
Computer Graphics Lab Manual (Batch - 2020-21) (DSY-SEM-III)

Experiment 01

Name : Digital Differential Analyzer Line Drawing Algorithm

Algorithm :

Step 1: Accept the end point co-ordinates of the line segment AB i.e. A (x_1, y_1) and B (x_2, y_2)

Step 2: Calculate

$$dx = x_2 - x_1$$

$$dy = y_2 - y_1$$

Step 3: If $\text{abs}(dx) \geq \text{abs}(dy)$ then
steps = $\text{abs}(dx)$

Else

$$\text{steps} = \text{abs}(dy)$$

Step 4: Let $x_{\text{increment}} = dx/\text{steps};$

$$y_{\text{increment}} = dy/\text{steps};$$

Step 5: Display the pixel at starting position
putpixel (x_1, y_1 , WHITE)

Step 6: Compute the next co-ordinate position along the line path.

$$x_{k+1} = x_k + x_{\text{increment}}$$

$$y_{k+1} = y_k + y_{\text{increment}}$$

putpixel (x_{k+1}, y_{k+1} , WHITE)

Step 7: If $x_{k+1} = x_2$ OR/AND

$$y_{k+1} = y_2$$

Then Stop

Else go to step 4.

Program :

```
/* DDA LINE DRAWING ALGORITHM */
```

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

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

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

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

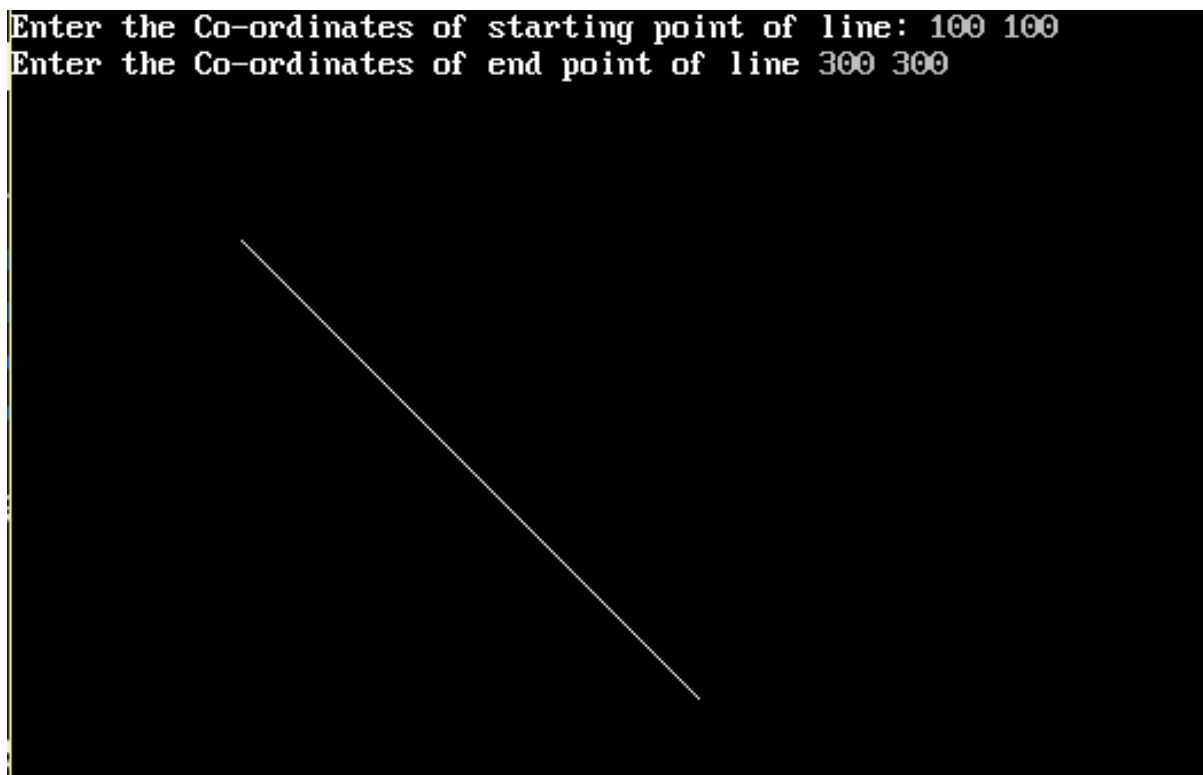
```
////////// DDA LINE DRAWING FUNCTION //////////
```

```
void ddaLine(int x1,int y1,int x2,int y2)
```

```
{  
    float x=x1,y=y1,dx,dy;  
    int step,i;  
  
    putpixel(x1,y1,WHITE);  
  
    if(abs(x2-x1)>=abs(y2-y1))  
        step=abs(x2-x1);  
    else  
        step=abs(y2-y1);  
    dx=(float)(x2-x1)/step;  
    dy=(float)(y2-y1)/step;  
    for(i=1;i<=step;i++)  
    {  
        x=x+dx;  
        y=y+dy;  
        putpixel((int) x,(int) y,WHITE);  
    }  
}  
  
void main()  
{  
    int x1,y1,x2,y2,gd,gm;  
    detectgraph(&gd,&gm);  
    initgraph(&gd, &gm, "C:\\TC\\BGI");  
  
    printf("Enter the Co-ordinates of starting point of line: ");  
    scanf("%d %d",&x1,&y1);
```

```
printf("Enter the Co-ordinates of end point of line ");  
scanf("%d %d",&x2,&y2);  
  
ddaLine(x1,y1,x2,y2);  
getch();  
CLOSEGRAPH();  
}
```

Output :



Advantages :

- (1) It avoids using the multiplication operation which is costly in terms of time complexity.
- (2) The primary advantages of a DDA over the conventional analog differential analyzer are greater precision of the results and the lack of drift/noise/slip/lash in the calculations.

Applications :

- (1) DDA algorithm uses floating points i.e. Real Arithmetic
- (2) DDA algorithm uses multiplication and division in its operations

-
- (3) DDA algorithm uses an enormous number of floating-point multiplications so it is expensive.
 - (4) DDA algorithm round off the coordinates to integer that is nearest to the line.
 - (5) It is the simplest algorithm and it does not require special skills for implementation
 - (6) After execution of simulation codes of DDA Algorithm, I come to the conclusion that for DDA algorithm, slope is the crucial factor in line generation.

Limitations :

- (1) In DDA algorithm, each time the calculated pixel co-ordinates are rounded to nearest integer values. There is possibility that the points may drift away from the true line path due to rounding of pixels position.
- (2) Rounding off is time consuming. As floating point operations are involved in the operation the DDA is not faster algorithm.

-X-X-X-

Experiment 02

Name : Midpoint Circle Generation Algorithm

Algorithm :

Step 1:

Accept the radius r and center (x_c, y_c) of a circle. The first point of the circumference of a circle with center as origine is
 $(x_0, y_0) = (0, r)$

Step 2:

Calculate the initial decision parameter as
 $P_0 = 5/4 - r \approx 1 - r$ (Θ radius is integer value)

Step 3:

At each x_k position starting at $k = 0$ perform the following test.

If $P_k < 0$

Then

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

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

$$P_{k+1} = P_k + 2x_k + 3$$

Otherwise

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

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

$$P_{k+1} = P_k + 2(x_k - y_k) + 5$$

Step 4:

Determine the symmetry points in other seven octants.

Step 5:

Translate each calculated pixel position by $T(x_k, y_k)$ and display the pixel.

$$x = x_{k+1} + x_c$$

$$y = y_{k+1} + y_c$$

putpixel (x, y, WHITE)

Step 6:

Repeat step 3 through 5 until $x \geq y$.

Step 7:

Stop

Program :

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<dos.h>
void main()
{
    int gd,gm;
    int i,r,x,y,xc,yc;
    float p;

    detectgraph(&gd,&gm);
    initgraph(&gd,&gm,"C:\\TURBOC3\\BGI");
    printf("Enter center of circle=");
    scanf("%d %d",&xc,&yc);
    printf("enter radius of circle=");
    scanf("%d",&r);
    x=0;
    y=r;
    p=1.25-r;
    do
    {
        if(p<0.0)
        {
            x += 1;
            p += (2*x) + 3;
        }
    }
```

```
else
{
    x += 1;
    y -= 1;
    p += 2*(x-y) + 5;
}
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);
delay(10);
}while(x<y);
getch();
}
```

Output :**Advantages :**

- It is a powerful and efficient algorithm.
- The entire algorithm is based on the simple equation of circle $X^2 + Y^2 = R^2$.
- It is easy to implement from the programmer's perspective.
- This algorithm is used to generate curves on raster displays.

Disadvantages :

- Accuracy of the generating points is an issue in this algorithm.
- The circle generated by this algorithm is not smooth. (Zagged effect can be seen)
- This algorithm is time consuming.

-X-X-X-

Experiment 03 (A)

Name : Boundary Fill Algorithm

Theory :

Boundary Fill Algorithm starts at a pixel inside the polygon to be filled and paints the interior proceeding outwards towards the boundary. This algorithm works **only if** the color with which the region has to be filled and the color of the boundary of the region are different. If the boundary is of one single color, this approach proceeds outwards pixel by pixel until it hits the boundary of the region.

Boundary Fill Algorithm is recursive in nature. It takes an interior point(x, y), a fill color, and a boundary color as the input. The algorithm starts by checking the color of (x, y). If it's color is not equal to the fill color and the boundary color, then it is painted with the fill color and the function is called for all the neighbours of (x, y). If a point is found to be of fill color or of boundary color, the function does not call its neighbours and returns. This process continues until all points up to the boundary color for the region have been tested.

Algorithm :

Recursive procedure to fill area in 4 connected way:

Procedure Boundary_Fill_(x, y, Fill-colour, Boundary-colour)

START:

```
{
    current = getpixel (x, y);
    If (current ≠ boundary-colour) AND
    (current ≠ fill-colour)
    Then
    {
        setpixel(x, y, Fill-Colour);
        Boundary_Fill_(x+1,y,Fill-Colour, Boundary-Colour);
        Boundary_Fill_(x-1,y,Fill-Colour, Boundary-Colour);
        Boundary_Fill_(x,y+1,Fill-Colour, Boundary-Colour);
        Boundary_Fill_(x,y-1,Fill-Colour, Boundary-Colour);
    }
} END
```

Recursive procedure to fill area in 8-connected way:

Procedure Boundary_Fill_(x, y, Fill-Colour, Boundary-Colour)

START:

```
{
    current = getpixel (x, y);
    If (current ≠ boundary-colour) AND
    (current ≠ boundary-colour)
    Then
    {
        setpixel(x, y, Fill-Colour);
        Boundary_Fill_(x+1,y,Fill-Colour, Boundary-Colour);
        Boundary_Fill_(x+1,y-1,Fill-Colour, Boundary-Colour);
        Boundary_Fill_(x,y-1,Fill-Colour, Boundary-Colour);
        Boundary_Fill_(x-1,y-1,Fill-Colour, Boundary-Colour);
        Boundary_Fill_(x-1,y,Fill-Colour, Boundary-Colour);
        Boundary_Fill_(x-1,y+1,Fill-Colour, Boundary-Colour);
        Boundary_Fill_(x,y+1,Fill-Colour, Boundary-Colour);
        Boundary_Fill_(x+1,y+1,Fill-Colour, Boundary-Colour);
    }
} END
```

Program :

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

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

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

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

```
#include<dos.h>
```

```
void fill_right(x,y)
```

```
int x , 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);
    }
}

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

void main()
{
    int x , y ,a[10][10];
    int gd, gm ,n,i;
    detectgraph(&gd,&gm);
    initgraph(&gd,&gm,"c:\\tc\\bgi");
    printf("\n\n\tEnter the no. of edges of polygon : ");
    scanf("%d",&n);
```

```
printf("\n\n\tEnter the cordinates of polygon :\n\n\n ");
for(i=0;i<n;i++)
{
    printf("\tX%d Y%d : ",i,i);
    scanf("%d %d",&a[i][0],&a[i][1]);
}
a[n][0]=a[0][0];
a[n][1]=a[0][1];
printf("\n\n\tEnter the seed pt. : ");
scanf("%d%d",&x,&y);
cleardevice();
setcolor(WHITE);
for(i=0;i<n;i++) /*- draw poly -*/
{
    line(a[i][0],a[i][1],a[i+1][0],a[i+1][1]);
}
fill_right(x,y);
fill_left(x-1,y);
getch();
}
```

Output :

Enter the no. of edges of polygon : 4

Enter the co-ordinates of polygon :

XY0 : 100 100

X1 Y1 : 300 100

X2 Y2 : 300 300

X3 Y3 : 100 300

Enter the seed pt. : 200 200



-X-X-X-

Experiment 03 (B)

Name : Flood Filling Algorithm

Theory :

Flood fill algorithm is used to fill an area that is not defined with a single boundary color. Flood fill algorithm replaces the specified interior color with the given fill color.

It accepts:

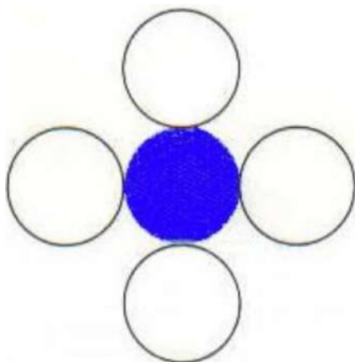
- i. The coordinates of interior pixel (x,y)
- ii. Interior color value.
- iii. Desired fill color value

Starting from (x,y), the algorithm starts from neighboring pixels to determine if they are of interior color. If Yes, then the pixel is painted with the desired color and their neighbors are tested in recursion.

Neighboring pixels are tested either by using 4-connected or 8-connected methods.

4 connected :

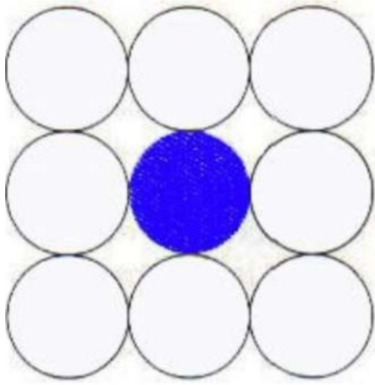
In this technique 4-connected pixels, we are putting the pixels above, below, to the right, and to the left side of the current pixels and this process will continue until we find a boundary with different color.



8 connected :

In this technique 8-connected pixels, we are putting pixels above, below, right and left side of the current pixels as we were doing in 4-connected technique.

In addition to this, we are also putting pixels in diagonals so that entire area of the current pixel is covered. This process will continue until we find a boundary with different color.

**Algorithm for 4-connected method:**

```
void flood_fill(int x,int y,int new_col,int old_col)
{
    if(getpixel(x,y)==old_col)
    {
        putpixel(x,y,new_col);
        flood_fill(x+1,y,new_col,old_col);
        flood_fill(x-1,y,new_col,old_col);
        flood_fill(x,y+1,new_col,old_col);
        flood_fill(x,y-1,new_col,old_col);
    }
}
```

Algorithm for 8-connected method:

```
void flood_fill(int x,int y,int new_col,int old_col)
{
    if(getpixel(x,y)==old_col)
    {
        putpixel(x,y,new_col);
        flood_fill(x+1,y,new_col,old_col);
        flood_fill(x-1,y,new_col,old_col);
        flood_fill(x,y+1,new_col,old_col);
```

```
        flood_fill(x,y-1,new_col,old_col);
        flood_fill(x+1,y+1,new_col,old_col);
        flood_fill(x+1,y-1,new_col,old_col);
        flood_fill(x-1,y+1,new_col,old_col);
        flood_fill(x-1,y-1,new_col,old_col);
    }
}
```

Program :

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

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

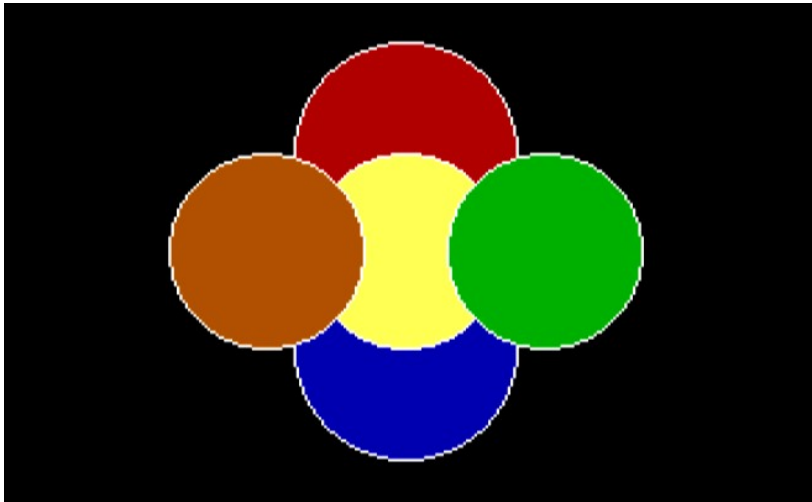
```
void flood_fill(int x,int y,int new_col,int old_col)
```

```
{
    if(getpixel(x,y)==old_col)
    {
        putpixel(x,y,new_col);
        flood_fill(x+1,y,new_col,old_col);
        flood_fill(x-1,y,new_col,old_col);
        flood_fill(x,y+1,new_col,old_col);
        flood_fill(x,y-1,new_col,old_col);
    }
}
```

```
void main()
```

```
{
```

```
int gdriver,gmode;
clrscr();
detectgraph(&gdriver,&gmode);
initgraph(&gdriver,&gmode,"c:\\turbo3\\bgi");
circle(200,150,35);
circle(100,150,35);
arc(150,150,45,135,35);
arc(150,150,225,315,35);
arc(150,115,0,180,40);
arc(150,185,180,360,40);
flood_fill(200,150,GREEN,getpixel(200,150));
flood_fill(150,150,YELLOW,getpixel(150,150));
flood_fill(80,150,BROWN,getpixel(80,150));
flood_fill(150,100,RED,getpixel(150,100));
flood_fill(150,200,BLUE,getpixel(150,200));
getch();
closegraph();
}
```

Output :**Advantages:**

- i. Flood fill colors an entire area in an enclosed figure through interconnected pixels using a single-color.
- ii. It is an easy way to fill color in the graphics. One just takes the shape and starts flood fill. The algorithm works in a manner so as to give all the pixels inside the boundary the same colour leaving the boundary and the pixels outside.
- iii. Flood Fill is also sometimes referred to as Seed Fill as you plant a seed and more and more seeds are planted by the algorithm. Each seed takes the responsibility of giving the same colour to the pixel at which it is positioned. There are many variations of Flood Fill algorithm that are used depending upon requirements.

Disadvantages:

- i. Flood fill algorithm is not advisable for filling larger polygons as quite larger stack is required for them.
- ii. Also it is slow since it requires a recursive function call to be given to the `getpixel()` command time and time again.
- iii. Initial pixel required more knowledge about surrounding pixels.

-X-X-X-

Experiment 04

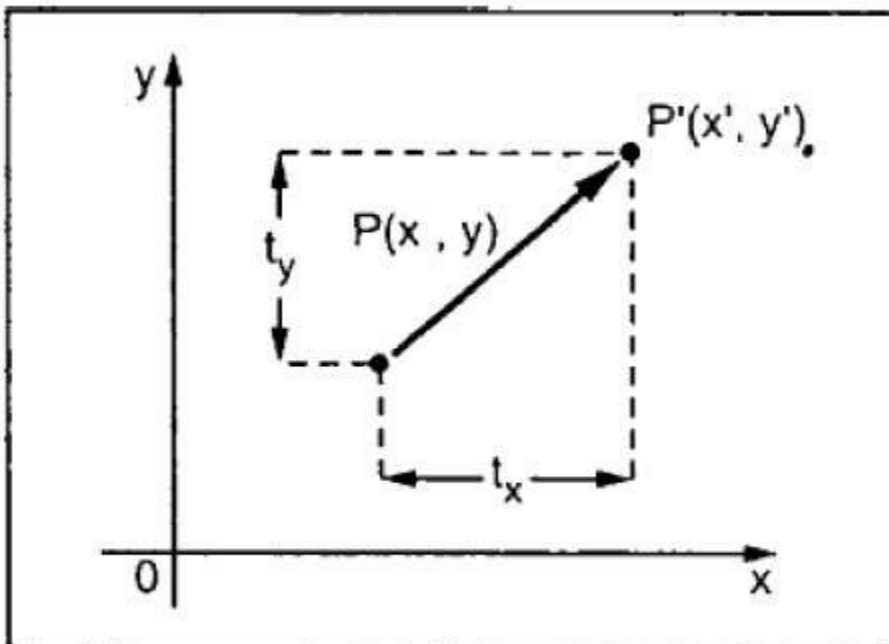
Name : 2D geometric transformation on an object like translation, rotation.

Theory :

Transformation means changing some graphics into something else by applying rules. We can have various types of transformations such as translation, scaling up or down, rotation, shearing, etc. When a transformation takes place on a 2D plane, it is called 2D transformation.

Translation

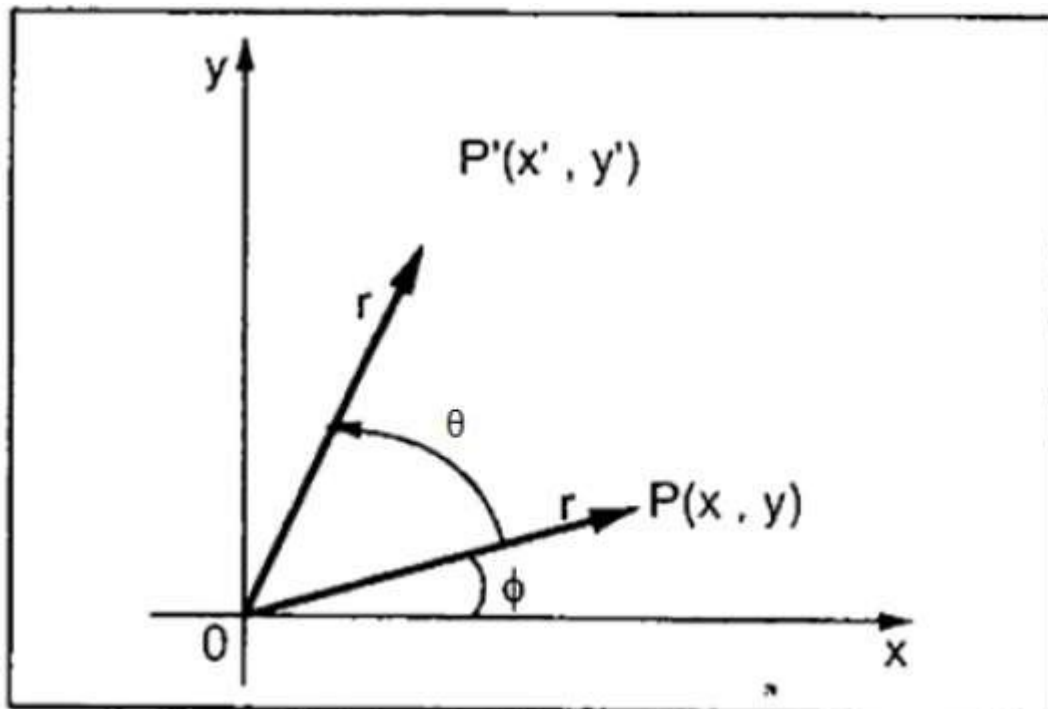
A translation moves an object to a different position on the screen. You can translate a point in 2D by adding translation coordinate (t_x, t_y) to the original coordinate X, Y to get the new coordinate X', Y' .



Rotation

In rotation, we rotate the object at particular angle θ from its origin. From the following figure, we can see that the point $P(x, y)$ is located at angle ϕ from the horizontal X coordinate with distance r from the origin.

Let us suppose you want to rotate it at the angle θ . After rotating it to a new location, you will get a new point $P'(x', y')$.



Program :

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

```
void main()
{
int gd,gm,m,n,i,a[10][2],b[10][2],tx,ty,sx,sy;
float t;
detectgraph(&gd,&gm);
initgraph(&gd,&gm,"C:\\TURBOC3\\BGI");
//Accepting vertices and its coordinates
printf("Accept the no of vertices of polygon - ");
```

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

for(i=0;i<n;i++)
{
printf("Accept end pt coordinates of vertices");
scanf("%d %d", &a[i][0],&a[i][1]);
}
// to close polygon
a[n][0]=a[0][0];
a[n][1]=a[0][1];

line(320,0,320,480);
line(0,240,640,240);


for(i=0;i<n;i++)
{
line(320+a[i][0], 240+a[i][1], 320+a[i+1][0], 240+a[i+1][1]);
}


printf("Enter your Choice\n1.Translation\n2.Rotation");
scanf("%d", &m);
switch(m)
{
case 1:
printf("Enter translation parameters:");
```

```
scanf("%d %d",&tx,&ty);

for(i=0;i<n;i++)
{
b[i][0]=a[i][0]+tx; //x'=x+tx
b[i][1]=a[i][1]+ty; //y'=y+ty
}
//to close new poly
b[n][0]=b[0][0];
b[n][1]=b[0][1];

//display new poly
for(i=0;i<n;i++)
{
line(320+b[i][0],240+b[i][1],320+b[i+1][0],240+b[i+1][1]);
}break;

//2D Rotation
case 2:
printf("Enter angle of rotation");
scanf("%f",&t);
t=(t*3.142)/180;

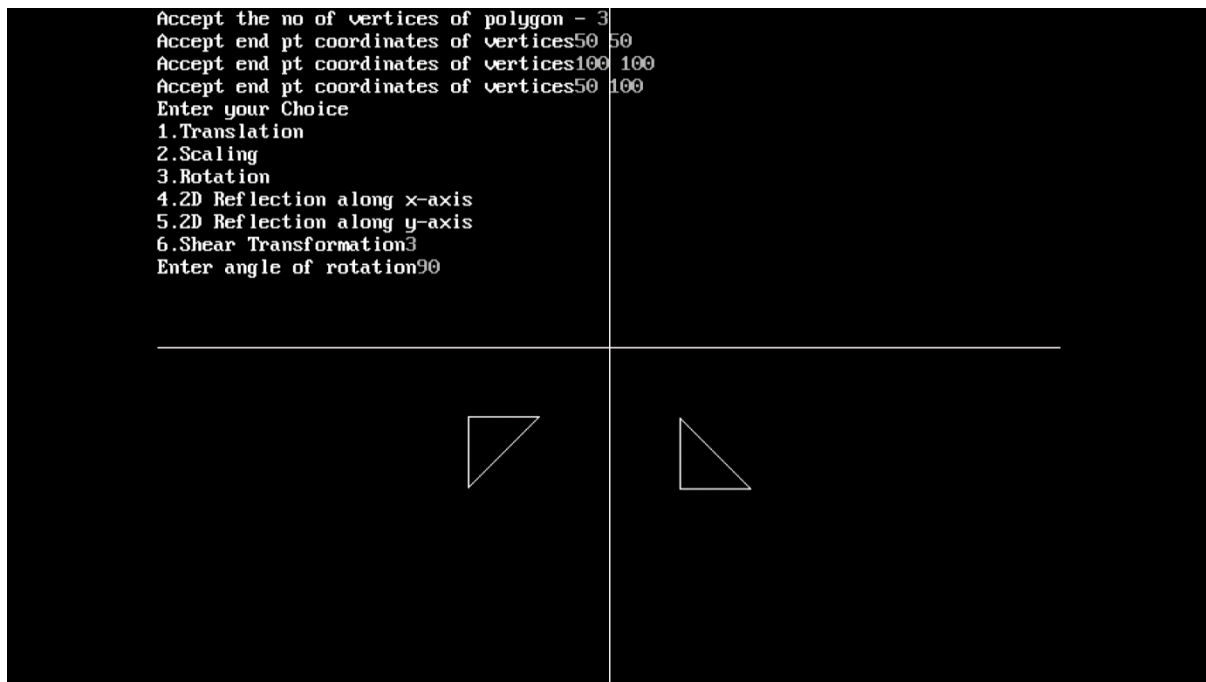
for(i=0;i<n;i++)
{
```

```
b[i][0]=(a[i][0]*cos(t)-a[i][1]*sin(t)); //x'=x.cos(t)-y.sin(t)
b[i][1]=(a[i][0]*sin(t)+a[i][1]*cos(t)); //y'=x.sin(t)+y.cos(t)
}
b[n][0]=b[0][0];
b[n][1]=b[0][1];
//Display new poly
for(i=0;i<n;i++)
{
line(320+b[i][0],240+b[i][1],320+b[i+1][0],240+b[i+1][1]);
}break;

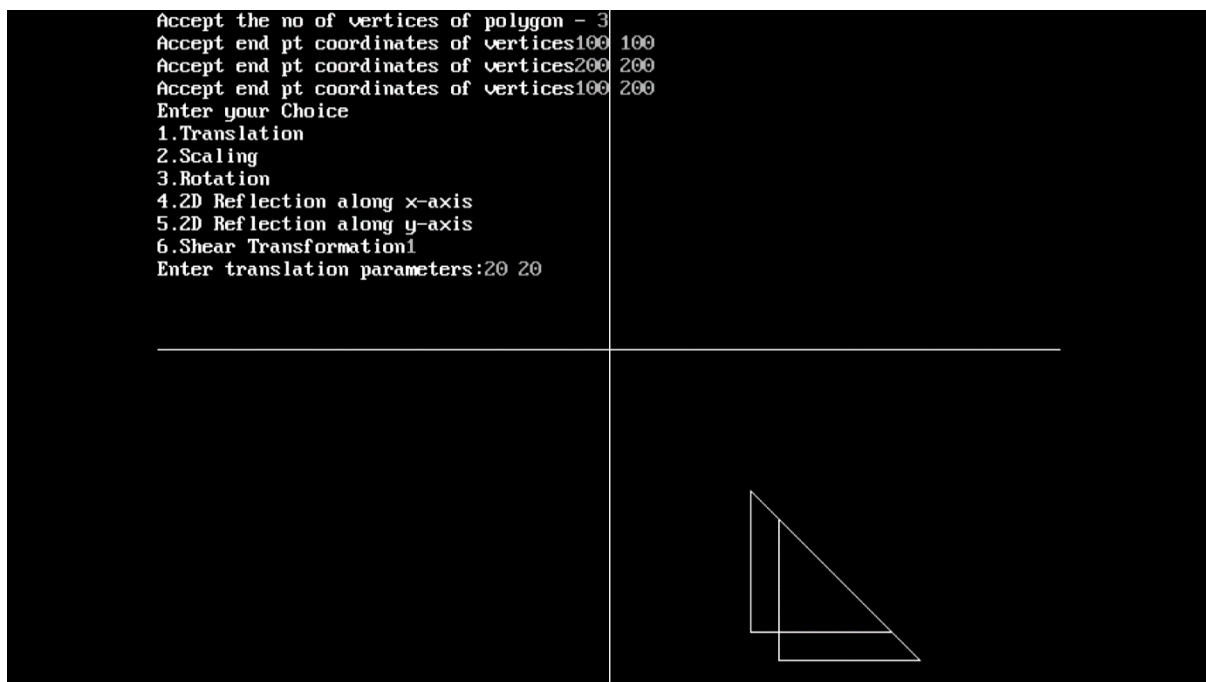
default:
printf("INVALID CHOICE");
}
getch();
}
```

Output :

Rotation Transformation



Translation Transformation



Experiment 06 (A)

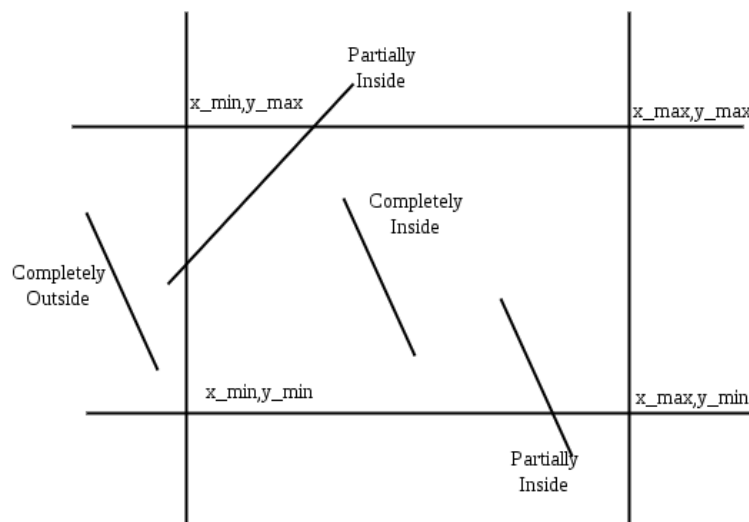
Name : Cohen Sutherland's Line Clipping Algorithm

Theory :

Cohen-Sutherland algorithm divides a two-dimensional space into 9 regions and then efficiently determines the lines and portions of lines that are inside the given rectangular area.

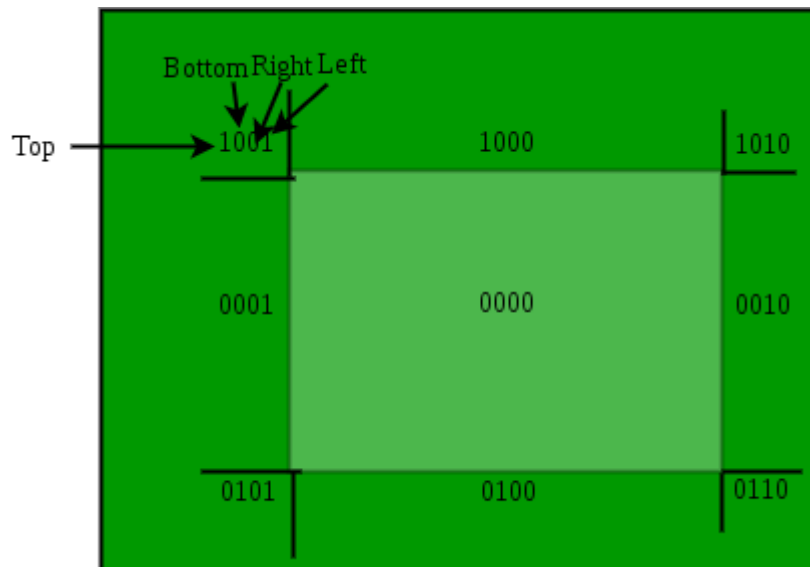
There are three possible cases for any given line.

1. **Completely inside the given rectangle:** Bitwise OR of region of two end points of line is 0 (Both points are inside the rectangle)
2. **Completely outside the given rectangle:** Both endpoints share at least one outside region which implies that the line does not cross the visible region. (bitwise AND of endpoints! = 0).
3. **Partially inside the window:** Both endpoints are in different regions. In this case, the algorithm finds one of the two points that is outside the rectangular region. The intersection of the line from outside point and rectangular window becomes new corner point and the algorithm repeats



Pseudo Code :

1. Accept end point co-ordinates of line AB i.e. $A(x_1, y_1)$ and $B(x_2, y_2)$ and window co-ordinate (X_{wmin}, Y_{wmin}) (X_{wmax}, Y_{wmax})
2. Assign 4 bit Region Code to both end pts of line AB



If $X < X_{wmin}$ then $B_1 = 1$ else 0

If $X > X_{wmax}$ then $B_2 = 1$ else 0

If $Y < Y_{wmin}$ then $B_3 = 1$ else 0

If $Y > Y_{wmax}$ then $B_4 = 1$ else 0

3. Check Status Line AB

a. Completely IN

If the region code for both end points are 0000 then line AB is completely IN.

Display line AB

Stop.

b. Completely OUT

If the logical AND operation between 2 end point codes is NOT 0000 then line AB is completely OUT

Discard line AB

Stop

c. Clipping Candidate

If case a and b fails, then line AB is clipping candidate

Go to step 4

4. Determine Intersection boundary. Consider region code of outside point

If $B_1 = 1$ line intersect with left boundary

$B_2 = 1$ line intersect with right boundary

$B_3 = 1$ line intersect with bottom boundary

$B_4 = 1$ line intersect with top boundary

5. Determine the intersection point

a. Left/Right Boundary

$$X' = X_{wmin}(L)$$

OR

$$= X_{wmax}(R)$$

$$(X'-X_1)/(X_2-X_1) = (Y'-Y_1)/(Y_2-Y_1)$$

$$Y' = Y_1 + m(X'-X_1)$$

$$I(X',Y')$$

b. Bottom/Up

$$Y' = Y_{\min}(B)$$

OR

$$= Y_{\max}(T)$$

$$X' = X_1 + (Y'-Y_1)/m$$

$$I'(X',Y')$$

6. To determine region code for I' go to step 2.

Program :

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

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

```
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)//for setting the 4 bit code
{
    PT ptemp;

    if(p.y<100)
        ptemp.code[0]='1';//Top area of window
    else
        ptemp.code[0]='0';
```

```

    if(p.y>350)
        ptemp.code[1]='1';//Bottom area of window
    else
        ptemp.code[1]='0';

    if(p.x>450)
        ptemp.code[2]='1';//Right area of window
    else
        ptemp.code[2]='0';

    if(p.x<150)
        ptemp.code[3]='1';//Left area of window
    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);
}

void main()
{
    int gd=DETECT,v,gm;
    PT p1,p2,p3,p4,ptemp;

```

```
printf("\nEnter x1 and y1\n");
scanf("%d %d",&p1.x,&p1.y);
printf("\nEnter x2 and y2\n");
scanf("%d %d",&p2.x,&p2.y);

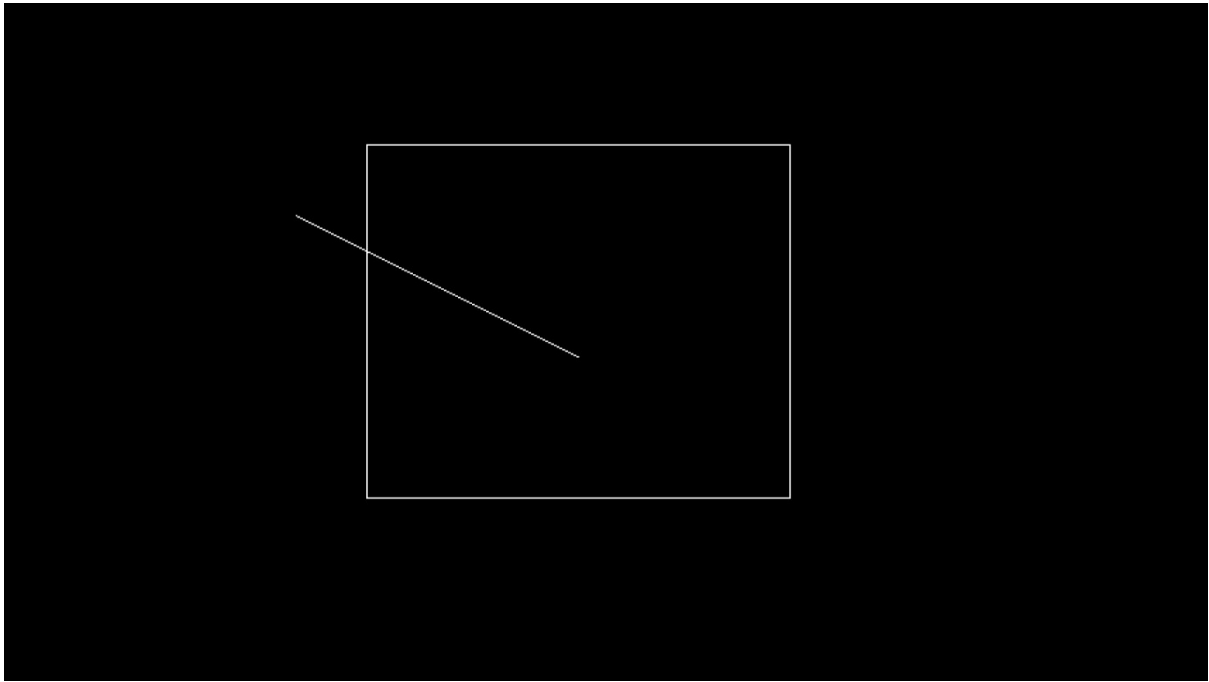
initgraph(&gd,&gm,"c:\\turbo3\\bgi");
drawwindow();
delay(500);

drawline(p1,p2);
delay(5000);
cleardevice();

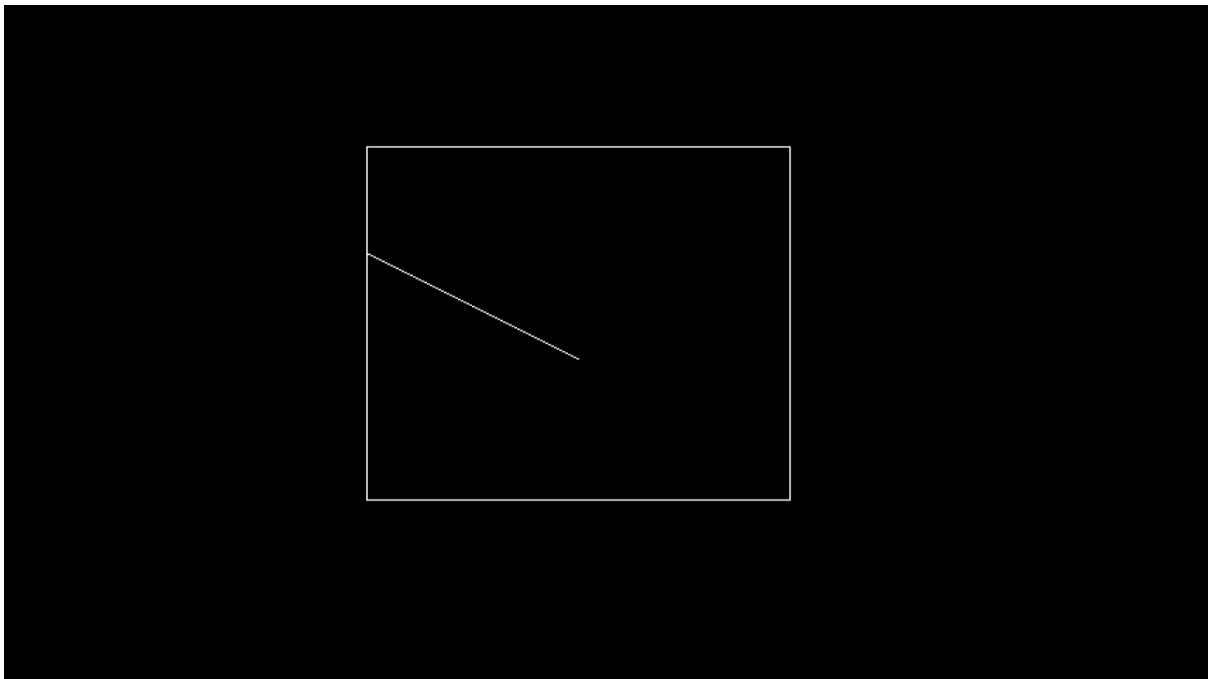
delay(500);
p1=setcode(p1);
p2=setcode(p2);
v=visibility(p1,p2);
delay(500);
switch(v)
{
case 0: drawwindow();
        delay(500);
        drawline(p1,p2);
        break;
case 1: drawwindow();
        delay(500);
        break;
case 2: p3=resetendpt(p1,p2);
        p4=resetendpt(p2,p1);
        drawwindow();
        delay(500);
        drawline(p3,p4);
        break;
}
delay(5000);
closegraph();
}
```

Output :

Before Clipping



After Clipping:



ADVANTAGES :

- It calculates end-points very quickly and rejects and accepts lines quickly.
- It can clip pictures much large than screen size.

DISADVANTAGES :

- Clipping window region can be rectangular in shape only and no other polygonal shaped window is allowed.
- Edges of rectangular shaped clipping window has to be parallel to the x-axis and y-axis.
- Complex mathematical calculations are involved and it is time consuming process.

APPLICATIONS :

- When a window is "placed" on the world, only certain objects and parts of objects can be seen. Points and lines which are outside the window are "cut off" from view. This process of "cutting off" parts of the image of the world is called clipping.
- In clipping, we examine each line to determine whether or not it is completely inside the window, completely outside the window, or crosses a window boundary. If inside the window, the line is displayed. If outside the window, the lines and points are not displayed. If a line crosses the boundary, we must determine the point of intersection and display only the part which lies inside the window.

-X-X-X-

Experiment 06 (B)

Name : Liang Barsky Line Clipping Algorithm

Theory :

Algorithm :

Step 1:

Accept Window Extents (Xw_{min} , Yw_{min}) (Xw_{max} , Yw_{max})

Accept End point coordinates of a line segment (x_1 , y_1) (x_2 , y_2)

Step 2:

Calculate p_k , q_k and r_k where $r_k = \frac{p_k}{q_k}$

k	p_k	q_k	$r_k = \frac{p_k}{q_k}$
1	$-\Delta x$	$x_1 - Xw_{min}$	
2	Δx	$Xw_{max} - x_1$	
3	$-\Delta y$	$y_1 - Yw_{min}$	
4	Δy	$Yw_{max} - y_1$	

Step 3:

During the calculation for any k if $p_k = 0$ and $q_k < 0$ implies line is parallel to one of the edge and is outside, therefore reject the line and **STOP**

Step 4:

Calculate the two point of intersections say u_1 and u_2 as given below

$$u_1 = \max\{0, r_k\} \text{ for all } p_k < 0$$

$$u_2 = \min(0, r_k) \text{ for all } p_k > 0$$

Step 5:

If $u_1 > u_2$ implies line is totally outside, therefore reject and **STOP**

Step 6:

1. Else Calculate the point of intersection $I'(x', y')$ and $I''(x'', y'')$ by using u_1 and u_2 respectively as

$$\begin{aligned} x &= x_1 + u_1(\Delta x) & y &= y_1 + u_1(\Delta y) \\ x &= x_2 + u_2(\Delta x) & y &= x_2 + u_2(\Delta y) \end{aligned}$$

Step 7:

Display the line between I' and I''

Step 8:

return.

Program :

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

void main()
{
    int i,gd,gm;
    int x1,y1,x2,y2,xmin,xmax,ymin,ymax,xx1,xx2,yy1,yy2,dx,dy;
    float t1,t2,p[4],q[4],temp;
    detectgraph(&gd,&gm);
    initgraph(&gd,&gm,"c:\\turboc3\\bgi");
    printf("Enter Starting point coordinates x1,y1 of line AB ");
    scanf("%d %d",&x1,&y1);
    printf("Enter ending point coordinates x2,y2 of line AB ");
    scanf("%d %d", &x2,&y2);
    setcolor(GREEN);
    line(x1,y1,x2,y2);

    setcolor(RED);
    printf("Accept window boundary xwmin,ywmin,xwmax,ywmax");
    scanf("%d %d %d %d",&xmin,&ymin,&xmax,&ymax);
    /*x1=120;
    y1=120;
    x2=300;
    y2=300;
```

```
xmin=100;
ymin=100;
xmax=250;
ymax=250; */
rectangle(xmin,ymin,xmax,ymax);

setcolor(WHITE);

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

p[0]=-dx;  q[0]=x1-xmin;
p[1]=dx;   q[1]=xmax-x1;
p[2]=-dy;  q[2]=y1-ymin;
p[3]=dy;   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;

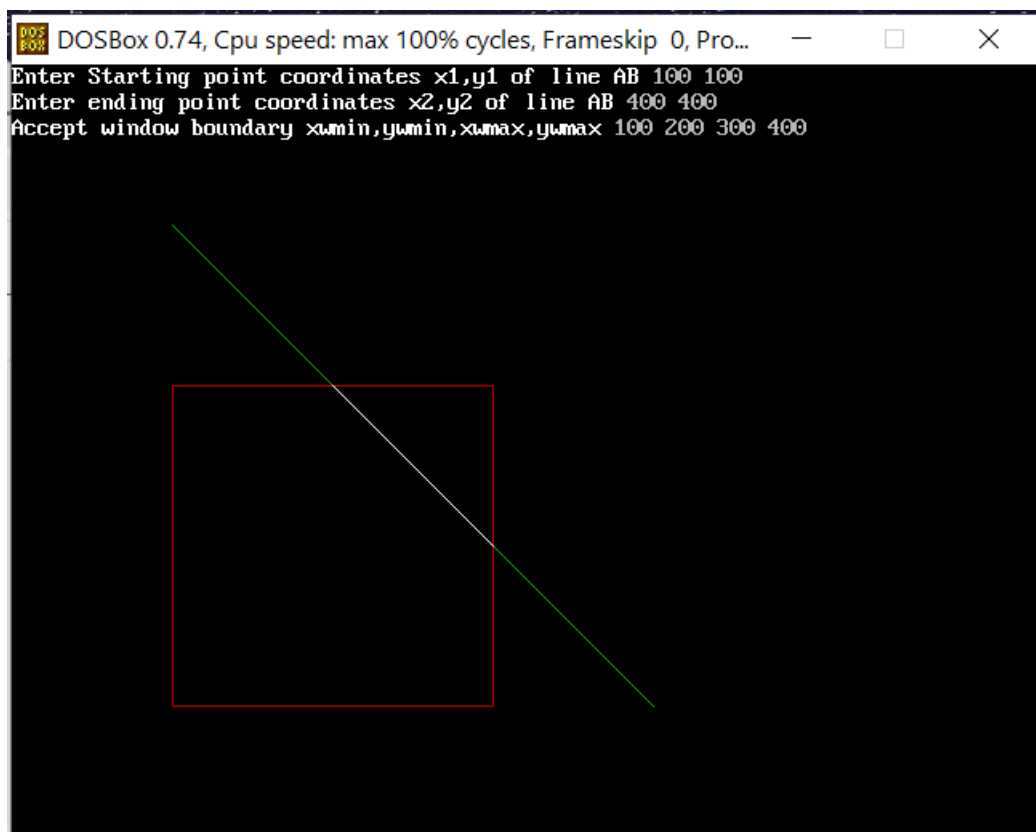
    for(i=0;i<4;i++)
    {
        temp=q[i]/p[i];

        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(10);  
    getch();  
    // closegraph();  
}
```

Output :



ADVANTAGES :

-
- Since intersection calculations are reduced.
 - Each update of parameters require only one division & window intersection of lines are calculated once, when final values have been computed.

DISADVANTAGES :

- Liang-Barsky algorithm involves the parametric equation of line.
- We have to derive equations to check whether the line is inside or outside the clipping window.

APPLICATIONS :

- When a window is "placed" on the world, only certain objects and parts of objects can be seen. Points and lines which are outside the window are "cut off" from view. This process of "cutting off" parts of the image of the world is called clipping.
- In clipping, we examine each line to determine whether or not it is completely inside the window, completely outside the window, or crosses a window boundary. If inside the window, the line is displayed. If outside the window, the lines and points are not displayed. If a line crosses the boundary, we must determine the point of intersection and display only the part which lies inside the window.

-X-X-X-

Experiment 08

Name : Study and implement perspective projection of a cube

Theory :

In perspective projection farther away object from the viewer, small it appears. This property of projection gives an idea about depth. The artist use perspective projection from drawing three-dimensional scenes.

Two main characteristics of perspective are vanishing points and perspective foreshortening. Due to foreshortening object and lengths appear smaller from the center of projection. More we increase the distance from the center of projection, smaller will be the object appear.

Vanishing Point :

It is the point where all lines will appear to meet. There can be one point, two point, and three point perspectives.

One Point: There is only one vanishing point as shown in fig (a)

Two Points: There are two vanishing points. One is the x-direction and other in the y -direction as shown in fig (b)

Three Points: There are three vanishing points. One is x second in y and third in two directions.

In Perspective projection lines of projection do not remain parallel. The lines converge at a single point called a center of projection. The projected image on the screen is obtained by points of intersection of converging lines with the plane of the screen. The image on the screen is seen as of viewer's eye were located at the centre of projection, lines of projection would correspond to path travel by light beam originating from object.

Important terms related to perspective

1. **View plane:** It is an area of world coordinate system which is projected into viewing plane.
2. **Center of Projection:** It is the location of the eye on which projected light rays converge.
3. **Projectors:** It is also called a projection vector. These are rays start from the object scene and are used to create an image of the object on viewing or view plane.

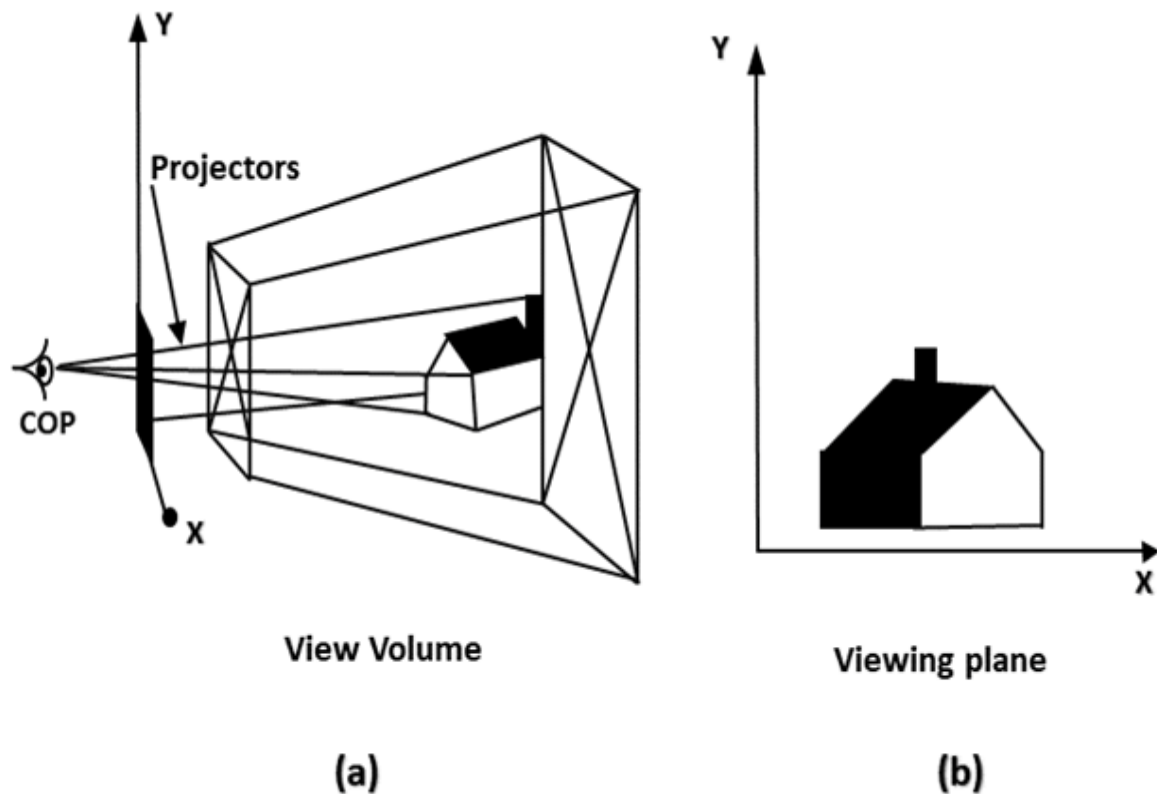
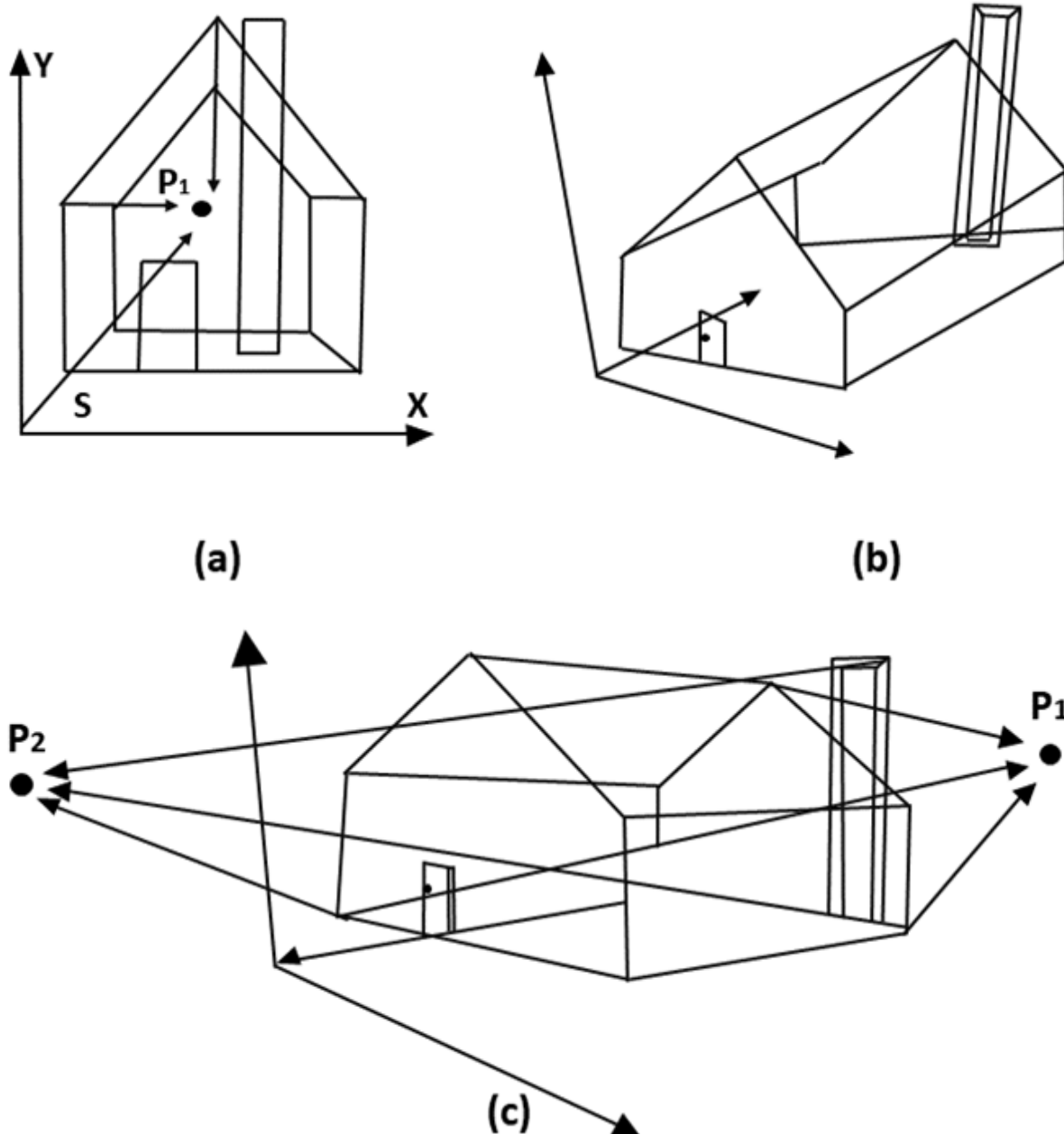


Fig: Perspective Projection

Anomalies in Perspective Projection :

It introduces several anomalies due to these object shape and appearance gets affected.

1. **Perspective foreshortening:** The size of the object will be small of its distance from the center of projection increases.
2. **Vanishing Point:** All lines appear to meet at some point in the view plane.
3. **Distortion of Lines:** A range lies in front of the viewer to back of viewer is appearing to six rollers.



Foreshortening of the z -axis in fig (a) produces one vanishing point, P_1 . Foreshortening the x and z -axis results in two vanishing points in fig (b). Adding a y -axis foreshortening in fig (c) adds vanishing point along the negative y -axis.

Program :

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

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

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

```
main()
```

```
{
```

```
    int x1,y1,x2,y2,gd,gm;
```

```
    int ymax,a[4][8];
```

```
    float par[4][4],b[4][8];
```

```
    int i,j,k,m,n,p;
```

```
    int xp, yp, zp, x, y, z;
```

```
    a[0][0] = 100; a[1][0] = 100; a[2][0] = -100;
```

```
    a[0][1] = 200; a[1][1] = 100; a[2][1] = -100;
```

```
    a[0][2] = 200; a[1][2] = 200; a[2][2] = -100;
```

```
    a[0][3] = 100; a[1][3] = 200; a[2][3] = -100;
```

```
    a[0][4] = 100; a[1][4] = 100; a[2][4] = -200;
```

```
    a[0][5] = 200; a[1][5] = 100; a[2][5] = -200;
```

```
    a[0][6] = 200; a[1][6] = 200; a[2][6] = -200;
```

```
    a[0][7] = 100; a[1][7] = 200; a[2][7] = -200;
```

```
    detectgraph(&gd,&gm);
```

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

```
ymax = getmaxy();
xp = 300; yp = 320; zp = 100;

for(j=0; j<8; j++)
{
    x = a[0][j]; y = a[1][j]; z = a[2][j];
    b[0][j] = xp - ( (float)( x - xp )/(z - zp)) * (zp);
    b[1][j] = yp - ( (float)( y - yp )/(z - zp)) * (zp);
}
```

```
/*- front plane display -*/
for(j=0;j<3;j++)
{
    x1=(int) b[0][j]; y1=(int) b[1][j];
    x2=(int) b[0][j+1]; y2=(int) b[1][j+1];
    line( x1,ymax-y1,x2,ymax-y2);
}
```

```
x1=(int) b[0][3]; y1=(int) b[1][3];
x2=(int) b[0][0]; y2=(int) b[1][0];
line( x1, ymax-y1, x2, ymax-y2);
/*- back plane display -*/
setcolor(11);
for(j=4;j<7;j++)
{
    x1=(int) b[0][j]; y1=(int) b[1][j];
    x2=(int) b[0][j+1]; y2=(int) b[1][j+1];
```

```
        line( x1, ymax-y1, x2, ymax-y2);
    }
    x1=(int) b[0][7]; y1=(int) b[1][7];
    x2=(int) b[0][4]; y2=(int) b[1][4];
    line( x1, ymax-y1, x2, ymax-y2);
    setcolor(7);
    for(i=0;i<4;i++)
    {
        x1=(int) b[0][i]; y1=(int) b[1][i];
        x2=(int) b[0][4+i]; y2=(int) b[1][4+i];
        line( x1, ymax-y1, x2, ymax-y2);
    }
    getch();
}
```

Output :



-X-X-X-

Experiment 09

Name : Implement Bezier curve for given set of control points

Theory :

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 –

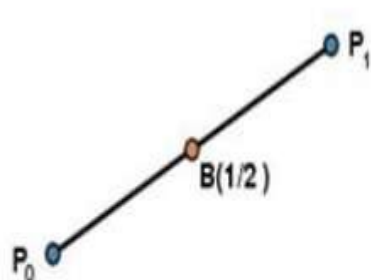
$$\sum_{k=0}^n P_k B_k(t)$$

Where p_i is the set of points and $B_i(t)$ represents the Bernstein polynomials which are given by –

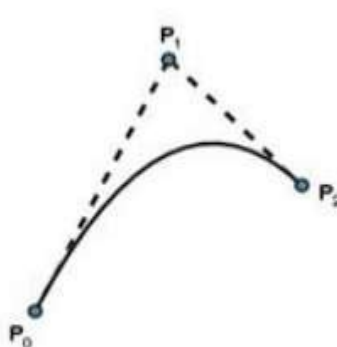
$$B_i(t) = \binom{n}{i} (1-t)^{n-i} t^i$$

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

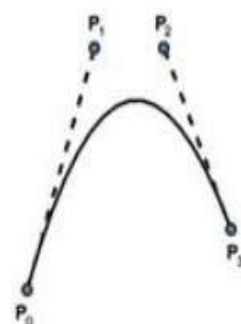
The simplest Bézier curve is the straight line from the point P_0 to P_1 . A quadratic Bezier curve is determined by three control points. A cubic Bezier curve is determined by four control points.



Simple Bezier Curve



Quadratic Bazier Curve



Cubic Bazier Curve

Properties of Bezier Curves :

Bezier curves have the following properties –

- They generally follow the shape of the control polygon, which consists of the segments joining the control points.
- They always pass through the first and last control points.
- They are contained in the convex hull of their defining control points.
- The degree of the polynomial defining the curve segment is one less than the number of defining polygon point. Therefore, for 4 control points, the degree of the polynomial is 3, i.e. cubic polynomial.

-
- A Bezier curve generally follows the shape of the defining polygon.
 - The direction of the tangent vector at the end points is same as that of the vector determined by first and last segments.
 - The convex hull property for a Bezier curve ensures that the polynomial smoothly follows the control points.
 - No straight line intersects a Bezier curve more times than it intersects its control polygon.
 - They are invariant under an affine transformation.
 - Bezier curves exhibit global control means moving a control point alters the shape of the whole curve.
 - A given Bezier curve can be subdivided at a point $t=t_0$ into two Bezier segments which join together at the point corresponding to the parameter value $t=t_0$.

Program :

```
#include<graphics.h>

#include<math.h>

#include<conio.h>

#include<stdio.h>

void main()

{

int x[4],y[4],i;

double put_x,put_y,t;

int gr=DETECT,gm;

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


printf("\n***** Bezier Curve *****");

for(i=0;i<4;i++)

{

printf("\n Please enter x and y coordinates ");
```


```
scanf("%d%d",&x[i],&y[i]);
putpixel(x[i],y[i],3);
}

for(t=0.0;t<=1.0;t=t+0.001)
{
    put_x=pow(1-t,3)*x[0]+3*t*pow(1-t,2)*x[1]+3*t*t*(1-
t)*x[2]+pow(t,3)*x[3];
    put_y=pow(1-t,3)*y[0]+3*t*pow(1-t,2)*y[1]+3*t*t*(1-
t)*y[2]+pow(t,3)*y[3];
    putpixel(put_x,put_y,WHITE);
}

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

Output :

```
***** Bezier Curve *****  
Please enter x and y coordinates 300 300  
Please enter x and y coordinates 400 300  
Please enter x and y coordinates 350 350  
Please enter x and y coordinates 300 400
```



-X-X-X-