Lab Manual

For

CSE 3102 & CSE 422 (Computer Graphics Lab)

Department of Computer Science and Engineering

Varendra University

Rajshahi, Bangladesh

INDEX

Lab Instructions:

➢ Students should come with thorough preparation for the experiment to be conducted.

➢ Students will not be permitted to attend the laboratory unless they bring the practical record fully completed in all respects pertaining to the experiment conducted in the previous class.

➢ Be honest in developing and representing your program. If a particular program output appears wrong repeat the program carefully.

➢ Strictly observe the instructions given by the Faculty / Lab. Instructor.

➢ Take permission before entering in the lab and keep your belongings in the racks.

➢ NO FOOD, DRINK, IN ANY FORM is allowed in the lab.

➢ TURN OFF CELL PHONES! If you need to use it, please keep it in bags.

➢ Avoid all horseplay in the laboratory. Do not misbehave in the computer laboratory. Work quietly.

➢ Save often and keep your files organized

➢ Do not reconfigure the cabling/equipment without prior permission.

➢ Do not play games on systems.

➢ If you finish early, spend the remaining time to complete the laboratory report writing. Come equipped with calculator and other materials related to lab works.

➢ Handle instruments with care. Report any breakage or faulty equipment to the instructor. Shutdown your computer you have used for the purpose of your experiment before leaving the Laboratory.

➢ Violation of the above rules and etiquette guidelines will result in disciplinary action.

Course Syllabus

1	Faculty	Faculty of Science and Engineering
2	Department	Department of CSE
3	Program	B. Sc in Computer Science and Engineering
4	Name of the Course	Computer Graphics Lab
5	Course Code	CSE 3102, CSE 422
6	Bi-semester	
7	Co-requisites	
8	Status	
9	Credit Hours	
10	Section	
11	Class Hours	
12	Class Location	
13	Name(s) of the Academic staff/ Instructor(s)	
14	Contact	
15	Office	
16	Counseling Hours	
17	Text Books	1. **Computer Graphics,** *Prentice Hall*
18	Reference	1. **Computer Graphics: A Programming Approach,** *McGraw-Hill College* 2. **Fundamentals of Computer Graphics,** *Prentice Hall*
19	Equipments and Aids	1. Lab Sheet 2. Text Books 3. Code::Blocks 4. Glut Setup
20	Course Description	This course introduces the foundational concepts and practical skills of computer graphics through hands-on programming and visualization. Students will learn to create basic and complex geometric shapes, apply various transformations, and implement animations using OpenGL. Laboratory tasks are designed to gradually develop proficiency in graphical algorithms and OpenGL functionalities, ultimately enabling students to build interactive visual applications, animations, and simple games.

22	Course Objectives	The course is designed to provide the background of the following topics: 1. To introduce students to the basic concepts of computer graphics, including the use of OpenGL and graphical libraries for creating and manipulating 2D shapes. 2. To develop student skills in implementing standard graphics algorithms such as line, circle, and polygon drawing, clipping, and transformation techniques. 3. To enable students to apply graphical programming knowledge in designing and animating objects, ultimately building interactive visual systems and basic animation projects.
23	Course Outcomes	After the successful completion of this course, students will be able to 1. Utilize graphical libraries such as OpenGL to draw and manipulate primitive shapes using built-in and user-defined functions. 2. Implement and test fundamental shape-drawing and clipping algorithms for 2D graphical objects. 3. Design and develop animated visual scenes by programming arbitrary object movements and transformations on a 2D plane.
24	Teaching Methods	Lecture, Presentation, Problem solving
25	**Topic Outline**	

	Class	Topics	COs	Reading reference	Activities
	1-2	Introduction to Coordinate System	CLO1		Understanding OpenGL window, setting up 2D coordinate space
	3	Point Drawing	CLO1		Using basic OpenGL functions to draw individual points
	4-5	Line Drawing Algorithms (Basic)	CLO1		Implementing horizontal, vertical, and diagonal line drawing algorithms

	6	Line Drawing Using DDA Algorithm	CLO1		Writing and visualizing lines using DDA algorithm
	7	Line Drawing Using Bresenham's Algorithm	CLO1		Implementing Bresenham's algorithm and comparing with DDA
	8	Circle Drawing Using Midpoint Circle Algorithm	CLO1		Applying midpoint algorithm for drawing symmetric circles
	9	Mid Lab Test	CLO1		Assessment of understanding on point, line, and circle drawing techniques
	10-11	2D Geometric Transformations	CLO3		Performing translation, rotation, and scaling on 2D shapes
	12-13	Cohen-Sutherland Line Clipping	CLO2		Clipping lines within a rectangular window using Cohen-Sutherland algorithm
	14-15	Sutherland-Hodgman Polygon Clipping	CLO2		Implementing polygon clipping against a window boundary
	16	Final Lab Test	CLO2 CLO3		Comprehensive evaluation on all algorithms and transformations covered

26	Assessment Methods	

Assessment Types	**Marks**
Attendance	10%
Experiment	10%
Laboratory Viva-voce	10%
Laboratory Report	20%
Final Lab Viva	10%
Lab Final	15%
Lab Quiz	25%
Total	100%

<table>
<tr><td></td><td></td><td></td></tr>
<tr><td>27</td><td>Grading Policy</td><td>
<table>
<tr><th>Letter Grade</th><th>Marks %</th><th>Grade Point</th><th>Letter Grade</th><th>Marks %</th><th>Grade Point</th></tr>
<tr><td>A+ (Plus)</td><td>80-100</td><td>4.00</td><td>C+ (Plus)</td><td>50-54</td><td>2.50</td></tr>
<tr><td>A (Plain)</td><td>75-79</td><td>3.75</td><td>C (Plain)</td><td>45-49</td><td>2.25</td></tr>
<tr><td>A- (Minus)</td><td>70-74</td><td>3.50</td><td>D (Plain)</td><td>40-44</td><td>2.00</td></tr>
<tr><td>B+ (Plus)</td><td>65-69</td><td>3.25</td><td>F (Fail)</td><td><40</td><td>0.00</td></tr>
<tr><td>B (Plain)</td><td>60-64</td><td>3.00</td><td>I*</td><td>-</td><td>Incomplete</td></tr>
<tr><td>B- (Minus)</td><td>55-59</td><td>2.75</td><td>W*</td><td>-</td><td>Withdrawal</td></tr>
</table>
</td></tr>
<tr><td>28</td><td>Additional Course Policies</td><td>1.1 . Lab Reports
Reports on previous Experiment must be submitted before the beginning of new experiment. A bonus mark may be obtained if a sutdents submits a neat, clean and complete lab report.

2.2 Examination
There will be a lab exam at the end of the semester that will be a closed book exam.

3.3. Unfair means policy
In case of copying/ plagiarism in any of the assessments, the students involved will receive zero marks. Zero Tolarance will be shown in this regard. In case of severel offences, actions will be taken as per university rule.

4.4. Counseling
Students are expected to follow the counselling hours posted. In case of emergency/ unavoidable situation, students can e-mail the respective teachers for an appoinment.

5.5. Policy for Absence in Class/ Exam
If a student is absent in the class for anything other than medical reasons, he/she will not receive attedance. If a student misses a class for genuine medical reasons, he/she must apply with the supporting documents . He/she will then have to follow the instructions given by the instructor for make-up.

In case of absence in the mid/ final exam for medical grounds, the student must also get his/ her application forwarded by the head of the department before a make-up exam can be taken.

It is recommended that the student inform the instructor beforehand through mail if they feel that they will miss a class / evaluation due to medical reasons.</td></tr>
</table>

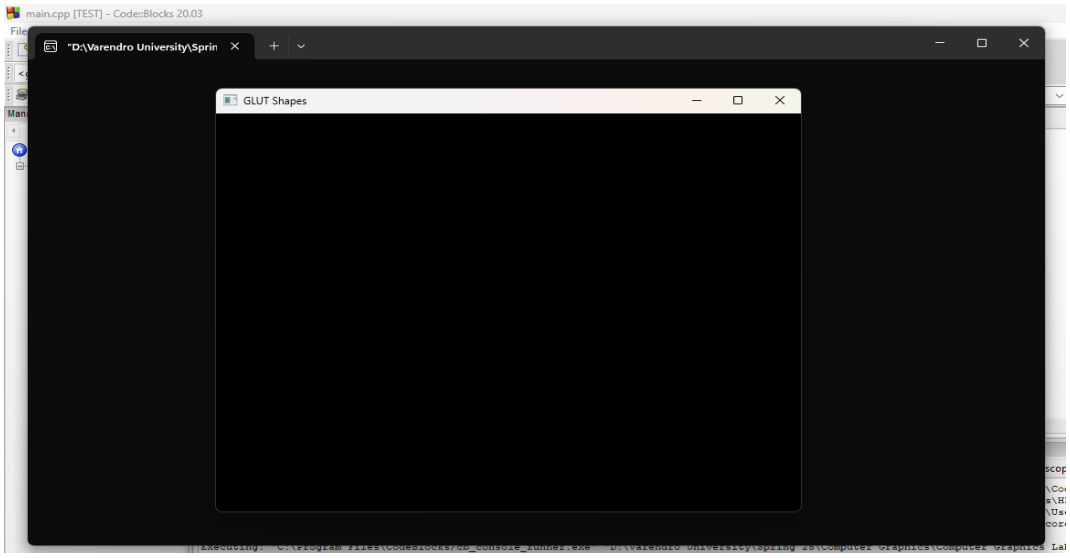
Lab-Introduction

Introduction:
This basic OpenGL program sets up a window using the GLUT (OpenGL Utility Toolkit) library. It initializes the display mode, window size, and position, and creates a window titled "GLUT Shapes". Although the display function is currently empty, it serves as a placeholder where graphical content can later be rendered. This code provides the foundational structure for building OpenGL applications in C++.

Code:

```
#include<windows.h>
#include <GL/glut.h>

void display()
{
    // Empty display function
}

int main(int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutInitWindowSize(640,480);
    glutInitWindowPosition(10,10);
    glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE | GLUT_DEPTH);

    glutCreateWindow("GLUT Shapes");
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Output:



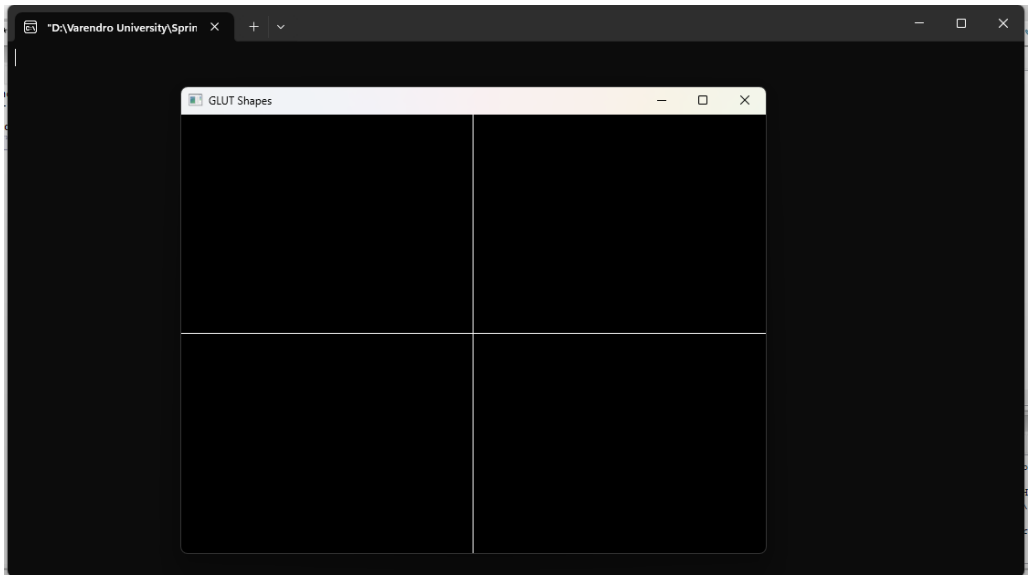
Lab – 1 Drawing the co-ordinates.

Introduction:
This OpenGL program uses GLUT to create a simple 2D coordinate system by drawing horizontal and vertical axes through the origin. It sets up a basic window and uses glVertex2f() within glBegin(GL_LINES) to define the lines. The glFlush() function ensures that the rendered output appears on the screen. This program serves as a foundation for learning graphical coordinate systems in OpenGL.

Code:

```
#include <windows.h>
#include <GL/glut.h>

void display() {
    glClear(GL_COLOR_BUFFER_BIT);

    glBegin(GL_LINES);
        glVertex2f(1.0, 0.0);
        glVertex2f(-1.0, 0.0);

        glVertex2f(0.0, 1.0);
        glVertex2f(0.0, -1.0);
    glEnd();

    glFlush();
}

int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(10, 10);
    glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE | GLUT_DEPTH);
    glutCreateWindow("GLUT Shapes");
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Output:



Lab – 2 Drawing the points.

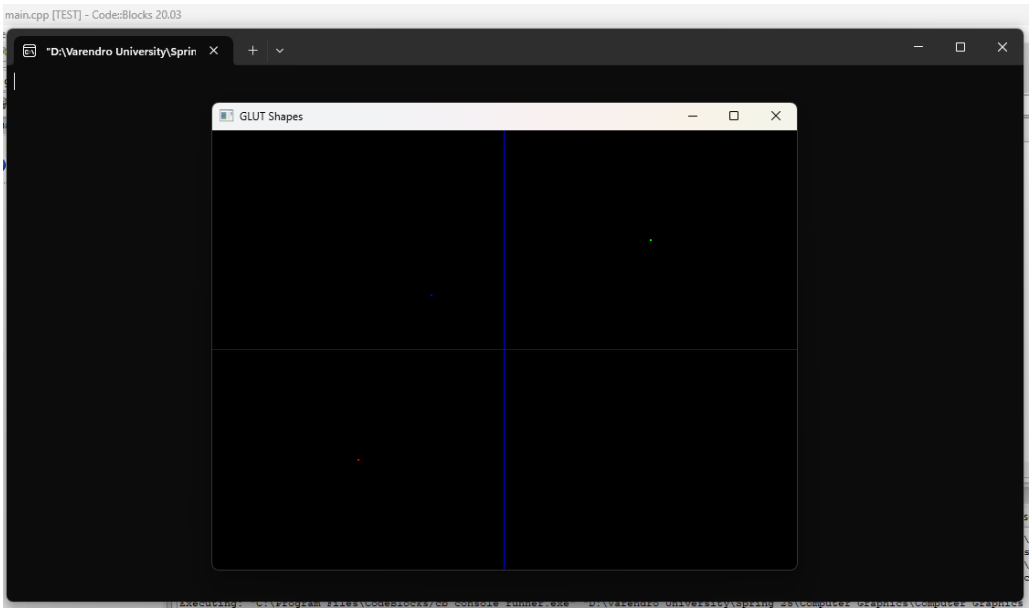
Introduction:
This OpenGL program demonstrates how to draw individual points with different colors using the GL_POINTS primitive. It also draws coordinate axes for reference using GL_LINES. The function glPointSize() sets the size of each point, and glColor3f() defines the color for each vertex. This basic exercise helps students understand the fundamentals of plotting colored points in a 2D coordinate space.

Code:

```
#include <windows.h>
#include <GL/glut.h>

void display() {
    glClear(GL_COLOR_BUFFER_BIT);

    glBegin(GL_LINES);
        glVertex2f(1.0, 0.0);
        glVertex2f(-1.0, 0.0);

        glVertex2f(0.0, 1.0);
        glVertex2f(0.0, -1.0);
    glEnd();

    glPointSize(2.0);
    glBegin(GL_POINTS);
        glColor3f(1.0, 0.0, 0.0);
        glVertex2f(-0.5, -0.5);

        glColor3f(0.0, 1.0, 0.0);
        glVertex2f(0.5, 0.5);

        glColor3f(0.0, 0.0, 1.0);
        glVertex2f(-0.25, 0.25);
    glEnd();

    glFlush();
}

int main(int argc, char *argv[]) {
```

```
    glutInit(&argc, argv);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(10, 10);
    glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE | GLUT_DEPTH);
    glutCreateWindow("GLUT Shapes");
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Output:



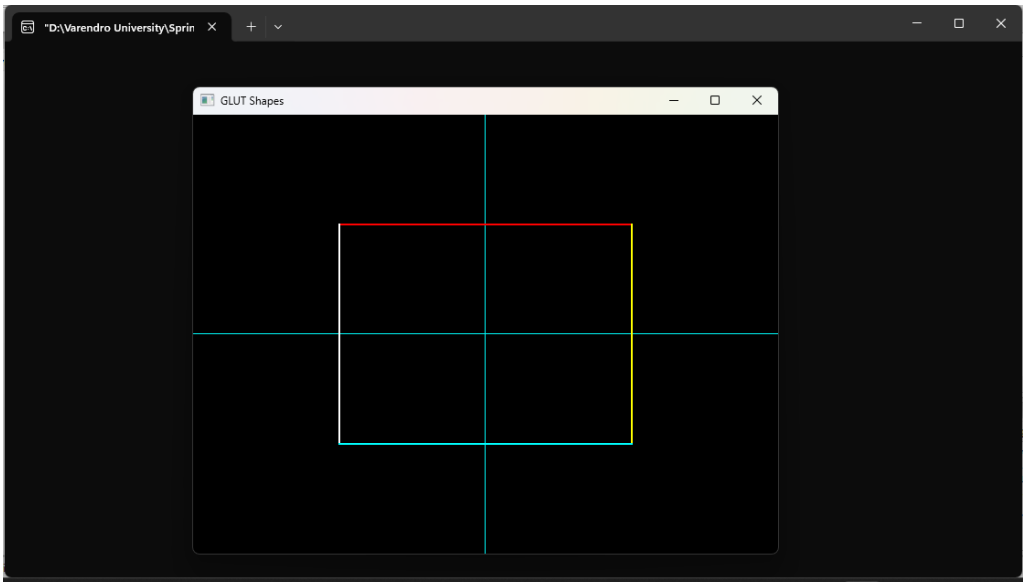
Lab – 3 Drawing lines using points 1) Vertical, 2) Horizontal – Lab Task

Introduction: This OpenGL program demonstrates how to draw horizontal and vertical lines using the GL_POINTS primitive instead of built-in line functions. The code draws multiple lines with different colors to visualize direction and placement. It also includes coordinate axes using GL_LINES for better orientation in the 2D space. This lab helps students understand how to manually simulate line drawing with point plotting.

Code:

```
#include <windows.h>
#include <GL/glut.h>

void display() {
    glClear(GL_COLOR_BUFFER_BIT);

    glBegin(GL_LINES);
        glVertex2f(1.0, 0.0);
        glVertex2f(-1.0, 0.0);

        glVertex2f(0.0, 1.0);
        glVertex2f(0.0, -1.0);
    glEnd();

    glPointSize(2.0);
    glBegin(GL_POINTS);

        glColor3f(1.0, 0.0, 0.0);
        for(float i = -0.5; i <= 0.5; i += 0.001) {
            glVertex2f(i, 0.5);  // Horizontal line (Red)
        }

        glColor3f(1.0, 1.0, 0.0);
        for(float i = -0.5; i <= 0.5; i += 0.001) {
            glVertex2f(0.5, i);  // Vertical line (Yellow)
        }

        glColor3f(1.0, 1.0, 1.0);
        for(float i = -0.5; i <= 0.5; i += 0.001) {
            glVertex2f(-0.5, i);  // Vertical line (White)
        }

        glColor3f(0.0, 1.0, 1.0);
        for(float i = -0.5; i <= 0.5; i += 0.001) {
            glVertex2f(i, -0.5);  // Horizontal line (Cyan)
        }

    glEnd();

    glFlush();
}

int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(10, 10);
    glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE | GLUT_DEPTH);
    glutCreateWindow("GLUT Shapes");
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Output:



Lab – 3 Drawing lines using points and user inputs 1) Diagonal m=1, 2) Diagonal m=-1 – Lab Task

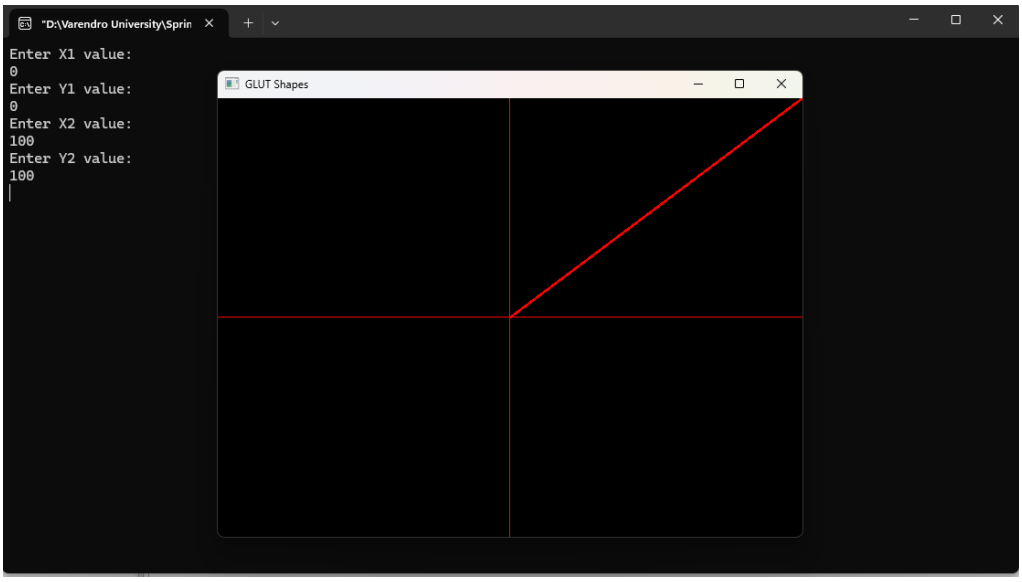
Introduction: This OpenGL program allows the user to input two coordinate points to draw diagonal lines using points. Based on the slope (either m = 1 or m = -1), the code plots pixels along the diagonal, simulating line drawing using manual point placement. It also displays coordinate axes for reference. This lab enhances understanding of slope-based line drawing logic and user-driven graphics programming.

Code:

```
#include <iostream>
#include <windows.h>
#include <GL/glut.h>
using namespace std;

float X1, X2, Y1, Y2;

void display() {
    glClear(GL_COLOR_BUFFER_BIT);

    glBegin(GL_LINES);
        glVertex2f(1.0, 0.0);
        glVertex2f(-1.0, 0.0);
        glVertex2f(0.0, 1.0);
        glVertex2f(0.0, -1.0);
    glEnd();
```

```
    glPointSize(2.0);
    glBegin(GL_POINTS);
        glColor3f(1.0, 0.0, 0.0);

        float x = X1 / 10.0;
        float y = Y1 / 10.0;
        X2 = X2 / 10.0;
        Y2 = Y2 / 10.0;

        if (X1 <= X2 && Y1 <= Y2) {
            for (float i = x; i <= X2; i += 0.001) {
                glVertex2f(x, y);
                x += 0.001;
                y += 0.001;
            }
        } else {
            for (float i = x; i <= X2; i += 0.001) {
                glVertex2f(x, y);
                x += 0.001;
                y -= 0.001;
            }
        }
    glEnd();

    glFlush();
}

int main(int argc, char *argv[]) {
    cout << "Enter X1 value:" << endl;
    cin >> X1;
    cout << "Enter Y1 value:" << endl;
    cin >> Y1;
    cout << "Enter X2 value:" << endl;
    cin >> X2;
    cout << "Enter Y2 value:" << endl;
    cin >> Y2;

    glutInit(&argc, argv);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(10, 10);
    glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE | GLUT_DEPTH);
    glutCreateWindow("GLUT Shapes");
    glutDisplayFunc(display);
    glutMainLoop();

    return 0;
}
```

Output:



Lab – 4 Drawing lines using DDA Line drawing Algorithm from user inputs -Lab Task

Introduction: This OpenGL program allows the user to input two coordinate points to draw diagonal lines using points. Based on the slope (either m = 1 or m = -1), the code plots pixels along the diagonal, simulating line drawing using manual point placement. It also displays coordinate axes for reference. This lab enhances understanding of slope-based line drawing logic and user-driven graphics programming.

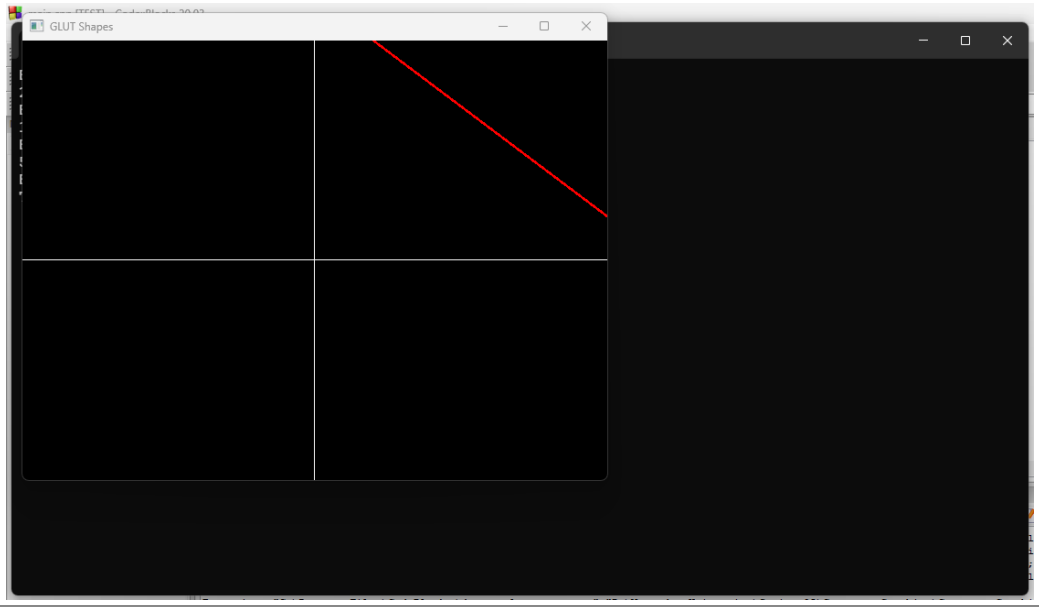
Code:

```
#include <GL/glut.h>
#include <iostream>
#include<cmath>

using namespace std;

double x_start, y_start, x_end, y_end;

void plotPoint(double x_plot, double y_plot) {
    glBegin(GL_POINTS);
    glVertex2i(x_plot, y_plot);
    glEnd();
}

void DDA_line()
{
    double m, change_x, change_y, x_plot, y_plot;

    m = (y_end - y_start) / (x_end - x_start);

    if(m<=1)
    {
        if(x_start<=x_end)
        {
            change_x = 1.0;
            change_y = m;
```

```
        }
        else
        {
            change_x = -1.0;
            change_y = -1.0 * m;
        }
    }
    else
    {
        if(x_start<=x_end)
        {
            change_x = 1.0 / m;
            change_y = 1.0;
        }
        else
        {
            change_x = -1.0 / m;
            change_y = -1.0;
        }
    }

    //cout<<"Intermediate Points: \n";
    //printf("(%.0lf,%.0lf)\n",x_start,y_start);
    plotPoint(x_start,y_start);

    while(x_plot!=x_end || y_plot!=y_end)
    {
        x_start = x_start + change_x;
        y_start = y_start + change_y;

        x_plot = round(x_start);
        y_plot = round(y_start);

        plotPoint(x_plot,y_plot);

        //printf("(%.0lf,%.0lf)\n",x_plot, y_plot);
    }
}


void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    DDA_line();
    glFlush();
}

void init() {
    glClearColor(1.0, 1.0, 1.0, 1.0);  // White background
    glColor3f(0.0, 0.0, 0.0);         // Black line
    glPointSize(3.0);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    int margin = 20;
    int left = min(x_start, x_end) - margin;
    int right = max(x_start, x_end) + margin;
    int bottom = min(y_start,y_end) - margin;
    int top = max(y_start,y_end) + margin;

    gluOrtho2D(left, right, bottom, top);
}

int main(int argc, char** argv)
```

```
{
    cout << "x_start = ";
    cin >> x_start;
    cout << "y_start = ";
    cin >> y_start;
    cout << "x_end = ";
    cin >> x_end;
    cout << "y_end = ";
    cin >> y_end;

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("DDA Line Drawing");

    init();
    glutDisplayFunc(display);
    glutMainLoop();

    return 0;
}
```

Output:



Lab – 5 Drawing lines using points Direct Bresenham Algorithm user inputs - Lab Task

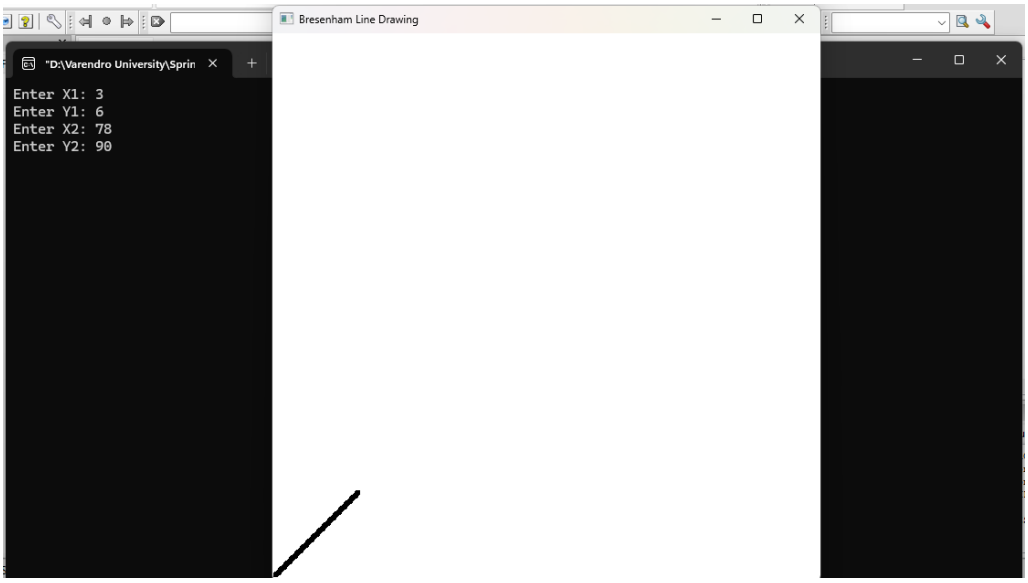
Introduction: This lab task demonstrates how to draw a straight line between two points using the **Bresenham Line Drawing Algorithm**, which is known for its efficiency in using only integer calculations. The program takes user input for the start and end coordinates of the line, then applies the Bresenham algorithm to plot each pixel on the screen. It avoids floating-point operations, making it ideal for raster displays and real-time graphics applications.

Code:

```
#include <GL/glut.h>
#include <iostream>
#include <cmath>
using namespace std;
int x_start, y_start, x_end, y_end; // Global variables to hold points

void plotPoint(int x, int y)
{
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
}
void Bresenham(int x1, int y1, int x2, int y2)
{
    int dx = abs(x2 - x1);
    int dy = abs(y2 - y1);
    int p = 2 * dy - dx;

    int x = x1;
    int y = y1;

    int x_inc = (x2 > x1) ? 1 : -1;
    int y_inc = (y2 > y1) ? 1 : -1;

    plotPoint(x, y); // Plot the initial point

    for (int i = 0; i < dx; i++) {
        x += x_inc;

        if (p < 0)
            p += 2 * dy;
        else {
            y += y_inc;
            p += 2 * (dy - dx);
        }

        plotPoint(x, y); // Plot each point
    }
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 0.0, 0.0); // Black color

    Bresenham(x_start, y_start, x_end, y_end);

    glFlush();
}

void init()
{
    glClearColor(1.0, 1.0, 1.0, 1.0); // White background
    glPointSize(5.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, 500, 0, 500); // Set coordinate system
}

int main(int argc, char** argv)
```

```
{
    cout << "Enter X1: ";
    cin >> x_start;
    cout << "Enter Y1: ";
    cin >> y_start;
    cout << "Enter X2: ";
    cin >> x_end;
    cout << "Enter Y2: ";
    cin >> y_end;
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Bresenham Line Drawing");

    init();
    glutDisplayFunc(display);
    glutMainLoop();

    return 0;
}
```

Output:



Lab – 6 Drawing circle using points, using Mid point circle drawing algorithms user inputs -Lab Task

Introduction:

In this lab, we implement the **Midpoint Circle Drawing Algorithm** using OpenGL to draw a circle based on user input for the center coordinates and radius. The algorithm utilizes symmetry and incremental integer calculations to efficiently determine the points on a circle's circumference. It avoids floating-point operations, making it ideal for raster displays and real-time graphics systems.

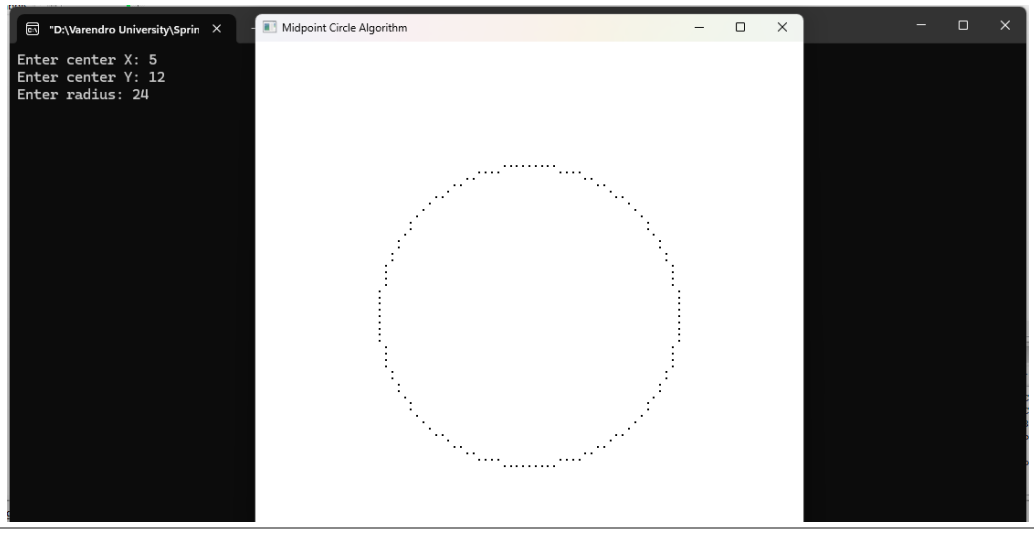
Code:

```
#include <GL/glut.h>
```

```
#include <iostream>
using namespace std;

// Global variables to hold user input
int center_x, center_y, r;

void plotCirclePoints(int x, int y)
{
    glBegin(GL_POINTS);

    glVertex2i(center_x + x, center_y + y);
    glVertex2i(center_x + y, center_y + x);
    glVertex2i(center_x + y, center_y - x);
    glVertex2i(center_x + x, center_y - y);
    glVertex2i(center_x - x, center_y - y);
    glVertex2i(center_x - y, center_y - x);
    glVertex2i(center_x - y, center_y + x);
    glVertex2i(center_x - x, center_y + y);

    glEnd();
}

void drawCircle()
{
    int x = 0;
    int y = r;
    int p = 1 - r;

    plotCirclePoints(x, y);

    while (y >= x)
    {
        if (p < 0)
        {
            p = p + 2 * x + 3;
            x = x + 1;
            y = y;
        }
        else
        {
            p = p + 2 * x - 2 * y + 5;
            x = x + 1;
            y = y - 1;
        }
        plotCirclePoints(x, y);
    }
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    drawCircle();
    glFlush();
}

void init()
{
    glClearColor(1.0, 1.0, 1.0, 1.0); // White background
    glColor3f(0.0, 0.0, 0.0);         // Black color for points
    glPointSize(2.0);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
```

```
    // Dynamically set the orthographic projection to always fit the circle
    int margin = 20;
    int left = center_x - r - margin;
    int right = center_x + r + margin;
    int bottom = center_y - r - margin;
    int top = center_y + r + margin;

    gluOrtho2D(left, right, bottom, top);
}

int main(int argc, char** argv)
{
    cout << "Enter center X: ";
    cin >> center_x;

    cout << "Enter center Y: ";
    cin >> center_y;

    cout << "Enter radius: ";
    cin >> r;

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Midpoint Circle Algorithm");

    init();
    glutDisplayFunc(display);
    glutMainLoop();

    return 0;
}
```

Output:



Lab – 7 Performing 2D Geometric Transformations (Translation, Rotation, Scaling, Reflection and Shearing) using user inputs – Lab Task

Introduction: In this lab, we explore fundamental **2D geometric transformations**—including translation, scaling, rotation, reflection, and shearing—using OpenGL. The user provides input for transformation parameters and polygon vertices. This interactive program visually demonstrates how each transformation alters the shape and position of a polygon on a 2D coordinate system, helping to understand the mathematical concepts behind computer graphics transformations.

Code:

```
// Topic: 2D Geometric Tranformations (Translation, Rotation, Scaling, Reflection, Shearing)
Algorithm Using OpenGL
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <vector>
#include <GL/glut.h>
using namespace std;

int P_X, P_Y, choice = 0, edges;
vector<int> Vec_X;
vector<int> Vec_Y;
int tx, ty;
double sx, sy, xf, yf, xr, yr;
double theta, thetaRad;
char reflectionAxis, shearingAxis;
int ShX, ShY;

double round(double d)
{
          return floor(d + 0.5);
}

void drawPolygon()
{
          glBegin(GL_POLYGON);
          glColor3f(1.0, 0.0, 0.0);
          for (int i = 0; i < edges; i++)
          {
                    glVertex2i(Vec_X[i], Vec_Y[i]);
          }
          glEnd();
}


void Translation(int x, int y)
{
          glBegin(GL_POLYGON);
          glColor3f(0.0, 1.0, 0.0);
          for (int i = 0; i < edges; i++)
          {
                    glVertex2i(Vec_X[i] + x, Vec_Y[i] + y);
          }
          glEnd();
}

void Scaling(double x, double y)
{
          glBegin(GL_POLYGON);
          glColor3f(0.0, 1.0, 0.0);
          for (int i = 0; i < edges; i++)
```

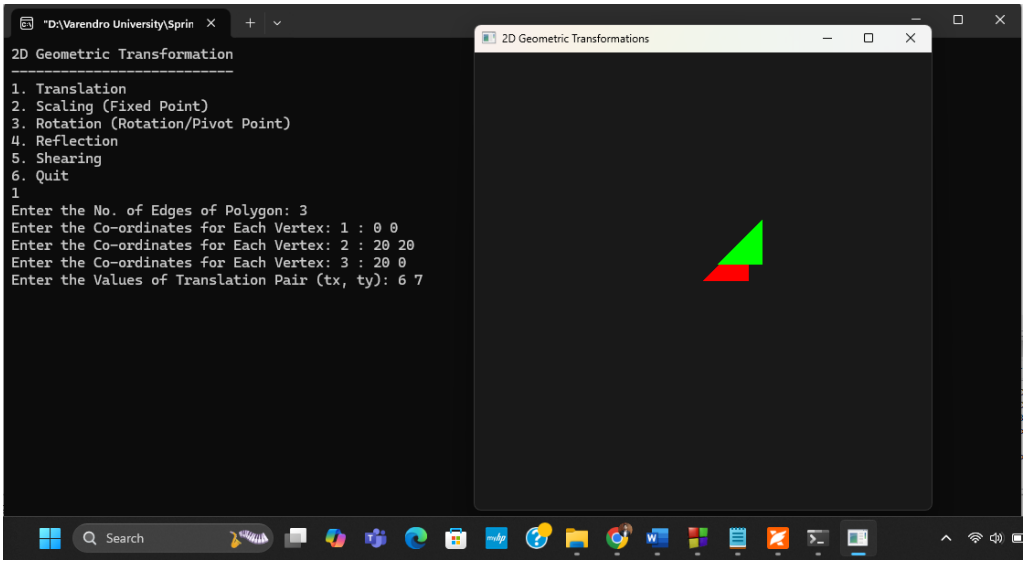
```
	{
		//glVertex2i(round(Vec_X[i] * x), round(Vec_Y[i] * y));
		glVertex2i(round((Vec_X[i]*x) + xf*(1-x)), round((Vec_Y[i]*y) + yf*(1-y)));
// Fixed Point Scaling
	}
	glEnd();
}

void Rotation(double thetaRad)
{
	glBegin(GL_POLYGON);
	glColor3f(0.0, 1.0, 0.0);
	for (int i = 0; i < edges; i++)
	{
		glVertex2i(round(xr + (Vec_X[i]-xr)*cos(thetaRad) - (Vec_Y[i]-
yr)*sin(thetaRad)), round(yr + (Vec_X[i]-xr)*sin(thetaRad) + (Vec_Y[i]-yr)*cos(thetaRad)));
	}
	glEnd();
}

void Reflection(char reflectionAxis)
{
	glBegin(GL_POLYGON);
	glColor3f(0.0, 1.0, 0.0);

	if (reflectionAxis == 'a' || reflectionAxis == 'A')   // X Axis
	{
		for (int i = 0; i < edges; i++)
		{
			glVertex2i(round(Vec_X[i]), round(Vec_Y[i] * -1));

		}
	}
	else if (reflectionAxis == 'b' || reflectionAxis == 'B')   // Y Axis
	{
		for (int i = 0; i < edges; i++)
		{
			glVertex2i(round(Vec_X[i] * -1), round(Vec_Y[i]));
		}
	}
    else if (reflectionAxis == 'c' || reflectionAxis == 'C')  // X=Y Axis
	{
		for (int i = 0; i < edges; i++)
		{
			glVertex2i(round(Vec_Y[i]), round(Vec_X[i]));
		}
	}
    else if (reflectionAxis == 'd' || reflectionAxis == 'D')  // X=Y=0 Origin
	{
		for (int i = 0; i < edges; i++)
		{
			glVertex2i(round(Vec_X[i] * -1), round(Vec_Y[i] * -1));
		}
	}
    else // Error
	{
		cout<<"Error"<<endl;
	}
	glEnd();
}

void Shearing()
{
	glBegin(GL_POLYGON);
```

```
          glColor3f(0.0, 1.0, 0.0);

          if (shearingAxis == 'x' || shearingAxis == 'X')
          {
                    glVertex2i(Vec_X[0], Vec_X[0]);

                    glVertex2i(Vec_X[1] + ShX, Vec_Y[1]);
                    glVertex2i(Vec_X[2] + ShX, Vec_Y[2]);

                    glVertex2i(Vec_X[3], Vec_Y[3]);
          }
          else if (shearingAxis == 'y' || shearingAxis == 'Y')
          {
                    glVertex2i(Vec_X[0], Vec_Y[0]);
                    glVertex2i(Vec_X[1], Vec_Y[1]);

                    glVertex2i(Vec_X[2], Vec_Y[2] + ShY);
                    glVertex2i(Vec_X[3], Vec_Y[3] + ShY);
          }
          glEnd();
}

void init(void)
{
    glClearColor(0.10, 0.10, 0.10, 0.0);         // Set Background Display Window Color
    glMatrixMode(GL_PROJECTION);                 // Set Projection Parameters
    gluOrtho2D(-100.0, 100.0, -100.0, 100.0);    // Set Orthogonal reference Frame's/ Graphs Limit
(X axis & Y axis)
}


void display(void)
{
          glClear(GL_COLOR_BUFFER_BIT);
          glColor3f(0.0, 0.0, 0.0);

          if (choice == 1)
          {
                    drawPolygon();
                    Translation(tx, ty);
          }
          else if (choice == 2)
          {
                    //drawPolygon();
                    Scaling(sx, sy);
                    drawPolygon();
          }
          else if (choice == 3)
          {
                    drawPolygon();
                    Rotation(thetaRad);
          }
          else if (choice == 4)
          {
                    drawPolygon();
                    Reflection(reflectionAxis);
          }
          else if (choice == 5)
          {
                    drawPolygon();
                    Shearing();
          }

          glFlush();
```

```
}

int main(int argc, char** argv)
{
          cout << "2D Geometric Transformation"<< endl;
          cout << "---------------------------"<< endl;

          cout << "1. Translation" << endl;
          cout << "2. Scaling (Fixed Point)" << endl;
          cout << "3. Rotation (Rotation/Pivot Point)" << endl;
          cout << "4. Reflection" << endl;
          cout << "5. Shearing" << endl;
          cout << "6. Quit" << endl;

          cin >> choice;

          if (choice == 6) {
                    return 0;
          }

          cout << "Enter the No. of Edges of Polygon: ";
          cin >> edges;

          for (int i = 0; i < edges; i++)
          {
                    cout << "Enter the Co-ordinates for Each Vertex: " << i + 1 << " : "; cin >> P_X
>> P_Y;
                    Vec_X.push_back(P_X);
                    Vec_Y.push_back(P_Y);
          }

          if (choice == 1)
          {
                    cout << "Enter the Values of Translation Pair (tx, ty): "; cin >> tx >> ty;
          }
          else if (choice == 2)
          {
                    cout << "Enter the Scaling Factors (sx, sy): "; cin >> sx >> sy;
                    cout << "Enter the Fixed Point (xf, yf): "; cin >> xf >> yf;
          }
          else if (choice == 3)
          {
                    cout << "Enter the Value of Theta (Angle) for Rotation (Degree): "; cin >>
theta;
                    thetaRad = theta * 3.1416 / 180;
                    cout << "Enter the Rotation/Pivot Point (xr, yr): "; cin >> xr >> yr;

          }
          else if (choice == 4)
          {
                    cout << "Choose Reflection Axis"<<endl;
                    cout << "----------------------"<<endl;
                    cout << "a) X Axis "<<endl;
                    cout << "b) Y Axis "<<endl;
                    cout << "c) X=Y Axis "<<endl;
                    cout << "d) About the Origin (X=Y=0) "<<endl;
                    cout << "----------------------"<<endl;
                    cout << "Enter Reflection Axis (Choose Option): "; cin >> reflectionAxis;
          }
          else if (choice == 5)
          {
                    cout << "Enter Shearing Axis ( X or Y ): "; cin >> shearingAxis;
                    if (shearingAxis == 'x' || shearingAxis == 'X')
                    {
```

```
                        cout << "Enter the Shearing Factor for X Axis (Shx): "; cin >> ShX;
                }
                else
                {
                        cout << "Enter the Shearing Factor for Y Axis (Shy): "; cin >> ShY;
                }
        }

    glutInit(&argc, argv);                      // Initalise GLUT
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);  // Set Display Mode

    glutInitWindowPosition(100, 100);               // Set Window Showing Position
    glutInitWindowSize(500, 500);                   // Set Window Size
    glutCreateWindow("2D Geometric Transformations"); // Set the Name of the Display Window


    init();                   // Execute Initialization Procedure
    glutDisplayFunc(display); // Send Graphics to Display Window ***
    glutMainLoop();           // Display Everything and Wait
    return 0;

}
```

Output:



Lab – 8 Implementing Cohen-Sutherland Line Clipping Algorithm using user inputs – Lab Task

Introduction: This lab focuses on the **Cohen-Sutherland Line Clipping Algorithm**, a popular method in computer graphics for efficiently clipping a line segment to a rectangular viewport or clipping window. The program takes user-defined coordinates for a line segment and visually demonstrates how parts of the line outside the viewing area are clipped. The original line is shown in **red**, while the clipped line (visible portion inside the window) is shown in **green**, all using OpenGL for rendering.

Code:

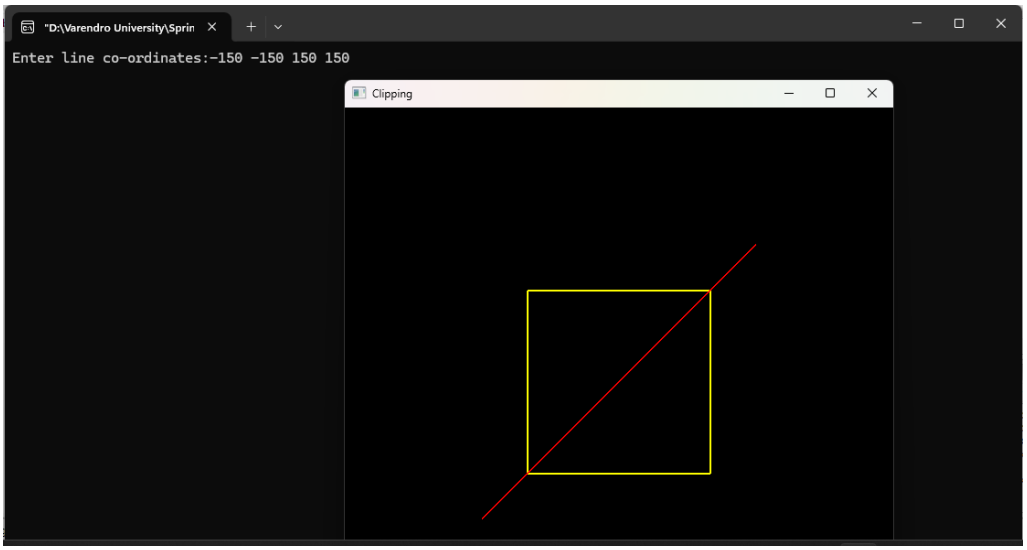
```
#include <windows.h>
```

```
#include<GL/glut.h>
#include<math.h>
#include<stdio.h>
#include<iostream>

void display();
using namespace std;
float xmin=-100;
float ymin=-100;
float xmax=100;
float ymax=100;
float xd1,yd1,xd2,yd2;
float xdd1,ydd1,xdd2,ydd2;



void init(void)
{

    glClearColor(0.0,0,0,0);
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(-300,300,-300,300);

}

int code(float x,float y)
{
    int c=0;
    if(y>ymax)c=8;
    if(y<ymin)c=4;
    if(x>xmax)c=c|2;
    if(x<xmin)c=c|1;
    return c;
}

void cohen_Line(float x1,float y1,float x2,float y2)
{
    int c1=code(x1,y1);
    int c2=code(x2,y2);
    float m=(y2-y1)/(x2-x1);
    while((c1|c2)>0)
    {
        if((c1 & c2)>0)
        {
           exit(0);
        }

    float xi=x1;float yi=y1;
    int c=c1;
    if(c==0)
    {
         c=c2;
         xi=x2;
         yi=y2;
    }
    float x,y;
    if((c & 8)>0)
    {
       y=ymax;
       x=xi+ 1.0/m*(ymax-yi);
    }
    else
      if((c & 4)>0)
      {
```

```
          y=ymin;
          x=xi+1.0/m*(ymin-yi);
      }
      else
       if((c & 2)>0)
       {
           x=xmax;
           y=yi+m*(xmax-xi);
       }
       else
       if((c & 1)>0)
       {
           x=xmin;
           y=yi+m*(xmin-xi);
       }

       if(c==c1)
       {
           xd1=x;
           yd1=y;
           c1=code(xd1,yd1);
       }

       if(c==c2)
       {
           xd2=x;
           yd2=y;
           c2=code(xd2,yd2);
       }
}
xdd1 = xd1;
ydd1 = yd1;

xdd2 = xd2;
ydd2 = yd2;

 display();

}

void mykey(unsigned char key,int x,int y)
{
    if(key=='c')
    {  cout<<"Hello";
        cohen_Line(xd1,yd1,xd2,yd2);
        glFlush();
    }
}
void display()
{

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0,1.0,0.0);

   glBegin(GL_LINE_LOOP);
   glVertex2i(xmin,ymin);
   glVertex2i(xmin,ymax);
   glVertex2i(xmax,ymax);
   glVertex2i(xmax,ymin);
   glEnd();

   glLineWidth (2);
   glColor3f(1.0,0.0,0.0);
   glBegin(GL_LINES);
```

```
   glVertex2i(xd1,yd1);
   glVertex2i(xd2,yd2);
   glEnd();

   glLineWidth (2);
   glColor3f(0.0,1.0,0.0);
   glBegin(GL_LINES);
   glVertex2i(xdd1,ydd1);
   glVertex2i(xdd2,ydd2);
   glEnd();

   glFlush();


}



int main(int argc,char** argv)
{
    printf("Enter line co-ordinates:");
    cin>>xd1>>yd1>>xd2>>yd2;
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(600,600);
    glutInitWindowPosition(0,0);
    glutCreateWindow("Clipping");
    glutDisplayFunc(display);
    glutKeyboardFunc(mykey);
    init();
    glutMainLoop();
    return 0;
}
```

Output:



Lab – 9 Implementing Sutherland-Hodgman Polygon Clipping Algorithm using user inputs – Lab Task

Introduction: This lab implements the **Sutherland-Hodgman Polygon Clipping Algorithm** using OpenGL. The algorithm clips a **convex or concave polygon** against a rectangular clipping window defined by two diagonal corners. When you left-click the window, the polygon is clipped to the boundary of the rectangular region, and the clipped polygon is displayed.
The algorithm processes each edge of the clipping window (left, right, top, bottom) one at a time and updates the polygon vertices after each clipping step.

Code:

```
// Sutherland Hodgman Ploygon Clipping Complete Code | implementation through openGL
/*
20 40
80 80
3
30 70
10 30
40 30




*/

#include<stdio.h>
#include<GL/gl.h>
#include<GL/glu.h>
#include<GL/glut.h>
#include<math.h>

typedef struct                                  // structure that holds the information of points
{
          float x;
          float y;
}PT;

// global variables

int n;

int i,j;

PT p1,p2,p[20],pp[20];

void left()                                     // left clipper
{
          i=0;j=0;

          for(i=0;i<n;i++)
          {
                    if(p[i].x<p1.x && p[i+1].x>=p1.x)               //Case-1: outside to inside
                    {
                              if(p[i+1].x-p[i].x!=0)
                              {
                                        pp[j].y=(p[i+1].y-p[i].y)/(p[i+1].x-p[i].x)*(p1.x-
p[i].x)+p[i].y;               // save point of intersection
                              }
                              else
                              {
                                        pp[j].y=p[i].y;
```

```
                        }
                        pp[j].x=p1.x;
                        j++;
                        pp[j].x=p[i+1].x;                               // save that point
that lie inside our clipping window // consult theory
                        pp[j].y=p[i+1].y;
                        j++;

                }

                if(p[i].x>=p1.x && p[i+1].x>=p1.x)               //Case-2: inside to inside
                {
                        pp[j].y=p[i+1].y;                               // only save second
point that lie inside our clipping window // consult theory
                        pp[j].x=p[i+1].x;
                        j++;
                }

                if(p[i].x>=p1.x && p[i+1].x<p1.x)                // Case-3:          inside to
outside
                {
                        if(p[i+1].x-p[i].x!=0)
                        {
                                pp[j].y=(p[i+1].y-p[i].y)/(p[i+1].x-p[i].x)*(p1.x-
p[i].x)+p[i].y;              // only save point of intersection
                        }
                        else
                        {
                                pp[j].y=p[i].y;
                        }
                        pp[j].x=p1.x;
                        j++;
                }
        }

        for(i=0;i<j;i++)
        {
                p[i].x=pp[i].x;
                p[i].y=pp[i].y;
        }

        p[i].x=pp[0].x;
        p[i].y=pp[0].y;
        n=j;
}


void right()                                            // right clipper
{
        i=0;j=0;

        for(i=0;i<n;i++)
        {
                if(p[i].x>p2.x && p[i+1].x<=p2.x)                //Case-1: outside to inside
                {
                        if(p[i+1].x-p[i].x!=0)
                        {
                                pp[j].y=(p[i+1].y-p[i].y)/(p[i+1].x-p[i].x)*(p2.x-
p[i].x)+p[i].y;                 // save point of intersection
                        }
                        else
                        {
                                pp[j].y=p[i].y;
                        }
```

```
                              pp[j].x=p2.x;
                              j++;
                              pp[j].x=p[i+1].x;                                                    //
save that point that lie inside our clipping window // consult theory
                              pp[j].y=p[i+1].y;
                              j++;
                    }

                    if(p[i].x<=p2.x && p[i+1].x<=p2.x)              // Case-2:         inside to
inside
                    {
                              pp[j].y=p[i+1].y;                                  // only save second
point that lie inside our clipping window // consult theory
                              pp[j].x=p[i+1].x;
                              j++;
                    }

                    if(p[i].x<=p2.x && p[i+1].x>p2.x)               // Case-3:         inside to
outside
                    {
                              if(p[i+1].x-p[i].x!=0)
                              {
                                        pp[j].y=(p[i+1].y-p[i].y)/(p[i+1].x-p[i].x)*(p2.x-
p[i].x)+p[i].y;                         // only save point of intersection
                              }
                              else
                              {
                                        pp[j].y=p[i].y;
                              }
                              pp[j].x=p2.x;
                              j++;
                    }

          }

          for(i=0;i<j;i++)
          {
                    p[i].x=pp[i].x;
                    p[i].y=pp[i].y;
          }

          p[i].x=pp[0].x;
          p[i].y=pp[0].y;

}

void top()                                                  // top clipper
{
          i=0;j=0;

          for(i=0;i<n;i++)
          {
                    if(p[i].y>p2.y && p[i+1].y<=p2.y)               //Case-1: outside to inside
                    {
                              if(p[i+1].y-p[i].y!=0)
                              {
                                        pp[j].x=(p[i+1].x-p[i].x)/(p[i+1].y-p[i].y)*(p2.y-
p[i].y)+p[i].x;                                   // save point of intersection
                              }
                              else
                              {
                                        pp[j].x=p[i].x;
                              }
                              pp[j].y=p2.y;
```

```
                              j++;
                              pp[j].x=p[i+1].x;                                        // save
that point that lie inside our clipping window // consult theory
                              pp[j].y=p[i+1].y;
                              j++;
                    }

                    if(p[i].y<=p2.y && p[i+1].y<=p2.y)              // Case-2:         inside to
inside
                    {
                              pp[j].y=p[i+1].y;                                        // only
save second point that lie inside our clipping window // consult theory
                              pp[j].x=p[i+1].x;
                              j++;
                    }

                    if(p[i].y<=p2.y && p[i+1].y>p2.y)               // Case-3:         inside to
outside
                    {
                              if(p[i+1].y-p[i].y!=0)
                              {
                                        pp[j].x=(p[i+1].x-p[i].x)/(p[i+1].y-p[i].y)*(p2.y-
p[i].y)+p[i].x;                         // only save point of intersection
                              }
                              else
                              {
                                        pp[j].x=p[i].x;
                              }
                              pp[j].y=p2.y;
                              j++;
                    }
          }

          for(i=0;i<j;i++)
          {
                    p[i].x=pp[i].x;
                    p[i].y=pp[i].y;
          }

          p[i].x=pp[0].x;
          p[i].y=pp[0].y;
          n=j;
}


void bottom()                                       // bottom clipper
{
          i=0;j=0;

          for(i=0;i<n;i++)
          {
                    if(p[i].y<p1.y && p[i+1].y>=p1.y)               // Case-1:         outside to
inside
                    {
                              if(p[i+1].y-p[i].y!=0)
                              {
                                        pp[j].x=(p[i+1].x-p[i].x)/(p[i+1].y-p[i].y)*(p1.y-
p[i].y)+p[i].x;                                   // save point of intersection
                              }
                              else
                              {
                                        pp[j].x=p[i].x;
                              }
                              pp[j].y=p1.y;
```

```
                                j++;
                                pp[j].x=p[i+1].x;                                                                       //
save that point that lie inside our clipping window // consult theory
                                pp[j].y=p[i+1].y;
                                j++;
                        }

                        if(p[i].y>=p1.y && p[i+1].y>=p1.y)               // Case-2:          inside to
inside
                        {
                                pp[j].x=p[i+1].x;                                               // only
save second point that lie inside our clipping window // consult theory
                                pp[j].y=p[i+1].y;
                                j++;
                        }

                        if(p[i].y>=p1.y && p[i+1].y<p1.y)                // Case-3:          inside to
outside
                        {
                                if(p[i+1].y-p[i].y!=0)
                                {
                                        pp[j].x=(p[i+1].x-p[i].x)/(p[i+1].y-p[i].y)*(p1.y-
p[i].y)+p[i].x;                         // only save point of intersection
                                }
                                else
                                {
                                        pp[j].x=p[i].x;
                                }
                                pp[j].y=p1.y;
                                j++;
                        }
                }

                for(i=0;i<j;i++)
                {
                        p[i].x=pp[i].x;
                        p[i].y=pp[i].y;
                }
                p[i].x=pp[0].x;
                p[i].y=pp[0].y;
                n=j;
}

void drawpolygon()
{
                glColor3f(1.0,0.0,0.0);
                for(i=0;i<n-1;i++)
                {
                        glBegin(GL_LINES);
                        glVertex2d(p[i].x,p[i].y);
                        glVertex2d(p[i+1].x,p[i+1].y);
                        glEnd();
                }
                glBegin(GL_LINES);
                glVertex2d(p[i].x,p[i].y);
                glVertex2d(p[0].x,p[0].y);
                glEnd();
}

void myMouse(int button, int state, int x, int y)
{
                if(button==GLUT_LEFT_BUTTON && state==GLUT_DOWN)                        // On output, please
left click on polygon then and only then clipping performs
                {
```

```
                    glClear(GL_COLOR_BUFFER_BIT);

                    glBegin(GL_LINE_LOOP);
                    glVertex2f(p1.x,p1.y);
                    glVertex2f(p2.x,p1.y);
                    glVertex2f(p2.x,p2.y);
                    glVertex2f(p1.x,p2.y);
                    glEnd();
                    left();
                    right();
                    top();
                    bottom();
                    drawpolygon();
          }
          glFlush();
}


void display(void)
{
          glClear(GL_COLOR_BUFFER_BIT);
          glColor3f(0.4,1.0,0.0);
          glBegin(GL_LINE_LOOP);
          glVertex2f(p1.x, p1.y);
          glVertex2f(p2.x,p1.y);
          glVertex2f(p2.x,p2.y);
          glVertex2f(p1.x,p2.y);
          glEnd();
          drawpolygon();
          glFlush();
}

void init(void)
{
          glClearColor(0.0,0.0,0.0,0.0);                              // clear screen usually black
          gluOrtho2D(0,500,0,500);
}

int main(int argc, char**argv)
{
          printf("Enter Window Coordinates:\n");
          printf("Please Enter two Points:\n");                                 // P1(x,y) is the
bottom left point for clipping window
          printf("Enter P1(x,y):\n");
          scanf("%f", &p1.x);                                 // if you don't know what value should be
given: enter 200
          scanf("%f", &p1.y);                                 // if you don't know what value should be
given: enter 200

          printf("Enter P2(x,y):\n");                                                                    //
P2(x,y) is the top right point for clipping window
          scanf("%f", &p2.x);                                 // if you don't know what value should be
given: enter 400
          scanf("%f", &p2.y);                                 // if you don't know what value should be
given: enter 400

          printf("\nEnter the no. of vertices:");                    // if you don't know what value
should be given: enter 3
          scanf("%d", &n);

          for(i=0;i<n;i++)
          {
                    printf("\nEnter V%d(x%d,y%d):\n" , i+1, i+1, i+1);
```

```
                    scanf("%f", &p[i].x);                   // if you don't know what value should be
given: enter V1(100,110), V2(340,210), V3(300,380)
                    scanf("%f", &p[i].y);
          }

          p[i].x=p[0].x;                                            // Assign last to first for
connected everything
          p[i].y=p[0].y;

          glutInit(&argc,argv);
          glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
          glutInitWindowSize(640,480);
          glutInitWindowPosition(0,0);
          glutCreateWindow("Sutherland Hodgman Polygon Clipping Algorithm ");
          init();

          glutDisplayFunc(display);
          glutMouseFunc(myMouse);                                   // notice mouse movement and
call user defined function
          glFlush();
          glutMainLoop();
          return 0;


}
```

Output: