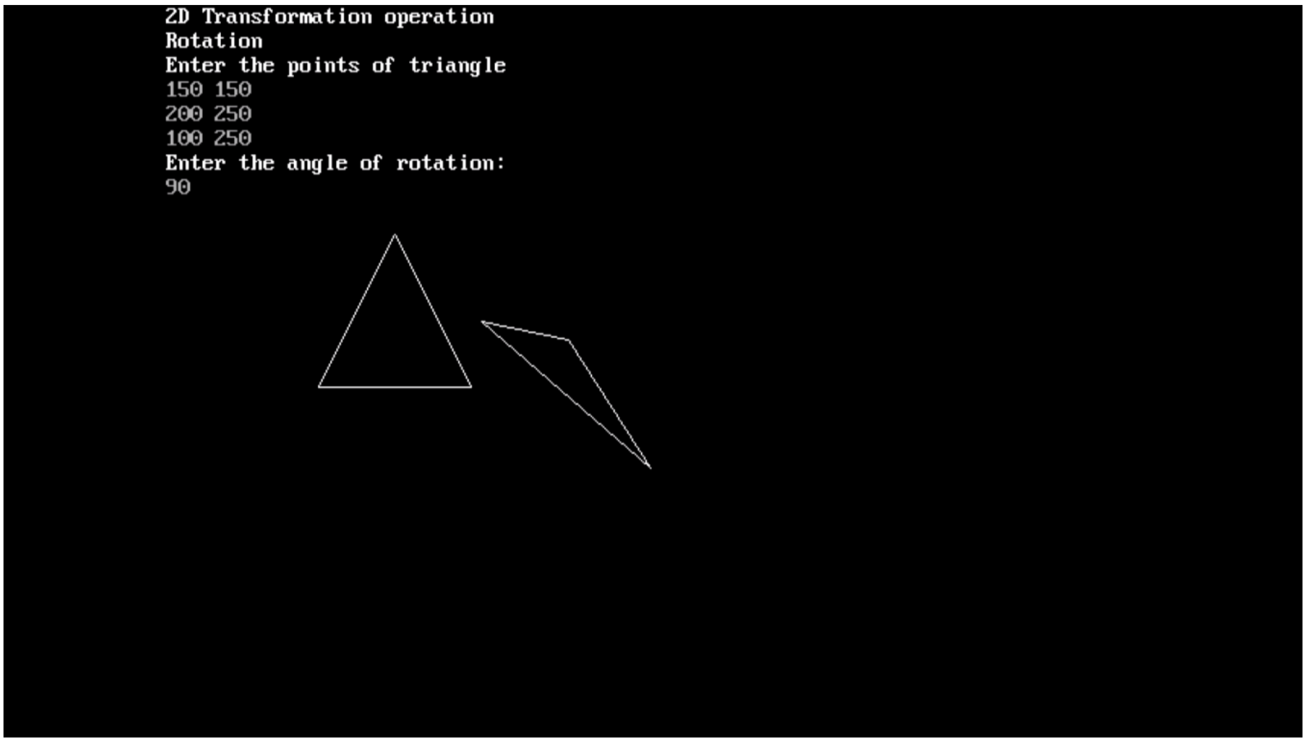
Output:

Translation



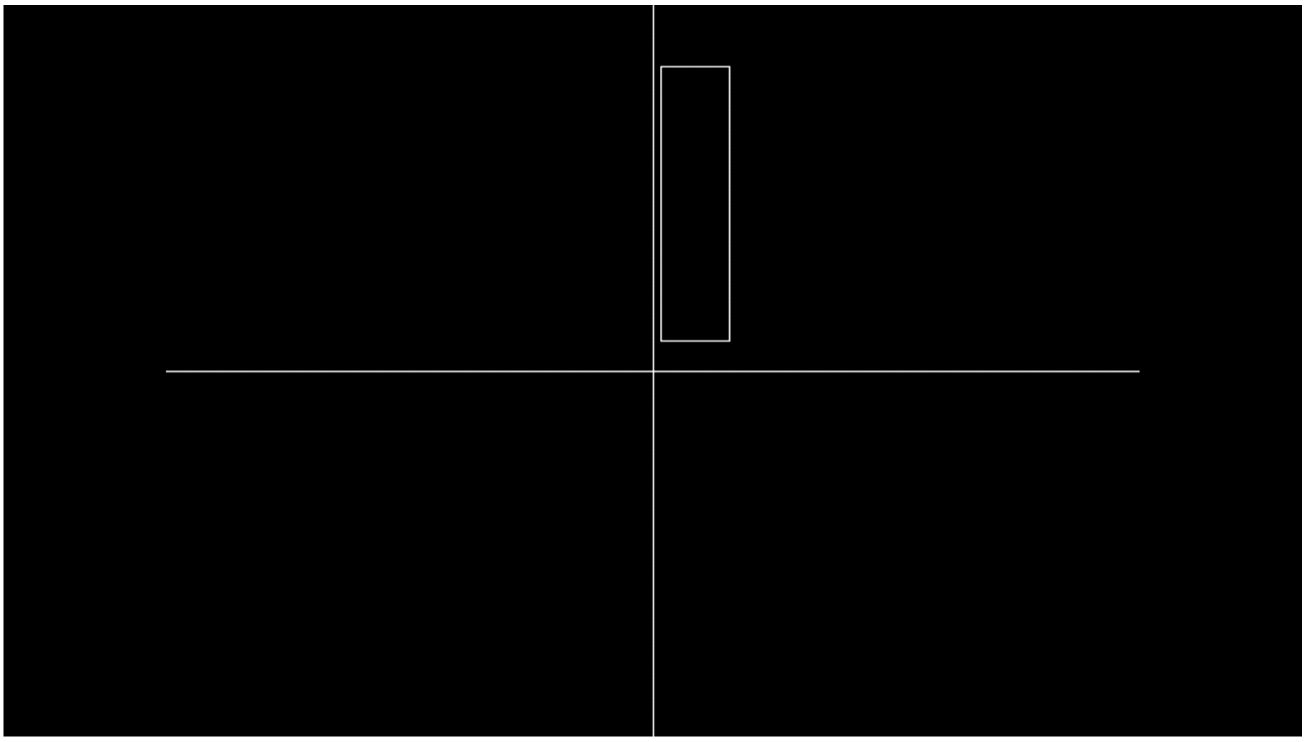
```
2D Transformation operation
Translation
Enter the points of triangle
150 150
100 250
200 250
Enter translation factor
100 100
```

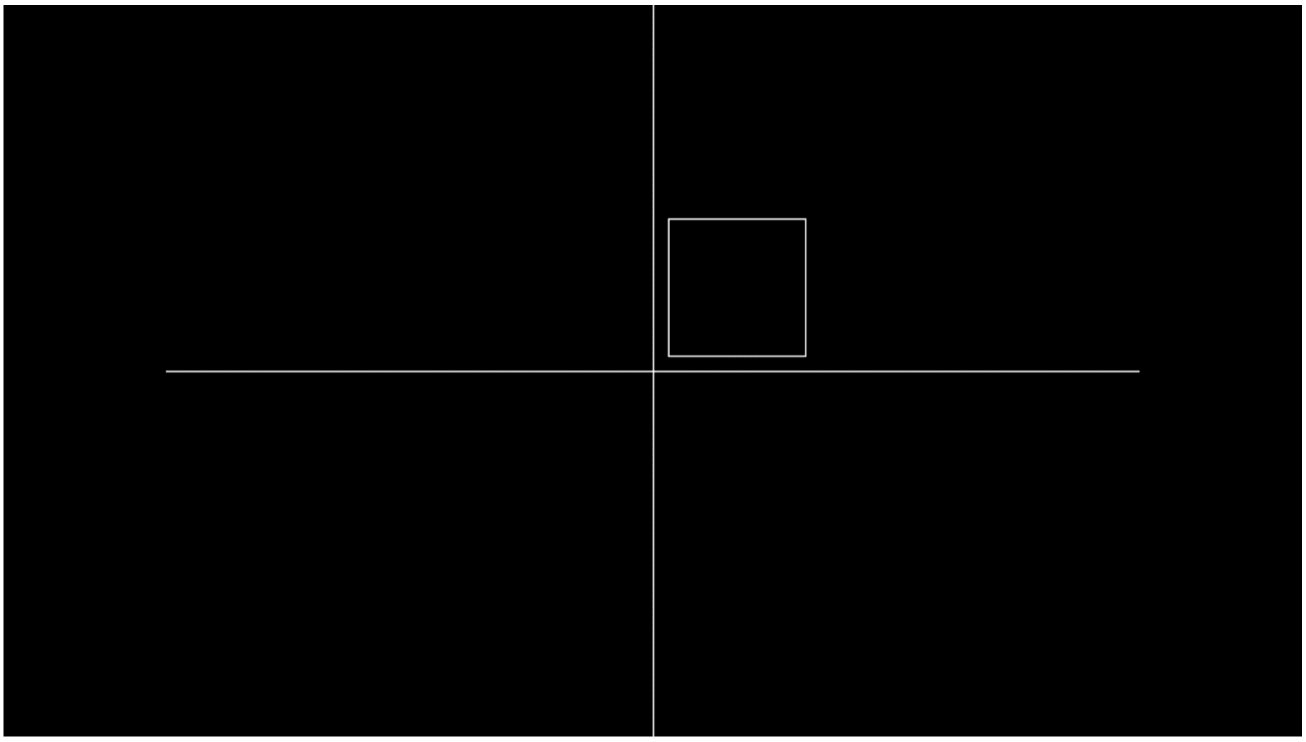
Rotation



```
2D Transformation operation
Rotation
Enter the points of triangle
150 150
200 250
100 250
Enter the angle of rotation:
90
```

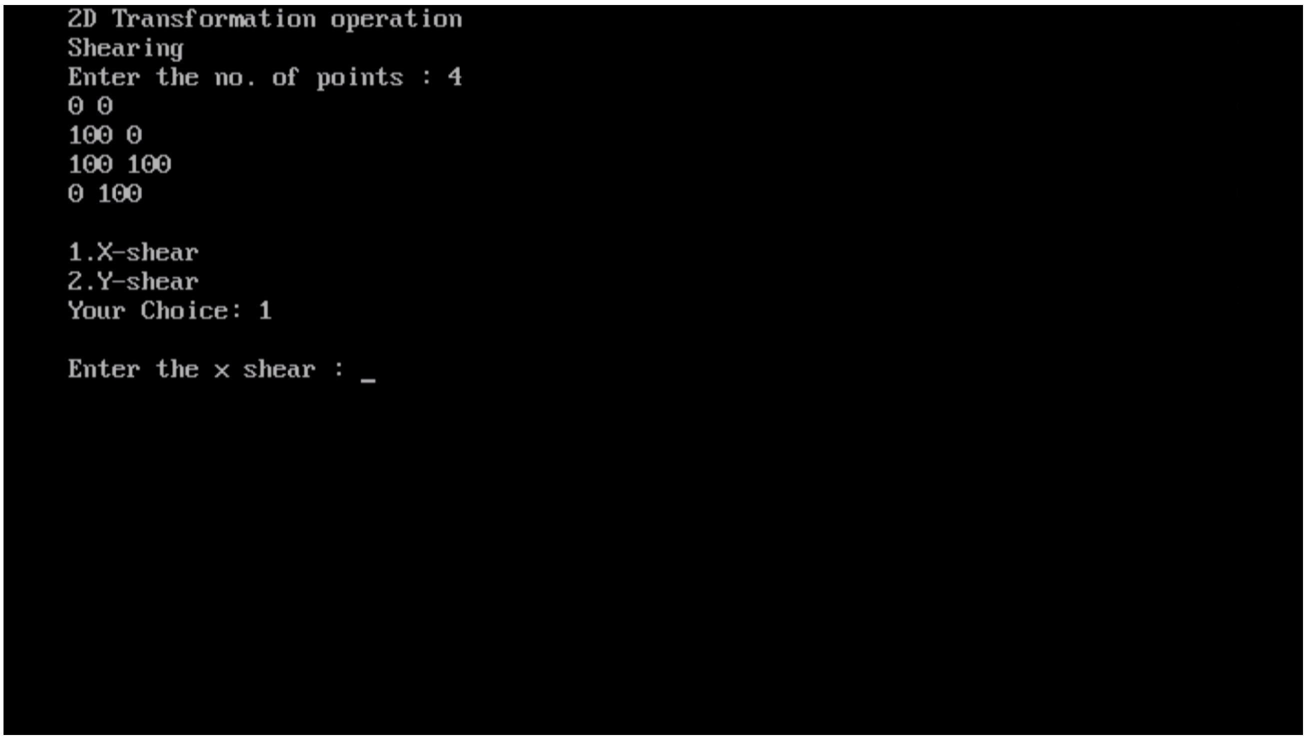
Scaling



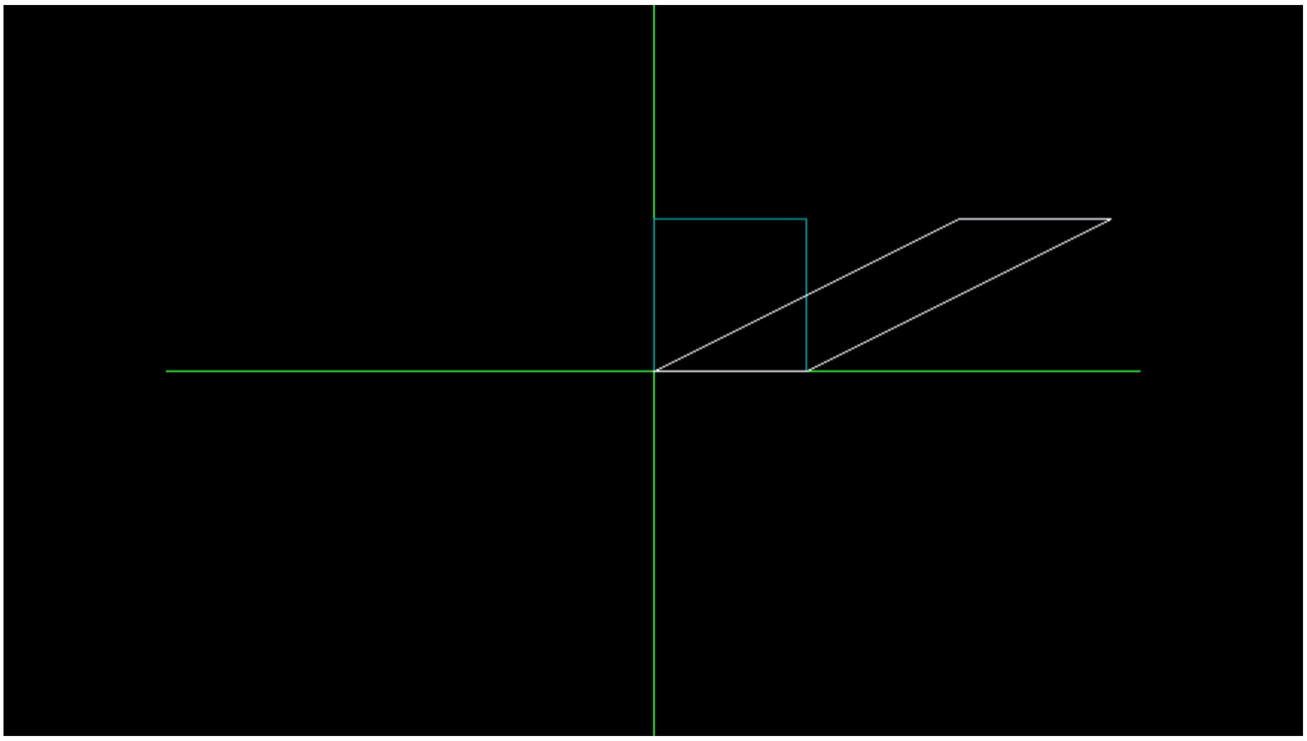


Shearing

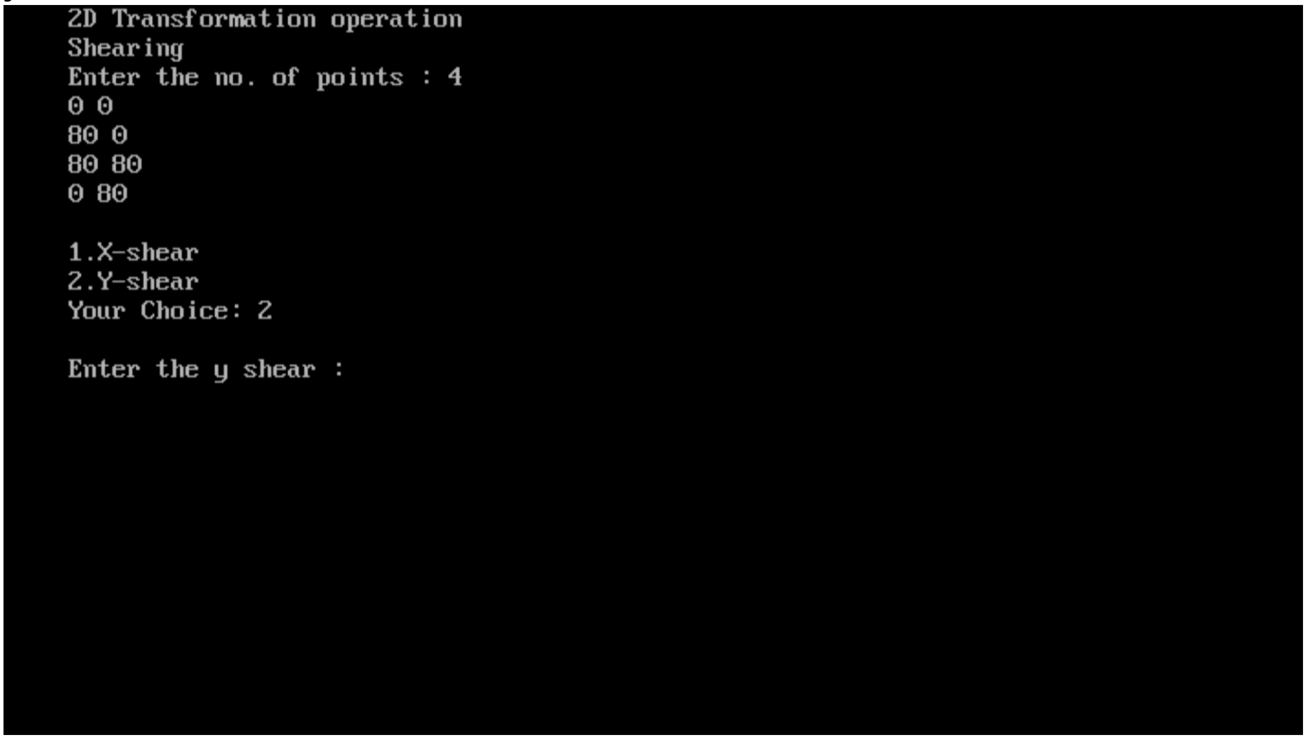
x-shear



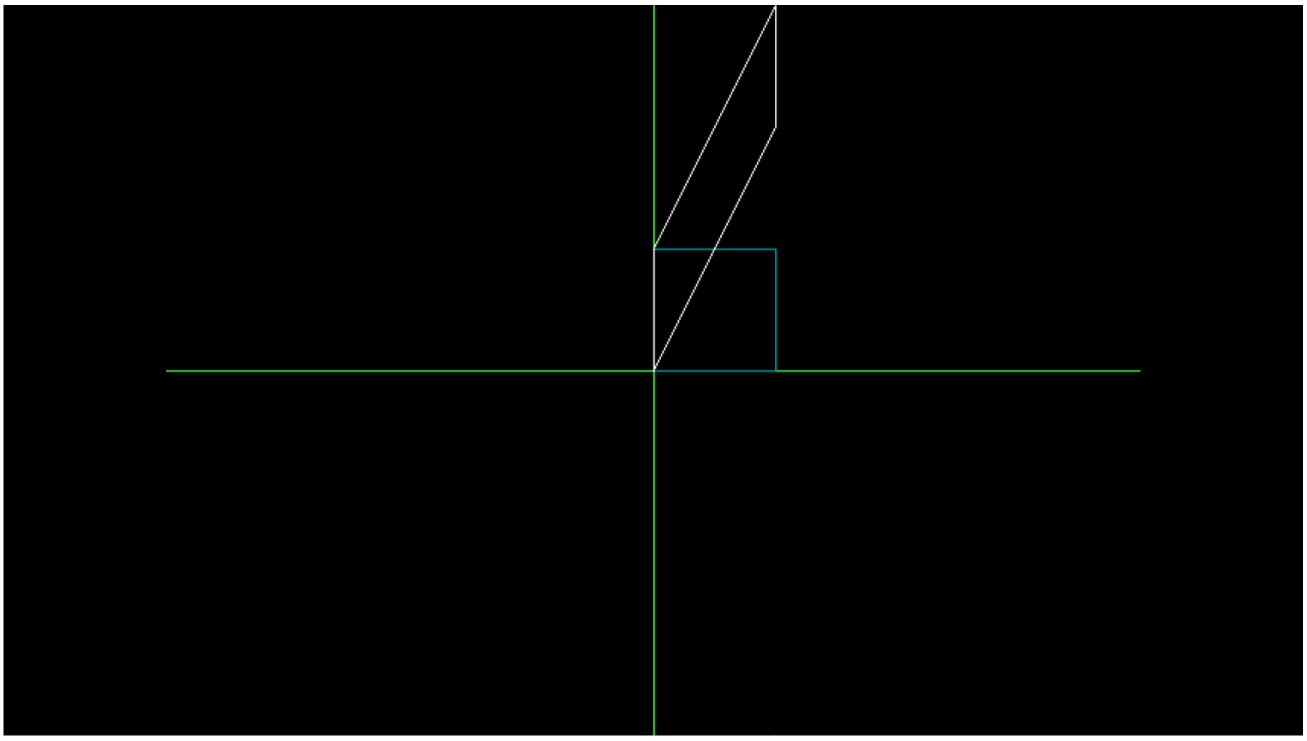
```
2D Transformation operation
Shearing
Enter the no. of points : 4
0 0
100 0
100 100
0 100

1.X-shear
2.Y-shear
Your Choice: 1

Enter the x shear : _
```



y-shear



```
2D Transformation operation
Shearing
Enter the no. of points : 4
0 0
80 0
80 80
0 80

1.X-shear
2.Y-shear
Your Choice: 2

Enter the y shear :
```



Program 9

Aim: Perform the following 3D Transformation operation:

a) Translation

b) Rotation

c) Scaling

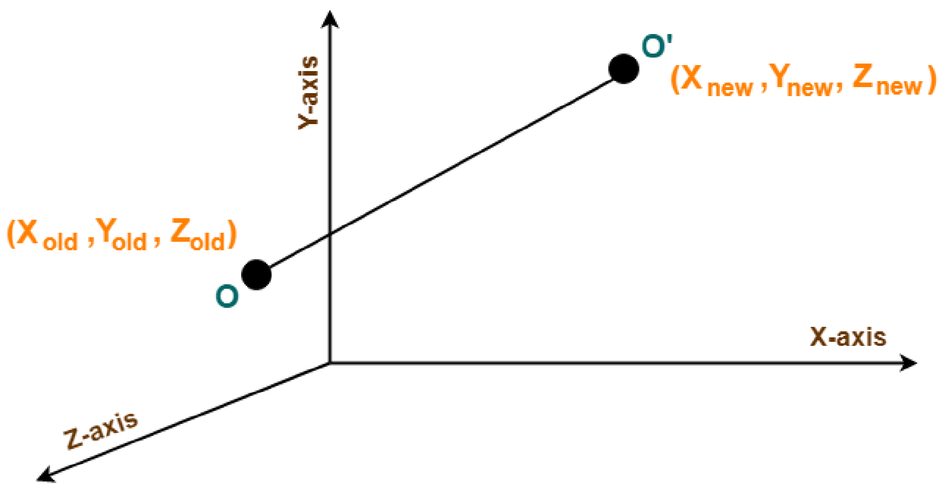
d) Sheering

Theory:

Translation

Consider a point object O has to be moved from one position to another in a 3D plane. Given a Translation vector (T_x, T_y, T_z)-

- T_x defines the distance the X_{old} coordinate has to be moved.
- T_y defines the distance the Y_{old} coordinate has to be moved.
- T_z defines the distance the Z_{old} coordinate has to be moved.



This translation is achieved by adding the translation coordinates to the old coordinates of the object as-

- $X_{new} = X_{old} + T_x$ (This denotes translation towards X axis)
- $Y_{new} = Y_{old} + T_y$ (This denotes translation towards Y axis)

- $Z_{new} = Z_{old} + T_z$ (This denotes translation towards Z axis)

In Matrix form, the above translation equations may be represented as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

Rotation

Consider a point object O has to be rotated from one angle to another in a 3D plane. Let-

- Initial coordinates of the object O = $(X_{old}, Y_{old}, Z_{old})$
- Initial angle of the object O with respect to origin = Φ
- Rotation angle = θ
- New coordinates of the object O after rotation = $(X_{new}, Y_{new}, Z_{new})$

In 3 dimensions, there are 3 possible types of rotation-

- X-axis Rotation
- Y-axis Rotation
- Z-axis Rotation

Rotation along X-direction: This rotation is achieved by using the following rotation equations-

- $X_{new} = X_{old}$
- $Y_{new} = Y_{old} \times \cos\theta - Z_{old} \times \sin\theta$
- $Z_{new} = Y_{old} \times \sin\theta + Z_{old} \times \cos\theta$

In Matrix form, the above rotation equations may be represented as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

Rotation along Y-axis: This rotation is achieved by using the following rotation equations-

- $X_{new} = Z_{old} \times \sin\theta + X_{old} \times \cos\theta$
- $Y_{new} = Y_{old}$
- $Z_{new} = Y_{old} \times \cos\theta - X_{old} \times \sin\theta$

In Matrix form, the above rotation equations may be represented as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

Rotation along Z-axis: This rotation is achieved by using the following rotation equations-

- $X_{new} = X_{old} \times \cos\theta - Y_{old} \times \sin\theta$
- $Y_{new} = X_{old} \times \sin\theta + Y_{old} \times \cos\theta$
- $Z_{new} = Z_{old}$

In Matrix form, the above rotation equations may be represented as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

Scaling

- Scaling may be used to increase or reduce the size of object.
- Scaling subjects the coordinate points of the original object to change.
- Scaling factor determines whether the object size is to be increased or reduced.
- If scaling factor > 1, then the object size is increased.
- If scaling factor < 1, then the object size is reduced.

Consider a point object O has to be scaled in a 3D plane. Let-

- Initial coordinates of the object O = (X_{old}, Y_{old},Z_{old})
- Scaling factor for X-axis = S_x
- Scaling factor for Y-axis = S_y
- Scaling factor for Z-axis = S_z
- New coordinates of the object O after scaling = (X_{new}, Y_{new}, Z_{new})

This scaling is achieved by using the following scaling equations-

- $X_{new} = X_{old} \times S_x$
- $Y_{new} = Y_{old} \times S_y$
- $Z_{new} = Z_{old} \times S_z$

In Matrix form, the above scaling equations may be represented as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

Shearing

In a three-dimensional plane, the object size can be changed along X direction, Y direction as well as Z direction. So, there are three versions of shearing-

1. Shearing in X direction

2. Shearing in Y direction
3. Shearing in Z direction

Consider a point object O has to be sheared in a 3D plane. Let-

- Initial coordinates of the object O = (X_{old}, Y_{old}, Z_{old})
- Shearing parameter towards X direction = Sh_x
- Shearing parameter towards Y direction = Sh_y
- Shearing parameter towards Z direction = Sh_z
- New coordinates of the object O after shearing = (X_{new}, Y_{new}, Z_{new})

Shearing in the X-direction: Shearing in X axis is achieved by using the following shearing equations-

- $X_{new} = X_{old}$
- $Y_{new} = Y_{old} + Sh_y \times X_{old}$
- $Z_{new} = Z_{old} + Sh_z \times X_{old}$

In Matrix form, the above shearing equations may be represented as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ Sh_y & 1 & 0 & 0 \\ Sh_z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

Shearing in the Y-direction: Shearing in Y axis is achieved by using the following shearing equations-

- $X_{new} = X_{old} + Sh_x \times Y_{old}$
- $Y_{new} = Y_{old}$
- $Z_{new} = Z_{old} + Sh_z \times Y_{old}$

In Matrix form, the above shearing equations may be represented as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & Sh_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & Sh_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

Shearing in the Z-direction: Shearing in Z axis is achieved by using the following shearing equations-

- $X_{new} = X_{old} + Sh_x \times Z_{old}$
- $Y_{new} = Y_{old} + Sh_y \times Z_{old}$
- $Z_{new} = Z_{old}$

In Matrix form, the above shearing equations may be represented as-

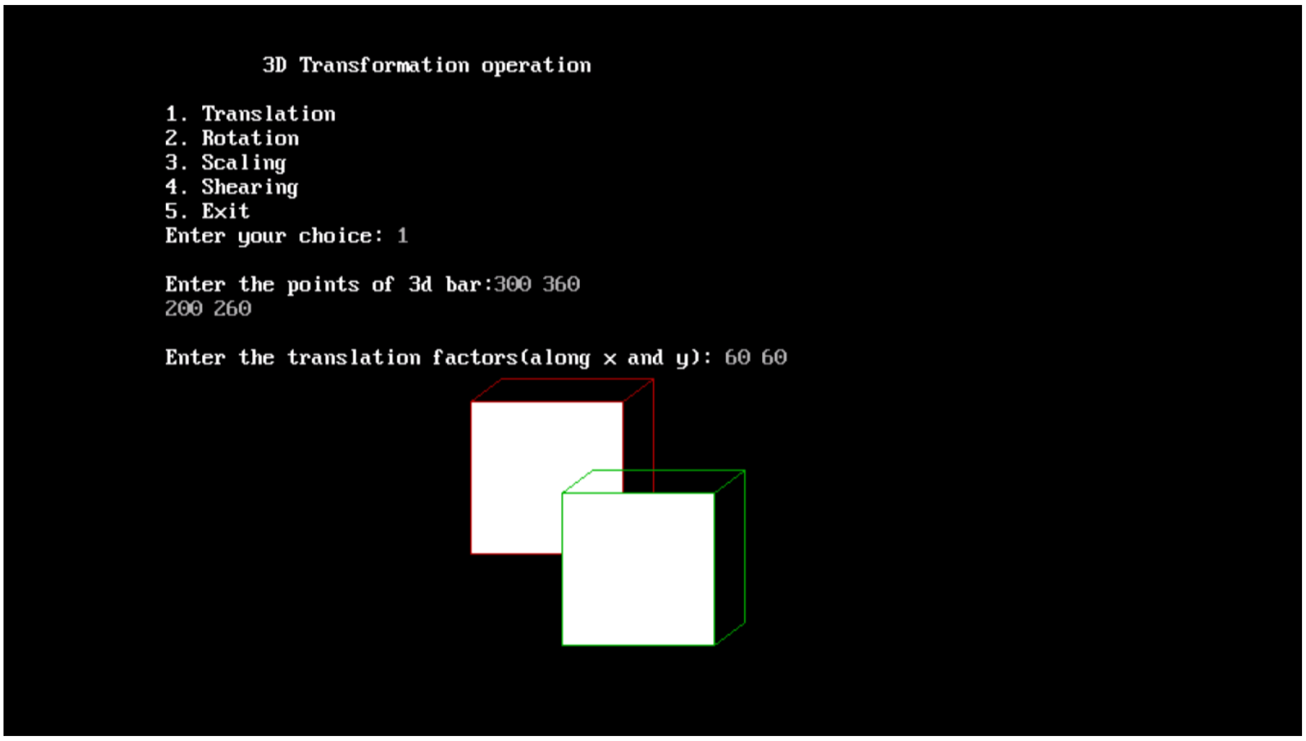
$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & Sh_x & 0 \\ 0 & 1 & Sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

Code:

```
#include <iostream.h>
#include <graphics.h>
#include <process.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>

int main(){
    int x1, x2, y1, y2, nx1, nx2, ny1, ny2, c, s, constant;
    int sx, sy, sz, depth = 20, xt, yt, zt, r, sh, shx, shy, shz;
    float t;
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");
    printf("\n\n\t3D Transformation operation");
    printf("\n\n1. Translation \n2. Rotation \n3. Scaling \n4. Shear
ing \n5. Exit");
    cout<<"\nEnter your choice: ";
    cin>>c;
    switch(c){
        case 1:
            printf("\nEnter the points of 3d bar:");
            scanf("%d %d %d %d", &x1, &y1, &x2, &y2);
            cout<<"\nEnter the translation factors(along x and y): "
;
            scanf("%d %d", &xt, &yt);
            nx1 = x1 + xt;
            ny1 = y1 + yt;
            nx2 = x2 + xt;
            ny2 = y2 + yt;
        break;
        case 2:
            cout<<"\nEnter the points of 3d bar:";
            scanf("%d %d %d %d", &x1, &y1, &x2, &y2);
            cout<<"\nEnter the rotating angle: ";
            scanf("%d", &sx);
            cout<<"\nRotating along z axis";
            nx1 = abs(x1 * cos(sx * 3.14 / 180) - y1 * sin(sx * 3.14
 / 180)) + 200;
            ny1 = abs(x1 * sin(sx * 3.14 / 180) + y1 * cos(sx * 3.14
 / 180)) + 200;
            nx2 = abs(x2 * cos(sx * 3.14 / 180) - y2 * sin(sx * 3.14
 / 180)) + 200;
            ny2 = abs(x2 * sin(sx * 3.14 / 180) + y2 * cos(sx * 3.14
 / 180)) + 200;
        break;
        case 3:
            cout<<"\nEnter the points of 3d bar:";
            scanf("%d %d %d %d", &x1, &y1, &x2, &y2);
            cout<<"\n Enter the scaling factor(for x,y,z): ";
            scanf("%d %d %d", &sx, &sy, &sz);
            nx1 = x1 * sx + 100;
            ny1 = y1 * sy + 100;
            nx2 = x2 * sx + 100;
            ny2 = y2 * sy + 100;
            depth = depth * sz;
        break;
        case 4:
            cout<<"\nEnter the points of 3d bar:";
            scanf("%d %d %d %d", &x1, &y1, &x2, &y2);
            cout<<"\nShear along: \t(1)x-axis \t(2)y-axis \t(3)z-
axis";
            cout<<"\nEnter choice (1 or 2 or 3): ";
            scanf("%d", &s);
            cout<<"\nEnter shearing parameter: ";
            cin>>sh;
            switch(s){
                case 1:
                    nx1 = x1 + sh * y1;
                    ny1 = y1;
                    nx2 = x2 + sh * y2;
                    ny2 = y2;
                break;
                case 2:
                    nx1 = x1;
                    ny1 = y1 + sh * x1;
                    nx2 = x2;
                    ny2 = y2 + sh * x2;
                break;
                case 3:
                    nx1 = x1 + sh * y1;
                    ny1 = y1 + sh * x1;
                    nx2 = x2 + sh * y2;
                    ny2 = y2 + sh * x2;
                break;
            }
        break;
        case 5:
            cout<<"\nExiting...";
            exit(0);
        default:
            cout<<"\nEnter the correct choice!!";
        break;
    }
    setcolor(RED);
    bar3d(x1, y1, x2, y2, 20, 1);
    getch();
    setcolor(GREEN);
    bar3d(nx1, ny1, nx2, ny2, depth, 1);
    getch();
    closegraph();
    return 0;
}
```

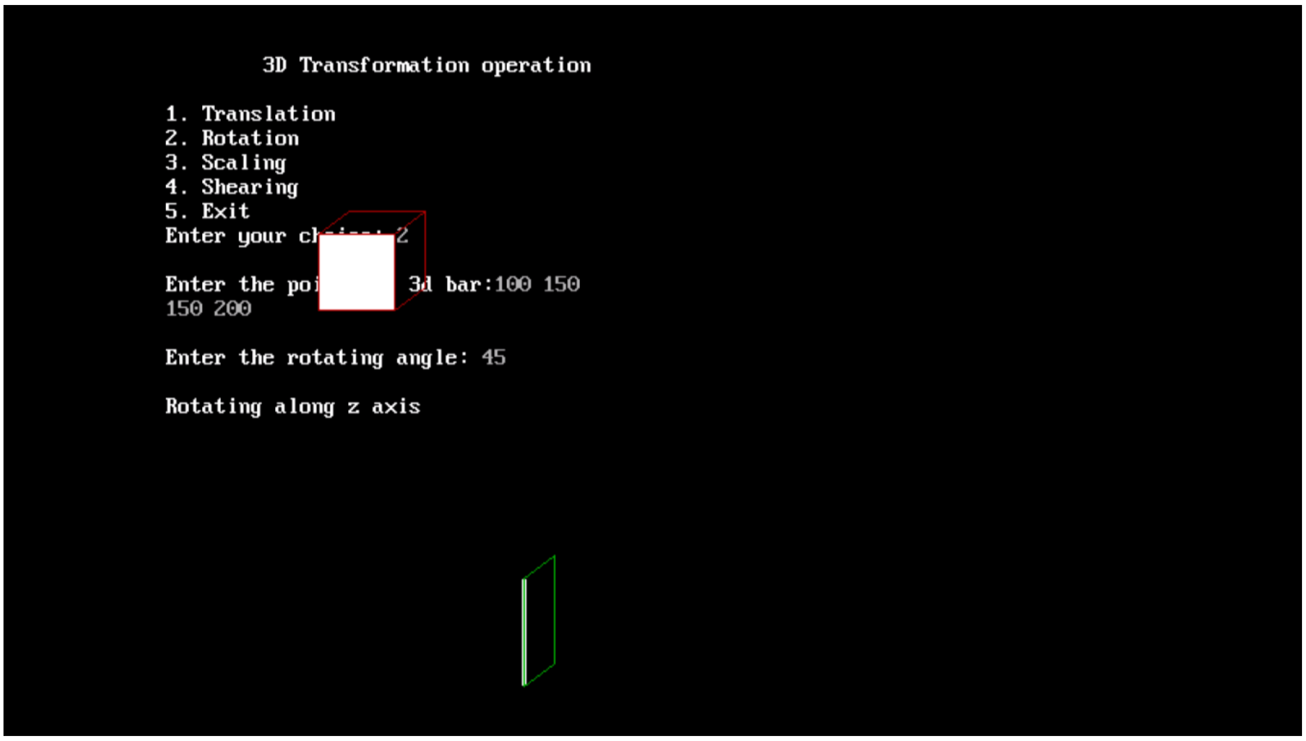
Output:

Translation



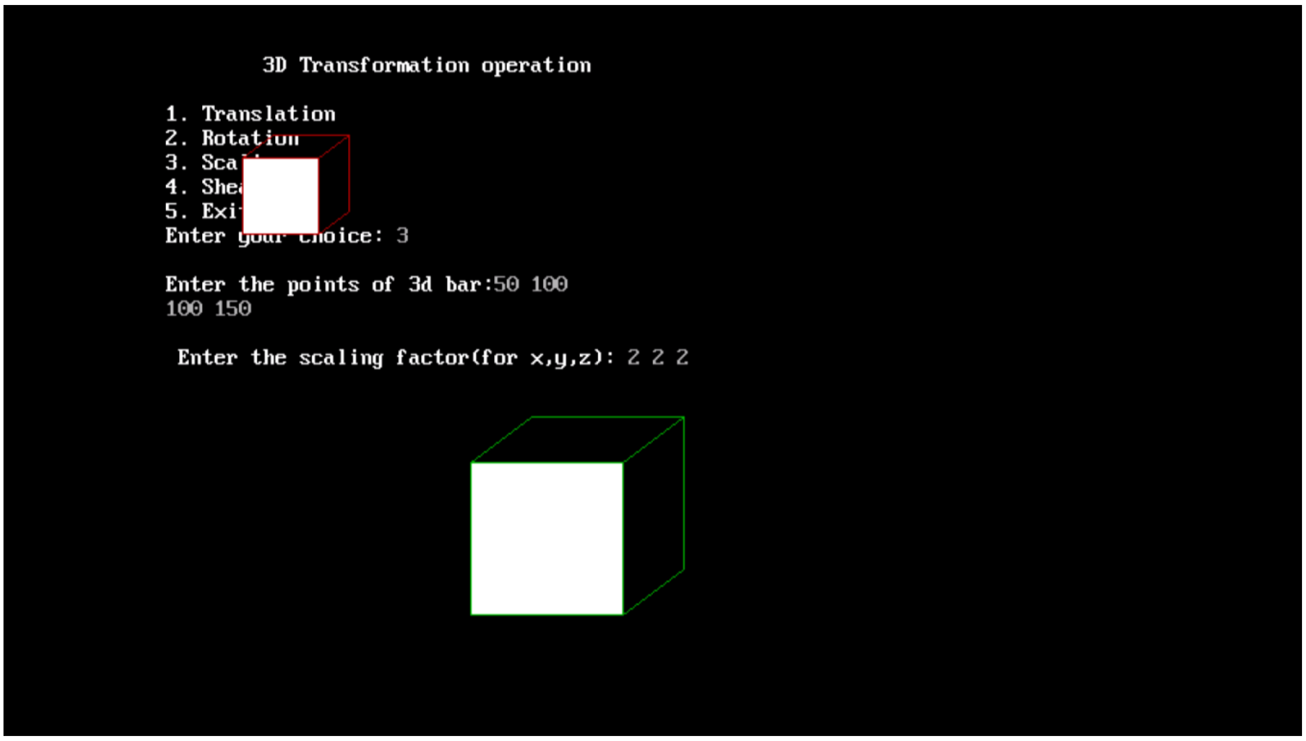
```
        3D Transformation operation

1. Translation
2. Rotation
3. Scaling
4. Shearing
5. Exit
Enter your choice: 1

Enter the points of 3d bar:300 360
200 260

Enter the translation factors(along x and y): 60 60
```

Rotation



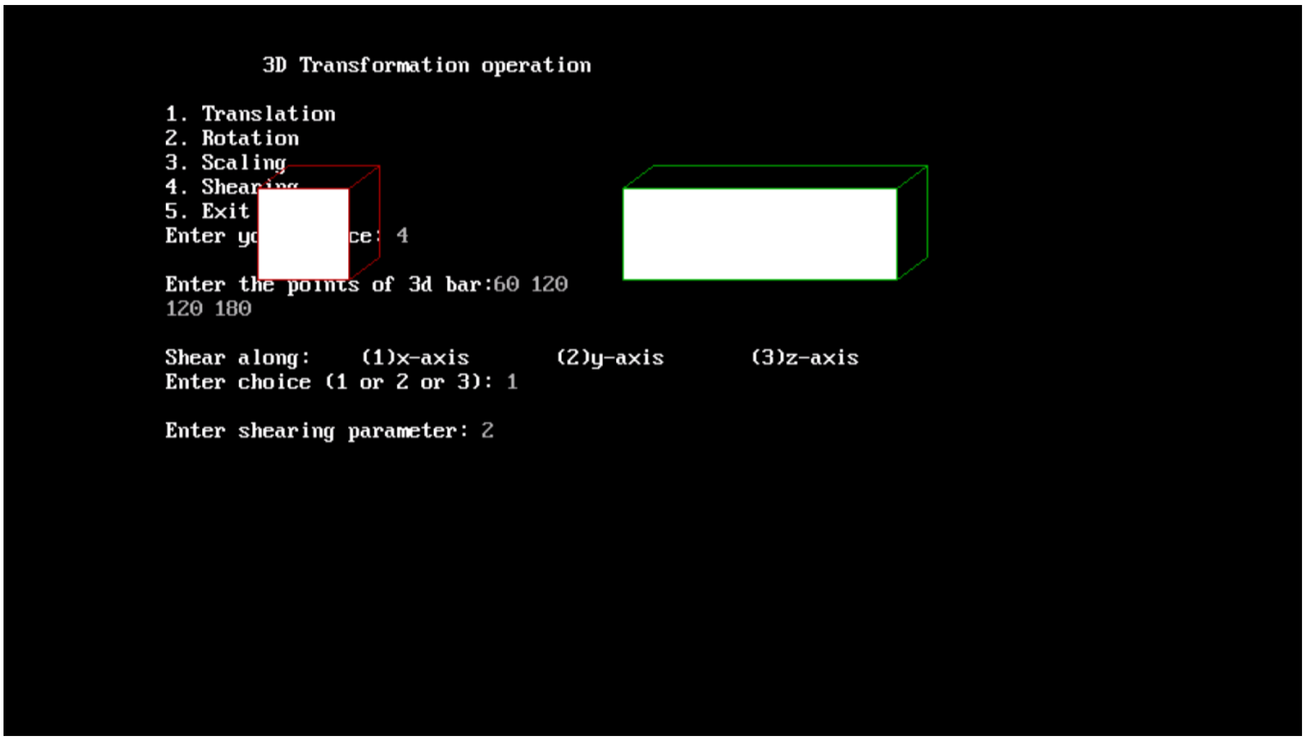
```
        3D Transformation operation

1. Translation
2. Rotation
3. Scaling
4. Shearing
5. Exit
Enter your choice: 2

Enter the points of 3d bar:100 150
150 200

Enter the rotating angle: 45

Rotating along z axis
```

Scaling



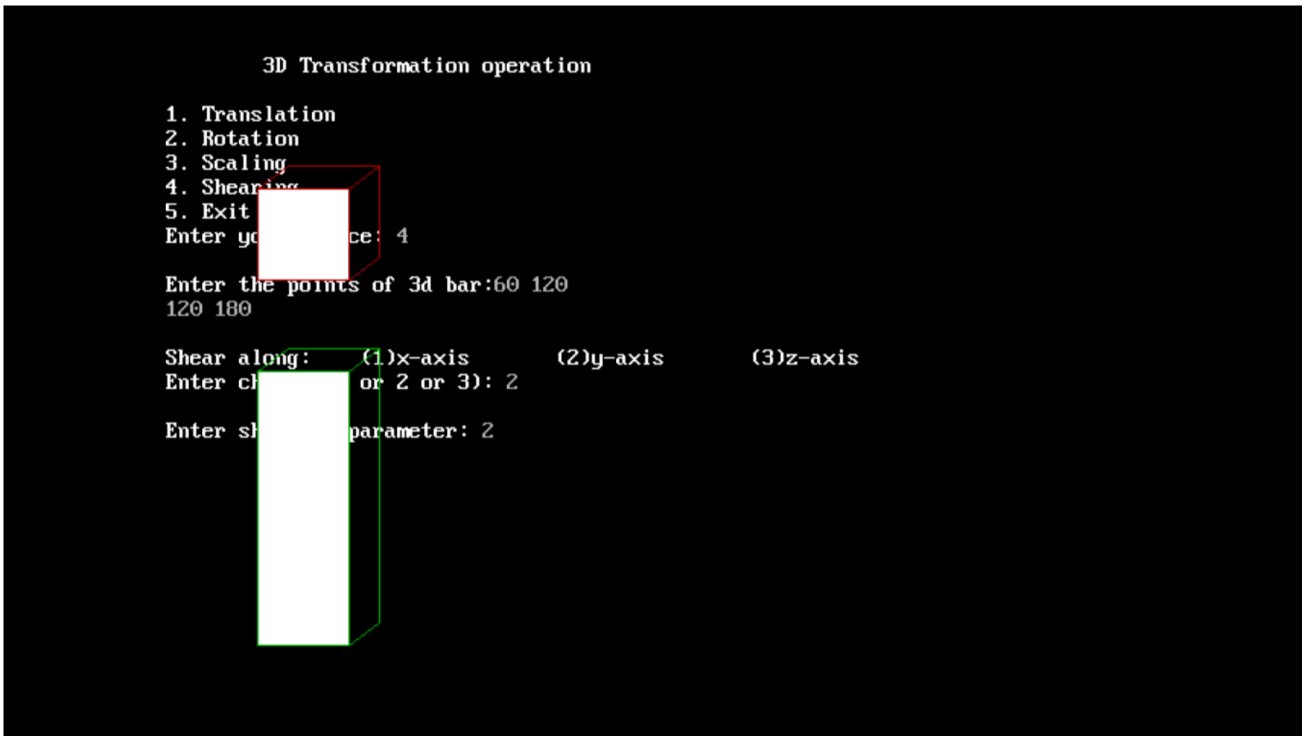
```
        3D Transformation operation

1. Translation
2. Rotation
3. Sca
4. She
5. Exi
Enter your choice: 3

Enter the points of 3d bar:50 100
100 150

 Enter the scaling factor(for x,y,z): 2 2 2
```

Shearing (x-shear)



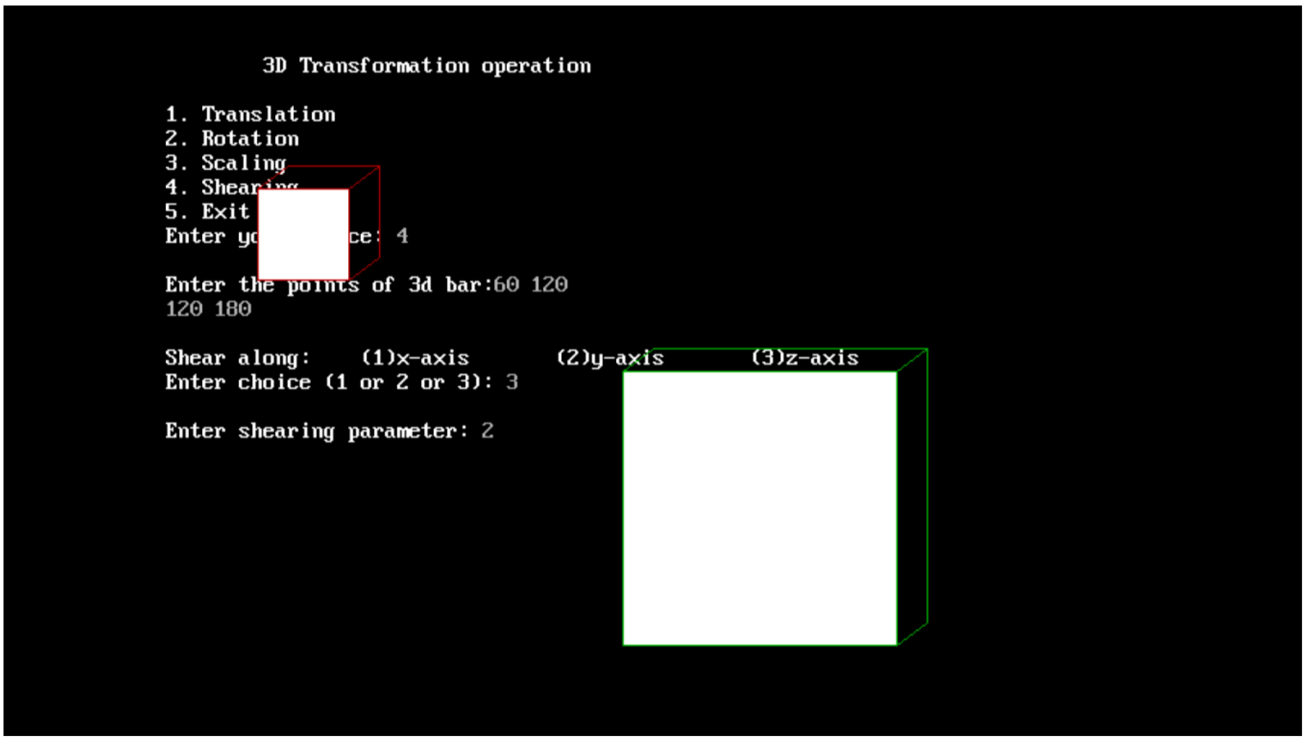
```
        3D Transformation operation

1. Translation
2. Rotation
3. Scaling
4. Shearing
5. Exit
Enter yo        ce: 4

Enter the points of 3d bar:60 120
120 180

Shear along:    (1)x-axis       (2)y-axis       (3)z-axis
Enter choice (1 or 2 or 3): 1

Enter shearing parameter: 2
```

Shearing (y-shear)



```
         3D Transformation operation

1. Translation
2. Rotation
3. Scaling
4. Shear
5. Exit
Enter yo        ce: 4

Enter the points of 3d bar:60 120
120 180

Shear along:    (1)x-axis       (2)y-axis       (3)z-axis
Enter ch        or 2 or 3): 2

Enter sh        parameter: 2
```

Shearing (z-shear)



```
         3D Transformation operation

1. Translation
2. Rotation
3. Scaling
4. Shear
5. Exit
Enter yo        ce: 4

Enter the points of 3d bar:60 120
120 180

Shear along:    (1)x-axis       (2)y-axis       (3)z-axis
Enter choice (1 or 2 or 3): 3

Enter shearing parameter: 2
```

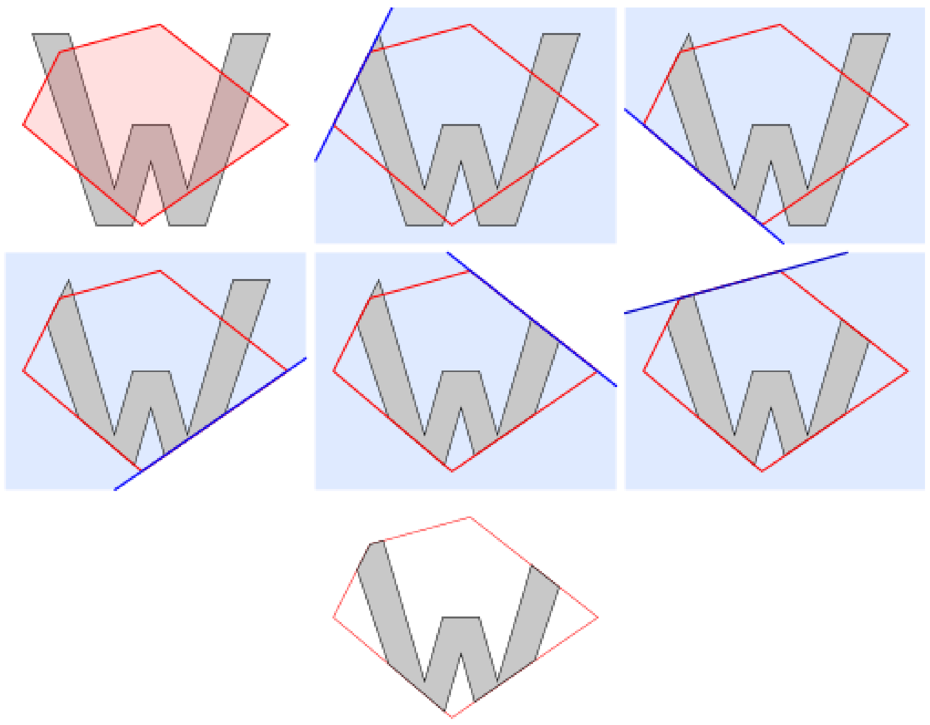
Program 10

Aim: Write a program to clip a Polygon using Sutherland Hodgeman Algorithm.

Theory:

It is performed by processing the boundary of polygon against each window corner or edge. First of all, entire polygon is clipped against one edge, then resulting polygon is considered, then the polygon is considered against the second edge, so on for all four edges. The algorithm begins with an input list of all vertices in the subject polygon. Next, one side of the clip polygon is extended infinitely in both directions, and the path of the subject polygon is traversed. Vertices from the input list are inserted into an output list if they lie on the visible side of the extended clip polygon line, and new vertices are added to the output list where the subject polygon path crosses the extended clip polygon line.

This process is repeated iteratively for each clip polygon side, using the output list from one stage as the input list for the next. Once all sides of the clip polygon have been processed, the final generated list of vertices defines a new single polygon that is entirely visible. Note that if the subject polygon was concave at vertices outside the clipping polygon, the new polygon may have coincident (i.e., overlapping) edges – this is acceptable for rendering, but not for other applications such as computing shadows.



Pseudocode:

```
List outputList = subjectPolygon;

for (Edge clipEdge in clipPolygon) do
    List inputList = outputList;
    outputList.clear();

    for (int i = 0; i < inputList.count; i += 1) do
        Point current_point = inputList[i];
        Point prev_point = inputList[(i − 1) % inputList.count];

        Point Intersecting_point = ComputeIntersection(prev_point, current_point, clipEdge)

        if (current_point inside clipEdge) then
            if (prev_point not inside clipEdge) then
                outputList.add(Intersecting_point);
            end if
            outputList.add(current_point);

        else if (prev_point inside clipEdge) then
            outputList.add(Intersecting_point);
        end if

    done
done
```

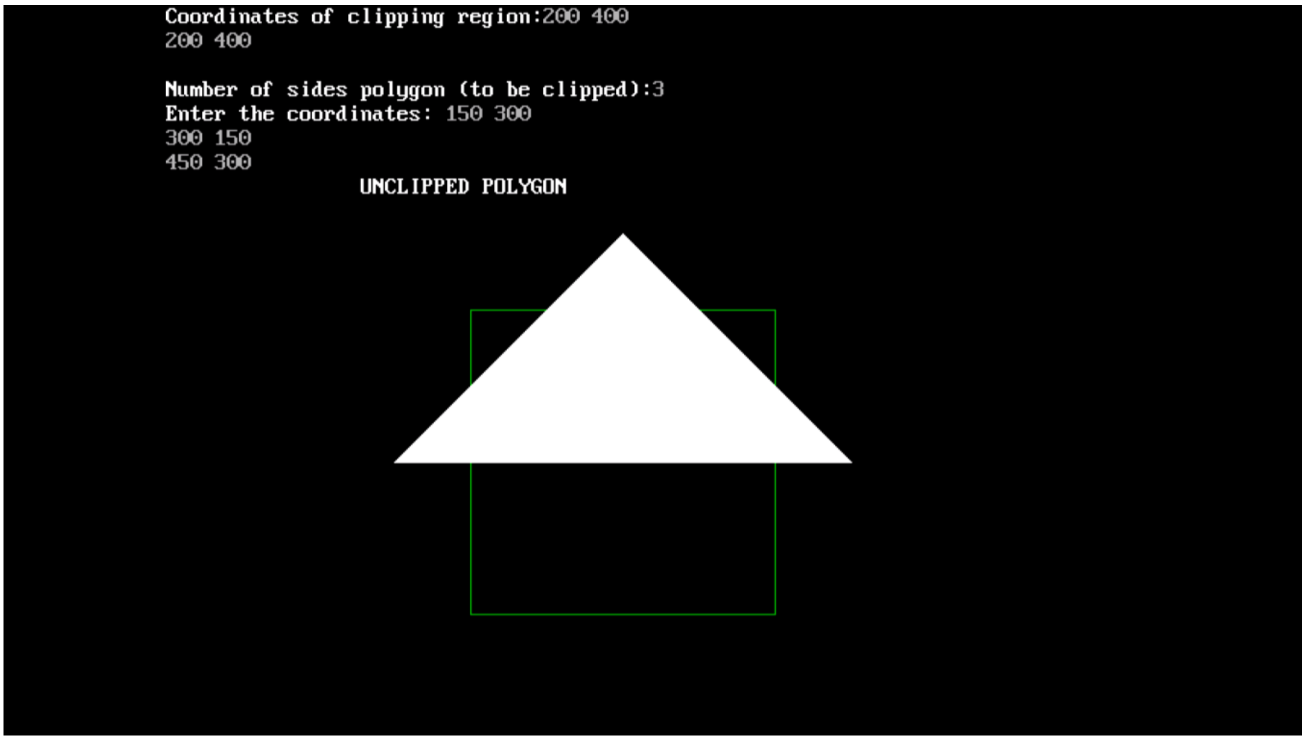
Code:

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

int round(int a){
    return ((int)(a+0.5));
}

float xmin, ymin, xmax, ymax, arr[20], m; int k;
void left_clip(float x1, float y1, float x2, float y2){
    if (x2 - x1)
        m = (y2 - y1) / (x2 - x1);
    else
        m = 100000;
    if(x1 >= xmin && x2 >= xmin){
        arr[k] = x2;
        arr[k + 1] = y2;
        k += 2;
    }
    if(x1 < xmin && x2 >= xmin){
        arr[k] = xmin;
        arr[k + 1] = y1 + m * (xmin - x1);
        arr[k + 2] = x2;
        arr[k + 3] = y2;
        k += 4;
    }
    if(x1 >= xmin && x2 < xmin){
        arr[k] = xmin;
        arr[k + 1] = y1 + m * (xmin - x1);
        k += 2;
    }
}

void top_clip(float x1, float y1, float x2, float y2){
    if (y2 - y1)
        m = (x2 - x1) / (y2 - y1);
    else
        m = 100000;
    if (y1 <= ymax && y2 <= ymax){
        arr[k] = x2;
        arr[k + 1] = y2;
        k += 2;
    }
    if (y1 > ymax && y2 <= ymax){
        arr[k] = x1 + m * (ymax - y1);
        arr[k + 1] = ymax;
        arr[k + 2] = x2;
        arr[k + 3] = y2;
        k += 4;
    }
    if (y1 <= ymax && y2 > ymax){
        arr[k] = x1 + m * (ymax - y1);
        arr[k + 1] = ymax;
        k += 2;
    }
}

void right_clip(float x1, float y1, float x2, float y2){
    if(x2 - x1)
```

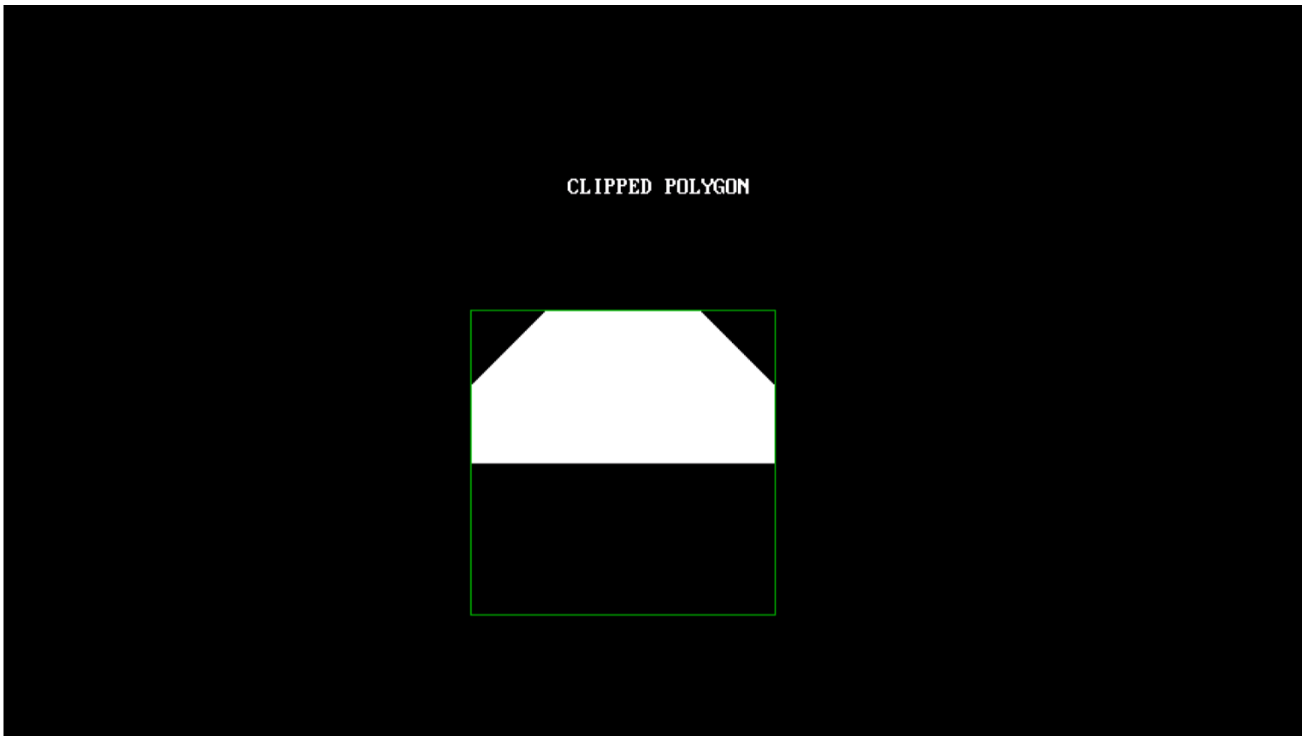
```
        m = (y2 - y1) / (x2 - x1);
    else
        m = 100000;
    if(x1 <= xmax && x2 <= xmax){
        arr[k] = x2;
        arr[k + 1] = y2;
        k += 2;
    }
    if(x1 > xmax && x2 <= xmax){
        arr[k] = xmax;
        arr[k + 1] = y1 + m * (xmax - x1);
        arr[k + 2] = x2;
        arr[k + 3] = y2;
        k += 4;
    }
    if(x1 <= xmax && x2 > xmax){
        arr[k] = xmax;
        arr[k + 1] = y1 + m * (xmax - x1);
        k += 2;
    }
}

void bottom_clip(float x1, float y1, float x2, float y2){
    if(y2 - y1)
        m = (x2 - x1) / (y2 - y1);
    else
        m = 100000;
    if(y1 >= ymin && y2 >= ymin){
        arr[k] = x2;
        arr[k + 1] = y2;
        k += 2;
    }
    if(y1 < ymin && y2 >= ymin){
        arr[k] = x1 + m * (ymin - y1);
        arr[k + 1] = ymin;
        arr[k + 2] = x2;
        arr[k + 3] = y2;
        k += 4;
    }
    if(y1 >= ymin && y2 < ymin){
        arr[k] = x1 + m * (ymin - y1);
        arr[k + 1] = ymin;
        k += 2;
    }
}

void main(){

    int g_driver = DETECT, g_mode;
    initgraph(&g_driver, &g_mode, "C:\\TURBOC3\\BGI");

    float xi, yi, xf, yf, polyy[20];
    int n, poly[20];
    //clrscr();
    cout<<"Coordinates of clipping region:";
    cin>>xmin>>xmax;
    cin>>ymin>>ymax;
    cout<<"\nNumber of sides polygon (to be clipped):";
    cin>>n;
    cout<<"Enter the coordinates: ";
    for(int i = 0; i < 2 * n; i++)
        cin>>polyy[i];
    polyy[i] = polyy[0];
    polyy[i + 1] = polyy[1];
    for (i = 0; i < 2 * n + 2; i++)
        poly[i] = round(polyy[i]);
    setcolor(GREEN);
    rectangle(xmin, ymax, xmax, ymin);
    cout<<"\t\tUNCLIPPED POLYGON";
    setcolor(WHITE);
    fillpoly(n, poly);
    getch();
    cleardevice();
    k = 0;
    for(i=0; i<2*n; i+=2)
        left_clip(polyy[i], polyy[i+ 1], polyy[i + 2], polyy[i
+ 3]);
    n = k / 2;
    for(i=0; i<k; i++)
        polyy[i] = arr[i];
    polyy[i] = polyy[0];
    polyy[i+1] = polyy[1];
    k=0;
    for(i=0; i<2*n; i+=2)
```

```
        top_clip(polyy[i], polyy[i+1], polyy[i+2], polyy[i+3]);
    n=k/2;
    for(i=0; i<k; i++)
        polyy[i] = arr[i];
    polyy[i] = polyy[0];
    polyy[i+1] = polyy[1];
    k=0;
    for(i=0;i<2*n;i+=2)
        right_clip(polyy[i],polyy[i+1],polyy[i+2],polyy[i+3]);
    n=k/2;
    for(i=0;i<k;i++)
        polyy[i]=arr[i];
    polyy[i]=polyy[0];
    polyy[i+1]=polyy[1];
    k = 0;
    for(i=0; i<2*n; i+=2)
        bottom_clip(polyy[i], polyy[i+1], polyy[i+2], polyy[i+3
]);
    for(i=0; i<k; i++)
        poly[i]=round(arr[i]);
    if(k)
        fillpoly(k/2,poly);
    setcolor(GREEN);
    rectangle(xmin, ymax, xmax, ymin);
    cout<<"CLIPPED POLYGON";
    getch();
    closegraph();
}
```

Output:



```
Coordinates of clipping region:200 400
200 400

Number of sides polygon (to be clipped):3
Enter the coordinates: 150 300
300 150
450 300
                UNCLIPPED POLYGON
```



```
CLIPPED POLYGON
```

Program 11

Aim: Write a program to implement Z-buffer algorithm. Perform the algorithm on a set of at least 5 objects in a viewing coordinate system.

Theory:

Z-buffer, which is also known as the Depth-buffer method is one of the commonly used method for hidden surface detection. It is an Image space method. Image space methods are based on the pixel to be drawn on 2D. For these methods, the running time complexity is the number of pixels times number of objects. And the space complexity is two times the number of pixels because two arrays of pixels are required, one for frame buffer and the other for the depth buffer.

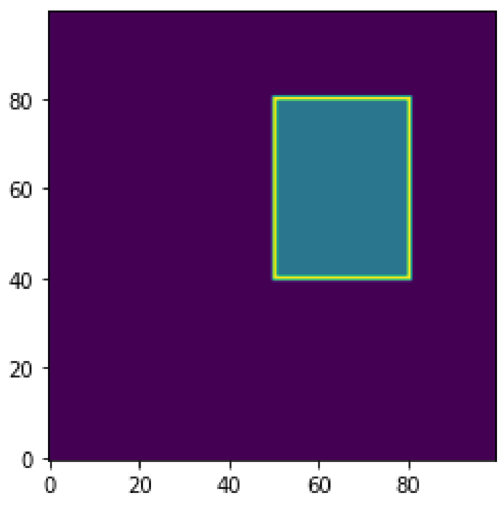
Algorithm:

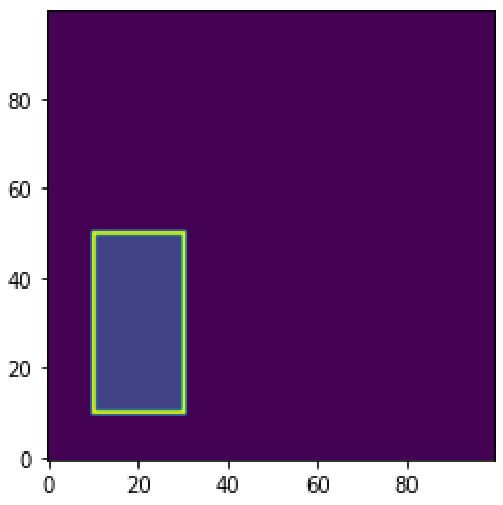
```
First of all, initialize the depth of each pixel.
i.e,  d(i, j) = infinite (max length)
Initialize the color value for each pixel
as c(i, j) = background color
for each polygon, do the following steps :

for (each pixel in polygon's projection)
{
    find depth i.e, z of polygon
    at (x, y) corresponding to pixel (i, j)

    if (z < d(i, j))
    {
        d(i, j) = z;
        c(i, j) = color;
    }
}
```

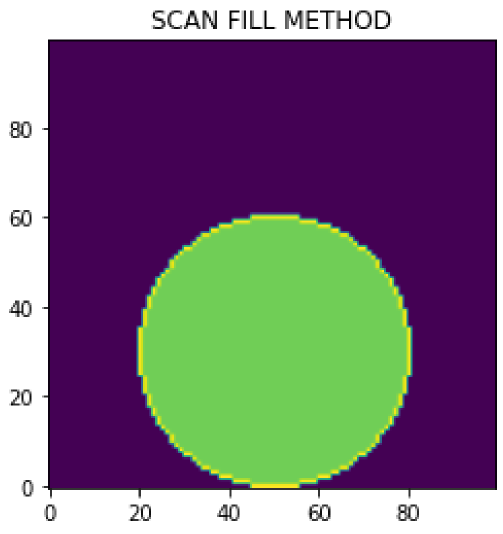
Code:

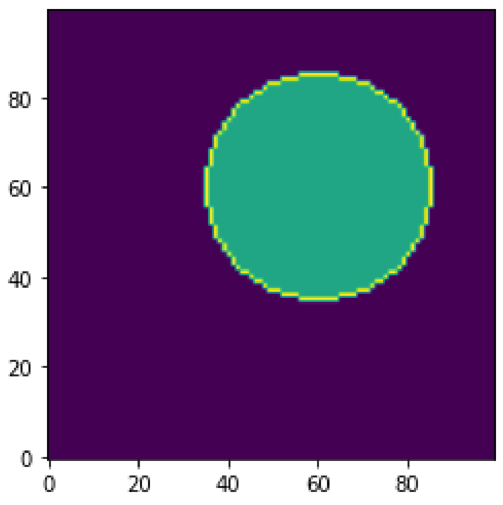
```python
import matplotlib.pyplot as plt
import numpy as np

def circle_imp(x0, y0, r):
    x = 0
    y = r
    pts_x = [x0+x, x0+x, x0+y, x0-y]
    pts_y = [y0+y, y0-y, y0+x, y0+x]
    d = 1 - r
    while y >= x:
        dE = 2*x + 1
        dSE = 2*x - 2*y + 1
        if d <= 0:
            d += dE
        else:
            d += dSE
            y = y - 1
        pts_x.append(x0+x)
        pts_x.append(x0-x)
        pts_x.append(x0+x)
        pts_x.append(x0-x)
        pts_x.append(x0+y)
        pts_x.append(x0-y)
        pts_x.append(x0+y)
        pts_x.append(x0-y)
        pts_y.append(y0+y)
        pts_y.append(y0+y)
        pts_y.append(y0-y)
        pts_y.append(y0-y)
        pts_y.append(y0+x)
        pts_y.append(y0+x)
        pts_y.append(y0-x)
        pts_y.append(y0-x)
        x += 1
    return pts_x, pts_y

def midpt_line(x0, y0, xn, yn):

    dy = yn-y0
    dx = xn-x0
    pts_x = []
    pts_y = []
    pts_x.append(x0)
    pts_y.append(y0)
    x = x0
    y = y0
    if dy >= 0 and dx >= 0:
        if dy > dx:
            d = 2*dx - dy
            dE = 2*dx
            dNE = 2*(dx-dy)
            for i in range(1, yn-y0 + 1):
                if d <= 0:
                    d = d + dE
                else:
                    d = d + dNE
                    x = x + 1
                pts_x.append(x)
                pts_y.append(y + i)
        else:
            d = 2*dy - dx
            dE = 2*dy
            dNE = 2*(dy-dx)
            for i in range(1, xn-x0+1):
                if d <= 0:
                    d = d + dE
                else:
                    d = d + dNE
                    y = y + 1
                pts_x.append(x + i)
                pts_y.append(y)
    else:
        if abs(dy) > abs(dx):
            dy = abs(dy)
            dx = abs(dx)
            d = 2*dx - dy
            dE = 2*dx
            dNE = 2*(dx-dy)
            for i in range(1, abs(yn-y0)+1):
                if d <= 0:
                    d = d + dE
                else:
                    d = d + dNE
                    x = x + 1
                pts_x.append(x)
                pts_y.append(y - i)
        else:
            dy = abs(dy)
            dx = abs(dx)
            d = 2*dy - dx
            dE = 2*dy
            dNE = 2*(dy-dx)
            for i in range(1, abs(xn-x0)+1):
                if d <= 0:
                    d = d + dE
                else:
                    d = d + dNE
                    y = y - 1
                pts_x.append(x + i)
                pts_y.append(y)
    return pts_x, pts_y

def FloodFill(screen, x, y, newC, oldC):

    currPix = screen[y, x]
    if currPix == oldC:
        screen[y, x] = newC
        FloodFill(screen, x + 1, y, newC, oldC)
        FloodFill(screen, x - 1, y, newC, oldC)
        FloodFill(screen, x, y + 1, newC, oldC)
        FloodFill(screen, x, y - 1, newC, oldC)

    x1 = np.zeros((100, 100))
    coor_y, coor_x = midpt_line(50, 40, 80, 40)
    for i, j in zip(coor_x, coor_y):
        x1[i, j] = 255
    coor_y, coor_x = midpt_line(50, 40, 50, 80)
    for i, j in zip(coor_x, coor_y):
        x1[i, j] = 255
    coor_y, coor_x = midpt_line(50, 80, 80, 80)
    for i, j in zip(coor_x, coor_y):
        x1[i, j] = 255
    coor_y, coor_x = midpt_line(80, 80, 80, 40)
    for i, j in zip(coor_x, coor_y):
        x1[i, j] = 255
    FloodFill(x1, 60, 50, 100, 0)
    plt.figure()
    plt.imshow(x1)
    plt.gca().invert_yaxis()
    obj1 = np.where(x1 > 0)

    plt.figure()
    x2 = np.zeros((100, 100))
    coor_y, coor_x = midpt_line(10, 10, 10, 50)
    for i, j in zip(coor_x, coor_y):
        x2[i, j] = 255
    coor_y, coor_x = midpt_line(10, 50, 30, 50)
    for i, j in zip(coor_x, coor_y):
        x2[i, j] = 255
    coor_y, coor_x = midpt_line(30, 50, 30, 10)
    for i, j in zip(coor_x, coor_y):
        x2[i, j] = 255
    coor_y, coor_x = midpt_line(10, 10, 30, 10)
    for i, j in zip(coor_x, coor_y):
        x2[i, j] = 255
    FloodFill(x2, 20, 20, 50, 0)
    plt.figure()
    plt.imshow(x2)
    plt.gca().invert_yaxis()
    obj2 = np.where(x2 > 0)

    plt.figure()
    x3 = np.zeros((100, 100))
    coor = []
    coor_y, coor_x = circle_imp(50, 30, 30)
    for i, j in zip(coor_x, coor_y):
        x3[i, j] = 255
        coor.append((j, i))
    FloodFill(x3, 50, 20, 200, 0)
    plt.figure()
    plt.imshow(x3)
    plt.title('SCAN FILL METHOD')
    plt.gca().invert_yaxis()
    obj3 = np.where(x3 > 0)

    plt.figure()
    x4 = np.zeros((100, 100))
    coor = []
    coor_y, coor_x = circle_imp(60, 60, 25)
    for i, j in zip(coor_x, coor_y):
        x4[i, j] = 255
        coor.append((j, i))
    FloodFill(x4, 60, 60, 150, 0)
    plt.figure()
    plt.imshow(x4)
    plt.gca().invert_yaxis()
    obj4 = np.where(x4 > 0)

    x5 = np.zeros((100, 100))
    coor_y, coor_x = midpt_line(70, 10, 90, 10)
    for i, j in zip(coor_x, coor_y):
        x5[i, j] = 255
    coor_y, coor_x = midpt_line(90, 10, 90, 30)
    for i, j in zip(coor_x, coor_y):
        x5[i, j] = 255
    coor_y, coor_x = midpt_line(70, 30, 90, 30)
    for i, j in zip(coor_x, coor_y):
        x5[i, j] = 255
    coor_y, coor_x = midpt_line(70, 10, 70, 30)
    for i, j in zip(coor_x, coor_y):
        x5[i, j] = 255
    FloodFill(x5, 80, 20, 250, 0)
    plt.figure()
    plt.imshow(x5)
    plt.gca().invert_yaxis()
    obj5 = np.where(x5 > 0)


    depth = np.full((100, 100), 10000)
    new = np.zeros((100, 100))

    for poly in [(obj1, 100, 5), (obj2, 50, 2), (obj3, 200, 10), (ob
j4, 150, 20), (obj5, 250, 1)]:
        d = poly[2]
        c = poly[1]
        poly = poly[0]
        for i, j in zip(poly[0], poly[1]):
            if depth[i, j] > d:
                depth[i, j] = d
                new[i, j] = c
    plt.figure()
    plt.imshow(new)
    plt.gca().invert_yaxis()
```

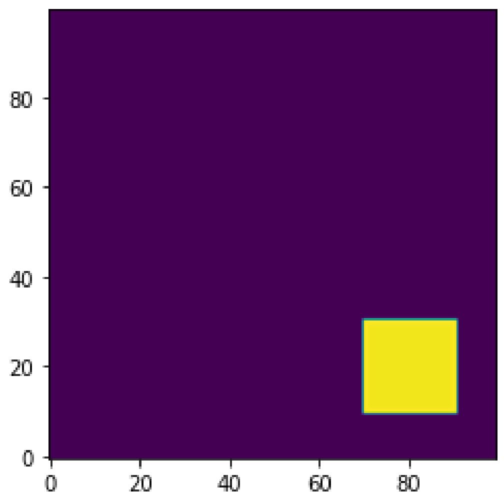
Output:

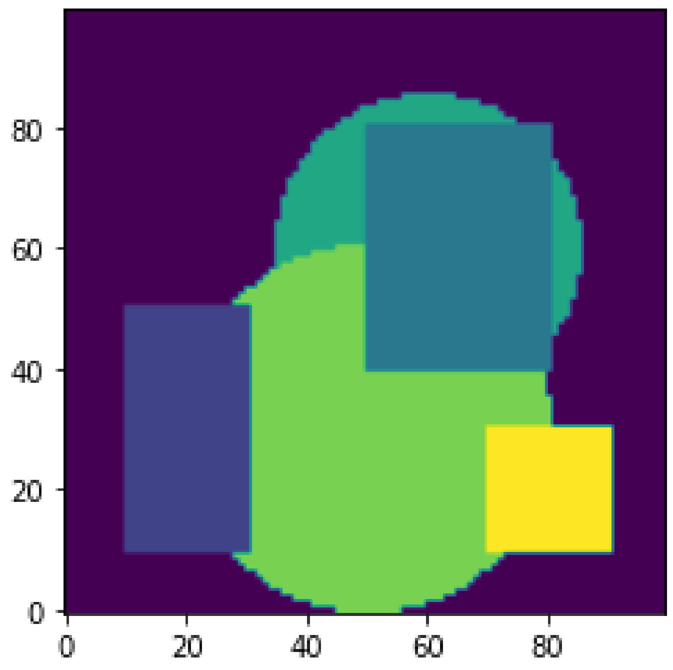

0
20
40
60
80


0
20
40
60
80


SCAN FILL METHOD
0
20
40
60
80


0
20
40
60
80


0
20
40
60
80


0
20
40
60
80
0
20
40
60
80

Program 12

Aim: Write A program to draw an arbitrary curve using Hermite and Bezier curve drawing method.

Theory:

A Hermite curve is a spline where every piece is a third-degree polynomial defined in Hermite form: that is, by its values and initial derivatives at the end points of the equivalent domain interval. Cubic Hermite splines are normally used for interpolation of numeric values defined at certain dispute values x_1, x_2, x_3, … x_n, to achieve a smooth continuous function. The data should have the preferred function value and derivative at each Xk. The Hermite formula is used to every interval (X_k, X_{k+1}) individually. The resulting spline become continuous and will have first derivative.

Cubic polynomial splines are specially used in computer geometric modeling to attain curves that pass via defined points of the plane in 3D space. In these purposes, each coordinate of the plane is individually interpolated by a cubic spline function of a divided parameter ‘t’.

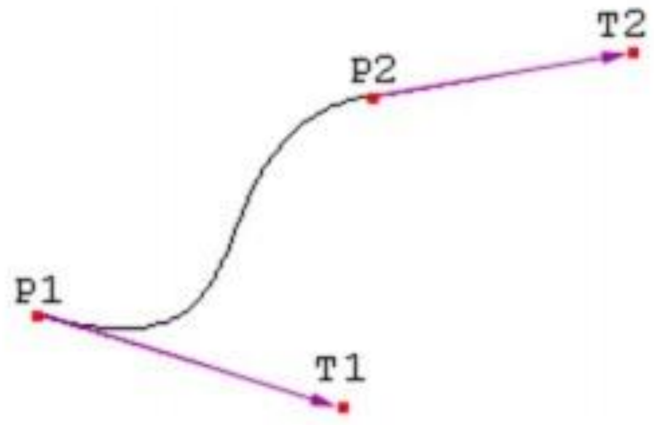


Fig.2.2. Hermite curve

Bezier curve is discovered by the French engineer **Pierre Bézier**. These curves can be generated under the control of other points. Approximate tangents by using control points are used to generate curve. The Bezier curve can be represented mathematically as –

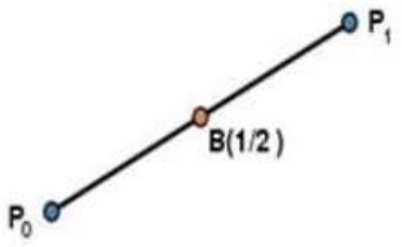
$$\sum_{k=0}^{n} P_i B_i^n(t)$$

Where p_i is the set of points and $B^n{}_i(t)$ represents the Bernstein polynomials which are given by –

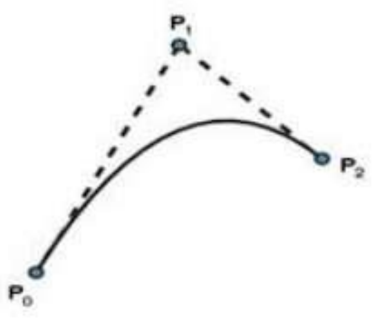
$$B_i^n(t) = \binom{n}{i}(1-t)^{n-i}t^i$$

Where **n** is the polynomial degree, **i** is the index, and **t** is the variable.

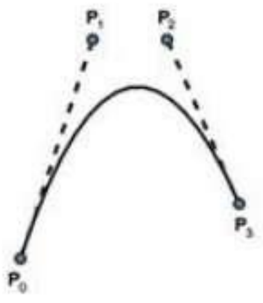
The simplest Bézier curve is the straight line from the point P_0 to P_1. A quadratic Bezier curve is determined by three control points. A cubic Bezier curve is determined by four control points.



Simple Bezier Curve



Quadratic Bazier Curve



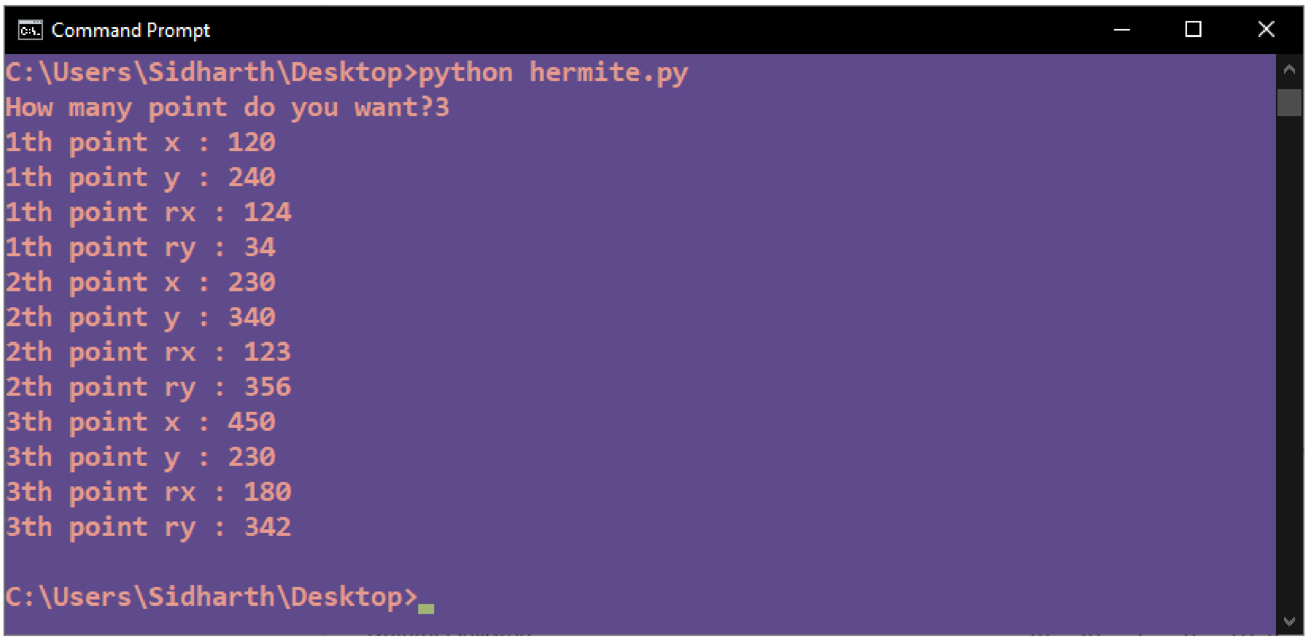
Cubic Bazier Curve

Code:

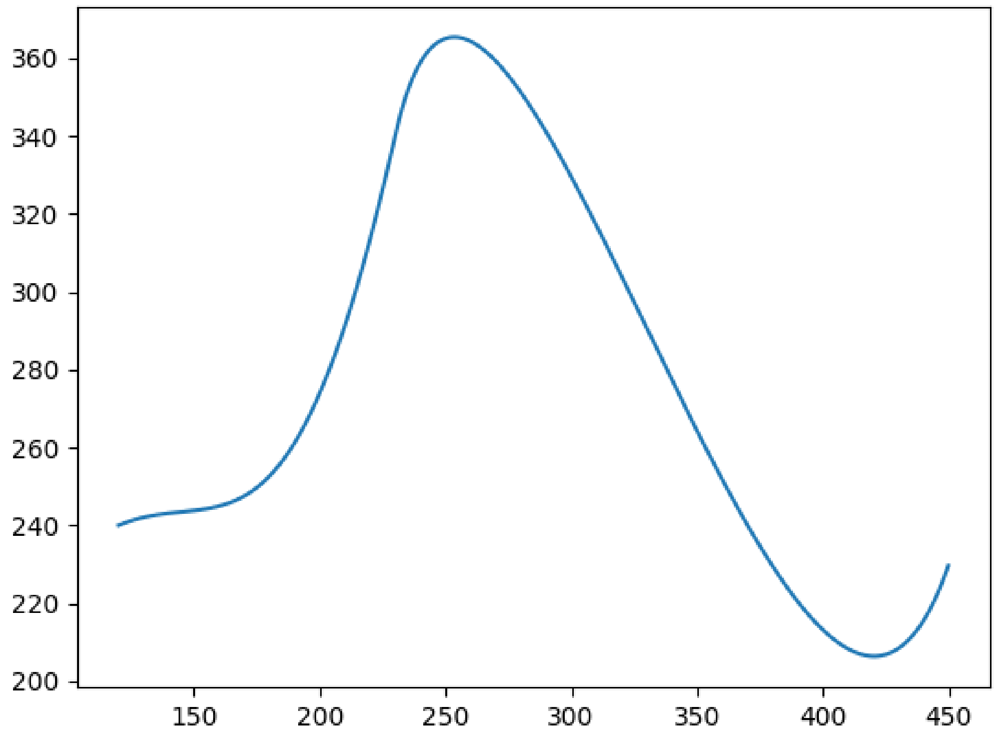
```
import math
import matplotlib.pyplot as plt

x = []
y = []
u = 0.001

def hermite(px , py , qx , qy , prx , pry , qrx , qry , u):
    tt = u
    while(tt < 1):
        resx = (1-3*math.pow(tt,2) + 2*math.pow(tt,3))*px +
(3*math.pow(tt,2) - 2*math.pow(tt,3))*qx + (tt-2*math.pow(tt,2) +
math.pow(tt,3))*prx + (-math.pow(tt,2)+math.pow(tt,3))*qrx
        resy = (1-3*math.pow(tt,2) + 2*math.pow(tt,3))*py +
(3*math.pow(tt,2) - 2*math.pow(tt,3))*qy+(tt-2*math.pow(tt,2) +
math.pow(tt,3))*pry + (-math.pow(tt,2)+math.pow(tt,3))*qry

        x.append(resx)
        y.append(resy)
        tt = tt+u

def main():
    n = int(input("How many point do you want?"))
    px , py , qx , qy , prx , pry , qrx , qry = [] , [] , [] , []
 , [] , [] , [] , []
    for i in range(n):
        px.append(int(input(str(i + 1) +"th point x : ")))
        py.append(int(input(str(i + 1) +"th point y : ")))
        prx.append(int(input(str(i + 1) +"th point rx : ")))
        pry.append(int(input(str(i + 1) +"th point ry : ")))
        if(i > 0):
            hermite(px[i - 1] , py[i - 1] , px[i] , py[i] , prx[i
 - 1] , pry[i - 1] , prx[i] , pry[i] , u)

    plt.plot(x , y)
    plt.show()

if __name__ == '__main__':
    main()
```

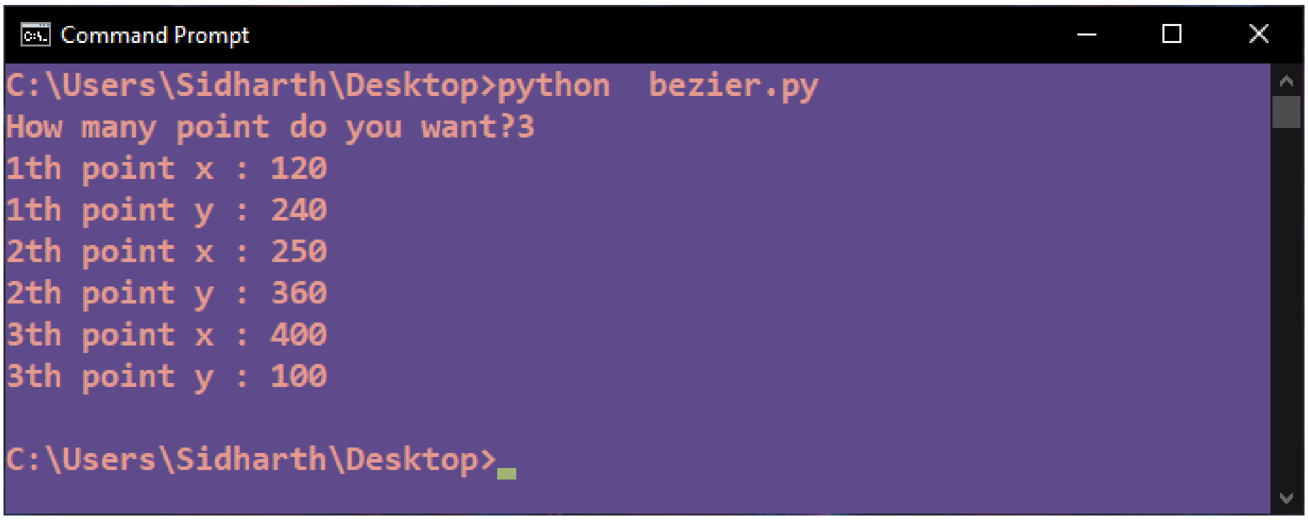
```python
import math
import matplotlib.pyplot as plt


u = 0.001

def binomial(i, n):
    return math.factorial(n) / float(
        math.factorial(i) * math.factorial(n - i))


def bezier(px , py , u , x , y):
    n = len(px)
    for i in range(n):
        bio = binomial(i , n - 1)
        t = 0
        while t < 1:
            resX = math.pow(t , i) * math.pow( (1 - t) , n - i - 1) *
px[i]
            resY = math.pow(t , i) * math.pow( (1 - t) , n - i - 1) *
py[i]
            x.append(bio * resX)
            y.append(bio * resY)
            t = t + u
    for i in range(n - 1):
        for j in range(1000):
            x[j] = x[j] + x[(1000*(i+1)) + j]
            y[j] = y[j] + y[(1000*(i+1)) + j]

def main():
    x = []
    y = []
    n = int(input("How many point do you want?"))
    px , py = [] , []
    for i in range(n):
        px.append(int(input(str(i + 1) +"th point x : ")))
        py.append(int(input(str(i + 1) +"th point y : ")))

    bezier(px , py , u , x , y)
    x = x[:1000]
    y = y[:1000]

    plt.plot(x , y)
    plt.show()


if __name__ == '__main__':
    main()
```

Output:

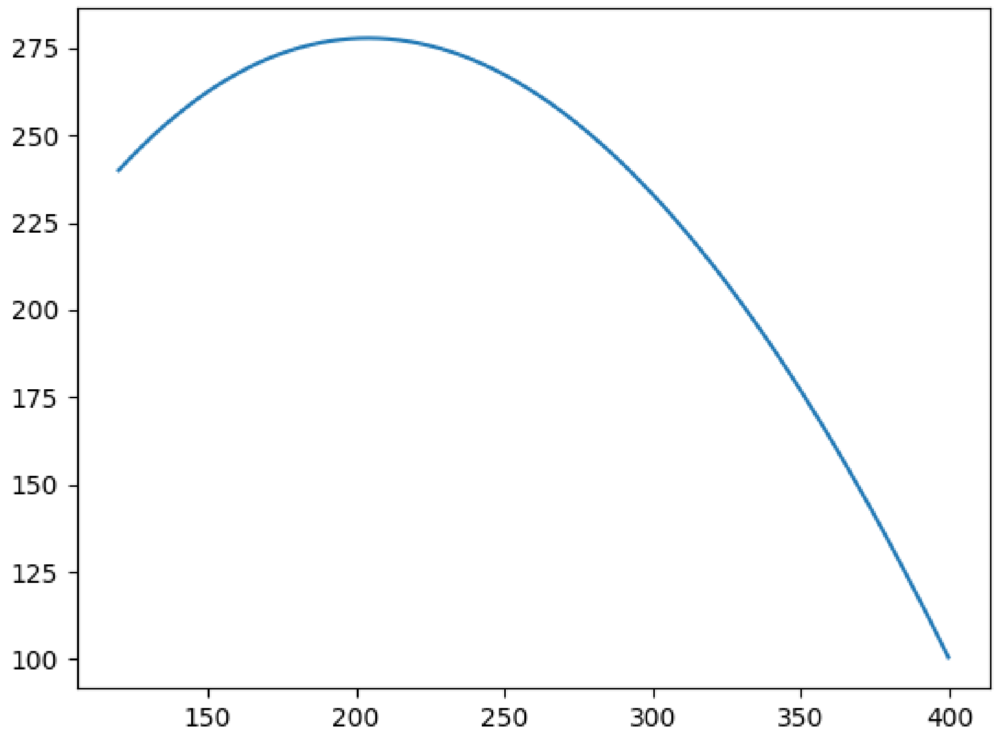


```
C:\Users\Sidharth\Desktop>python hermite.py
How many point do you want?3
1th point x : 120
1th point y : 240
1th point rx : 124
1th point ry : 34
2th point x : 230
2th point y : 340
2th point rx : 123
2th point ry : 356
3th point x : 450
3th point y : 230
3th point rx : 180
3th point ry : 342

C:\Users\Sidharth\Desktop>
```





```
C:\Users\Sidharth\Desktop>python  bezier.py
How many point do you want?3
1th point x : 120
1th point y : 240
2th point x : 250
2th point y : 360
3th point x : 400
3th point y : 100

C:\Users\Sidharth\Desktop>
```


275
250
225
200
175
150
125
100
150
200
250
300
350
400