# A lab Report on Computer Graphics

Mamata Maharjan(077BCT043)[*]
*Submitted to the Department of Electronics and Computer Engineering,*
*Pulchowk Campus, Institute of Engineering, Lalitpur.*
(Dated: August 19, 2023)

This lab report provides a comprehensive exploration of computer graphics, focusing on the practical implementation of various algorithms. Through a series of structured lab sessions, each algorithm is meticulously implemented using the C++ programming language. These algorithms encompass foundational techniques such as the Digital Differential Analyzer (DDA) and Bresenham's Algorithm for efficient line drawing, as well as advanced approaches including the Mid-Point Circle and Ellipse Algorithms. The implementation of line clipping using the Cohen-Sutherland and Liang-Barsky Algorithms, alongside the execution of 2D Transformations, enriches our understanding of geometric transformations. The report also delves into the art of crafting Bezier Curves and the application of Flood Filling for adding color. By navigating the interplay between algorithms and C++ programming, this report underscores the fusion of theory and practice, unveiling the intricacies of computer graphics algorithms and their tangible applications.

## I. INTRODUCTION

Computer Graphics refers to the creation, manipulation, and representation of visual images and graphical content through the use of computers. It encompasses a broad range of techniques and algorithms that enable the generation of visual content, from simple geometric shapes to complex, photo realistic scenes.

DDA Algorithm (Digital Differential Analyzer) is a fundamental line drawing algorithm used to approximate straight lines on a digital grid. It calculates the incremental changes in x and y coordinates and plots the closest pixel to achieve line rendering. Bresenham's Algorithm is another line drawing method that is more efficient than DDA, as it uses integer arithmetic and avoids floating-point calculations, resulting in smoother lines and better performance. The Mid-Point Circle Algorithm and Mid-Point Ellipse Algorithm are used for drawing circles and ellipses on digital screens, respectively. These algorithms calculate pixel coordinates based on a decision parameter, ensuring efficient and accurate circle and ellipse rendering.

For the purpose of efficient line clipping, two prominent algorithms are used: the Cohen-Sutherland Line Clipping Algorithm and the Liang-Barsky Line Clipping Algorithm. The former categorizes endpoints with respect to a rectangular window and clips lines based on their relative positions, while the latter employs parametric equations to find intersections with window boundaries and offers better efficiency for various cases. 2D Transformations encompass translation, rotation, scaling, and shearing, allowing graphical objects to be positioned, oriented, resized, and skewed. Bezier Curves are utilized for creating smooth curves between control points, offering flexible shape control, and are often used in computer-aided design and animation. Flood Filling is an algorithm used to color closed areas, starting from a seed point and filling adjacent pixels that meet specific criteria, which is a vital tool for painting and rendering solid regions in computer graphics. All of the aforementioned Algorithms were implemented using c++ as a part of the computer graphics lab.

### A. Initial Lab: Introduction to Computer Graphics (graphics.h)

- **Objective:** To get familiar with the computer graphics using c/c++ using graphics.h.
- **Theory:** The codes were written in graphics.h and $TURBOC++$ in lab but was completed with a 32 bit compiler and a visual studio code at home. The code uses a simple logic of static addressing the simple line drawing and arranging them in such a way that they take the shape of an object. Those static objects were made to move writing a few lines of logic and refreshing the window with certain delay.

**Source code**

```
#clock.c
#include<graphics.h>
#include<math.h>
#include<dos.h>

int main(int argc, char const *argv[])
{
    int gd = DETECT, gm;
    initgraph(&gd,&gm,(char*)"");

    int x1,x2;
```

[*] Email: 077bct043.mamata@pcampus.edu.np

```
    float a1;

    x1 = 380;
    x2 = 200;
    a1 = 0;

    setcolor(YELLOW);

    for(int i =0;i<60;i++)
    {
        circle(320,200,65);
        x1 =320 + int(60 * cos(-1.57075 +a1));
        x2 =200 + int(60 * sin(-1.57075 +a1));
        line(320,200,x1,x2);
        a1 = i*0.10472; //6 degrees division
        delay(1000);//1 seconds
        clearviewport();
    }

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

#vehicle.c
#include<graphics.h>
#include<math.h>
#include<dos.h>

int main(int argc, char const *argv[])
{
    int gd = DETECT, gm;
    initgraph(&gd,&gm,(char*)"");

    int x1,x2,x3,x4;
    float a1;

    x1 = 147;
    x2 = 317;
    x3 = 267;
    a1 = 0;

    setcolor(YELLOW);

    for(int i =0;i<400;i++)
    {
        //car body
        line(100+i,200,100+i,300);
        line(100+i,300,110+i,300);
        line(150+i,300,230+i,300);
        line(270+i,300,300+i,300);
        line(300+i,300,300+i,250);
        line(300+i,250,250+i,250);
        line(250+i,250,100+i,250);
        line(250+i,250,200+i,200);
        line(150+i,200,150+i,250);
        line(200+i,200,100+i,200);
        //car body

        //tires
        circle(130+i,300,20);
        circle(250+i,300,20);
        circle(130+i,300,17);
        circle(250+i,300,17);
        line(130+i,300,x1,x2);
        line(250+i,300,x3,x2);
        x1 =130+i+ int(17 * cos(a1));
        x2 =300 + int(17* sin(a1));
        x3 =250+i+ int(17* cos(a1));
        a1 = i*0.10472; //6 degrees division
        //tires

        //road
        line(0,320,800,320);

        delay(100);//1 seconds
        clearviewport();
    }

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

- **Results and Analysis**
  This initial lab helped us get familiar with the workings and coordinates of the graphics window. The two task provided seemed hard at the beginning but was very fun once things started working out as envisioned.

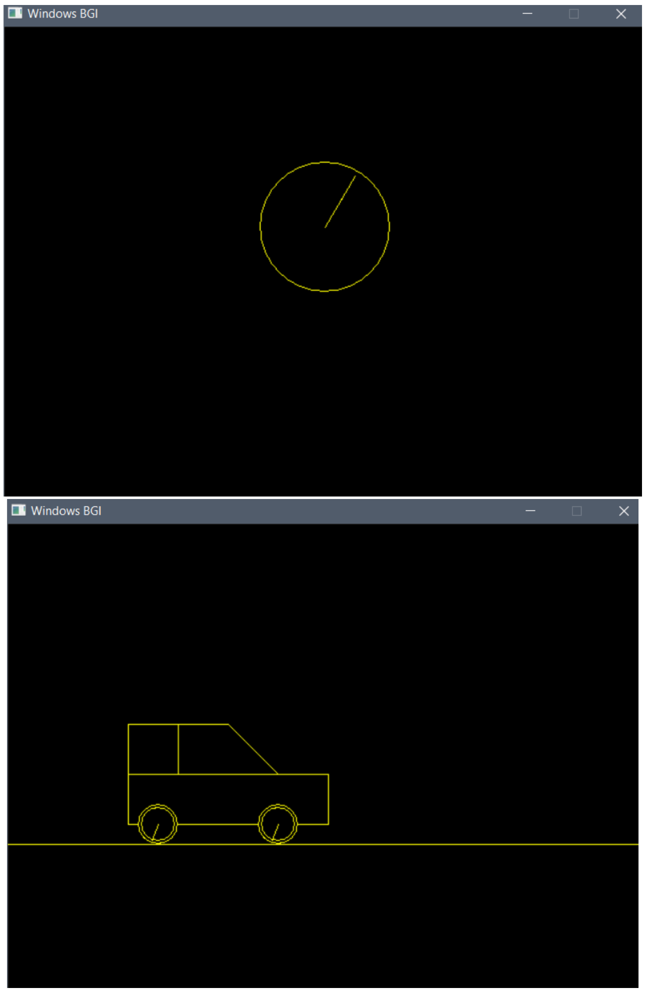


Figure 1. Fig: Rotating clock and moving vehicle

## B. Lab 1: Digital Differential Analyzer Algorithm

- **Objective:** To implement Digital Differential Analyzer Algorithm
- **Theory:** The algorithm starts at one point and moves step by step towards the other point. It calculates how much the x (horizontal) and y (vertical) values change between the two points. Then, it divides these changes into smaller steps and calculates the coordinates of each step. By doing this repeatedly, it identifies a series of pixels that, when colored, create a line between the two points.

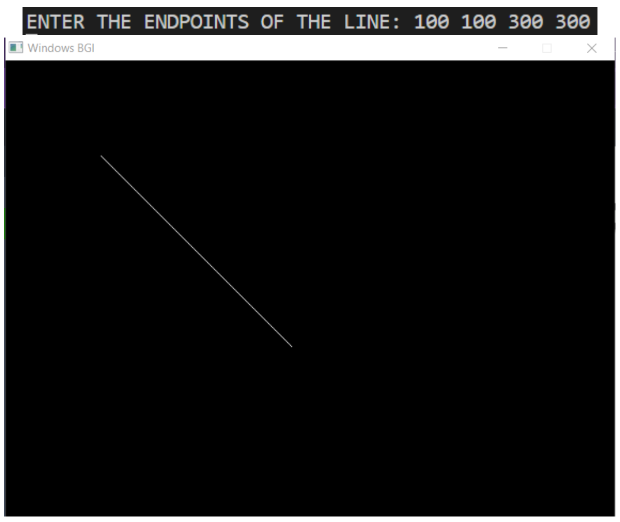
  In simple terms, the DDA algorithm takes a line between two points and breaks it into tiny steps. It figures out the pixels to color in order to draw the line on a digital screen. It's like connecting dots step by step to form a line.

### Source code

```
//DDA Algorithms
#include<graphics.h>
#include<math.h>
#include<conio.h>
#include<dos.h>

int main(int argc, char const *argv[])
{
    int gd = DETECT, gm;
    initgraph(&gd,&gm,(char*)"");

    int x1,x2,y1,y2,i;
    float delx,dely,steps,xin,yin;

printf("ENTER THE ENDPOINTS OF THE LINE: ");
scanf("%d %d %d %d",&x1,&y1,&x2,&y2);

    delx = x2 - x1;
    dely = y2 - y1;

    if(abs(delx)>=abs(dely))
        steps = abs(delx);
    else
        steps = abs(dely);

    xin = delx/steps;
    yin = dely/steps;

    for(i = 0;i<=steps;i++)
    {
        putpixel(int(x1),int(y1),WHITE);
        x1 +=xin;
        y1 += yin;
        delay(100);
    }

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

- **Results and Analysis**

The Digital Differential Analyzer (DDA) algorithm, which was implemented using C++, provided a straightforward method for drawing lines in computer graphics. By incrementally calculating pixel coordinates between two endpoints, the algorithm efficiently created lines on a digital screen. Its simplicity and effectiveness made it a valuable tool for rendering lines in various applications, offering a visual representation of lines connecting two points.



```
ENTER THE ENDPOINTS OF THE LINE: 100 100 300 300
```

Figure 2. Digital Differential Analyzer Algorithm

### C. Lab 2: Bresenham's Line Algorithm

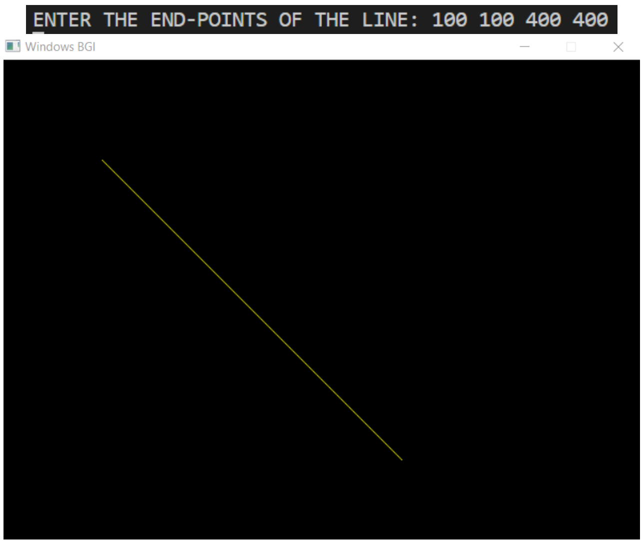
- **Objective:** To implement the Bresenham's Line Algorithm in C/C++
- **Theory:** Bresenham's Algorithm is a widely used method in computer graphics for drawing lines on a digital screen. Named after its creator Jack E. Bresenham, this algorithm efficiently determines which pixels to color in order to create a straight line between two given endpoints. It avoids the need for complex floating-point calculations by working with integer increments, making it faster and suitable for computers with limited processing power. Bresenham's Algorithm starts by calculating the initial pixel position and then uses incremental calculations to decide which pixel should be colored next based on the slope of the line. By comparing the distances between the real line and the candidate pixels, it intelligently selects the pixel closest to the actual line path. This approach ensures that the drawn line closely follows the intended line between the endpoints.

**Source code**

```
//BLA Algorithms
#include<iostream>
#include<graphics.h>
#include<cmath>
#include<conio.h>
#include<dos.h>

class BeserhamLineAlgorithm
{
private:
int a,b,x1,x2,y1,y2,i,delx,dely,p0;

public:
BeserhamLineAlgorithm():p0(0),i(0)
{
    getData();

    delx = fabs(x2-x1);
    dely = fabs(y2-y1);
    if(x2>x1)
    a = 1;
    else
    a = -1;

    if(y2>y1)
    b = 1;
    else
    b = -1;
}

void calculation()
{
    if(delx>dely)
    {
        p0 += 2*dely - delx;

        for(i =0 ;i<=delx;i++)
        {

            putpixel(x1,x2,RED);
            delay(50);

            if(p0<0)
            {
                x1 += a;
                p0 += 2*dely;
            }
            else
            {
                x1 += a;
                y1 += b;
                p0 += 2*dely - 2*delx;
            }
        }
    }
    else
    {
        p0 += 2*delx - dely;

        for(i = 0;i<dely;i++)
        {
            putpixel(x1,y1,YELLOW);
            delay(50);

        if(p0<= 0)
        {
            y1 += b;
            p0 += 2 * delx;
        }
        else
```

```
                {
                    x1 += a;
                    y1 += b;
                    p0 += 2*delx-2*dely;
                }
                }
            }
        }
        void getData()
        {
            std::cout<<"ENTER THE END-POINTS OF THE
                 LINE: ";
            std::cin>>x1>>y1>>x2>>y2;
        }

    };
    int main(int argc, char const *argv[])
    {
        int gd = DETECT, gm;
        initgraph(&gd,&gm,(char*)"");

        BeserhamLineAlgorithm BLA;
        BLA.calculation();

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

- **Results and Analysis**



```
ENTER THE END-POINTS OF THE LINE: 100 100 400 400
```

Figure 3. Bresenham's Line Algorithm

In the realm of computer graphics, Bresenham's Algorithm, which was implemented using C++, provided an ingenious method for drawing lines. By making use of integer arithmetic and incremental calculations, the algorithm precisely determined pixel positions to render lines on a digital screen.

### D. Lab 3: Midpoint Circle Algorithm

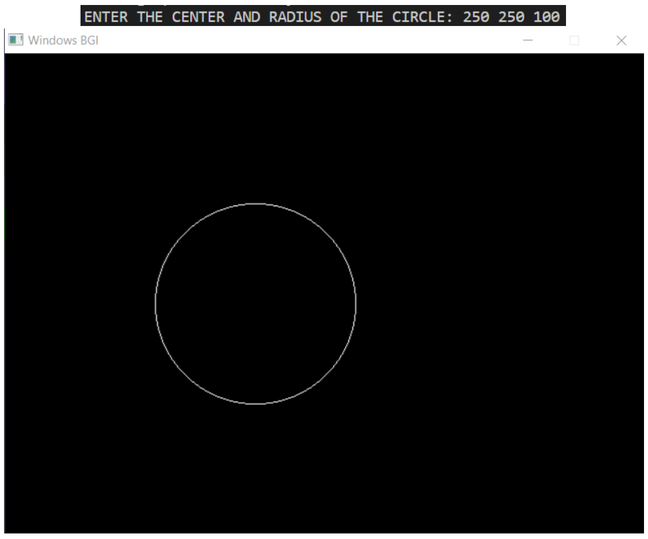
- **Objective:** To design n-bit (4-bit) sub tractor for unsigned integer binary numbers.
- **Theory:** The Mid-point Circle Algorithm is a technique used for drawing circles in computer graphics. It operates by calculating pixel positions symmetrically around the circle's center, incrementally adjusting these positions to create the circular shape. It is particularly efficient because it only requires calculations for one-eighth of the circle's circumference, then mirrors these positions to form the entire circle. The algorithm begins by setting an initial point and calculating other points symmetrically in eight octants of the circle. It uses the idea of "midpoints" to determine the best pixel choices, keeping track of the decision to color the interior or exterior pixel based on the midpoint's position. As the algorithm progresses, it moves along the circle's edges, incrementally calculating the pixel positions in each octant. By mirroring these points to form the complete circle, the algorithm produces a smooth and accurate circular shape on the screen.

#### Source code

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

class midpointCircleAlgorithm
{
    private:
    int a,b,r,xk,yk,pk,i;

    public:
    midpointCircleAlgorithm():i(0)
    {
        getData();

        xk = 0;
        yk = r;
        pk = 1-r;
    }

    void composeOneEighth()
    {
        delay(50);
        if(pk <= 0)
        {
            xk += 1;
            pk +=2 *xk +1;
        }
        else
        {
            xk += 1;
            yk -= 1;
            pk += 2*xk - 2*yk +1;
        }

    }

    void makeCircle()
    {
        for( i =0 ; xk<=yk; i++)
        {
           putpixel(yk+b,xk+a,WHITE);
            putpixel(yk+b,-xk+a,WHITE);
            putpixel(-yk+b, xk+a,WHITE);
            putpixel(-yk+b,-xk+a,WHITE);
            putpixel(xk+a,yk+b,WHITE);
            putpixel(-xk+a,yk+b,WHITE);
            putpixel(xk+a,-yk+b,WHITE);
            putpixel(-xk+a,-yk+b,WHITE);
            composeOneEighth();
        }
    }
    void getData()
    {
        std::cout<<"ENTER THE CENTER AND RADIUS
             OF THE CIRCLE: ";
        std::cin>>a>>b>>r;
    }

};
int main(int argc, char const *argv[])
{
    int gd = DETECT, gm;
    initgraph(&gd,&gm,(char*)"");

   midpointCircleAlgorithm mpa;
   mpa.makeCircle();

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

- **Results and Analysis**

The Mid-point Circle Algorithm, implemented with C++, stood as a remarkable approach for drawing circles in computer graphics. By calculating pixel positions based on a circle's midpoint and its symmetrical properties, the algorithm elegantly crafted circular shapes on a digital window.



```
ENTER THE CENTER AND RADIUS OF THE CIRCLE: 250 250 100
```

Figure 4. Fig: Midpoint Circle Algorithm

### E. Lab 4: Midpoint Ellipse Algorithm

- **Objective:** To design n-bit (4-bit) sub tractor for unsigned integer binary numbers.
- **Theory:** The Mid-point Ellipse Algorithm is a technique used in computer graphics to draw ellipses on a digital screen. Unlike circles, ellipses have varying lengths of axes, and this algorithm calculates pixel positions symmetrically around the ellipse's center, adjusting these positions to render the elliptical shape. It's efficient as it computes points only within one quadrant of the ellipse and mirrors them to complete the shape. The algorithm starts with the center of the ellipse and calculates pixel positions symmetrically in four quadrants of the ellipse. By considering the properties of ellipses, it determines the best pixel choices to create a smooth curve. The concept of "midpoints" helps guide the algorithm's decision on whether to color the interior or exterior pixel. As the algorithm progresses through each quadrant, it incrementally calculates pixel positions that closely outline the ellipse. By mirroring these points in all quadrants, it generates a visually accurate ellipse on the screen.

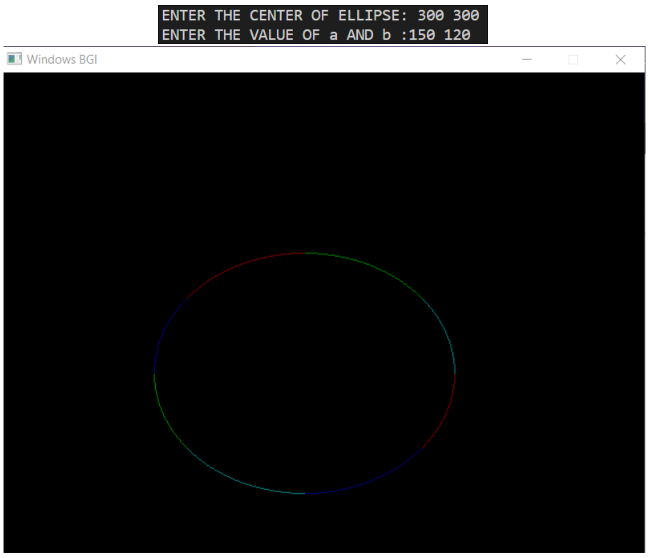
**Source code**

```
#include<iostream>
```

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

class midpointEllipseAlgo
{
    private:
    int h,k,a,b,i;
    float xk,yk,p1,p2;

    public:
    midpointEllipseAlgo()
    {
        getData();
        xk = 0;                     //2.
            initializing the first point
        yk = b;
    }

    void getData()
    {
        std::cout<<"ENTER THE CENTER OF ELLIPSE
            : ";
        std::cin>>h>>k;
        std::cout<<"ENTER THE VALUE OF a AND b
            :";
        std::cin>>a>>b;
    }

    //pvalue for region 1
    void regionI()
    {
        delay(50);
               p1 = b*b - b*a*a+ (a*a)/4;

     for(i=0;(2 *xk*b*b )<= (2*yk*a*a);i++){
        putpixel(xk+h,yk+k,1);
        putpixel(xk+h,-yk+k,2);
        putpixel(-xk+h,yk+k,3);
        putpixel(-xk+h,-yk+k,4);

            if(p1 <= 0)
            {
                xk += 1;
                p1 += (2 *xk *b* b)+(b*b);
            }
            else
            {
                xk += 1;
                yk -= 1;
                p1 += (2 *xk *b* b)+(b*b)-(2*yk*
                    a*a);
            }
     }
    }

    //pvalue for region 2
    void regionII()
    {
        delay(50);
         p2 = pow((xk + 0.5),2)*b*b+pow((yk-1)
             ,2)*a*a-a*a*b*b; //xk and yk are
             the last point of region I

        for(i=0;yk >= 0;i++){
        putpixel(xk+h,yk+k,4);
        putpixel(xk+h,-yk+k,3);
        putpixel(-xk+h,yk+k,2);
        putpixel(-xk+h,-yk+k,1);

            if(p2 <= 0)
            {
                xk += 1;

                yk -= 1;
                p2 += (2 *xk *b* b)+(a*a)-(2*yk*
                    a*a);
            }
            else
            {
                yk -= 1;
                p2 += (a*a)-(2*yk*a*a);
            }
        }
    }
    void makeEllipse()
    {

        regionI();


        regionII();
    }

};

int main(int argc, char const *argv[])
{
    int gd = DETECT, gm;
    initgraph(&gd,&gm,(char*)"");

    midpointEllipseAlgo EA;
    EA.makeEllipse();

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

- **Results and Analysis**

In the domain of computer graphics, the Midpoint Ellipse Algorithm, executed through C++ implementation, offered an adept technique for drawing ellipses. Leveraging incremental calculations and symmetrical properties, the algorithm skillfully generated ellipse shapes on a digital screen.



```
ENTER THE CENTER OF ELLIPSE: 300 300
ENTER THE VALUE OF a AND b :150 120
```

Figure 5. Fig: Midpoint Ellipse Algorithm

## F. Lab 5: Cohen Sutherland Line Clipping Algorithm

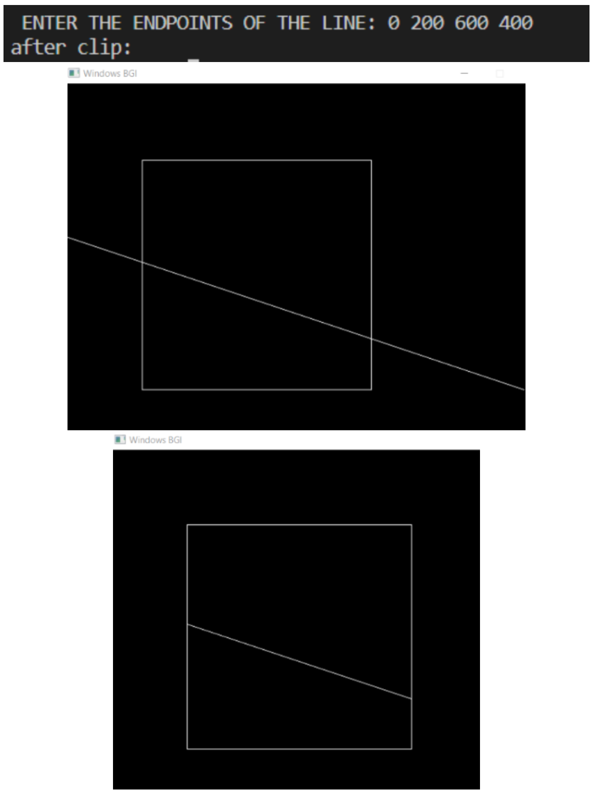
- **Objective:** To implement the Cohen Sutherland Line Clipping Algorithm
- **Theory:** The Cohen Sutherland Line Clipping Algorithm is a method in computer graphics used for efficiently clipping lines to a specific region or window. It divides the screen into regions using a binary code system and uses logical operations to determine if a line segment lies entirely inside, outside, or partially inside the region. This algorithm aids in drawing only the visible parts of lines within the specified region. The algorithm first assigns binary codes to the endpoints of the line segment based on their relative positions to the region. These codes indicate whether the endpoints are to the left, right, above, or below the region's boundaries. By analyzing these codes, the algorithm quickly identifies the position of the line with respect to the region. If both endpoints are within the region, the line is drawn as it is. If not, the algorithm uses logical operations to determine whether to clip the line and draw only the visible parts. This process ensures that only the relevant parts of the line are drawn, resulting in accurate and efficient line clipping.

**Source code**

```
//cohen sutherland line clipping algorithm
#include<iostream>
#include<cmath>
#include<graphics.h>
#include<dos.h>
#include<cstring>

class cohenSutherland{

private:
int xmin,ymin,xmax,ymax;
int inside,top,bottom,left,right,code1,code2;
int x1,y1,x2,y2;
float slope;

public:
cohenSutherland():xmin(100),ymin(100),xmax
     (400),ymax(400){

//TOP-BOTTOM-RIGHT-LEFT
top = 8;// {1,0,0,0};
bottom = 4; //{0,1,0,0};
left = 1;//{0,0,0,1};
right = 2; //{0,0,1,0};
inside = 0; //{0,0,0,0};

code1 = 0; //{0,0,0,0};
code2 = 0; //{0,0,0,0};
getLine();
drawWindow();
drawLine();
}

void getLine(){
    std::cout<<"\n ENTER THE ENDPOINTS OF THE
        LINE: ";
    std::cin>>x1>>y1>>x2>>y2;
    slope = (y2-y1)/(x2-x1);
}

int code(float x, float y){

    int c = inside;

    if(x<xmin)
    c |= left;
    else if(x> xmax)
    c |= right;
    if(y<ymin)
    c |= bottom;
    else if(y > ymax)
    c |= top;

    return c;
}


void trivialAcceptance(){

bool accept = false;
code1 = code(x1,y1);
code2 = code(x2,y2);
while(true){
    if((code1 == 0) && (code2 == 0)){
        accept = true;
        break;
    }
    else if(code1 & code2){
        //outside
        break;
    }
    else{

        int codeo;
        double x, y;
        //for first point

        if(code1 != 0)
        codeo = code1;
        else
        codeo = code2;

        if((codeo & top)){
            x = x1 + (x2 - x1) *(ymax - y1)/(y2
                 - y1);
             y = ymax;
        }
        else if((codeo & bottom)){

            x = x1 + (x2 - x1)*(ymin - y1)/(y2-
                y1);
            y = ymin;
        }
        else if((codeo & right)){

            y = y1 + (y2 - y1)* (xmax - x1) / (
                x2 - x1);
            x = xmax;
        }
        else if((codeo & left)){ //indicating
             left side

            y = y1+ (y2 - y1) * (xmin - x1) / (
                x2 - x1);
            x = xmin;
        }

        if(codeo == code1){
            x1 = x;
            y1 = y;
            code1 = code(x1,y1);
        }
        else{
            x2 = x;
            y2 = y;
            code2 = code(x2,y2);
        }

    }
}

if(accept){
    std::cout<<"after clip: ";
    delay(2000);
    clearviewport();
    drawWindow();
    drawLine();
}
else
{
    std::cout<<"the line is outside the
        boundary. ";
}
}

void drawWindow(){
    rectangle(xmin,ymin,xmax,ymax);
}
void drawLine(){
    line(x1,y1,x2,y2);
}

};

int main(int argc, const char* argv[]){
    int gd = DETECT,gm;
    initgraph(&gd,&gm,(char*)"");

    cohenSutherland cs;
    cs.trivialAcceptance();

    getch();

    closegraph();
    return 0;
}
```

- **Results and Analysis**

The Cohen-Sutherland Line Clipping Algorithm, realized using C++, presented a robust solution for line clipping in computer graphics. By categorizing regions and employing bitwise operations, the algorithm adeptly determined which portions of lines were visible on a digital screen. This method's efficacy in isolating visible segments of lines greatly enhanced the rendering of clipped lines in graphical contexts.



```
ENTER THE ENDPOINTS OF THE LINE: 0 200 600 400
after clip:
```

Figure 6. Cohen Sutherland Line Clipping Algorithm

## G. Lab 6: Liang Barsky Line Clipping Algorithm

- **Objective:** To implement Liang Barsky Line Clipping Algorithm

- **Theory:** The Liang Barsky Line Clipping Algorithm is used in computer graphics to efficiently clip lines against a given window or viewport. Unlike other techniques, Liang Barsky computes parameters for the intersections between the line and the window boundaries. Using these parameters, it determines if a line segment is completely within, partially within, or outside the window, and then clips and renders only the visible parts of the line. The algorithm calculates parameters that represent intersections between the line and the window boundaries. By analyzing these parameters, it determines whether the line segment is inside the window, outside, or partially inside. If the line is entirely within the window, it remains untouched. If it's partially inside, the algorithm calculates new endpoints for the visible portion. If it's entirely outside, the line is discarded. By utilizing these parameters, the algorithm efficiently decides which segments of the line should be clipped and drawn, ensuring accurate line clipping.

### Source code

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

#define TopLimit 400
#define BottomLimit 100
#define RightLimit 500
#define LeftLimit 200

void drawClippingWindow()
{
    line(LeftLimit, BottomLimit, RightLimit,
        BottomLimit);
    line(RightLimit, BottomLimit, RightLimit,
        TopLimit);
    line(RightLimit, TopLimit, LeftLimit,
        TopLimit);
    line(LeftLimit, TopLimit, LeftLimit,
        BottomLimit);
}

int clipTest(float numerator, float
    denominator, float *u1, float *u2)
{
    float t;
    int result = 1;
    t = denominator / numerator;

    if (numerator < 0.0)
    {
        if (t > *u2)
            result = 0;
        else if (t > *u1)
            *u1 = t;
    }
    else if (numerator > 0.0)
    {
        if (t < *u1)
            result = 0;
        else if (t < *u2)
            *u2 = t;
    }
    else
    {
        if (denominator < 0.0)
        {
            std::cout << "Line is parallel and
                outside the clipping window" <<
                 std::endl;
            result = 0;
        }
    }
    return (result);
}

void liangClipping(float x1, float y1, float
    x2, float y2)
```

```
{
    int flag = 0;
    float numerator[4], denominator[4];
    float u1 = 0;
    float u2 = 1;
    float dx = x2 - x1;
    float dy = y2 - y1;

    numerator[0] = -dx;
    numerator[1] = dx;
    numerator[2] = -dy;
    numerator[3] = dy;

    denominator[0] = x1 - LeftLimit;
    denominator[1] = RightLimit - x1;
    denominator[2] = y1 - BottomLimit;
    denominator[3] = TopLimit - y1;

    if (clipTest(numerator[0], denominator[0],
        &u1, &u2))
        if (clipTest(numerator[1], denominator
            [1], &u1, &u2))
            if (clipTest(numerator[2],
                denominator[2], &u1, &u2))
                if (clipTest(numerator[3],
                    denominator[3], &u1, &u2))
                {
                    if (u2 < 1.0)
                    {
                        x2 = x1 + u2 * dx;
                        y2 = y1 + u2 * dy;
                    }
                    if (u1 > 0.0)
                    {
                        x1 += u1 * dx;
                        y1 += u1 * dy;
                    }
                    flag = 1;
                    line(x1, y1, x2, y2);
                }
    if (flag == 0)
    {
        std::cout << "Line lies completely
            outside!" << std::endl;
    }
}

int main()
{
    float start_x, start_y, end_x, end_y;
    int gd = DETECT, gm;
    initgraph(&gd, &gm, (char *)"");
    start_x = 500;
    start_y = 400;
    end_x = 80;
    end_y = 80;
    drawClippingWindow();
    liangClipping(start_x, start_y, end_x,
        end_y);
    getch();
    closegraph();
    return 0;
}
```

- **Results and Analysis**

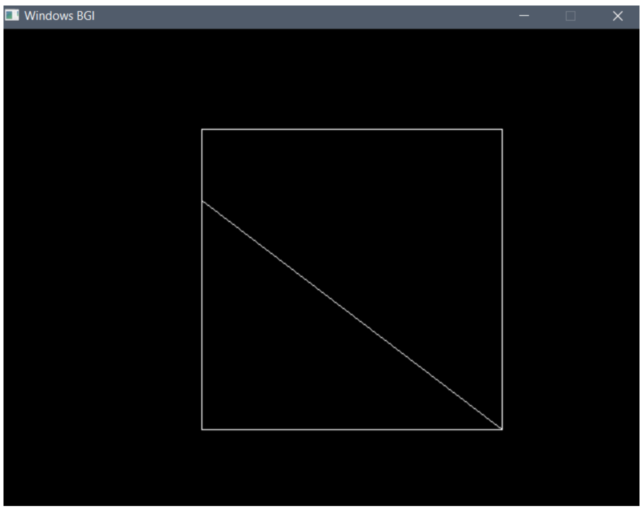


Figure 7. Fig: Liang Barsky Line Clipping Algorithm

The Liang-Barsky Line Clipping Algorithm, implemented via C++, provided a sophisticated approach to line clipping in computer graphics. Utilizing parameter calculations and logical operations, the algorithm skillfully identified visible portions of lines on a digital canvas. Its accuracy and versatility in handling various clipping scenarios greatly improved line rendering .

### H. Lab 7: 2D Transformations

- **Objective:** To implement 2D Transformations in computer graphics
- **Theory:** 2D Transformations are fundamental in computer graphics as they allow us to manipulate the positions, orientations, and sizes of 2D objects. These transformations include translation (shifting), rotation, and scaling operations. These operations modify the coordinates of object vertices to achieve the desired transformation effects. 2D Transformations apply mathematical formulas to object points to achieve changes. Translation shifts object points by adding fixed values to their coordinates. Rotation employs trigonometric functions to pivot points around a specified center. Scaling alters an object's size by multiplying coordinates. By repeatedly applying these

operations to all object points, we can transform the object's position, orientation, and size, resulting in a variety of visual effects and changes on the screen.

- **For Example:** The 2's complement of 0111 is 1001 and is obtained by leaving first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in the other three most significant bits. Let us see an example of subtraction using two binary numbers X = 10(1010) and Y = 9 (1001) and perform X-Y and Y-X.

### Source code

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

class transformation
{
    private:
    int i,j,k,n,prompt;
    int p[3][3],store[3][3];
    float cm[3][3],t[3][3];

    public:
    transformation():n(0),i(0),j(0),k(0)
    {
        //for 2d transformation creating 3 x 3
            matrix dynamically
        prompt = 0;
        for(i =0;i<3;i++){
            for(j =0;j<3;j++){
                if(i == j)
                    cm[i][j] = 1;
                else
                cm[i][j] = 0;
            }
        }

        line(300,0,300,500);
        line(0,300,600,300);
       // get the input points of a triangle
       std::cout<<"ENTER THE POINTS OF THE
           TRIANGLE IN MATRIX FORM: "<<std::
           endl;
        for(i =0;i<3;i++){
            for(j = 0;j<3;j++){
                    if(i == 2){
                        p[i][j] = 1;
                    }
                    else
                    std::cin>>p[i][j];
                }
            }
    for(i=0;i<3;i++){
        for(j=0;j<3;j++){
            store[i][j] = p[i][j];
        }
    }
    makeTriangle();
}

void makeTriangle()
{
    line(p[0][0],p[1][0],p[0][1],p[1][1]);
    line(p[0][0],p[1][0],p[0][2],p[1][2]);
    line(p[0][1],p[1][1],p[0][2],p[1][2]);
}

void chooseOperation()
{
    system("cls");
    char ch = 'y';
    while(ch == 'y'){
    std::cout<<"\n 1 for TRANSLATION \n 2
        FOR SCALING ABOUT ORIGIN \n 3 FOR
        SCALING ABOUT FIXED POINT \n 4 FOR
        ROTATION ABOUT ORIGIN \n 5 FOR
        ROTATING ABOUT A FIXED POINT\n 6
        FOR RELECTION ABOUT X-AXIS \n 7 FOR
         REFLECTION ABOUT Y-AXIS \n ";//8
        FOR REFLECTION ABOUT Y = MX +C \n 9
         FOR SHARING(XY)
    std::cin>>prompt;
    switch(prompt){
        case 1:{
            translation();
            transform();
            break;
        }
        case 2:{
            scalingAboutOrigin();
            transform();
            break;
        }
        case 3:{
            fixedPointScaling();
            transform();
            break;
        }
        case 4:{
            rotationAboutOrigin();
            transform();
            break;
        }
        case 5:{
            fixedPointRotation();
            transform();
            break;
        }
        case 6:{
            reflectionxy(1);
            break;
        }
        case 7:{
            reflectionxy(2);
            break;
        }
        case 8:{
            clearviewport();
            line(300,0,300,500);
            line(0,300,600,300);

    p[0][1] = 400;
    p[0][2] = 500;
    p[1][0] = 200;
    p[1][1] = 200;
    p[1][2] = 100;
    p[2][0] = 1;
    p[2][1] = 1;
    p[2][2] = 1;
    makeTriangle();
    delay(100);
            reflectiont(1);
            line(100,500,500,100);
            reflectiont(2);
            line(500,500,100,100);
            break;
        }
        default:{
        std::cout<<"choose Operation error"
             ;
        break;
        }
    }
    //matrix multiplication cm * p;
    //transform();
    makeTriangle();
    clearall();
    std::cout<<"\n Would you like to try
         next? 'y': ";
    std::cin>>ch;
    }
}

void clearall(){
    for(i =0;i<3;i++){
        for(j =0;j<3;j++){
            if(i == j){
                cm[i][j] = 1;
                t[i][j] = 1;
            }
            else{
            cm[i][j] = 0;
            t[i][j] = 0;
            }
        }
    }
            for(i=0;i<3;i++){
        for(j=0;j<3;j++){
            p[i][j] = store[i][j];
        }
    }
}
void reflectiont(int slope = 1){
    if(slope == 1){
        p[0][0] = 400;
        p[1][0] = 100;
        makeTriangle();
    }
    else{
        p[0][0] = 200;
        p[1][0] = 500;
        p[0][1] = 200;
        p[1][1] = 400;
        p[0][2] = 100;
        p[1][2] = 500;
        p[2][0] = 1;
        p[2][1] = 1;
        p[2][2] =1;
    }
}

void reflection(int m = 1,int c = 0){

    //y = mx+3
     for(i=0;i<3;i++){
        for(j=0;j<3;j++){
            cm[i][j] = p[i][j];
        }
    }
    float angle = atan(m);
    float temp[3][3];

    //translation(300,300-c)
    for(i=0;i<3;i++){
        for(j=0;j<3;j++){
            if(i == j)
            t[i][j] = 1;
            else
            t[i][j] = 0;
        }
        t[0][2] = 0;
        t[1][2] = -c;
    }
    matMul(t,cm);
    std::cout<<"translation ";
    //rotation(-theta)
            for(i=0;i<3;i++){
        for(j=0;j<3;j++){
            if(i == j)
            t[i][j] = 1;
            else
            t[i][j] = 0;
        }}
    t[0][0] *= cos(-angle);
    t[0][1] -= sin(-angle);
    t[1][0] += sin(-angle);
    t[1][1] *= t[0][0];

    matMul(t,cm);
    std::cout<<"rotation ";

    //reflection about x-axis
                    for(i=0;i<3;i++){
        for(j=0;j<3;j++){
            if(i == j)
            t[i][j] = 1;
            else
            t[i][j] = 0;
        }}
    for(i=0;i<3;i++){
        for(j =0;j<3;j++){
            temp[i][j] = 0;
            //X-axis
                    if(i == 1)
                    t[1][1] = (2*300/cm[i][j
                        ])-1;

            for(k =0;k<3;k++){
                //std::cout<<i<<j<<k;
                temp[i][j] += t[i][k]*cm[k][
                    j];
            }
        }
    }
    for(i=0;i<3;i++){
        for(j =0;j<3;j++){
                cm[i][j] = temp[i][j];
        }
    }
    std::cout<<"reflection ";

    //rotation(+)
    for(i=0;i<3;i++){
        for(j=0;j<3;j++){
            if(i == j)
            t[i][j] = 1;
            else
            t[i][j] = 0;
        }
    }
    t[0][0] *= cos(angle);
    t[0][1] -= sin(angle);
    t[1][0] += sin(angle);
    t[1][1] *= t[0][0];
    matMul(t,cm);
    std::cout<<"rotation ";

    //t(0,c)
    for(i=0;i<3;i++){
        for(j=0;j<3;j++){
            if(i == j)
            t[i][j] = 1;
            else
            t[i][j] = 0;
        }
        t[0][2] = 300;
        t[1][2] = 300+c;
    }
    matMul(t,cm);
    std::cout<<"translation ";

         for(i=0;i<3;i++){
        for(j=0;j<3;j++){
            p[i][j] = cm[i][j];
            std::cout<<p[i][j];
        }
    }
}

void translation(){
    std::cout<<"enter the translation
        vector: ";
    //get the translation vector
    std::cin>>cm[0][2]>>cm[1][2];
}

void scaling(){
    std::cout<<"ENTER THE SCALING VECTOR: "
        ;
    std::cin>>cm[0][0]>> cm[1][1] ;
}

void scalingAboutOrigin(){
     for(i=0;i<3;i++){
        for(j=0;j<3;j++){
            if(i == j)
            t[i][j] = 1;
            else
            t[i][j] = 0;
        }
        t[0][2] = 300;
        t[1][2] = 300;
    }
    scaling();
    matMul(t,cm);

        t[0][2] = -p[0][0];
        t[1][2] = -p[1][0];
    matMul(cm,t);
}
void tvec(int check=0){

    for(i=0;i<3;i++){
        for(j=0;j<3;j++){
            if(i == j)
            t[i][j] = 1;
            else
            t[i][j] = 0;
        }
    }
    if(check == 1){
        t[0][2] = -p[0][0];
        t[1][2] = -p[1][0];
    }
    else{
        t[0][2] = p[0][0];
        t[1][2] = p[1][0];
    }
}

void fixedPointScaling(){
    tvec();
    scaling();
    matMul(t,cm);
    tvec(1);
    matMul(cm,t);
}
```

```
void rotationAboutOrigin(){
     for(i=0;i<3;i++){
        for(j=0;j<3;j++){
            if(i == j)
            t[i][j] = 1;
            else
            t[i][j] = 0;
        }
        t[0][2] = 300;
        t[1][2] = 300;
    }
    rotation();
    matMul(t,cm);

        t[0][2] = -p[0][0];
        t[1][2] = -p[1][0];
    matMul(cm,t);
}
void rotation(){
    float angle;
    std::cout<<"\n ENTER THE ANGLE FOR THE
        ROTATION:";
    std::cin>>angle;
    cm[0][0] *= cos(3.1415 * (angle/180.0))
        ;
    cm[0][1] -= sin(3.1415 * (angle/180.0))
        ;
    cm[1][0] += sin(3.1415 * (angle/180.0))
        ;
    cm[1][1] *= cm[0][0];

}

void fixedPointRotation(){
    tvec();
    rotation();
    matMul(t,cm);

    tvec(1);
   matMul(cm,t);
}

void reflectionxy(int check = 0){
    std::cout<<"transform \n";
            for(i=0;i<3;i++){
        for(j =0;j<3;j++){

    std::cout<<"\n"<<p[i][j];
        }
            }
    float temp[3][3];
    for(i=0;i<3;i++){
        for(j =0;j<3;j++){
            temp[i][j] = 0;
                if(check == 1){ //X-axis
                    if(i == 1)
                    cm[1][1] = (2*300/p[i][j
                        ])-1;
                }
                else {
                    if(i == 0)
                    cm[0][0] = (float(2*300)/
                        p[i][j])-1;
                }

            for(k =0;k<3;k++){
                //std::cout<<i<<j<<k;
                temp[i][j] += cm[i][k]*p[k][
                    j];
            }
        }
    }
    for(i=0;i<3;i++){
        for(j =0;j<3;j++){
                p[i][j] = int(temp[i][j]);

    std::cout<<"\n"<<p[i][j];
        }
    }

    std::cout<<"transform sucess \n";
}

void transform(){
    std::cout<<"transform \n";
    float temp[3][3];
    for(i=0;i<3;i++){
        for(j =0;j<3;j++){
            temp[i][j] = 0;
            for(k =0;k<3;k++){
                //std::cout<<i<<j<<k;
                temp[i][j] += cm[i][k]*p[k][
                    j];
            }
        }
    }
    for(i=0;i<3;i++){
        for(j =0;j<3;j++){
                p[i][j] = int(temp[i][j]);

    std::cout<<"\n"<<p[i][j];
        }
    }

    std::cout<<"transform sucess \n";
}

void matMul(float c[][3],float t[][3]){
    float temp[3][3];
    for(i=0;i<3;i++){
        for(j =0;j<3;j++){
            temp[i][j] = 0;
            for(k =0;k<3;k++){
                //std::cout<<i<<j<<k;
                temp[i][j] += c[i][k]*t[k][j
                    ];
            }
        }
    }
    for(i=0;i<3;i++){
        for(j =0;j<3;j++){
```

```
                    cm[i][j] = temp[i][j];
                }
            }
        }
};
int main(int argc, char const *argv[])
{
    int gd = DETECT, gm;
    initgraph(&gd,&gm,(char*)"");
            transformation tr;
            tr.chooseOperation();

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

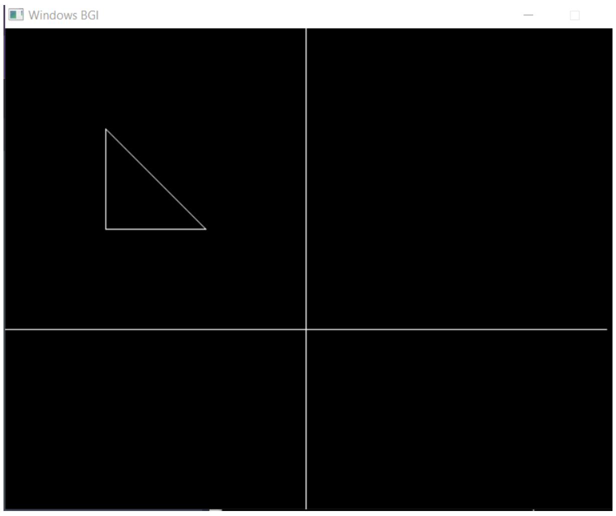
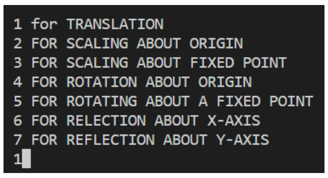


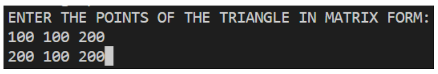
Figure 10. draw triangle

- **Results and Analysis**



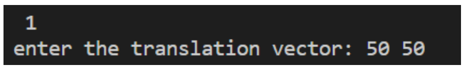
```
1 for TRANSLATION
2 FOR SCALING ABOUT ORIGIN
3 FOR SCALING ABOUT FIXED POINT
4 FOR ROTATION ABOUT ORIGIN
5 FOR ROTATING ABOUT A FIXED POINT
6 FOR RELECTION ABOUT X-AXIS
7 FOR REFLECTION ABOUT Y-AXIS
1
```

Figure 11. Chose operation



```
ENTER THE POINTS OF THE TRIANGLE IN MATRIX FORM:
100 100 200
200 100 200
```

Figure 8. define triangle



```
 1
enter the translation vector: 50 50
```

Figure 12. choose translation vector

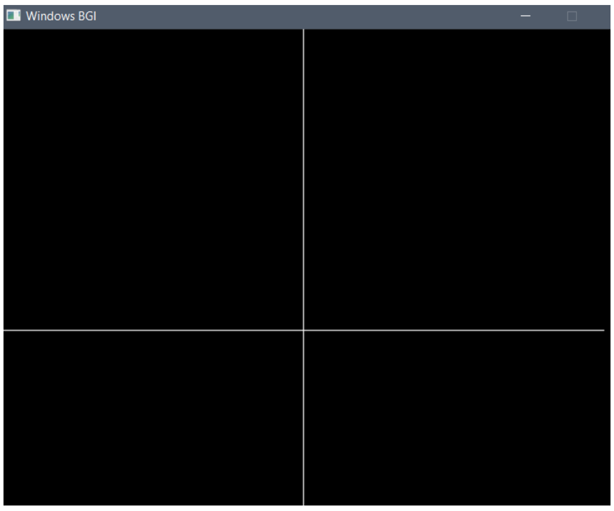


Figure 9. Drawing Axes

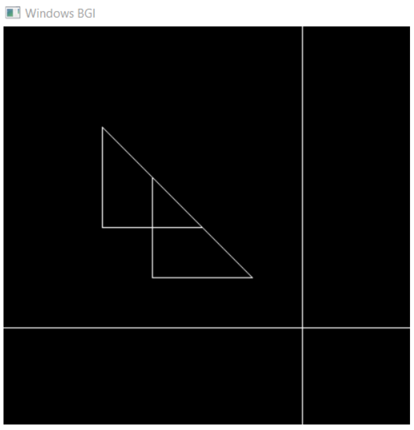
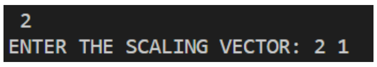


Figure 13. Translate the triangle



```
 2
ENTER THE SCALING VECTOR: 2 1
```

Figure 14. choose scaling vector

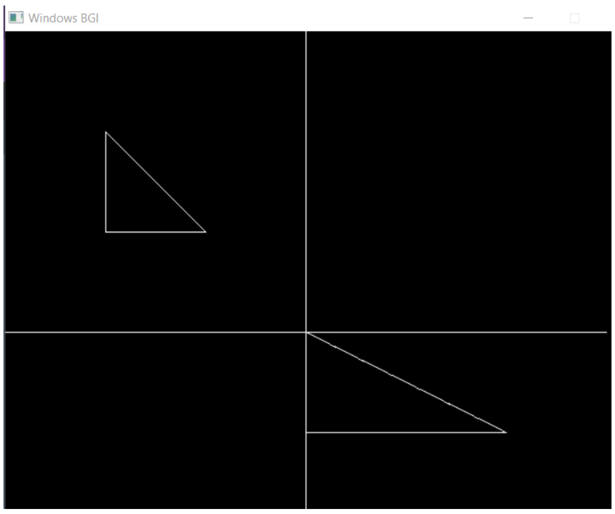
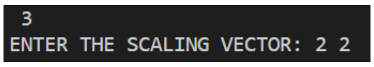


Figure 15. scaling about origin



```
 3
ENTER THE SCALING VECTOR: 2 2
```

Figure 16. choose scaling vector

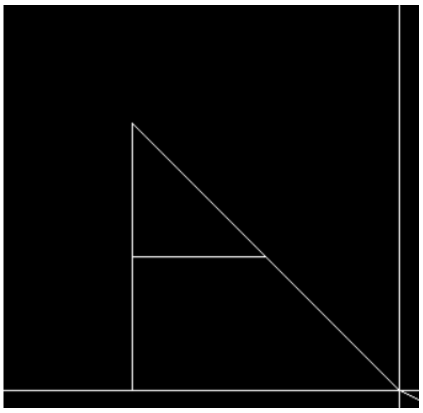
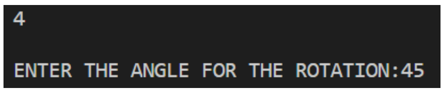


Figure 17. scaling about a fixed point



```
4

ENTER THE ANGLE FOR THE ROTATION:45
```

Figure 18. choose angle

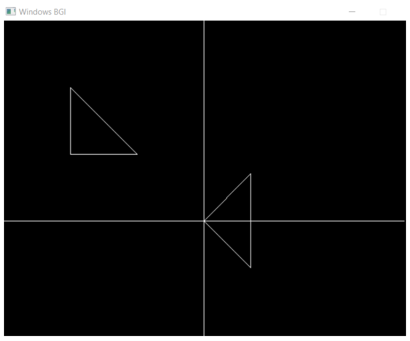
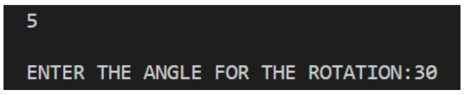


Figure 19. rotation about origin



```
5

ENTER THE ANGLE FOR THE ROTATION:30
```

Figure 20. chose angle

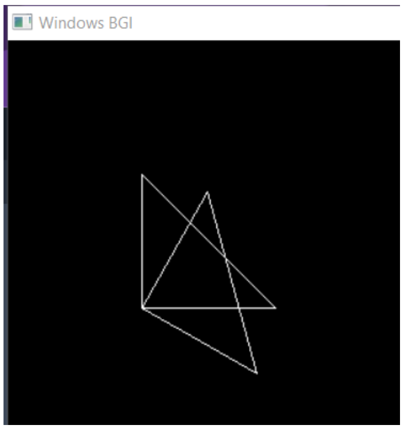


Figure 21. rotation about fixed point

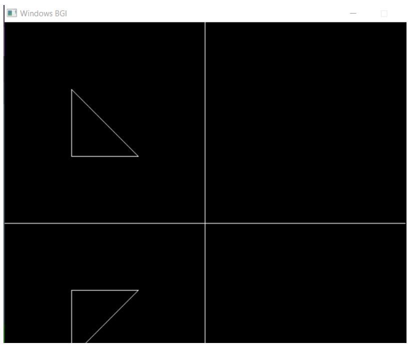


Figure 22. X-Axis Reflection

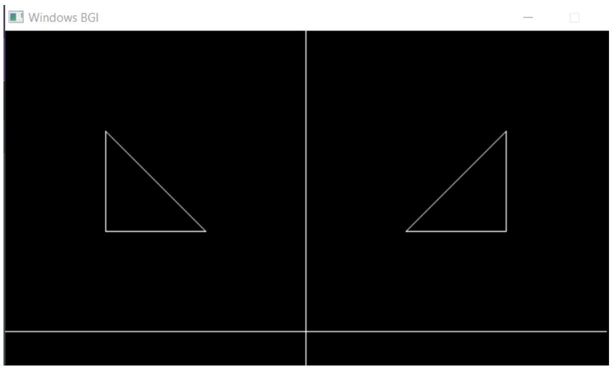


Figure 23. Y-Axis Reflection

Fig: 2D Transformations

In the realm of computer graphics, the implementation of 2D Transformations using C++ brought about a transformative approach. Through translation, rotation, and scaling operations, this algorithm skillfully manipulated geometric triangle on a digital screen.

## I. Lab 8: Bezier Curve

- **Objective:** To implement Bezier Curve Algorithm
- **Theory:** Bezier Curves are fundamental constructs in computer graphics for creating smooth and visually pleasing curves. These curves are defined by control points that guide their shape. With varying degrees such as quadratic (3 control points) and cubic (4 control points), Bezier Curves offer precise control over curvature, making them versatile tools for generating curves in graphics applications. Bezier Curves employ control points to define curve shapes. In a quadratic Bezier Curve, the first and last control points establish endpoints, while the middle point influences shape. For a cubic Bezier Curve, all four points collectively shape the curve. By adjusting control point positions, a wide range of curves can be created, from gentle arcs to intricate forms.
- **For Example:**

### Source code

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

class CustomPoint {
public:
    int xPos, yPos;

    CustomPoint(int _x = 0, int _y = 0) : xPos(
        _x), yPos(_y) {}
};

int customBinomialCoeff(int n, int k) {
    if (k == 0 || k == n)
        return 1;
    return customBinomialCoeff(n - 1, k - 1) +
        customBinomialCoeff(n - 1, k);
}

CustomPoint calculateCustomBezierPoint(
    CustomPoint p[], double t, int n) {
    double xVal = 0, yVal = 0;
    int customBinCoeff;

    for (int i = 0; i < n; i++) {
        customBinCoeff = customBinomialCoeff(n
            - 1, i);
        xVal += customBinCoeff * p[i].xPos *
            pow(1 - t, n - 1 - i) * pow(t, i);
        yVal += customBinCoeff * p[i].yPos *
            pow(1 - t, n - 1 - i) * pow(t, i);
    }

    return CustomPoint(static_cast<int>(xVal),
        static_cast<int>(yVal));
}

int main() {
    int graphicsDriver = DETECT, graphicsMode;
    initgraph(&graphicsDriver, &graphicsMode, (
        char*)"");

    int numControlPoints = 4;
    CustomPoint controlPoints[numControlPoints]
        = {{150, 400}, {300, 100}, {450,
        500}, {600, 200}};

    for (int i = 0; i < numControlPoints; i++)
        {
        putpixel(controlPoints[i].xPos,
            controlPoints[i].yPos, WHITE);
        if (i < numControlPoints - 1) {
            line(controlPoints[i].xPos,
                controlPoints[i].yPos,
                controlPoints[i + 1].xPos,
                controlPoints[i + 1].yPos);
        }
    }

    for (double t = 0; t <= 1; t += 0.001) {
        CustomPoint curvePoint =
            calculateCustomBezierPoint(
            controlPoints, t, numControlPoints)
            ;
        putpixel(curvePoint.xPos, curvePoint.
            yPos, GREEN);
    }

    delay(10000);
    closegraph();
    return 0;
}
```

- **Results and Analysis**

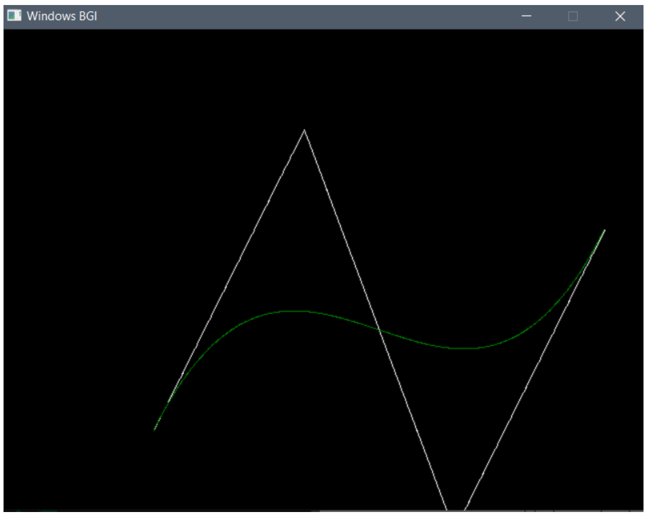


Figure 24. Fig: Midpoint Ellipse Algorithm

The application of Bezier Curves, facilitated by C++ implementation, introduced an elegant method for drawing curves in computer graphics. By controlling control points and interpolation, this algorithm expertly generated smooth and visually appealing curves on a digital window.

### J. Lab 9: Flood Filling

- **Objective:** To implement the Flood Filling Algorithm
- **Theory:** Flood Filling is a technique utilized in computer graphics to fill enclosed regions with a specified color. It commences at a seed point and progressively colors adjacent pixels sharing the same initial color until the entire region is filled. This algorithm is efficient and commonly applied for coloring areas in applications like paint programs and graphics editing software. Flood Filling starts by choosing a starting pixel within the target area. It examines the color of this pixel and compares it to the desired fill color. If they match, the process moves to adjacent pixels (up, down, left, right), evaluating and filling them if needed. By iteratively extending to linked pixels, the algorithm methodically colors the enclosed region with the designated color. Ensuring appropriate boundary conditions and stopping criteria is essential to prevent the algorithm from spreading indefinitely or exceeding the intended region.

#### Source code

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

void floodFill(int x, int y, int fillColor,
    int oldColor) {
    if (x < 0 || x >= getmaxx() || y < 0 || y
        >= getmaxy() || getpixel(x, y) !=
        oldColor)
        return;

    putpixel(x, y, fillColor);

    floodFill(x + 1, y, fillColor, oldColor);
        // Right
    floodFill(x - 1, y, fillColor, oldColor);
        // Left
    floodFill(x, y + 1, fillColor, oldColor);
        // Down
    floodFill(x, y - 1, fillColor, oldColor);
        // Up
}

int main() {
    int gd = DETECT, gm;
    initgraph(&gd, &gm, (char*)"");

    rectangle(200, 200, 350, 350);
    floodFill(250, 250, RED, BLACK);

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

- **Results and Analysis**

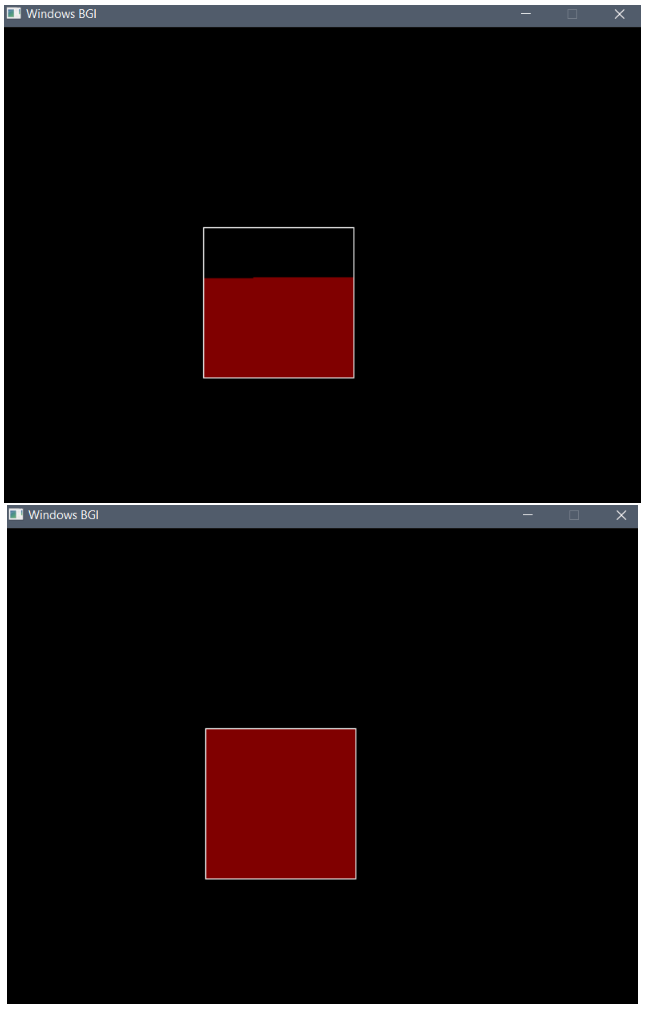


Figure 25. Flood Filling Algorithm

Flood Filling, implemented using C++, emerged as a powerful technique for coloring closed regions in computer graphics. By iteratively filling adjacent pixels based on a specified color criterion, the algorithm effectively colored enclosed areas on a digital screen. Its precision and ability to fill complex shapes greatly contributed to vibrant and detailed visuals.

## II. DISCUSSION

Our journey through the computer graphics lab was a fascinating fusion of creativity and education. Exploring an array of techniques, we uncovered the artistry of visual wonders on screens. From the foundational DDA and Bresenham's Algorithms for efficient line drawing to the intricate Mid-Point Circle and Ellipse Algorithms producing flawless curves, mathematics revealed its magic in crafting captivating visuals. Our discoveries extended to line clipping with Cohen-Sutherland and Liang-Barsky Algorithms, ensuring precise fitting of creations on screens. The realm of 2D transformations offered effortless object manipulation, and Bezier Curves introduced us to crafting elegant curves. Our exploration culminated in the vibrant art of Flood Filling, illuminating the path to filling shapes with a burst of color. With newfound skills, we depart equipped to infuse screens with our creative imagination, armed with the tools to etch our digital artistry into the tapestry of technology.

## III. CONCLUSION

Hence through hands-on exploration, we bridged the gap between theoretical concepts and practical implementation. Starting with the fundamentals of line drawing, we progressed to crafting a comprehensive computer graphics project, a journey that encapsulated the essence of our learning. The lab sessions served as a crucible where we honed our skills and translated theoretical knowledge into practical dexterity. Our progression through the algorithms and the project work exemplified the seamless transition from classroom theory to tangible accomplishments in the realm of computer graphics.

## IV. REFERENCES

- Hearn, D., & Baker, M. P. (1998). Computer Graphics (2nd ed.).
- Maharjan, M. (2023). Assignments/Graphics Repository. Retrieved from My GitHub