Name: Pragisha Prajapati

CSIT 3rd Sem, 'A'
Roll no.: 12

INDEX

SN	Title	Date	Signature
1.	DDA line drawing algorithm to generate a line.	26th July, 2022.	8/2
2.	Bresenham's line drawing algorithm to generate a line	2nd August, 2022	8/5
3.	Mid-point circle drawing algorithm to draw circle.	5th August, 2022	8/9
4.	Mid-point ellipse drawing algorithm to draw an ellipse	8th August, 2022.	8/26
5.	Implementation of 2D transformations.	12th August, 2022.	8/26
6.	Bezier Curve implementation using C program:	26th August, 2022	9/2
7.	Implementation of 3D transformation	2nd September, 2022	9/6
8	Implementation of Cohen-Sutherland's line clipping algorithm.	6th September, 2022	11/11
9.	Implementation of Liang-Barsky algorithm.	18th September, 2022	11/11
10.	Installation of Code Blocks.	22nd Nov, 2022	12/20
11.	To draw a line using OpenGL	22nd Nov, 2022	12/20
12.	To draw triangle using openGL.	22nd Nov, 2022	12/20

Title: DDA line drawing algorithm to generate a line

Theory:

Digital Differential Analyzer (DDA) is scan conversion line drawing algorithm based on calculating either Δy or Δx using equation. We sample the line at unit intervals in one condition coordinate and find corresponding integer values nearest the line path for the other coordinate.

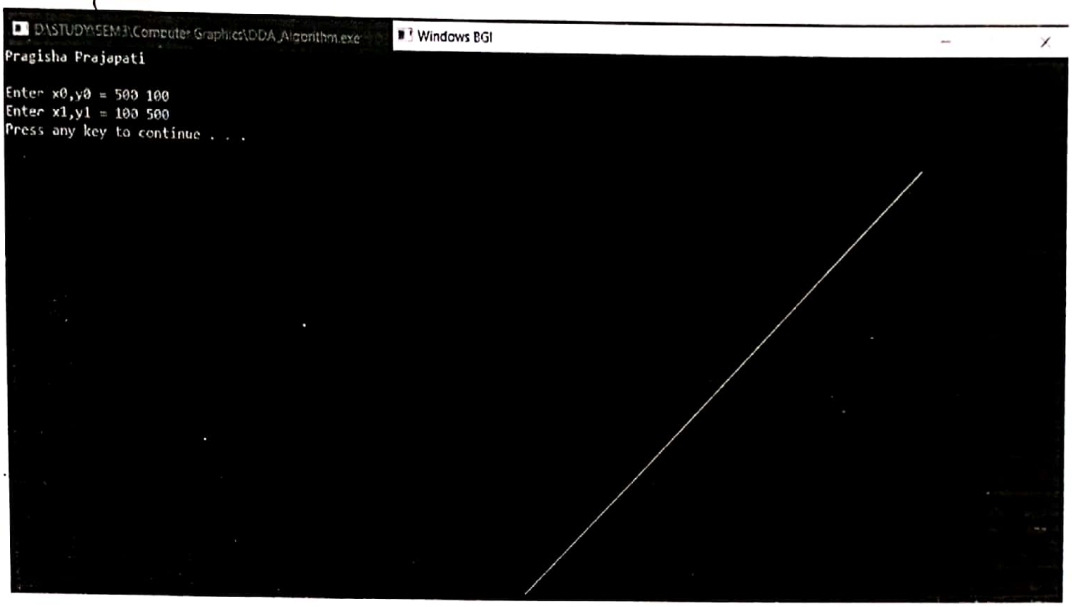
Algorithm:

```
DDA (x1, y1, x2, y2){
        dx = x2 - x1;
        dy = y2 - y1;
        if (abs (dx) > abs(dy)) {
              steps = abs(dx);
        }
        else {
              steps = abs (dy);
        }
        xinc = dx / steps;
        yinc = dy / steps;
        for (i=1; i<= steps ; i++){
              putpixel (round (x1), round (y1));
              x1 = x1 + xinc;
              y1 = y1 + yinc;
        }
}
```

Source code:

```
#include <graphics.h>
#include <conio.h>
#include <stdio.h>
int main(){
    int gd=DETECT,gm,i;
    float x,y,dx,dy,steps;
    int x0,x1,y0,y1;
    printf("Enter x0 and x1:");
    scanf("%d %d",&x0,&x1);
    printf("Enter y0 and y1:");
    scanf("%d %d",&y0,&y1);
    initgraph(&gd,&gm,(char*)"");
    dx=(float)(x1-x0);
    dy=(float)(y1-y0);
    if(dx>=dy){
        steps=dx;
    }
    else{
        steps=dy;
    }
    dy=dy/steps;
    dx=dx/steps;
    x=x0;
    y=y0;
    i=1;
    while(i<=steps){
        putpixel(x,y,RED);
        x=x+dx;
        y=y+dy;
        i=i+1;
    }
    getch();
    closegraph();
}
```

Output:



Conclusion:

Hence, we were able to generate a line using DDA algorithm in C program using graphics.h.

8/2

Title: Bresenham's algorithm to draw a line

Theory:

BLA is a more efficient method used to plot pixel position along a straight line. It involves only integer addition, subtractions and multiplication operations. These operations can be performed very rapidly so lines can be generated quickly.

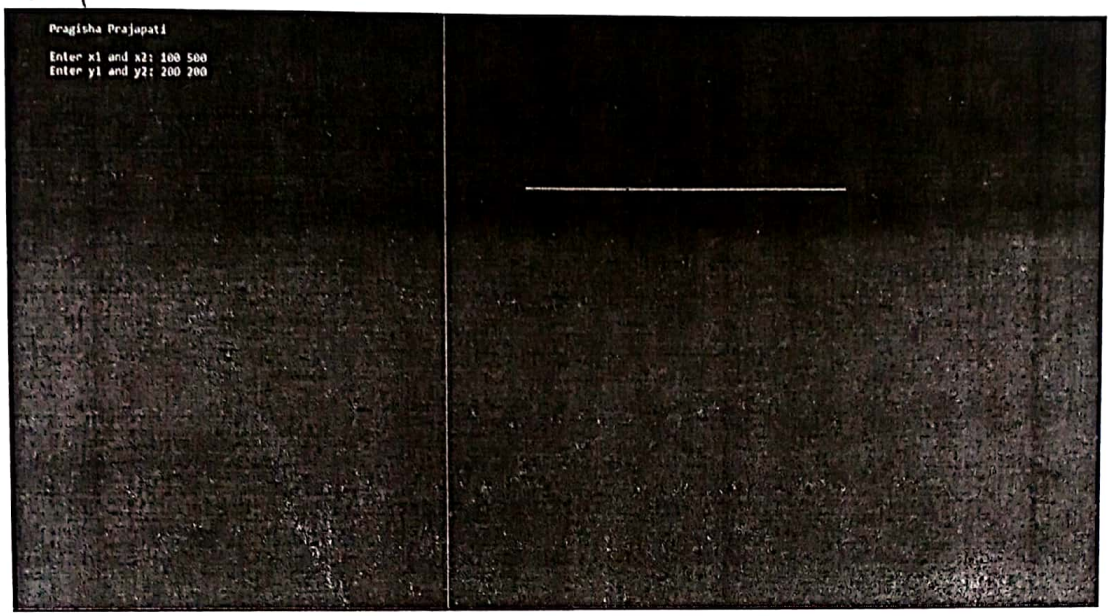
Algorithm:

```
BLA (x1, y1, x2, y2) {
    x = x1;
    y = y1;
    dx = x2 - x1;
    dy = y2 - y1;
    p = 2dy - dx
    while (x <= x2) {
        putpixel (x, y);
        x++;
        if (p < 0) {
            p = p + 2dy;
        }
        else {
            p = p + 2dy - 2dx
            y++;
        }
    }
}
```

Source code:

```
//Bresenham's line drawing algorithm
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
int main(){
        int gd=DETECT,gm,i;
        float x,y,dx,dy,steps,p1;
        int x1,y1,y2,x2;
        initgraph(&gd,&gm,(char*)" ");
        printf("\n\t Pragisha Prajapati \n\n");
        printf("\t Enter x1 and x2: ");
        scanf("%d %d",&x1,&x2);
        printf("\t Enter y1 and y2: ");
        scanf("%d %d",&y1,&y2);
        dx=(float)x2-x1;
        dy=(float)y2-y1;
        p1=2*dy-dx;
        x=x1;
        y=y1;
        while(x<=x2){
            putpixel(x,y,WHITE);
            x++;
            if(p1<0){
                p1=p1+2*dy;
            }
            else{
                p1=p1+(2*dy)-(2*dx);
                y++;
            }
        closegraph();
}
```

Output:



Conclusion:

Hence, we were able to generate a line implementing Bresenham's line drawing algorithm in C graphic.h library.

8/5

Title: Mid-point circle drawing algorithm to draw circle

Theory:

The mid-point circle drawing algorithm used to determine the points needed for rasterizing a circle. We use mid-point algorithm to calculate all the parameter points of the circle in first octant and then print them along with their mirror points in other octants.

Algorithm:

Step 1: Input radius r and circle center (x_c, y_c) and obtain the first point on the circumference of a circle as: $(x_0, y_0) = (0, r)$

Step 2: Calculate the initial value of decision paramter as: $p_0 = 1 - r$

Step 3: At each x_k position, starting at $k = 0$, perform:
- if $p_k < 0$, the next point along the circle centred on $(0,0)$ is (x_{k+1}, y_k) and $p_{k+1} = p_k + 2x_{k+1} + 1$
- if otherwise, the next point along the circle is (x_{k+1}, y_{k+1}) and $p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$
- where $2x_{k+1} = 2x_k + 2$ and $2y_{k+1} = 2y_k - 2$.

Step 4: Determine symmetry points in other seven octants.

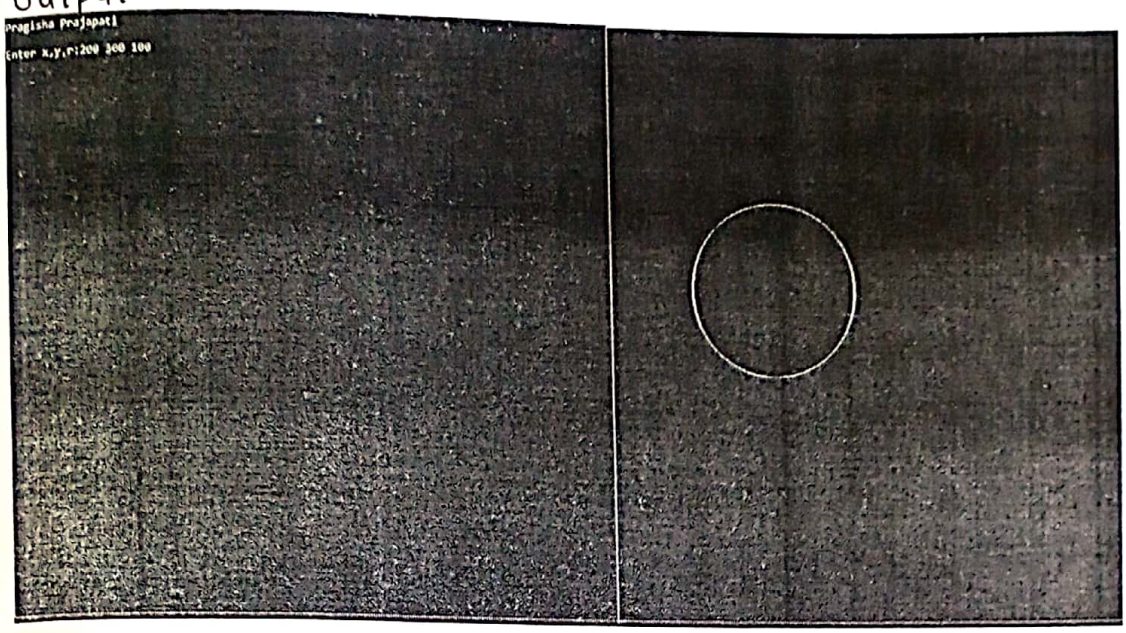
Step 5: Move each calculated pixel position (x, y) onto the circular path centered on (x_c, y_c) and plot:
$x = x + x_c$, $y = y + y_c$;

Step 6: Repeat steps 3 and 5 until $x \geq y$.

Source code :

```
// Mid-point circle drawing algorithm
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
int main(){
    int gd=DETECT,gm,i;
    int xo,yo,r,p;
    int x,y;
    initgraph(&gd,&gm,(char *)"");
    printf("Pragisha Prajapati \n\n");
    printf("Enter x,y,r: ");
    scanf("%d %d %d",&xo,&yo,&r);
    x=0;
    y=r;
    p=(1-r);
    while(y>x){
        putpixel(xo+x,yo+y,white);
        putpixel(xo+y,yo+x,WHITE);
        putpixel(xo-y,yo+x,WHITE);
        putpixel(xo-x,yo+y,WHITE);
        putpixel(xo-x,yo-y,WHITE);
        putpixel(xo-y,yo-x,WHITE);
        putpixel(xo+y,yo-x,WHITE);
        putpixel(xo+x,yo-y,WHITE);
        if(p<0){
            p=p+1+2*x;
        }
        else{
            y=y-1;
            p=p+1+(2*x)-(2*y);
        }x=x+1;
}
getch();
closegraph();
}
}
```

Output:



Conclusion:

Hence, we were able to generate a circle using midpoint-circle drawing algorithm in c program using graphic.h.

8/9

Title: Midpoint Ellipse drawing algorithm.

Theory:

In computer graphics, the midpoint ellipse algorithm is an incremented method of drawing an ellipse. It is used to calculate all the perimeter points of an ellipse. Here, mid-point of two pixel is calculated, decision parameter is calculated to determine region, boundary and pixels are generated on that basis.

Algorithm:

Step 1: Start

Step 2: Declare $rx, ry, x, y, x, dx, dy, p, p_2$

Step 3: Initialize initial point of region 1 as:

$x = 0, y = ry$

Step 4: Calculate $p = ry^2 + rx^2/4 - rx^2 ry$.

$dx = 2ry^2 x$

$dy = 2rx^2 y$.

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

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

plot (x, y)

if $(p < 0)$

update $x = x + 1$;

$p = p + ry^2 [2x + 3]$

else

update $x = x + 1$; $y = y - 1$;

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

Step 8: Calculate $p2 = ry^2 (x + 1/2)^2 * rx^2 (y-1)^2 - rx^2 ry^2$.

Step 9: Repeat till $(y > 0)$ if $(p2 > 0)$ update $y = y - 1$

$p2 = p2 - 2yrx^2 + rx^2$ else $x = x + 1; y = y - 1$

$p2 = p2 + 2ry^2 (2x) - 2yrx^2 + rx^2$

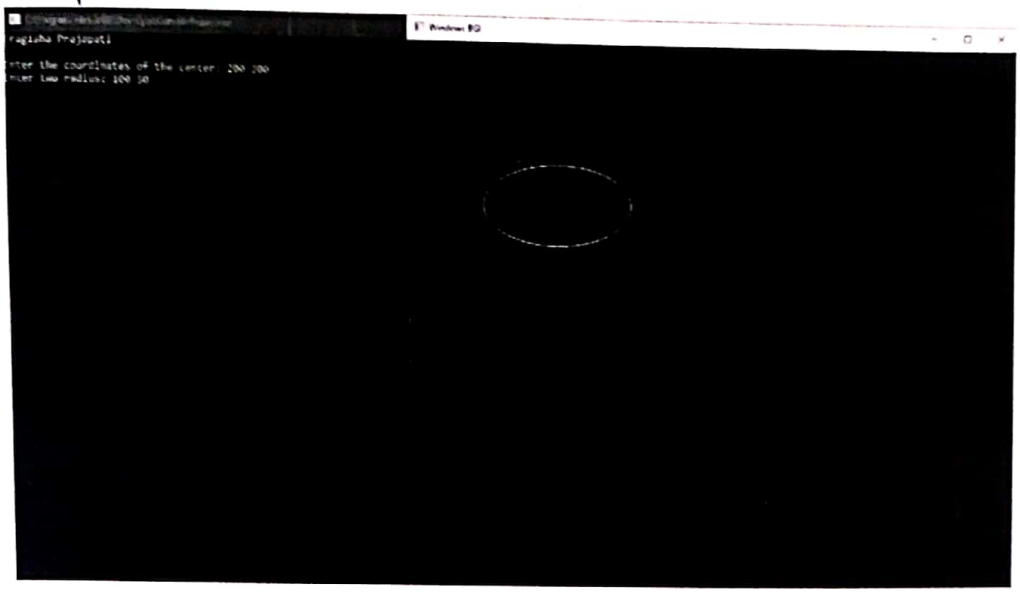
Step 10: Stop

Source code:

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
int main(){
    int gd = DETECT, gm;
    int rn, ry;
    float p;
    float nc, yc;
    initgraph(&gd, &gm, (char *)" ");
    printf("Pragisha Prajapati \n\n");
    printf("Enter the cordinates of the center: ");
    scanf("%f%f", &nc, &yc);
    printf("Enter two radius: ");
    scanf("%d%d", &rn, &ry);
    int n=0, y=ry;
    p = ry*ry - rn*rn*ry + rn*rn/4;
    while((2*ry*ry*n) <= (2*rn*rn*y)){
        if (p<0){
            n++;
            p=p+(2*ry*ry*n)+ry*ry;
        }else{
            n++;
            y--;
            p=p+2*ry*ry*n-2*rn*rn*y-
                ry*ry;
        }
        putpixel(nc+n, yc+y, WHITE);
        putpixel(nc+n, yc-y, WHITE);
        putpixel(nc-n, yc+y, WHITE);
        putpixel(nc-n, yc-y, WHITE);
    }
```

```
p = ry*ry*(x+0.5)*(x+0.5)+rx*rx*(y-1)*
    (y-1)-rx*rx*ry*ry;
while(y>0){
        if(p<=0){
            x++;
            y--;
            p=p+2*ry*ry*x-2*rx*rx*y+
              rx*rx;
        }
        else{
            y--;
            p=p-2*rx*rx*y+rx*rx;
        }
        putpixel(xc+x,yc+y,WHITE);
        putpixel(xc+x,yc-y,WHITE);
        putpixel(xc-x,yc+y,WHITE);
        putpixel(xc-x,yc-y,WHITE);
    }
    getch();
    closegraph();
}
```

Output:



Conclusion:

Hence, we were able to generate an ellipse using midpoint ellipse drawing algorithm in C program.

8/26

Title: Implementation of 2D transformations.

Theory:

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

Transformation plays an important role in cg to reposition the graphics on screen and change their size or orientation. Looking forward, we have implemented some 2D transformation and they are:

1) Reflection

A reflection is a mirror image of the shape. An image will reflect through a line known as the line of reflection:

Reflection over x-axis: $(x, y) = (x, -y)$

Reflection over y-axis: $(x, y) = (-x, y)$

2) Translation

Translation means moving an object without changing its size. While translating, all the points on the shape will be shifted by the same number.

$x' = x + tx$

$y' = y + ty$

3) Scaling

Scaling of an object means multiplying every linear dimension of it by same factor.

$x' = x * sx$

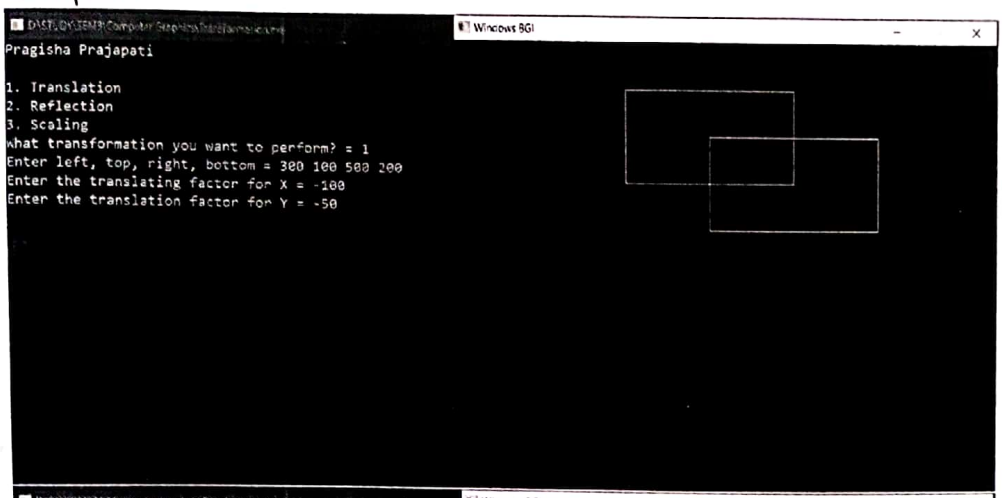
$y' = y * sy$

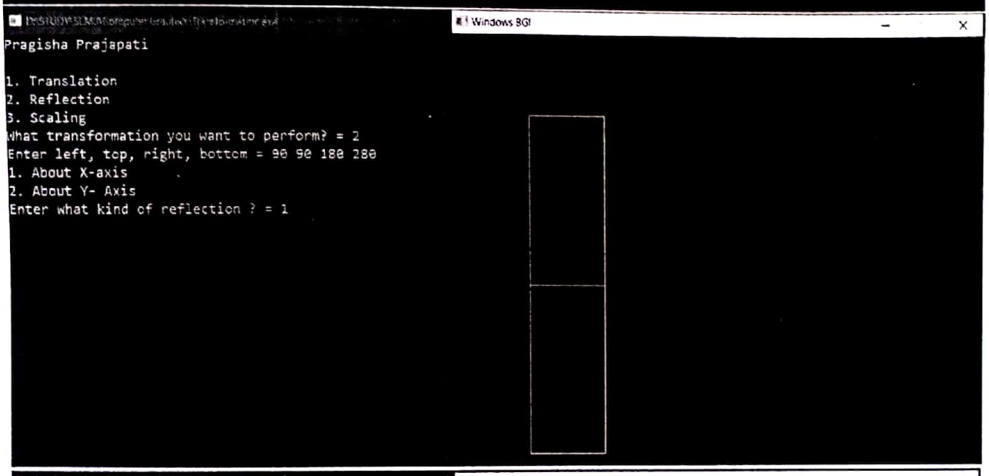
Source code :

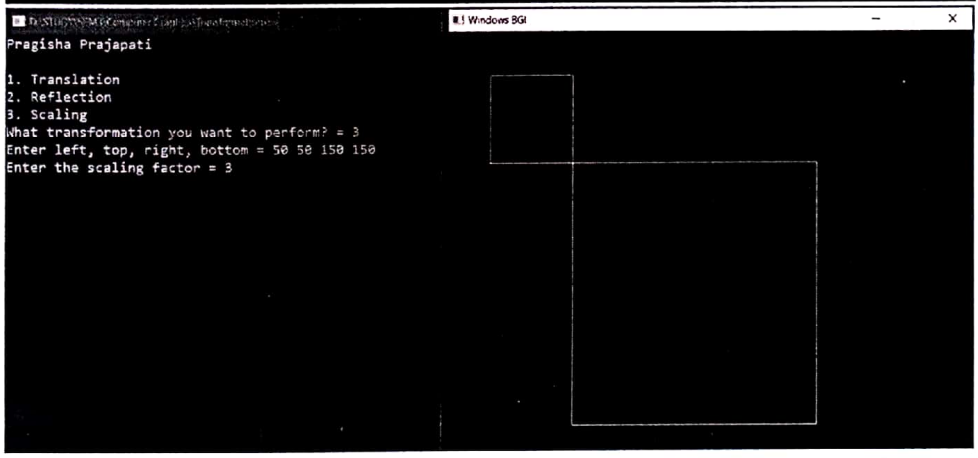
```
#include <iostream>
#include <graphics.h>
using namespace std;
int translation (int n , int tn){
        return (n+tn);
}
int scalar (int k , int n){
       return (k *n);
}
int main(){
      int n, gd= DETECT , gm , i , left , top , right , bottom , k,
      nr, yr;
      cout <<" Pragisha Prajapati " <<endl <<endl;
      cout <<" 1 Translation \n 2. Reflection \n 3. Scaling <<endl;
      cout <<" what transformation you want to perform?
       <<endl;
      cin >>n ;
      cout <<"Enter left , top , right , bottom = ";
       cin >> left >> top >> right >> bottom;
       rectangle (left , right , top , bottom);
       switch (n){
           case 1:
           {
               cout <<" Enter translating factor for n=";
               cin >>k;
                left =translation (left , k);
                right = translation (right , k);
                cout <<" Enter translating factor for y=";
                cin >>k;
                top = translation (top , k);
                bottom = translation (bottom , k);
                     rectangle (left , top , right , bottom);
                     getch();
            }
            break:
```

```
case 2:
    cout<<"1. About x-axis \n 2. About y-axis";
    cout<<"Enter what kind of reflection?";
    cin>>n;
    if (n==1){
        k= fabs (bottom - top);
        top = bottom;
        rectangle (left, top, right, bottom+k);
    }
    else if (n==2){
        k= fabs (left-right);
        left = right;
        rectangle (left, top, bottom, right+k);
    }
    else {
        cout<<"Wrong choice";
    }
    getch();
}
break:
    case 3:
    {
        cout<<"Enter the scaling factor:";
        cin>>k;
        top = scalar (k, top);
        left = scalar (k, left);
        right = scalar (k, right);
        bottom = scalar (k, bottom);
        rectangle (left, top, right, bottom);
        getch();
    }
}
system ("pause");
closegraph ();
return 0;
}
```

Output:







Conclusion:

Hence, we were able to implement 2D transformations using C program.

8/26

Title: Bezier Curve

Theory:

Bezier splines are highly useful, easy to implement and convenient for curve surface design. In general, a Bezier curve can be fitted to any number of central points. The number of central points to be approximated and their relation position determine the degree the Bezier polynomial. As with the interpolation splines, a Bezier curve can be specified with Boundary conditions, with the characterising matrix or with Blending function.

Suppose, we are given $n+1$ central points: $P_k(x_k, y_k, z_k)$ with k varying from 0 to n. These coordinate points can be blended to produce the following position vector $P(u)$, which describes the path of an approximating Bezier polynomial function between P_0 and P_n.

$$P(u) = \sum_{k=0}^{n} P_k \, BEZ_{k,n}(u), \quad 0 \le u \le 1$$

This vector equation represents a set of parametric equations for individual curve coordinate:

$$x(u) = \sum_{k=0}^{n} x_k \, BEZ_{k,n}(u),$$

$$y(u) = \sum_{k=0}^{n} y_k \, BEZ_{k,n}(u),$$

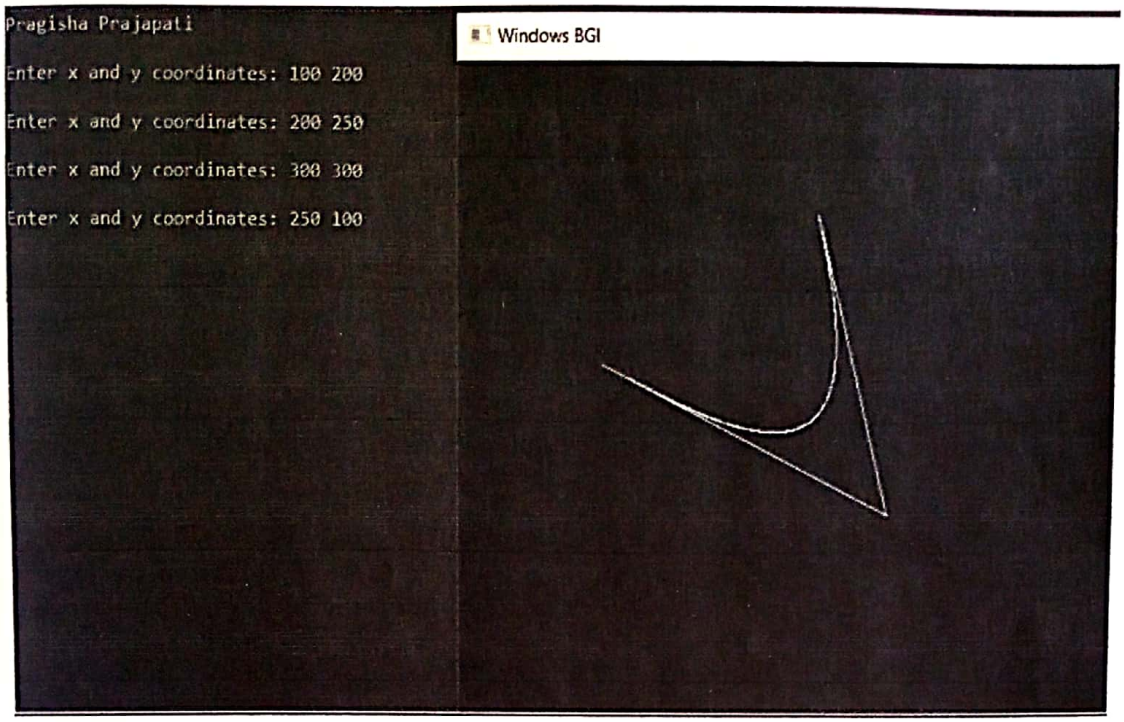
$$z(u) = \sum_{k=0}^{n} z_k \, BEZ_{k,n}(u),$$

Source code:

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

int main() {
        int  x[4], y[4], i;
        double  put-x, put-y, t;
        int  gr = DETECT, gm;
        initgraph (&gr, &gm, (char * )" ");
        printf ("Pragisha Prajapati\n");
        printf ("Bezier Curve ");
        printf ("\n Enter x and y coordinates:\n");
        for (i = 0; i<4; i++){
            scanf ("%d %d", &x [i], &y [i]);
        }
        for (i=0; i< 3; i++){
            line (x [i], y [i], x [i+1], y [i+1]);
        }
        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:



Conclusion:

Hence, we were able to implement Bezier curve using C program in graphics.

9/6

Title: 3D transformations

Theory:

In very general terms, a 3D model is a mathematical representation of a physical entity that occupies space. In more practical term, a 3D model is made of a description of its shape and a description of its color appearance. 3D transformation is process of manipulating the view of a 3D object with respect to its original position by modifying its physical attributes through various methods of transformation like Translation, Scaling, Rotation, etc.

Types of transformation:

1) Translation
2) Scaling
3) Rotation
4) Shear
5) Reflection

1) Translation:

It moves the object without changing its size.

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

2) Scaling:

It multiplies every dimension of it by same factor.

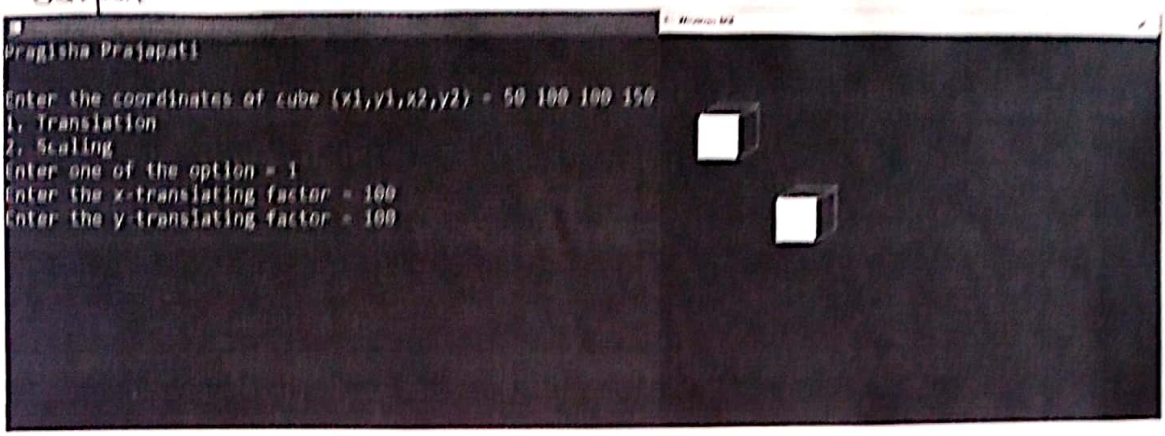
formula:
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

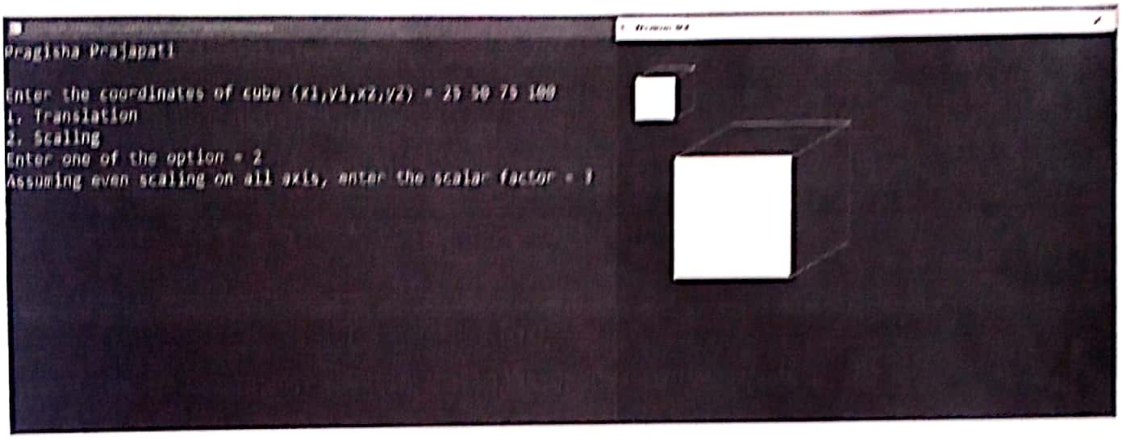
Source code:

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <graphics.h>
#include <stdlib.h>
int translation (int n, int tn){
        return (n + tn);
}
int   scalar (int k, int n){
        return (k * n);
}
int main (){
        int gd = DETECT, gm;
        initgraph (&gd, &gm, (char *)" ");
        printf ("Pragisha Prajapati ");
        printf ("Enter the coordinates of cube (x1, y1, x2,
        y2) = ");
        int left, right, bottom, top, depth;
        scanf ("%d, %d %d %d", &left, &top, &right,
        &bottom);
        depth = fabs ((right - left) /2);
        bar3d (left, top, right, bottom, depth, 1);
        getch();
        printf ("1. Translation \n 2. Scaling \n ");
        printf ("Enter one of the option=" );
        int k;
        scanf ("%d", &k);
        switch (k){
        case 1 :
        {
                printf ("Enter the x-translating factor =");
                scanf ("%d", &k);
                left = translation (left, k);
```

```
        right = translation (right, k);
        printf ("Enter the y-tranlating factor = ");
        scanf ("%d", &k);
        top = translation (top, k);
        bottom = translation (bottom, k);
        bar3d (left, top, right, bottom, depth, i);
        getch();
        break;
    }
    case 2:
    {
        printf ("Assuming even scaling on all axis,
                enter the scalar factor = ");
        scanf ("%d", &k);
        left = scalar (left, k);
        right = scalar (right, k);
        top = scalar (top, k);
        bottom = scalar (bottom, k);
        depth = fabs ((right - left)/2);
        bar3d (left, top, right, bottom, depth, i);
        getch();
        break;
    }
    default:
    {
        printf ("Invalid selection !");
    }
    closegraph();
    return 0;
}
```

Output :





Conclusion :

In this way, we were able to implement 3D transformations using C program in graphics.

9/6

Title: Cohen-Sutherland line clipping algorithm

Theory:

Cohen-Sutherland line clipping algorithm is oldest and most popular line clipping algorithm. In this method, coordinate system is divided into nine regions. All regions are assigned a four digit code to each region as below:

1001	1000	1010
0001	0000	0010
0101	0100	0110

Algorithm:

Step 1: Establish region code for all line end point.

- First bit is 1 if $x < x_{wmin}$ (point lies to left of window), else set it to 0.
- Second bit is 1 if $x > x_{woman}$ (point lies to right of window), else set it to 0.
- Third bit is 1 if $y < y_{wmin}$ (point lies to below of window), else set it to 0.
- Fourth bit is 1 if $y > y_{wman}$ (point lies to above window), else set it to 0.

Step 2: Determine which line are completely inside window and which are not.

a) If both end points of line has region code '0000' line is completely inside window.

b) If logical AND operation of region codes of two end points is NOT '0000', the line is completely outside.

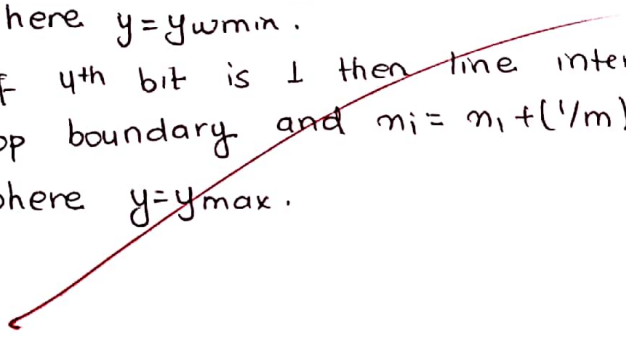
Step 3: If both fail tests then line is partially visible so need to find the intersection with boundaries of window:

a) If 1st bit is 1, then line intersect with left boundary and $y_i = y_i + m(x - x_1)$ x_{wmin}.

b) If 2nd bit is 1 then intersect with right boundary and $y_i = y_1 + m(x - x_1)$; $x = x_{wmax}$

c) If 3rd bit is 1, then line intersect with bottom boundary and $x_i = x_1 + (1/m)(y - y_1)$ where $y = y_{wmin}$.

d) If 4th bit is 1 then line intersect with top boundary and $x_i = x_1 + (1/m)(y - y_1)$ where $y = y_{max}$.



Source code:

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
void main (){
    int rcode-begin [4] = {0,0,0,0}, rcode-end [4]
    = {0,0,0,0}, region-code [4];
    int xmax, ymax, xmin, ymin, flag=0;
    printf ("\n Pragisha Prajapati");
    float slope;
    int x, y, x1, y1, i, xc, yc;
    int gr = DETECT, gm;
    initgraph (&gr, &gm, (char *)" ");
    printf ("Enter xmin, ymin = \n");
    scanf ("%d %d", &xmin, &ymin);
    printf ("Enter xmax, ymax = \n");
    scanf ("%d %d", &xman, &yman);
    printf ("Enter initial point x and y = ");
    scanf ("%d %d", &x, &y);
    printf ("Enter final point x1 & y1 = ");
    scanf ("%d %d", &x1, &y1);
    cleardevice ();
    rectangle (xmin, ymin, xman, yman);
    line (x, y, x1, y1);
    line (0, 0, 600, 0);
    line (0, 0, 0, 600);
    if (y > yman) {
        rcode-begin [0] = 1;
        flag = 1;
}
```

```
if (y<ymin){
    rcode-begin[1]=0;
    flag=1;
}
if (n>nmax){
    rcode-begin[2]=1;
    flag=1;
}
if (n<nmin){
    rcode-begin[3]=1;
    flag=1;
}
if (y1>ymax){
    rcode-end[0]=1;
    flag=1;
}
if (y1<ymin){
    rcode-end[1]=1;
    flag=1;
}
if (n1>nmax){
    rcode-end[2]=1;
    flag=1;
}
if (n1<ymin){
    rcode-end[3]=1;
    flag=1;
}
if (flag==0){
    printf("No need to clip as it is
     already in the window\n");
}
flag=1;
```

```
for (i=0; i<4; i++){
    region_code [i] = rcode_begin [i] &&
    rcode_end [i];
    if (flag ==0){
        printf ("line is outside window \n");
    }
    else {
        slope = (float) (y1-y)/(x1-x);
        if (rcode_begin [2]==0 &&
            rcode_begin [3]=1){
            y=y+(float)(xmin-x) * slope;
            x=xmin;
        }
        if (rcode_begin [2]==1 &&
            rcode_begin [3]==0){
            y=y+(float)(xmax-x) * slop;
            x=xmax;
        }
        if (rcode_begin [0]==1 &&
            rcode_begin [1]==0){
            x=x+(float)(ymax-y)/slope;
            y=ymax;
        }
        if (rcode_begin [0]==0 &&
            rcode_begin [1]==1){
            x=x+(float)(ymin-y)/slope;
            y=ymin;
        }
        if (rcode_end [2]=0 &&
            rcode_end [3]==1){
            y1=y1+(float)(xmin-x1) *slope;
            x1=xmin;
        }
```

```
        if (rcode.end[2]==1 && rcode.end[3]==0){
            y1 = y1 +(float)(xmax - x1) * slope;
            x1 = xmax;
        }
        if (rcode.end[0]==1 && rcode.end[1]==0){
            x1 = x1 + (float)(ymax - y1) * slope;
            y1 = ymax;
        }
        if (rcode.end[0]==0 && rcode.end[1]==1){
            x1 = x1 + (float)(ymin - y1) /slope;
            y1 = ymin;
        }
    }
    delay (9000);
    clearviewport();
    rectangle (xmin, ymin, xmax, ymax);
    line (0,0,600,0);
    line (0,0,0,600);
    selectcolor (RED);
    line (x,y,x1,y1);
    getch();
    closegraph();
}
```

Output:

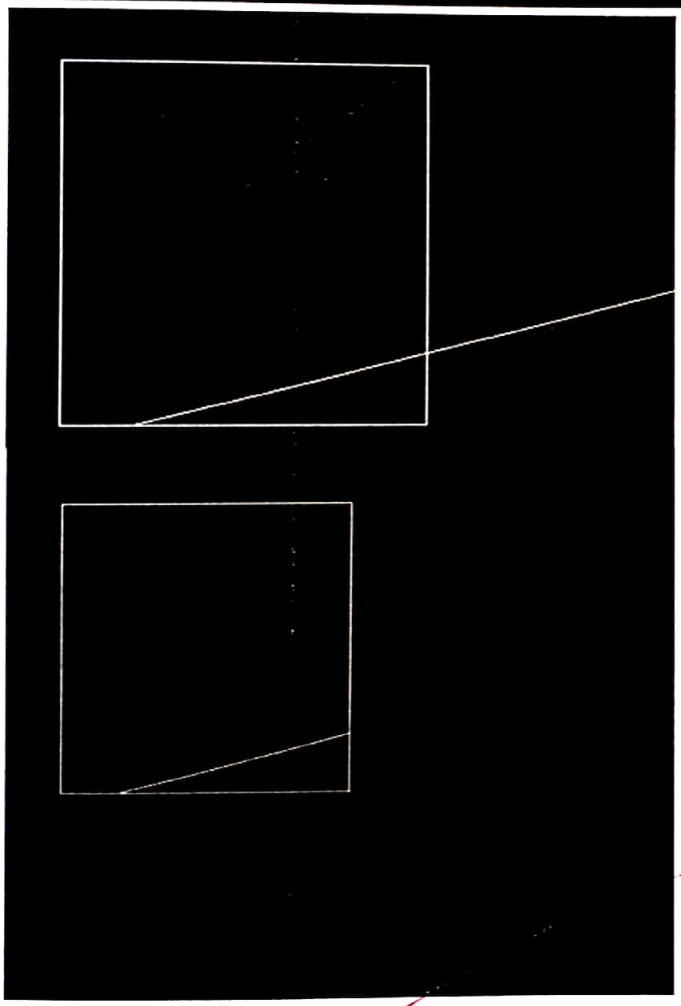
```
Pragisha Prajapati

Enter XMin, YMin= 50 50

Enter XMax, YMax =300 300

Enter intial point x and y= 500 200

Final point x1 and y1= 100 300
```



11/11

Title: Liang-Barsky line clipping algorithm.

Algorithm:

Step 1: Input $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$

Step 2: Input window clipping boundary.
(xmin, xmax, ymin, ymax)

Step 3: Determine p_i and q_i as follows:

$p_1 = -\Delta x$ $\quad q_1 = x_1 - x_{min}$ (left boundary)

$p_2 = \Delta x$ $\quad q_2 = x_{max} - x_1$ (right boundary)

$p_3 = -\Delta y$ $\quad q_3 = y_1 - y_{min}$ (bottom boundary)

$p_4 = \Delta y$ $\quad q_4 = y_{max} - y_1$ (top boundary)

Step 4: If $(p_i = 0)$ and $(q_i < 0)$ for any line is parallel to window boundary and is outside of window so line segment is discorted.

Step 5: Set two parameters t_1 & t_2;
$t_1 = 0$; $t_2 = 1$;

Step 6: Determine t_i as follows:

$T_i = \frac{q_i}{p_i}$, where $I = 1, 2, 3, 4, 5$.

Step 7: Find all y_i for which $p_i < 0$
set $t_1 = max(0, rp)$

Step 8: Find all y_i for $p_i > 0$. set $t_2 = min(1, rp)$

Step 9: If $(t_1 > t_2)$ line is completely outside window so discard line else if $(t_1 = 0$ & $t_2 = 1)$ line is completely inside window else clip line segment & determine (x_1', y_1') & (x_2', y_2') as; $x_1' = x_1 + t_1 \Delta x$, $x_2' = x_1 + t_2 \Delta x$,
$y_1' = y_1 + t_1 \Delta y$, $y_2' = y_1 + t_2 \Delta y$.

Step 10: Stop

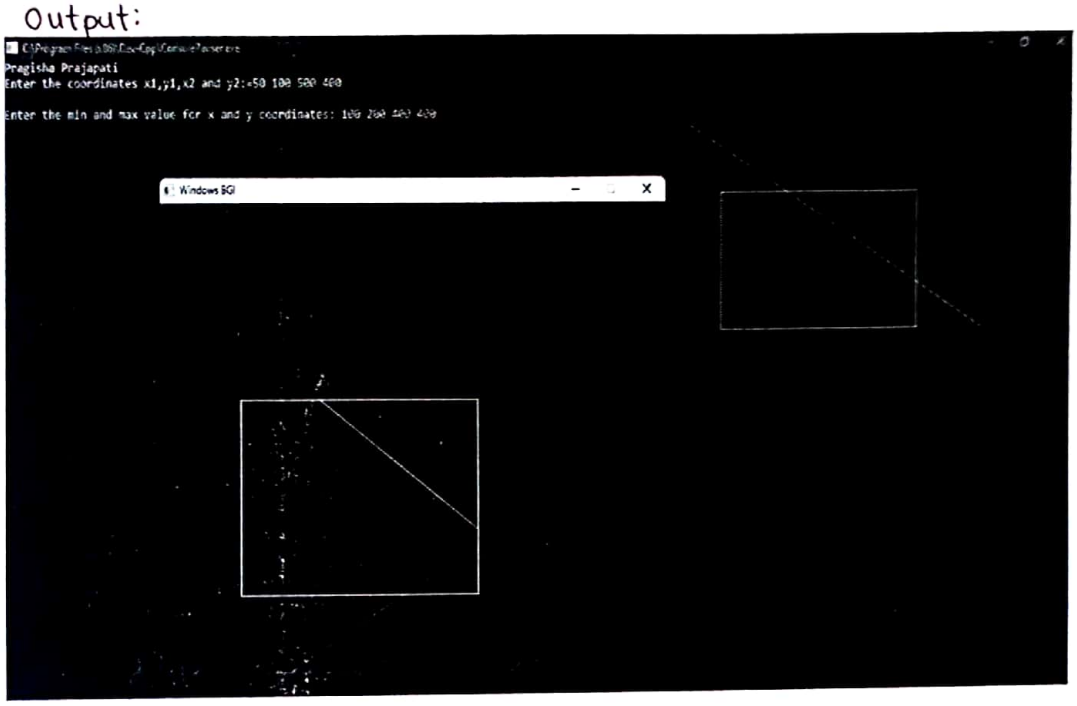
Source code:

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <graphics.h>
#include <dos.h>
int main(){
    int i, gd=DETECT, gm;
    int x1, y1, x2, y2, xmin, xmax, ymin, ymax,
    xx1, xx2, yy1, yy2, dx, dy;
    float t1, t2, p[4], q[4], temp;
    printf("Enter the top values (xmin, ymin):");
    scanf("%d %d", &xmin, &ymin);
    printf("Enter the starting points");
    scanf("%d, %d %d %d", &y1, &x2, &x1, &y2);
    initgraph(&gd, &gm, (char*)"");
    rectangle(xmin, ymin, xmax, ymax);
    line(x1, y1, x2, y2);
    dx = (x2-x1);
    dy = y2-y1;
    p[0] = -dx;
    p[1] = dx;
    p[2] = -dy;
    p[3] = dy;
    q[0] = x1-xmin;
    q[1] = xmax-x1;
    q[2] = y1-ymin;
    q[3] = ymax-y1;
    for(i=0; i<4; i++){
        if(p[i]==0)
            printf("line is parallel");
```

```
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){
    nn1 = n1+t1 * p[i];
    nn2 = n1 + t2 * p[1];
    yy1 = y1+t1 *p[3];
    yy2= y1 + t2 * p[3];
    delay (1000);
    clear view port ();
    rectangle (nmin, ymin, nman, yman);
    line (nn1, yy1, nn2, yy2);
}
delay (7000);
closegraph ();
}
```

Output:



Conclusion:

Hence, we were able to implement ~~cohen~~ Liang Barsky ~~Sutherland's~~ line clipping algorithm using C program.

11/11

Title: OpenGL and its installation

Theory:

Opengl is a software interface that allows programmer to communicate with graphics hardware. Opengl is the short form for "Open Graphics library". It is an application programming Interface (API) designed for rendering 2D and 3D graphics. It provides a common set of commands that can be used to manage graphics in different applications and on multiple platforms.

Opengl Utility Toolkit (GLUT) has been created to aid in the development of more complicated 3D objects such as a sphere, a torus and even a teapot. Like all software libraries, opengl creates of a series of functions calls that can be invoked from our own programs.

Install opengl on windows in codeBlock

1) Download CodeBlock and install it.
2) Go to Freeglut windows development libraries and download zip file from the download link that appear after free glutwingw package with having link name as Download freeglut 3.0.0 for MinGW and extract it.
3) Open notepad with run as administrato and open file from
   a) This PC > C: (c-drive > program files (x86) > codeBlocks share > codeBlocks > templates, then click to show all files)

b) Next open glut.cbp and search all glut 32 and replace with freeglut.

4. Then open from this PC > C > program files (x86) > code Blocks > share > codeBlocks > templates > wizard > glut then click to show all files.

5. open wizard.script and here, also replace all glut32 with freeglut.

6. Then goto freeglut floder (where it was downloaded) and
   a) Include > GL and copy all four file from there.
   b) Goto this PC > C > program files (x86) > clodeBlocks > MINGW > include > GL and plate it.
   c) Then from download folder freeglut > li6.copy two files and goto this PC > C > program files (x86) > CodeBlocks > MINGW > lib and paste it.
   d) Again goto download folder freeglut > bin and copy one file (freeglut.all) from here and goto This PC > C > windows > syswindow64 and paste this file.

7. Now open code Blocks.
   a) Select file > New Project > GLUT project > Next
   b) Give project title anything and then choose next.
   c) for selecting GLUT is location: This PC > C > programs files (X86) > code Blocks > MinGW.
   d) Press OK > Next > finish.

   Now, codeBlocks is ready to test for openGL file.

Title: Creating lines in openGL

Theory:

In openGL, the term line refers to a line segment. There are easy ways to specify a connected series of line segments or even closed. Connected series of segments.

With openGL, we can specify lines with different width and lines that are stippled in various ways - dotted, dashed and so on. Drawing a line with openGL is apparently quite simple using following code:

```
glbegin(GL_lines);
glvertex2i(x0,y0);
glvertex2i(x1,y1);
glEnd();
```

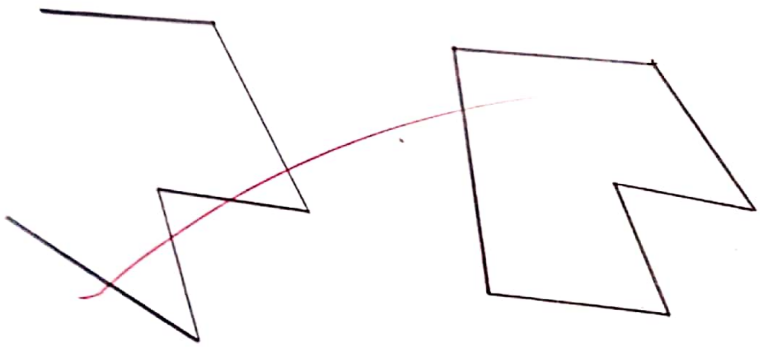
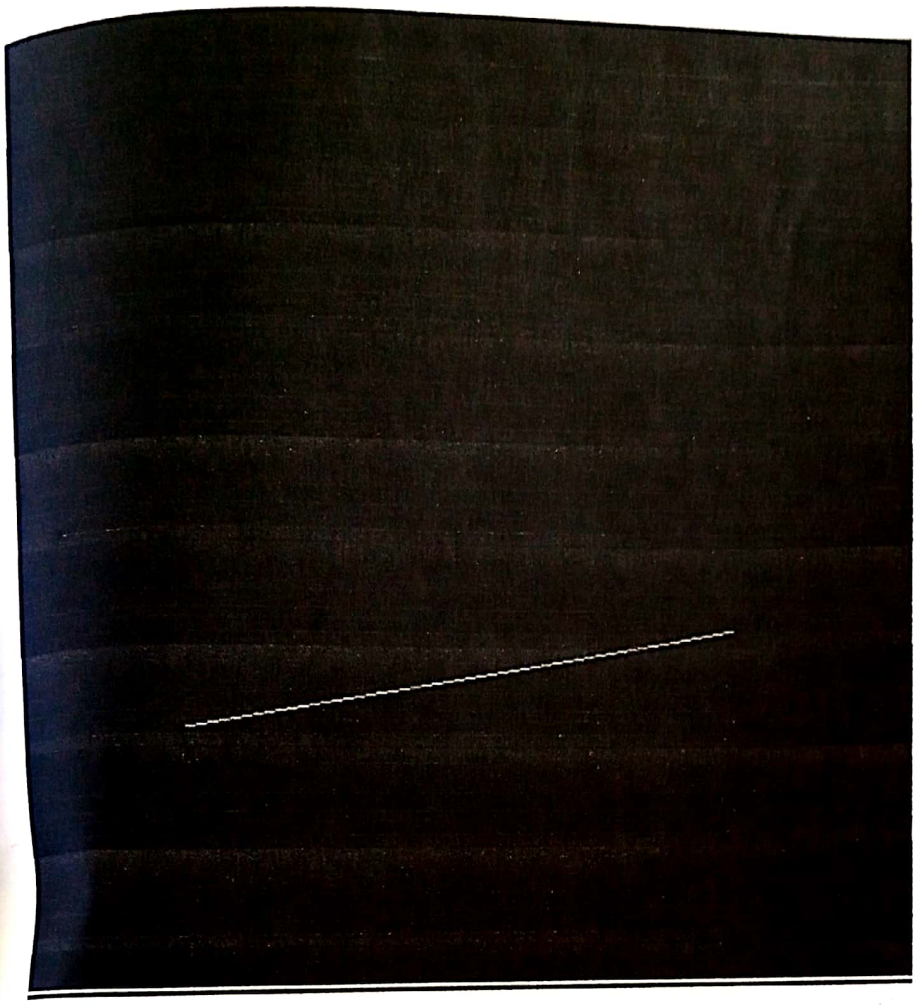


Fig: Two conneted series of line segments.

```
#ifdef-APPLE
#include <GLUT/glut.h>
#else
#include <windows.h>
#include <GL/glut.h>
#endif
#include<stdlib.h>
void init (void){
    glclearcolor (0,0,0,0);
    glviewport (0,0,500,500);
    glmatrix mode (GL-projection);
    glLoadIdentity ();
    glortho (0,500,0,500,1,-1);
    glorthoidentity ();
}
void draw (void){
    glclear (GL-color-Buffer-BIT);
    glcolor3F (1.0,1.0,1.0);
    gl pointsize (13.0);
    glBegin (GL-lines);
    glvertex 2d (275,450);
    glvertex2d (165,25);
    glEnd ();
    glFlush ();
}
int main (int orgc, char **argv){
    glutinit (&argc, argv);
    glutInitWindow position (10,10);
    glut Init windowsize (500,500);
    glutInit Displaymode (GLUT-Single/GLUT-RGB);
    glut Createwindow ("Pragisha line output");
    init ();
    glutDisplayTune (drawlines);
    glutMain loop ();
}
```

Output



Conclusion:

Hence, we were able to draw a line using OpenGL in lab.

12/20

Title: Creating Triangle using openGL

Theory:

Triangles are simply polygons. Polygons in openGL are drawn by fitting in all the pixels enclosed within the boundary but we can also draw them as outlined polygons or simply points at the vertices. We can draw triangle using following segments.

```
glBegin(GL_TRIANGLE);
glVertex3f(x0, y0, z0); // Top
glVertex3f(x1, y1, z1); // Bottom left
glVertex3f(x2, y2, z2); // Bottom right
glEnd();
```

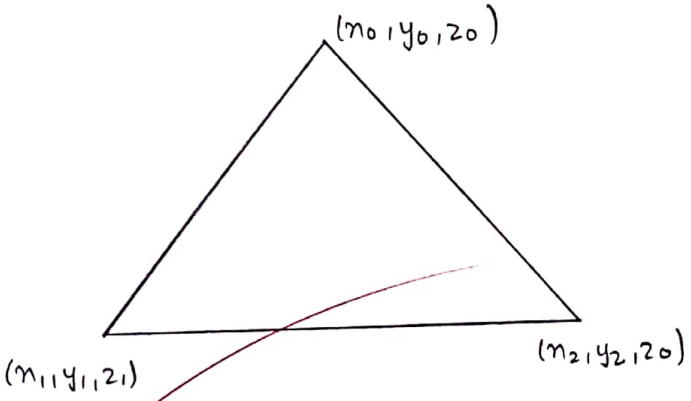


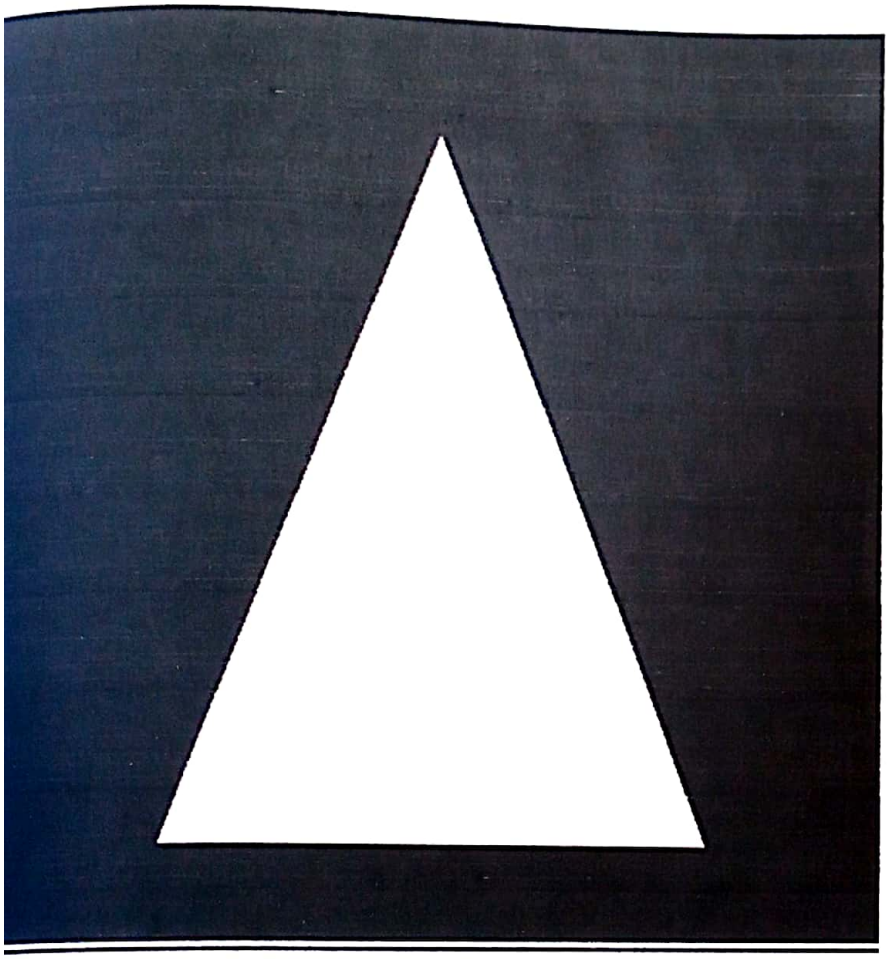
Fig: Triangle using openGL

Opengl Source Code

```
#include <windows.h>
#ifdef __APPLE__
#include <GLUT/glut.h>
#endif
#include <stdlib.h>
void display(){
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POLYGON);
    glColor3f(1,1,1); glVertex3f(-0.6,-0.75,0.5);
    glColor3f(1,1,1); glVertex3f(0.6,-0.75,0);
    glColor3f(1,1,1); glVertex3f(0,0.75,0);
    glEnd();
    glFlush();
}

int main(int argc, char **argv){
    glutInit(&argc, argv);
    glutInitWindowPosition(80,80);
    glutCreateWindowSize(400,300);
    glutCreateWindow("Pragisha Triangle output");
    glutDisplayFunction(display);
    glutMainLoop();
}
```

Output



Conclusion:

Hence, we were able to draw a triangle using OpenGL in lab.

12/20