SCHOOL OF COMPUTER SCIENCE

UNIVERSITY OF PETROLEUM AND ENERGY STUDIES
DEHRADUN, UTTARAKHAND

COMPUTER GRAPHICS
LABORATORY FILE
(2024-2025)

For

Vth Semester

Submitted To:
Mr. Dinesh Bafila

Submitted By:
Akshat Negi
500106533(SAP ID)
R2142220414(Roll No.)
B.Tech. CSF (Batch-1)

COMPUTER GRAPHICS LAB-VTH SEM

NAME: Akshat Negi SAP-ID: 500106533 SPL/BATCH: 1

S.NO	COMPUTER GRAPHICS PROGRAM NAME	PAGE NO	DATE	REMARK/ GRADE
1	Experiment 1: Introduction to OpenGL: [Lab Environment Setup] a. What is OpenGL? b. What is GLU/GLUT? c. What is OpenGL Architecture? d. Setting up the environment. e. First OpenGL Program: This initializes a window of green color. f. Draw a Hut.		09/09/24	
2	Experiment 2: Drawing a line [Usage of Open GL] a. Draw a line using equation of line Y=m*X+C. b. Draw a line using DDA algorithm for slope m<1 and m>1. c. Draw a line using Bresenham algorithm for slope m<1 and m>1. # Take the input from user for all the three scenarios i.e. value of (x1, y1) and (x2, y2).		16/09/24	
3	Experiment 3: Drawing a Circle and an Ellipse [Done on OpenGL] a. Draw the circle with the help of polar equations b. Draw the circle with the help of mid-point method. c. Draw the Ellipse with the mid-point method. # Take the value of radius, major axis and minor axis as input from the user.		23/09/24	
4	Experiment 4: Seed Fill Algorithms [Small Project will be given for demonstration] a. WAP to fill the polygon using scan lines. b. WAP to fill a region using boundary fill algorithm using 4 or 8 connected approaches. c. WAP to fill a region using flood fill algorithm using 4 or 8 connected approaches. # Take the value of seed point, intensity of new color as input from user.		21/10/24	
5	Experiment 5: Viewing and Clipping [Geographical Animation for demonstration] a. Write an interactive program for line clipping using Cohen Sutherland line clipping algorithm. b. Write an interactive program for line clipping using Liang-Barsky line clipping algorithm. c. Write an interactive program for polygon clipping using Sutherland – Hodgeman polygon clipping algorithm. # Take the window coordinates as input from the user, also take polygon coordinates as input.		11/11/24	

6	Experiment 6 : Basic 2D & 3D Transformations a. Write an interactive program for following basic transformation. b. Translation c. Rotation d. Scaling e. Reflection about axis. f. Reflection about a line Y=mX+c and aX+bY+c=0. g. Shear about an edge and about a vertex. # Perform all the experiment for 3-D transformation. # Take the following values as input from user: Theta (angle of rotation), translation factor, scaling factor and other values. Make necessary assumptions.	✓	11/11/24	
7	Experiment 7: Drawing Bezier curves. [Virtual GLUT based demonstration] a. Write a program to draw a cubic spline. b. WAP to draw a Bezier curve. # Take necessary values as input from the user like degree of the Bezier curve.		11/11/24	
8	Experiment 8: Event Handling a. Implement mouse input functionality. b. Implement keypress functionality. c. Implement another call back functions. #Implement above with the help of animation.		11/11/24	
9	Experiment 9: Creating 3D Shapes like Cube, Sphere, and others.		11/11/24	

LAB EXPERIMENT – 1

Introduction to OpenGL:

(Lab Environment Setup)

a) What is OpenGL?

OpenGL (Open Graphics Library) is a powerful and versatile application programming interface (API) used for rendering 2D and 3D vector graphics. Essentially, it's a set of functions that allow software to communicate with a computer's graphics hardware (GPU) to create stunning visuals.

b) What is GLU/GLUT?

GLU: OpenGL Utility Library

(OpenGL Utility Library) is a higher-level library built on top of OpenGL. It provides a set of functions that simplify common tasks in 3D graphics programming. These functions offer a more convenient interface for performing operations like:

- **Projection and viewing transformations:** Defining the camera's position and orientation.
- **Quadric surfaces:** Creating shapes like spheres, cylinders, and cones.
- **NURBS:** Handling non-uniform rational B-splines for complex curves and surfaces.
- **Tessellation:** Breaking down complex shapes into simpler polygons.
- **Error handling:** Managing OpenGL errors.

While GLU was useful in the past, it's gradually being phased out as modern OpenGL provides more direct ways to accomplish these tasks.

GLUT: OpenGL Utility Toolkit

(OpenGL Utility Toolkit) is a cross-platform windowing library designed to simplify creating OpenGL applications. It provides basic functions for:

- **Window creation and management:** Opening, closing, and resizing windows.
- **Input handling:** Managing keyboard and mouse events.

- **OpenGL context management:** Creating and destroying OpenGL rendering contexts.
- **Idle callback:** Executing code when the application is idle.

GLUT is primarily used for educational purposes and small-scale projects. For more complex applications, modern windowing libraries like GLFW or Qt are often preferred.

c) What is OpenGL Architecture?

The OpenGL architecture refers to the design and structure of the OpenGL system, which is a software interface to graphics hardware. This architecture outlines how OpenGL interacts with the underlying hardware, software, and the various components that make up the OpenGL system.

Key Components of OpenGL Architecture:

OpenGL Client and Server Model:

a) **Client:** The application that makes OpenGL API calls is considered the "client."

b) **Server:** The "server" is typically the graphics hardware or the driver software that executes the OpenGL commands. In a distributed system, the server could be a remote machine with the necessary hardware.

c) OpenGL operates in a client-server model, where the client sends drawing commands to the server, which then processes these commands and renders the graphics.

OpenGL Pipeline: The OpenGL rendering pipeline is a sequence of steps that the system follows to transform 3D models into a 2D image on the screen.
Vertex Processing: Vertices (points in 3D space) are processed, including transformations (e.g., scaling, rotation) and lighting calculations.
Primitive Assembly: Vertices are assembled into geometric primitives, like points, lines, or triangles.
Rasterization: The primitives are converted into fragments, which are potential pixels on the screen.
Fragment Processing: Each fragment is processed to determine its final colour and other attributes, taking into account textures, shading, and lighting.
Framebuffer Operations: The processed fragments are written to the framebuffer, where they become pixels that will be displayed on the screen.

OpenGL Context:

d) The OpenGL context is an environment in which OpenGL functions operate. It includes all the state information needed to perform rendering operations, such as textures, shaders, and buffer objects.

e) Each window or rendering surface has its own OpenGL context, and multiple contexts can share resources.

State Machine:

f) OpenGL operates as a state machine, meaning it maintains various states (e.g., current color, current texture) that persist until explicitly changed by the application.

g) The state machine allows for efficient rendering, as OpenGL doesn't need to repeatedly set the same parameters unless they change.

Extensions:

h) OpenGL is designed to be extensible. Hardware vendors can introduce new features through extensions, allowing developers to access advanced capabilities beyond the core OpenGL specification.

i) Extensions provide a way to experiment with new features before they become part of the official OpenGL standard.

GLU and GLUT:

As mentioned earlier, GLU and GLUT are utility libraries that work on top of the core OpenGL architecture to provide additional functionality and simplify certain tasks.

These libraries help manage higher-level operations (GLU) and handle windowing and input (GLUT).

Shaders and Programmable Pipeline:

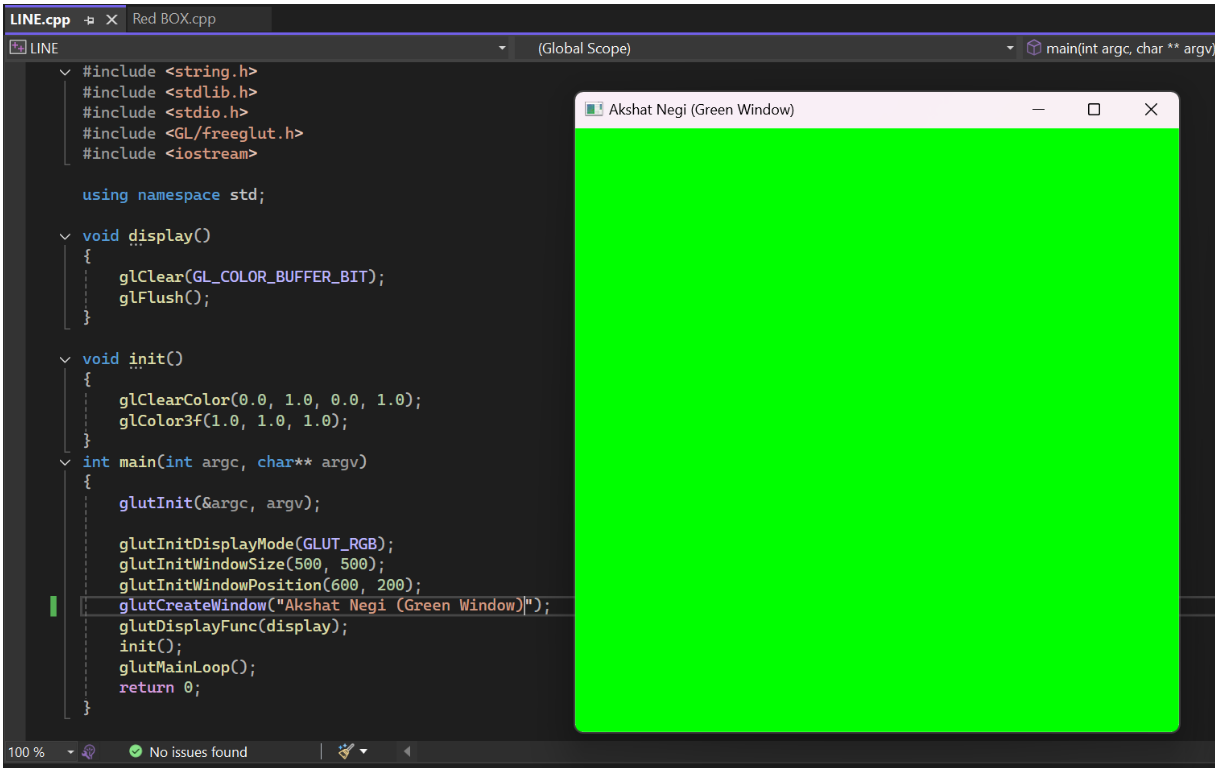
In modern OpenGL, the fixed-function pipeline has largely been replaced by the programmable pipeline, where shaders (small programs written in GLSL, the OpenGL Shading Language) control various stages of the rendering process.

- **Vertex Shader:** Handles the transformation and lighting of individual vertices.
- **Fragment Shader:** Determines the color and other attributes of individual fragments.

d) Setting up the environment.

e) First OpenGL Program: This initializes a window of green color.

```cpp
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <GL/freeglut.h>
#include <iostream>
using namespace std;
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
}

void init()
{
    glClearColor(0.0, 1.0, 0.0, 1.0);
    glColor3f(1.0, 1.0, 1.0);
}
int main(int argc, char** argv)
{
    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(600, 200);
    glutCreateWindow("Akshat Negi (Green Window)");
    glutDisplayFunc(display);
    init();
    glutMainLoop();
    return 0;

}
```



f) Draw a Hut.

```
#include <GL/freeglut.h>
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.6f, 0.4f, 0.2f); // Brown color
    glBegin(GL_POLYGON);
    glVertex2f(-0.5f, -0.5f);
    glVertex2f(0.5f, -0.5f);
    glVertex2f(0.5f, 0.0f);
    glVertex2f(-0.5f, 0.0f);
    glEnd();
    glColor3f(0.8f, 0.2f, 0.0f); // Red color
    glBegin(GL_TRIANGLES);
    glVertex2f(-0.6f, 0.0f);
    glVertex2f(0.6f, 0.0f);
    glVertex2f(0.0f, 0.5f);
    glEnd();
    glColor3f(0.3f, 0.2f, 0.1f); // Dark brown color
    glBegin(GL_POLYGON);
    glVertex2f(-0.1f, -0.5f);
    glVertex2f(0.1f, -0.5f);
    glVertex2f(0.1f, -0.2f);
    glVertex2f(-0.1f, -0.2f);
    glEnd();
    glColor3f(0.0f, 0.6f, 1.0f); // Blue color
    glBegin(GL_POLYGON);
    glVertex2f(-0.4f, -0.2f);
    glVertex2f(-0.2f, -0.2f);
    glVertex2f(-0.2f, 0.0f);
    glVertex2f(-0.4f, 0.0f);
    glEnd();
    glFlush();
}
void init() {
    glClearColor(0.5f, 0.8f, 1.0f, 1.0f); // Light blue background
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
}
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Simple Hut");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Experiment 2: Drawing a line [Usage of Open GL]

a. **Draw a line using equation of line Y=m*X+C.**

b. **Draw a line using DDA algorithm for slope m<1 and m>1.**

```
#include <GL/freeglut.h>
#include <iostream>
#include <cmath>
using namespace std; // Function to plot points
void plot(int x, int y) {
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
    glFlush();
}  // DDA Line Drawing Algorithm
void DDA(int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    int steps = abs(dx) > abs(dy) ? abs(dx) : abs(dy);  // Maximum steps
    float xIncrement = dx / (float)steps;
    float yIncrement = dy / (float)steps;
    float x = x1;
    float y = y1; // Draw the line by plotting points
    for (int i = 0; i <= steps; i++) {
        plot(round(x), round(y));
        x += xIncrement;
        y += yIncrement;
    }
}// Function to get input from the user and call DDA
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    int x1, y1, x2, y2;
    cout << "Enter the coordinates of the first point (x1, y1): ";
    cin >> x1 >> y1;
    cout << "Enter the coordinates of the second point (x2, y2): ";
    cin >> x2 >> y2;
    DDA(x1, y1, x2, y2);
} // Initialize the OpenGL Graphics
void init() {
    glClearColor(1.0, 1.0, 1.0, 0.0);  // Background color
    glColor3f(0.0, 0.0, 0.0);          // Drawing color
    glPointSize(2.0);                  // Point size
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 500.0, 0.0, 500.0);  // Define the drawing area
}
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);       // Window size
    glutInitWindowPosition(100, 100);   // Window position
    glutCreateWindow("DDA Line Drawing Algorithm");
    init();
    glutDisplayFunc(display);           // Register display function
    glutMainLoop();                     // Enter the event-processing loop
    return 0;
}
```

c. Draw a line using Bresenhan algorithm for slope m<1 and m>1.

```
#include <GL/freeglut.h>
#include <stdio.h> // Function to set pixel at (x, y)
void setPixel(int x, int y) {
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
    glFlush();
} // Bresenham's algorithm for slope |m| < 1 (dy < dx)
void bresenhamLineLow(int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    int D = 2 * dy - dx;
    int y = y1;
    for (int x = x1; x <= x2; x++) {
        setPixel(x, y);
        if (D > 0) {
            y += (y2 > y1) ? 1 : -1;  // Increase/decrease y depending on the
slope direction
            D = D + (2 * (dy - dx));
        }
        else {
            D = D + 2 * dy;
        }
    }
}// Bresenham's algorithm for slope |m| > 1 (dy > dx)
void bresenhamLineHigh(int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    int D = 2 * dx - dy;
    int x = x1;
    for (int y = y1; y <= y2; y++) {
        setPixel(x, y);
        if (D > 0) {
            x += (x2 > x1) ? 1 : -1;  // Increase/decrease x depending on the
slope direction
            D = D + (2 * (dx - dy));
        }
        else {
            D = D + 2 * dx;
        }
    }
} // Main function that checks the slope and calls the appropriate function
void drawLine(int x1, int y1, int x2, int y2) {
    if (abs(y2 - y1) < abs(x2 - x1)) {
        if (x1 > x2) {
            bresenhamLineLow(x2, y2, x1, y1);  // Line from (x2, y2) to (x1, y1)
        }
        else {
            bresenhamLineLow(x1, y1, x2, y2);  // Line from (x1, y1) to (x2, y2)
        }
    }
    else {
        if (y1 > y2) {
            bresenhamLineHigh(x2, y2, x1, y1); // Line from (x2, y2) to (x1, y1)
        }
        else {
            bresenhamLineHigh(x1, y1, x2, y2); // Line from (x1, y1) to (x2, y2)
        }
    }
} // User input handling and initialization
```

```
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    int x1, y1, x2, y2;
    printf("Enter coordinates of the first point (x1, y1): ");
    scanf_s("%d %d", &x1, &y1);
    printf("Enter coordinates of the second point (x2, y2): ");
    scanf_s("%d %d", &x2, &y2);
    drawLine(x1, y1, x2, y2);
}
void init() {
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glColor3f(0.0, 0.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, 500, 0, 500);  // Set the orthographic projection
}
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutCreateWindow("Bresenham's Line Algorithm");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

 # Take the input from user for all the three scenarios i.e. value of (x1, y1) and (x2, y2).

LAB EXPERIMENT – 2

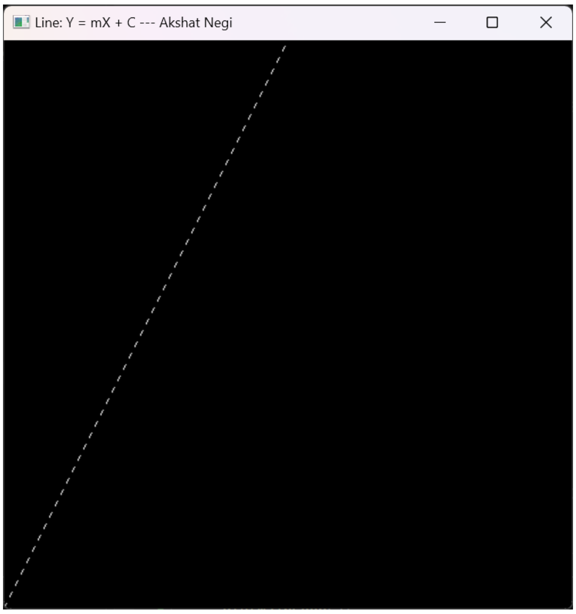
DRAWING A LINE

[Usage of Open GL]

 # Take the input from user for all the three scenarios i.e. value of (x1, y1) and (x2, y2).

a) Draw a line using equation of line Y=m*X+C.

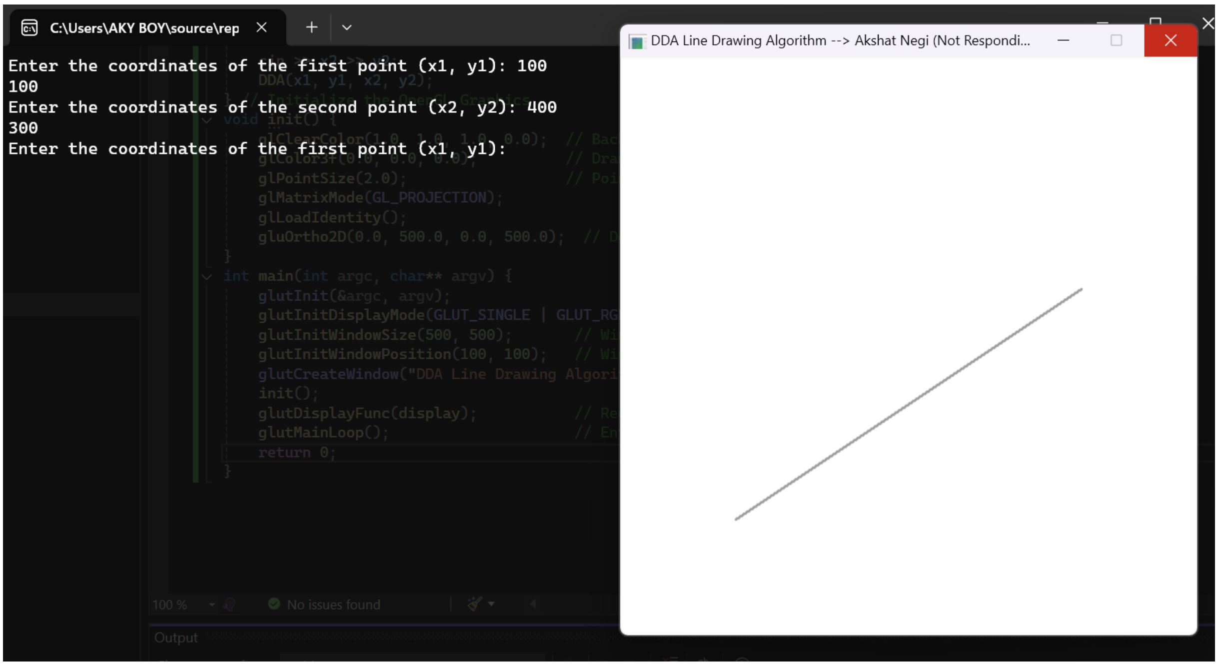
```
#include <GL/freeglut.h>

float m = 2.0f;
float C = 1.0f;

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);

    glBegin(GL_LINES);
    for (float x = -1.0f; x <= 1.0f; x += 0.01f)
    {
        float y = m * x + C;
        glVertex2f(x, y);
    }
    glEnd();

    glFlush();
}

void init()
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glColor3f(1.0, 1.0, 1.0);
    gluOrtho2D(-1.0, 1.0, -1.0, 1.0);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Line: Y = mX + C --- Akshat Negi");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
   }
```



b) Draw a line using DDA algorithm for slope m<1 and m>1.

```
#include <GL/freeglut.h>
#include <iostream>
#include <cmath>
using namespace std; // Function to plot points
void plot(int x, int y) {
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
    glFlush();
}  // DDA Line Drawing Algorithm
void DDA(int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    int steps = abs(dx) > abs(dy) ? abs(dx) : abs(dy);  // Maximum steps
    float xIncrement = dx / (float)steps;
    float yIncrement = dy / (float)steps;
    float x = x1;
    float y = y1; // Draw the line by plotting points
    for (int i = 0; i <= steps; i++) {
        plot(round(x), round(y));
        x += xIncrement;
        y += yIncrement;
    }
}// Function to get input from the user and call DDA
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    int x1, y1, x2, y2;
    cout << "Enter the coordinates of the first point (x1, y1): ";
    cin >> x1 >> y1;
    cout << "Enter the coordinates of the second point (x2, y2): ";
    cin >> x2 >> y2;
    DDA(x1, y1, x2, y2);
} // Initialize the OpenGL Graphics
void init() {
```

```
    glClearColor(1.0, 1.0, 1.0, 0.0);  // Background color
    glColor3f(0.0, 0.0, 0.0);          // Drawing color
    glPointSize(2.0);                  // Point size
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 500.0, 0.0, 500.0);  // Define the drawing area
}
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);       // Window size
    glutInitWindowPosition(100, 100);   // Window position
    glutCreateWindow("DDA Line Drawing Algorithm --> Akshat Negi");
    init();
    glutDisplayFunc(display);           // Register display function
    glutMainLoop();                     // Enter the event-processing loop
    return 0;
}
```

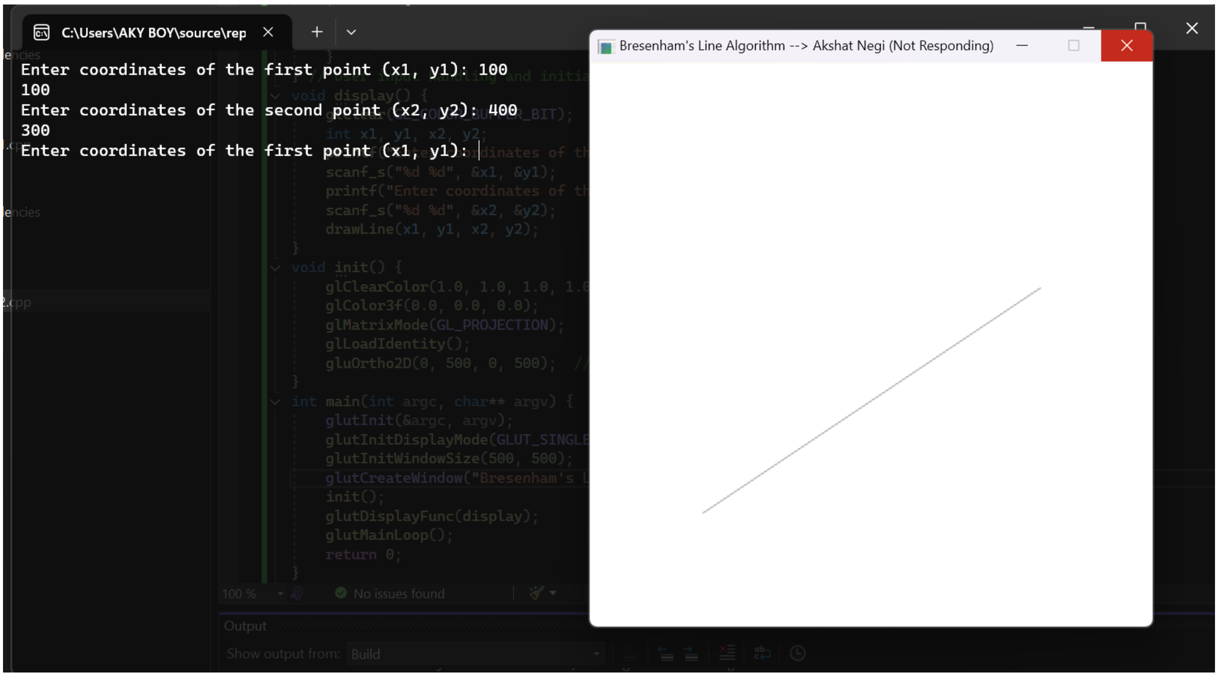


c) Draw a line using Bresenham algorithm for slope m<1 and m>1.

```
#include <GL/freeglut.h>
#include <stdio.h> // Function to set pixel at (x, y)
void setPixel(int x, int y) {
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
    glFlush();
} // Bresenham's algorithm for slope |m| < 1 (dy < dx)
void bresenhamLineLow(int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    int D = 2 * dy - dx;
    int y = y1;
    for (int x = x1; x <= x2; x++) {
        setPixel(x, y);
        if (D > 0) {
            y += (y2 > y1) ? 1 : -1;  // Increase/decrease y depending on the
slope direction
```

```
            D = D + (2 * (dy - dx));
        }
        else {
            D = D + 2 * dy;
        }
    }
}// Bresenham's algorithm for slope |m| > 1 (dy > dx)
void bresenhamLineHigh(int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    int D = 2 * dx - dy;
    int x = x1;
    for (int y = y1; y <= y2; y++) {
        setPixel(x, y);
        if (D > 0) {
            x += (x2 > x1) ? 1 : -1;  // Increase/decrease x depending on the
slope direction
            D = D + (2 * (dx - dy));
        }
        else {
            D = D + 2 * dx;
        }
    }
} // Main function that checks the slope and calls the appropriate function
void drawLine(int x1, int y1, int x2, int y2) {
    if (abs(y2 - y1) < abs(x2 - x1)) {
        if (x1 > x2) {
            bresenhamLineLow(x2, y2, x1, y1);  // Line from (x2, y2) to (x1, y1)
        }
        else {
            bresenhamLineLow(x1, y1, x2, y2);  // Line from (x1, y1) to (x2, y2)
        }
    }
    else {
        if (y1 > y2) {
            bresenhamLineHigh(x2, y2, x1, y1); // Line from (x2, y2) to (x1, y1)
        }
        else {
            bresenhamLineHigh(x1, y1, x2, y2); // Line from (x1, y1) to (x2, y2)
        }
    }
} // User input handling and initialization
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    int x1, y1, x2, y2;
    printf("Enter coordinates of the first point (x1, y1): ");
    scanf_s("%d %d", &x1, &y1);
    printf("Enter coordinates of the second point (x2, y2): ");
    scanf_s("%d %d", &x2, &y2);
    drawLine(x1, y1, x2, y2);
}
void init() {
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glColor3f(0.0, 0.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, 500, 0, 500);  // Set the orthographic projection
}
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutCreateWindow("Bresenham's Line Algorithm --> Akshat Negi");
```

```
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```



LAB EXPERIMENT – 3

Drawing a Circle and an Ellipse

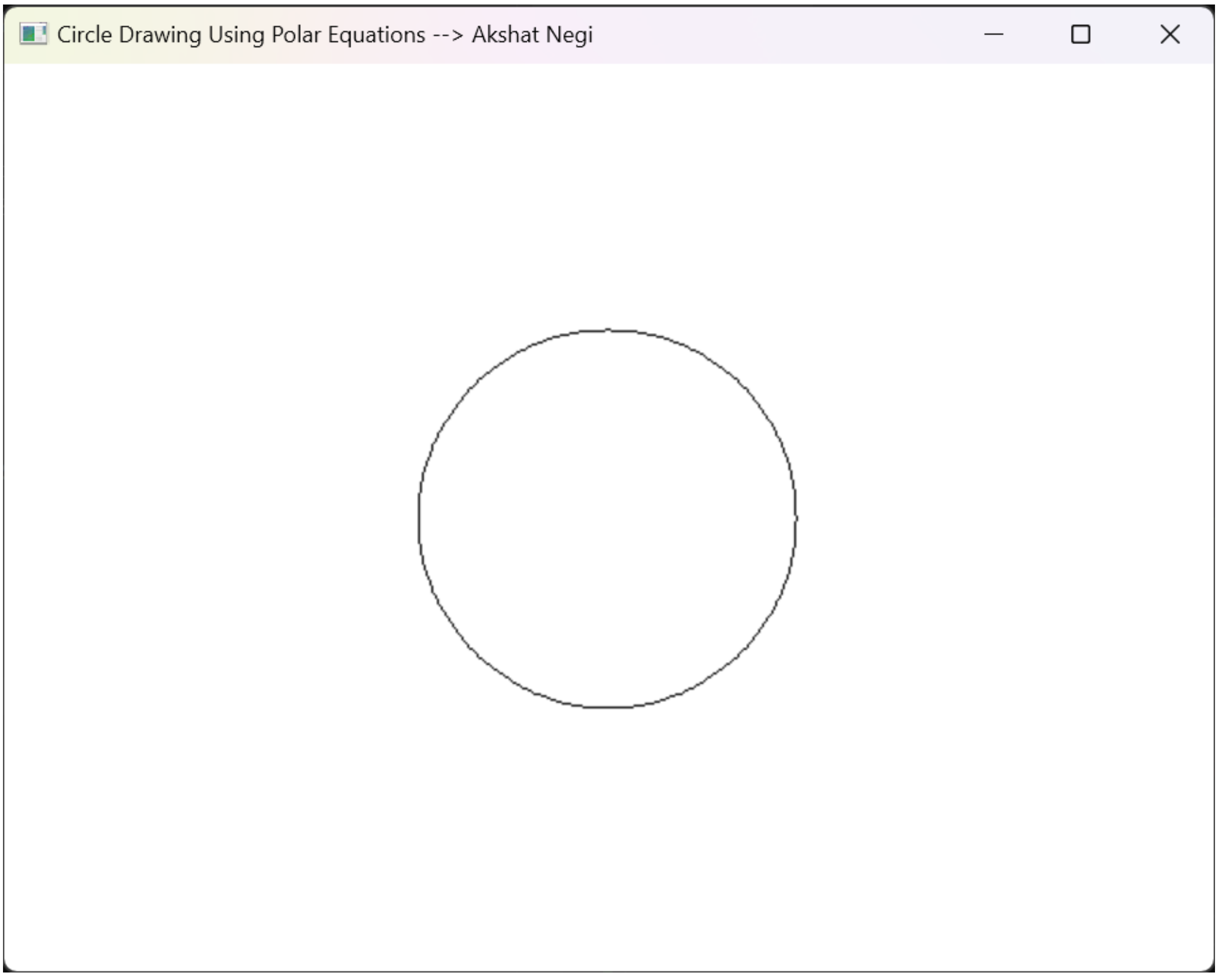
[Usage of Open GL]

 # Take the value of radius, major axis and minor axis as input from the user.

a) Draw the circle with the help of polar equations

```
#include <GL/freeglut.h>
#include <iostream>
#include <cmath>
#include <math.h>
# define M_PI           3.14159265358979323846  /* pi */
using namespace std;

int radius = 100;
int centerX = 320;
int centerY = 240;

void init() {
    glClearColor(1.0, 1.0, 1.0, 1.0);
    gluOrtho2D(0, 640, 0, 480);
}

void drawCirclePolarEquation() {
    glBegin(GL_LINE_LOOP);
    for (double angle = 0; angle <= 360; angle += 1) {
        double x = centerX + radius * cos(angle * M_PI / 180);
        double y = centerY + radius * sin(angle * M_PI / 180);
        glVertex2i(x, y);
    }
    glEnd();
}

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

    // Draw the circle using polar equations
    drawCirclePolarEquation();

    glFlush();
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Circle Drawing Using Polar Equations --> Akshat Negi");
    init();
    glutDisplayFunc(display);
```

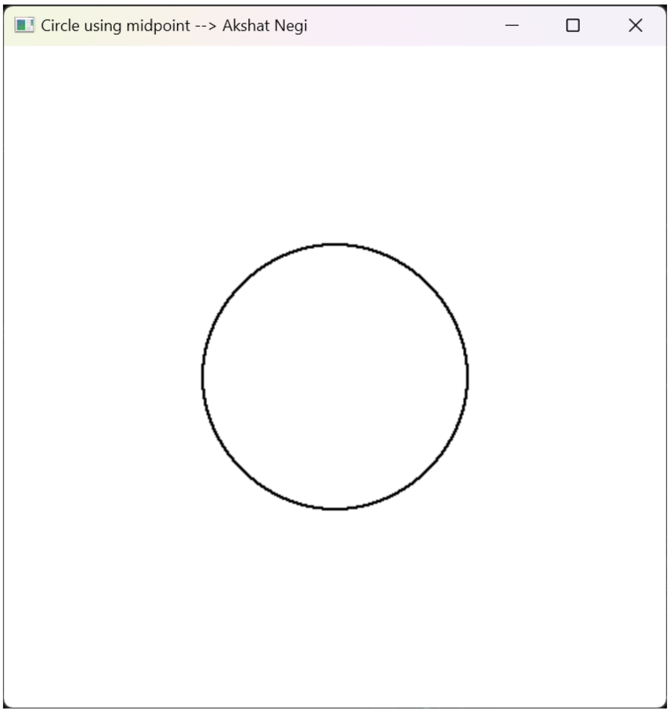
```
 glutMainLoop();
 return 0;
}
```



b) Draw the circle with the help of mid-point method.

```
#include <iostream>
#include<math.h>
#include<GL/freeglut.h>

using namespace std;

void circle() {
	glColor3f(0.0, 0.0, 0.0);
	glPointSize(2.0);
	float r = 100;
	float x = 0, y = r;
	float p = 1 - r;
	glBegin(GL_POINTS);
	while (x != y)
	{
		x++;
```

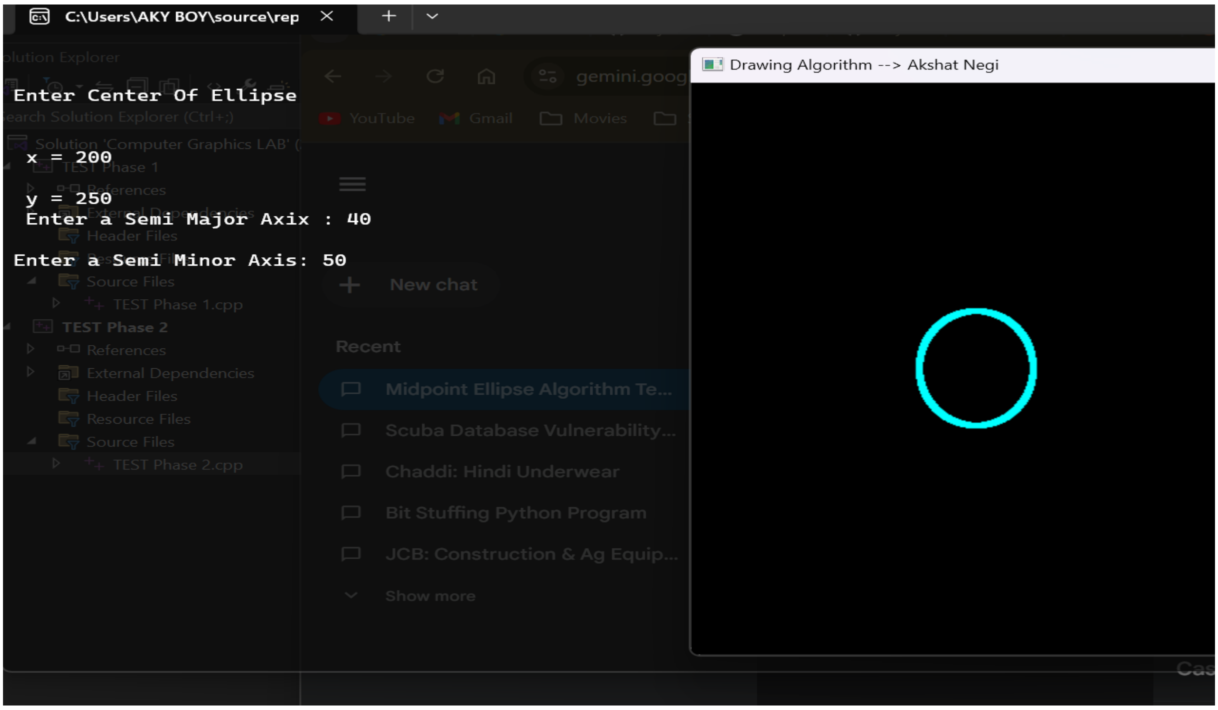
```
            if (p < 0) {
                p += 2 * (x + 1) + 1;
            }
            else {
                y--;
                p += 2 * (x + 1) + 1 - 2 * (y - 1);
            }
            glVertex2i(x, y);
            glVertex2i(-x, y);
            glVertex2i(x, -y);
            glVertex2i(-x, -y);

            glVertex2i(y, x);
            glVertex2i(-y, x);
            glVertex2i(y, -x);
            glVertex2i(-y, -x);
        }
        glEnd();
        glFlush();
}
int main(int argc, char** argv) {
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(500, 500);
        glutInitWindowPosition(100, 100);
        glutCreateWindow("Circle using midpoint --> Akshat Negi");
        glClearColor(1.0, 1.0, 1.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT);
        gluOrtho2D(-250, 250, -250, 250);
        glMatrixMode(GL_PROJECTION);
        glViewport(0, 0, 500, 500);
        glutDisplayFunc(circle);
        glutMainLoop();
        return 0;
}
```



c) Draw the Ellipse with the mid-point method.

```
#include <GL/freeglut.h>
#include <iostream>
using namespace std;
int rx, ry;
int xi, yi;

void ellipseMidPoint() {
	int x = 0, y = ry;
	int p1 = (ry * ry) + (rx * rx * 0.25) - (ry * rx * rx);
	int dx = 2 * x * (ry * ry);
	int dy = 2 * y * (rx * rx);
	while (dy > dx) {
		glVertex2i(x + xi, y + yi);
		glVertex2i(x + xi, -y + yi);
		glVertex2i(-x + xi, -y + yi);
		glVertex2i(-x + xi, y + yi);
		if (p1 < 0) {
			x++;
			dx = 2 * x * (ry * ry);
			p1 += dx + (ry * ry);
		}
		else {
			x++;
			y--;
			dx = 2 * x * (ry * ry);
			dy = 2 * y * (rx * rx);
			p1 += dx + (ry * ry) - dy;
		}
	}
	int p2 = (ry * ry * (x + 0.5) * (x + 0.5)) + (rx * rx * (y - 1) * (y - 1))
- (rx * rx * ry * ry);
	while (y > 0) {
		glVertex2i(x + xi, y + yi);
		glVertex2i(x + xi, -y + yi);
		glVertex2i(-x + xi, -y + yi);
		glVertex2i(-x + xi, y + yi);
		if (p2 > 0) {
			y--;
			dy = 2 * y * (rx * rx);
			p2 += (rx * rx) - dy;
		}
		else {
			y--;
			x++;
			dy -= 2 * (rx * rx);
			dx += 2 * (ry * ry);
			p2 += dx + (rx * rx) - dy;
		}
	}
}

void display() {
	//glClear(GL_COLOR_BUFFER_BIT); already mentioned in main program
	glColor3f(0.0, 1.0, 1.0);
	glPointSize(5.0);
	glBegin(GL_POINTS);
	//int rx = 40, ry = 50, xi = 200, yi = 250;
	ellipseMidPoint();
```

```
        glEnd();
        glFlush();
}
int main(int argc, char** argv)
{
        cout << "\n\nEnter Center Of Ellipse  \n\n";
        cout << "\n x = ";
        cin >> xi;

        cout << "\n y = ";
        cin >> yi;

        cout << " Enter a Semi Major Axix : ";
        cin >> rx;
        cout << " \nEnter a Semi Minor Axis: ";
        cin >> ry;


        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(500, 500);
        glutInitWindowPosition(100, 100);
        glutCreateWindow("Drawing Algorithm --> Akshat Negi");
        glClearColor(0.0, 0.0, 0.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT);
        gluOrtho2D(0, 500, 0, 500);
        glMatrixMode(GL_PROJECTION);
        glViewport(0, 0, 500, 500);
        glutDisplayFunc(display);
        glutMainLoop();
        return 0;
}
```



LAB EXPERIMENT – 4

Seed Fill Algorithms

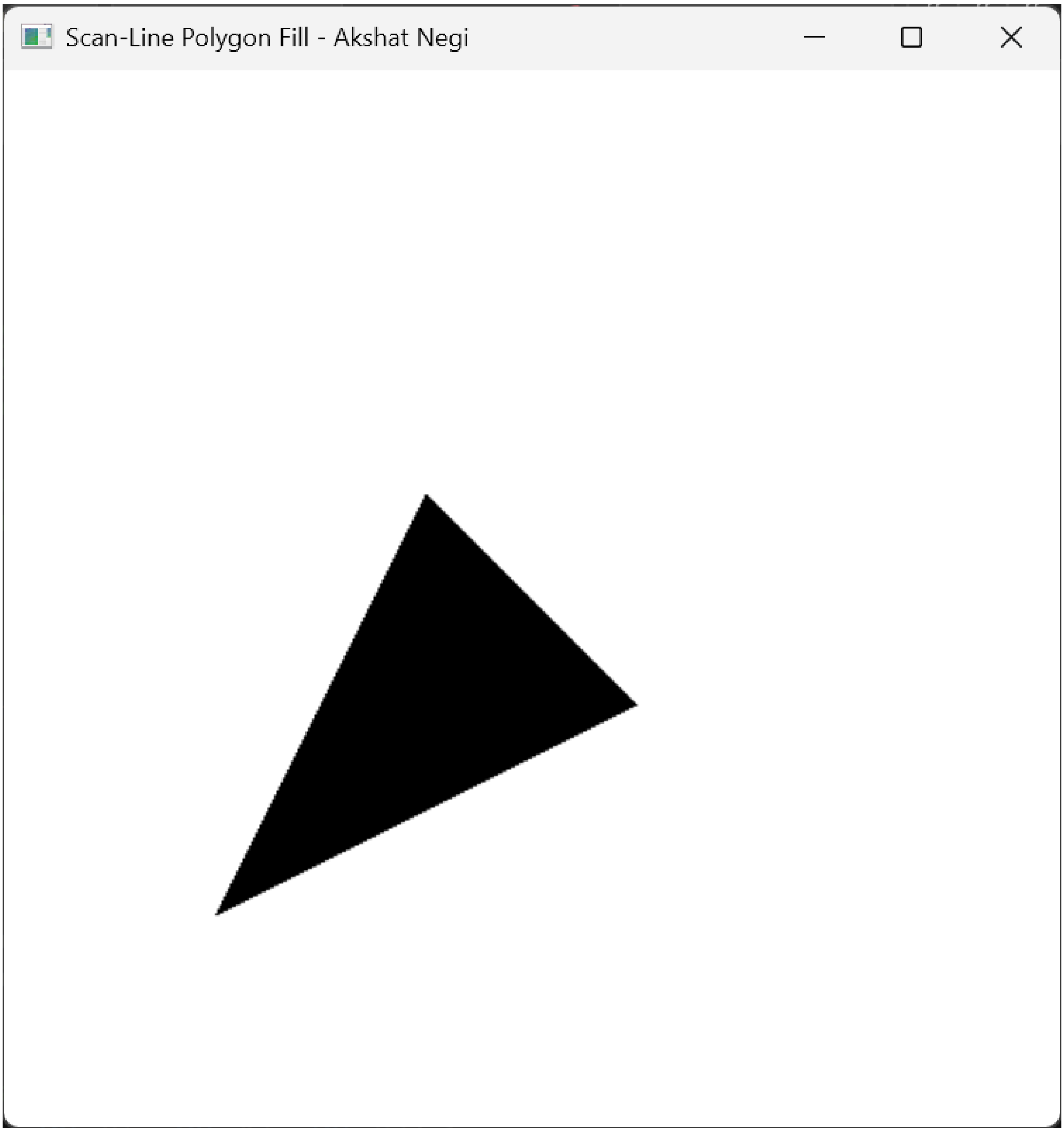
[Small Project will be given for demonstration]

 # Take the value of seed point, intensity of new color as input from user.

a. WAP to fill the polygon using scan lines.

```
#include <GL/freeglut.h>
#include <iostream>
#include <vector>
#include <algorithm>

// Global Variables
std::vector<int> x_coords;
std::vector<int> y_coords;
int edges;

// Function to draw a line between two points
void drawLine(int x1, int y1, int x2, int y2) {
    glBegin(GL_LINES);
    glVertex2i(x1, y1);
    glVertex2i(x2, y2);
    glEnd();
    glFlush();
}

// Function to implement scan-line polygon filling
void scanFill()
{
    int i, j, temp;
    int xmin = *std::min_element(x_coords.begin(), x_coords.end());
    int xmax = *std::max_element(x_coords.begin(), x_coords.end());

    // Scan each scan-line within the polygon's vertical extent
    for (i = xmin; i <= xmax; i++) {
        // Initialize an array to store the intersection points
        std::vector<int> interPoints;

        for (j = 0; j < edges; j++) {
            int next = (j + 1) % edges;

            // Check if the current edge intersects with the scan line
            if ((y_coords[j] > i && y_coords[next] <= i) || (y_coords[next] > i
&& y_coords[j] <= i)) {
                int interX = x_coords[j] + (i - y_coords[j]) * (x_coords[next] -
x_coords[j]) / (y_coords[next] - y_coords[j]);
                interPoints.push_back(interX);
            }
        }

        // Sort the intersection points in ascending order
        std::sort(interPoints.begin(), interPoints.end());
```

```
        // Fill the pixels between pairs of intersection points
        for (j = 0; j < interPoints.size(); j += 2) {
            if (j + 1 < interPoints.size()) {
                drawLine(interPoints[j], i, interPoints[j + 1], i);
            }
        }
    }
}

// Display callback for OpenGL
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    scanFill();
    glFlush();
}
// Function to initialize OpenGL

void init() {
    // Set the background color to white and the drawing color to black
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glColor3f(0.0, 0.0, 0.0);

    // Set up 2D orthographic projection with the window size
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(0.0, 500.0, 0.0, 500.0);  // Adjust window size as needed
}

int main(int argc, char** argv) {
    // Define the polygon vertices
    x_coords = { 100, 200, 300 };
    y_coords = { 100, 300, 200 };
    edges = 3;

    // Initialize GLUT
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);   // Window size
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Scan-Line Polygon Fill - Akshat Negi");

    init();  // Set up OpenGL

    // Register the display callback function
    glutDisplayFunc(display);

    // Enter the GLUT main loop
    glutMainLoop();

    return 0;
}
```

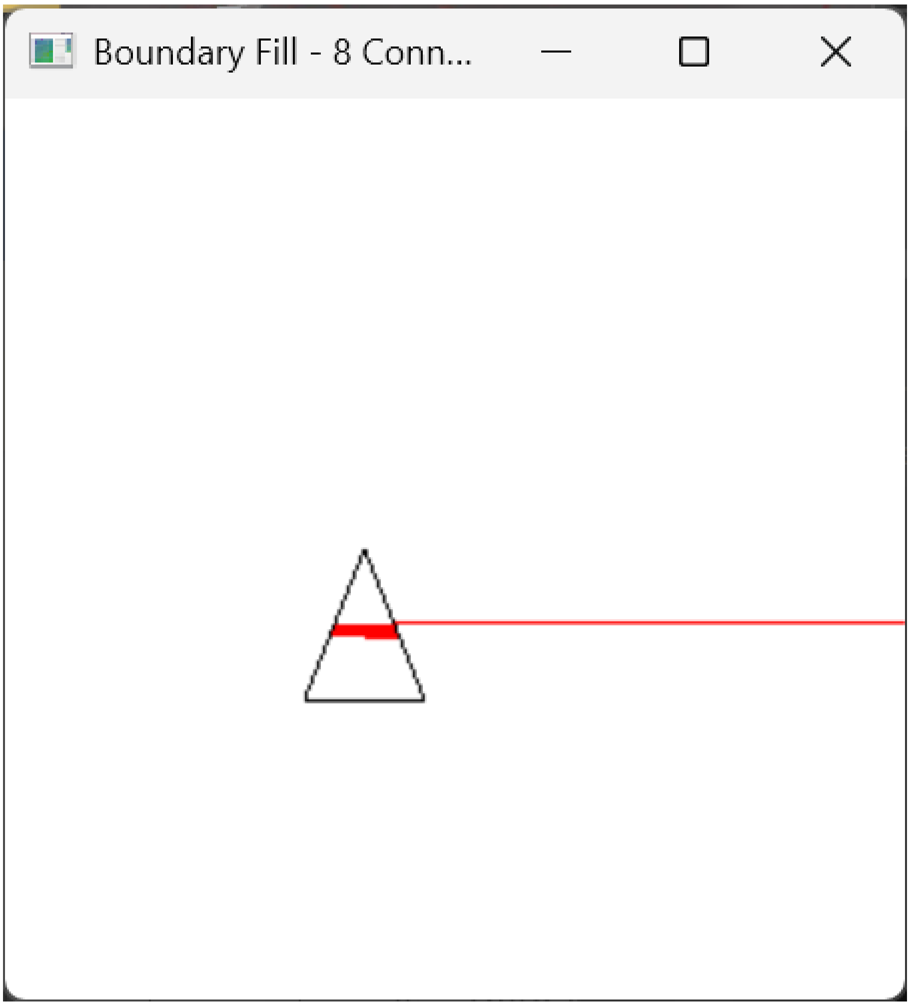


b. WAP to fill a region using boundary fill algorithm using 4 or 8 connected approaches.

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

float fillColor[3] = { 1.0, 0.0, 0.0 };  // Red color for filling
float borderColor[3] = { 0.0, 0.0, 0.0 }; // Black color for the boundary
float epsilon = 0.001;  // Tolerance for color comparison

// Function to set a pixel with a specific color
void setPixel(int x, int y, float* color) {
    glColor3fv(color);
```

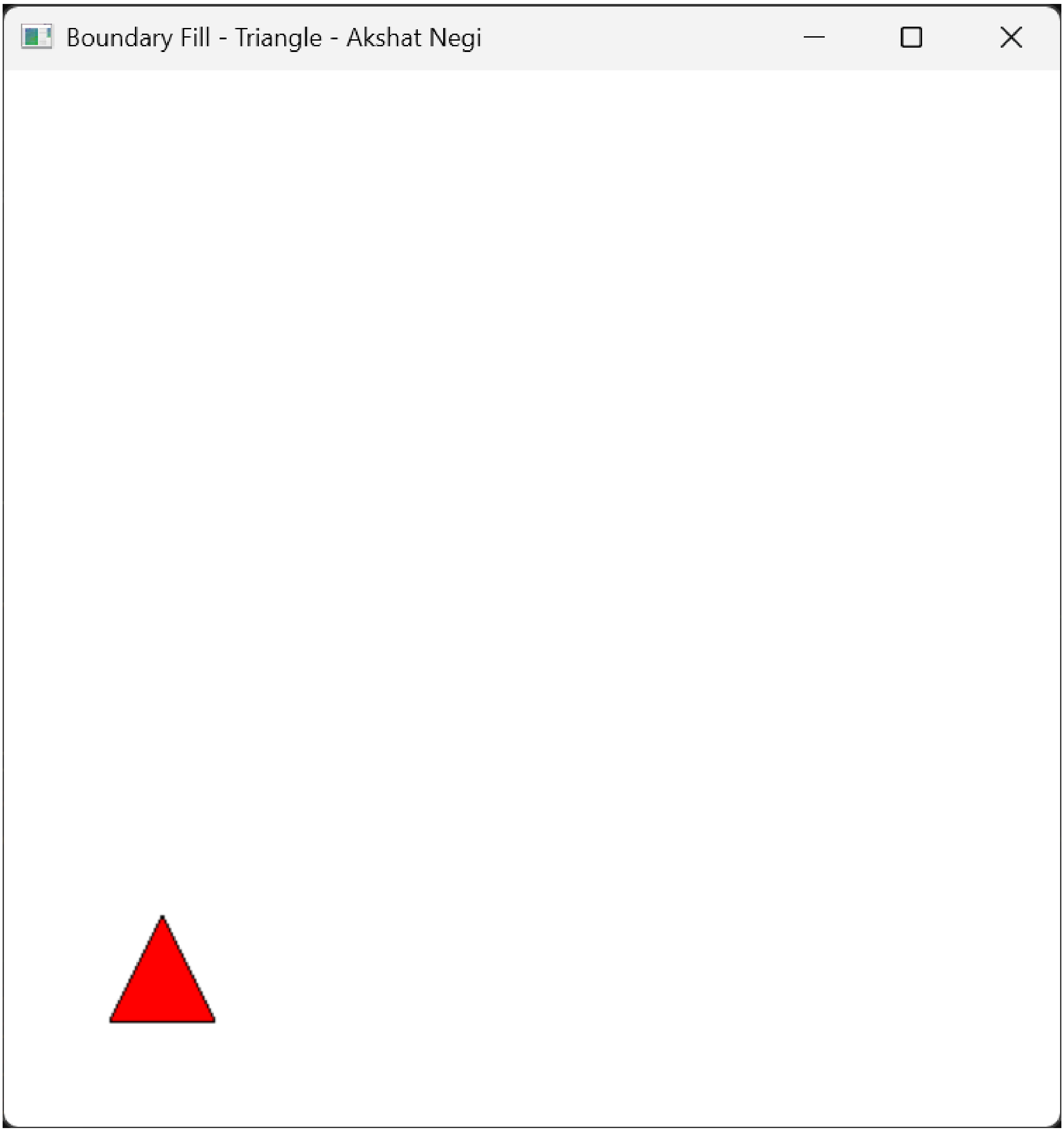
```
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
    glFlush();
}

// Function to get the color of a pixel at coordinates (x, y)
void getPixelColor(int x, int y, float* color) {
    glReadPixels(x, y, 1, 1, GL_RGB, GL_FLOAT, color);
}

// Helper function to compare two colors with a tolerance
bool isSameColor(float* color1, float* color2) {
    return (fabs(color1[0] - color2[0]) < epsilon &&
        fabs(color1[1] - color2[1]) < epsilon &&
        fabs(color1[2] - color2[2]) < epsilon);
}

// Boundary Fill Algorithm (8-connected)
void boundaryFill(int x, int y, float* fillColor, float* boundaryColor) {
    float currentColor[3];
    getPixelColor(x, y, currentColor);

    // If the pixel is neither the boundary nor the fill color, fill it
    if (!isSameColor(currentColor, boundaryColor) && !isSameColor(currentColor,
fillColor)) {
        setPixel(x, y, fillColor);

        // 8-connected neighbors
        boundaryFill(x + 1, y, fillColor, boundaryColor); // Right
        boundaryFill(x - 1, y, fillColor, boundaryColor); // Left
        boundaryFill(x, y + 1, fillColor, boundaryColor); // Up
        boundaryFill(x, y - 1, fillColor, boundaryColor); // Down
        boundaryFill(x + 1, y + 1, fillColor, boundaryColor); // Up-Right
        boundaryFill(x - 1, y + 1, fillColor, boundaryColor); // Up-Left
        boundaryFill(x + 1, y - 1, fillColor, boundaryColor); // Down-Right
        boundaryFill(x - 1, y - 1, fillColor, boundaryColor); // Down-Left
    }
}

// Function to draw a smaller triangle
void drawTriangle() {
    glColor3fv(borderColor); // Set border color (black)
    glBegin(GL_LINE_LOOP);
    glVertex2i(120, 150); // Top vertex
    glVertex2i(100, 100); // Bottom-left vertex
    glVertex2i(140, 100); // Bottom-right vertex
    glEnd();
    glFlush();
}

void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    drawTriangle();  // Draw triangle on screen

    // Starting the boundary fill from a point inside the triangle
    boundaryFill(120, 120, fillColor, borderColor);
}

void init() {
    glClearColor(1.0, 1.0, 1.0, 1.0); // Set background color to white
    glColor3f(0.0, 0.0, 0.0);         // Set drawing color to black
    gluOrtho2D(0, 300, 0, 300);       // Set the coordinate system for the window
```

```
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(300, 300);    // Decrease window size
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Boundary Fill - 8 Connected Triangle - Akshat Negi");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```



```cpp
#include <GL/freeglut.h>
#include <iostream>
#include <cmath>

float fillColor[3] = { 1.0, 0.0, 0.0 };  // Red color for filling
float borderColor[3] = { 0.0, 0.0, 0.0 }; // Black color for the boundary
float epsilon = 0.001;  // Tolerance for color comparison

// Function to set a pixel with a specific color
void setPixel(int x, int y, float* color) {
    glColor3fv(color);
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
    glFlush();
}

// Function to get the color of a pixel at coordinates (x, y)
void getPixelColor(int x, int y, float* color) {
    glReadPixels(x, y, 1, 1, GL_RGB, GL_FLOAT, color);
}

// Helper function to compare two colors with a tolerance
bool isSameColor(float* color1, float* color2) {
    return (fabs(color1[0] - color2[0]) < epsilon &&
        fabs(color1[1] - color2[1]) < epsilon &&
        fabs(color1[2] - color2[2]) < epsilon);
}

// Boundary Fill Algorithm (4-connected)
void boundaryFill(int x, int y, float* fillColor, float* boundaryColor) {
    float currentColor[3];
    getPixelColor(x, y, currentColor);

    // If the pixel is neither the boundary nor the fill color, fill it
    if (!isSameColor(currentColor, boundaryColor) && !isSameColor(currentColor,
fillColor)) {
        setPixel(x, y, fillColor);

        boundaryFill(x + 1, y, fillColor, boundaryColor);
        boundaryFill(x - 1, y, fillColor, boundaryColor);
        boundaryFill(x, y + 1, fillColor, boundaryColor);
        boundaryFill(x, y - 1, fillColor, boundaryColor);
    }
}

// Function to draw a triangle
void drawTriangle() {
    glColor3fv(borderColor); // Set border color (black)
    glBegin(GL_LINE_LOOP);
    glVertex2i(50, 50);  // Vertex 1 (Bottom-left corner)
    glVertex2i(100, 50); // Vertex 2 (Bottom-right corner)
    glVertex2i(75, 100); // Vertex 3 (Top corner)
    glEnd();
    glFlush();
}

void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    drawTriangle();  // Draw triangle on screen

    // Starting the boundary fill from a point inside the triangle
    boundaryFill(75, 60, fillColor, borderColor); // Adjusted point for filling
```

```
}

void init() {
    glClearColor(1.0, 1.0, 1.0, 1.0); // Set background color to white
    glColor3f(0.0, 0.0, 0.0);         // Set drawing color to black
    gluOrtho2D(0, 500, 0, 500);       // Set the coordinate system for the window
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Boundary Fill - Triangle - Akshat Negi");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

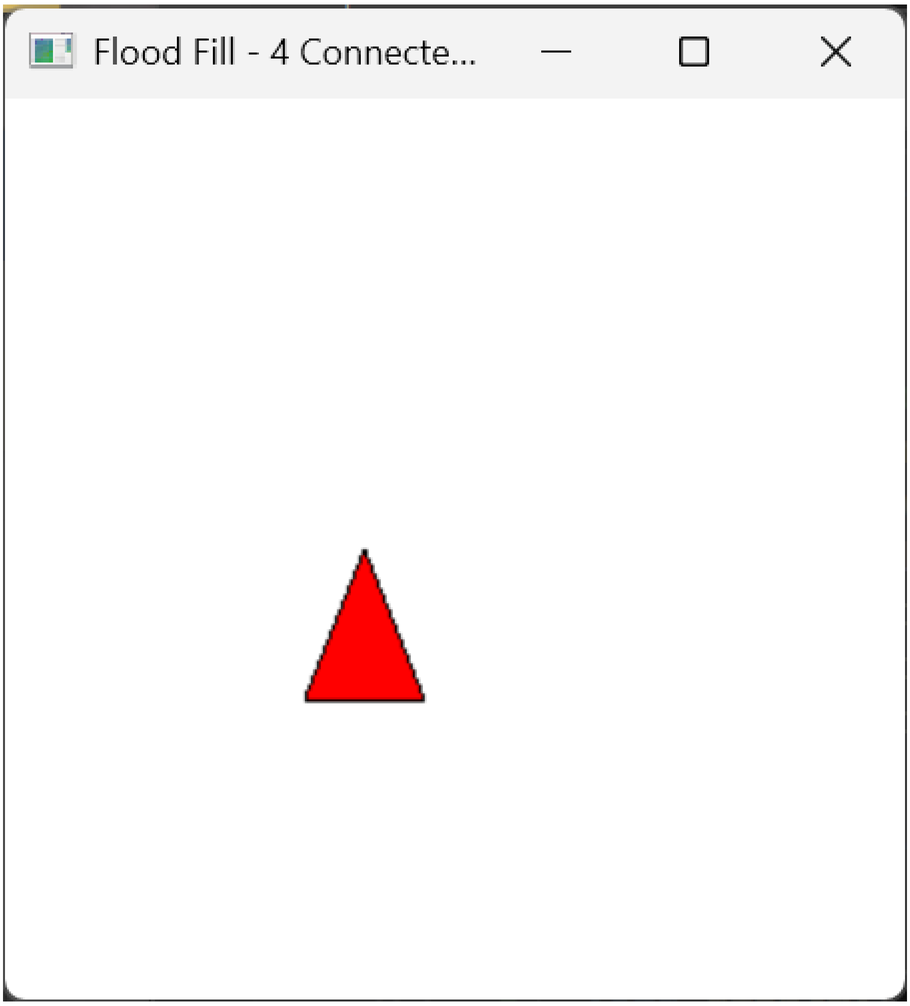


c. WAP to fill a region using flood fill algorithm using 4 or 8 connected approaches.

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

float fillColor[3] = { 1.0, 0.0, 0.0 };  // Red color for filling
float borderColor[3] = { 0.0, 0.0, 0.0 }; // Black color for the boundary
float epsilon = 0.001;  // Tolerance for color comparison

// Function to set a pixel with a specific color
void setPixel(int x, int y, float* color) {
    glColor3fv(color);
    glBegin(GL_POINTS);
```

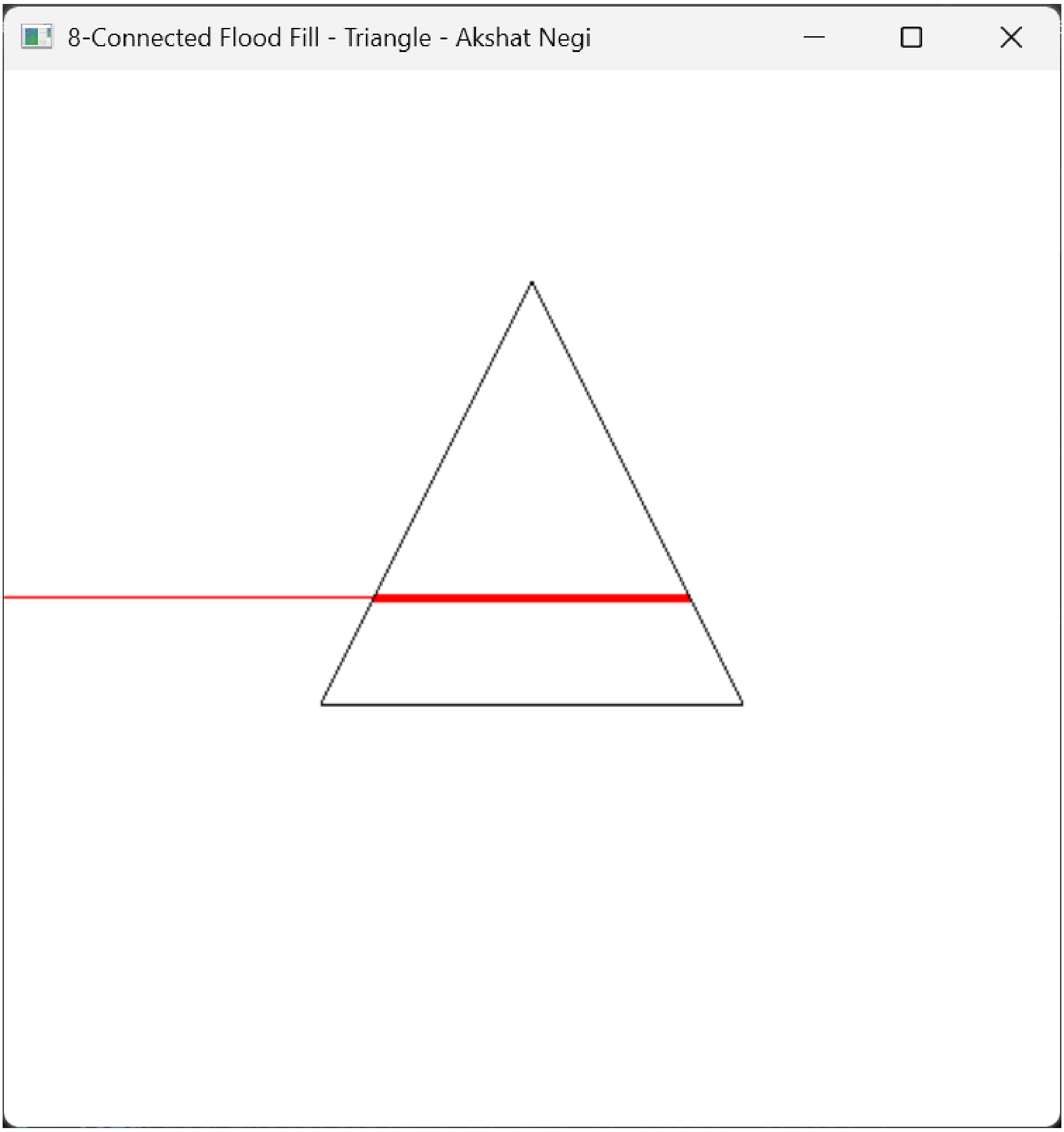
```
    glVertex2i(x, y);
    glEnd();
    glFlush();
}

// Function to get the color of a pixel at coordinates (x, y)
void getPixelColor(int x, int y, float* color) {
    glReadPixels(x, y, 1, 1, GL_RGB, GL_FLOAT, color);
}

// Helper function to compare two colors with a tolerance
bool isSameColor(float* color1, float* color2) {
    return (fabs(color1[0] - color2[0]) < epsilon &&
        fabs(color1[1] - color2[1]) < epsilon &&
        fabs(color1[2] - color2[2]) < epsilon);
}

// 4-Connected Flood Fill Algorithm
void floodFill(int x, int y, float* fillColor, float* boundaryColor) {
    float currentColor[3];
    getPixelColor(x, y, currentColor);

    // If the pixel is neither the boundary nor the fill color, fill it
    if (!isSameColor(currentColor, boundaryColor) && !isSameColor(currentColor,
fillColor)) {
        setPixel(x, y, fillColor);

        // 4-connected neighbors
        floodFill(x + 1, y, fillColor, boundaryColor); // Right
        floodFill(x - 1, y, fillColor, boundaryColor); // Left
        floodFill(x, y + 1, fillColor, boundaryColor); // Up
        floodFill(x, y - 1, fillColor, boundaryColor); // Down
    }
}

// Function to draw a smaller triangle
void drawTriangle() {
    glColor3fv(borderColor); // Set border color (black)
    glBegin(GL_LINE_LOOP);
    glVertex2i(120, 150); // Top vertex
    glVertex2i(100, 100); // Bottom-left vertex
    glVertex2i(140, 100); // Bottom-right vertex
    glEnd();
    glFlush();
}

void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    drawTriangle();  // Draw triangle on screen

    // Starting the flood fill from a point inside the triangle
    floodFill(120, 120, fillColor, borderColor);
}

void init() {
    glClearColor(1.0, 1.0, 1.0, 1.0); // Set background color to white
    glColor3f(0.0, 0.0, 0.0);         // Set drawing color to black
    gluOrtho2D(0, 300, 0, 300);       // Set the coordinate system for the window
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
```

```
    glutInitWindowSize(300, 300);     // Decrease window size
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Flood Fill - 4 Connected Triangle - Akshat Negi");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```



```cpp
#include <GL/freeglut.h>
#include <iostream>
#include <cmath>

float fillColor[3] = { 1.0, 0.0, 0.0 };  // Red color for filling
float borderColor[3] = { 0.0, 0.0, 0.0 }; // Black color for the boundary
float epsilon = 0.001;  // Tolerance for color comparison

// Function to set a pixel with a specific color
void setPixel(int x, int y, float* color) {
    glColor3fv(color);
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
    glFlush();
}

// Function to get the color of a pixel at coordinates (x, y)
void getPixelColor(int x, int y, float* color) {
    glReadPixels(x, y, 1, 1, GL_RGB, GL_FLOAT, color);
}

// Helper function to compare two colors with a tolerance
bool isSameColor(float* color1, float* color2) {
    return (fabs(color1[0] - color2[0]) < epsilon &&
        fabs(color1[1] - color2[1]) < epsilon &&
        fabs(color1[2] - color2[2]) < epsilon);
}

// Flood Fill Algorithm (8-connected)
void floodFill(int x, int y, float* fillColor, float* boundaryColor) {
    float currentColor[3];
    getPixelColor(x, y, currentColor);

    // If the pixel is neither the boundary nor the fill color, fill it
    if (!isSameColor(currentColor, boundaryColor) && !isSameColor(currentColor,
fillColor)) {
        setPixel(x, y, fillColor);

        // Recursively call floodFill for 8-connected neighbors
        floodFill(x + 1, y, fillColor, boundaryColor); // Right
        floodFill(x - 1, y, fillColor, boundaryColor); // Left
        floodFill(x, y + 1, fillColor, boundaryColor); // Up
        floodFill(x, y - 1, fillColor, boundaryColor); // Down
        floodFill(x + 1, y + 1, fillColor, boundaryColor); // Top-Right
        floodFill(x - 1, y - 1, fillColor, boundaryColor); // Bottom-Left
        floodFill(x + 1, y - 1, fillColor, boundaryColor); // Bottom-Right
        floodFill(x - 1, y + 1, fillColor, boundaryColor); // Top-Left
    }
}

// Function to draw a triangle
void drawTriangle() {
    glColor3fv(borderColor); // Set border color (black)
    glBegin(GL_LINE_LOOP);
    glVertex2i(250, 400); // Top vertex
    glVertex2i(150, 200); // Bottom-left vertex
    glVertex2i(350, 200); // Bottom-right vertex
    glEnd();
    glFlush();
}

void display() {
```

```
    glClear(GL_COLOR_BUFFER_BIT);
    drawTriangle();  // Draw triangle on screen

    // Starting the flood fill from a point inside the triangle
    floodFill(250, 250, fillColor, borderColor);
}

void init() {
    glClearColor(1.0, 1.0, 1.0, 1.0); // Set background color to white
    glColor3f(0.0, 0.0, 0.0);         // Set drawing color to black
    gluOrtho2D(0, 500, 0, 500);       // Set the coordinate system for the window
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("8-Connected Flood Fill - Triangle - Akshat Negi");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```


8-Connected Flood Fill - Triangle - Akshat Negi

LAB EXPERIMENT – 5

Viewing and Clipping

[Geographical Animation for demonstration]

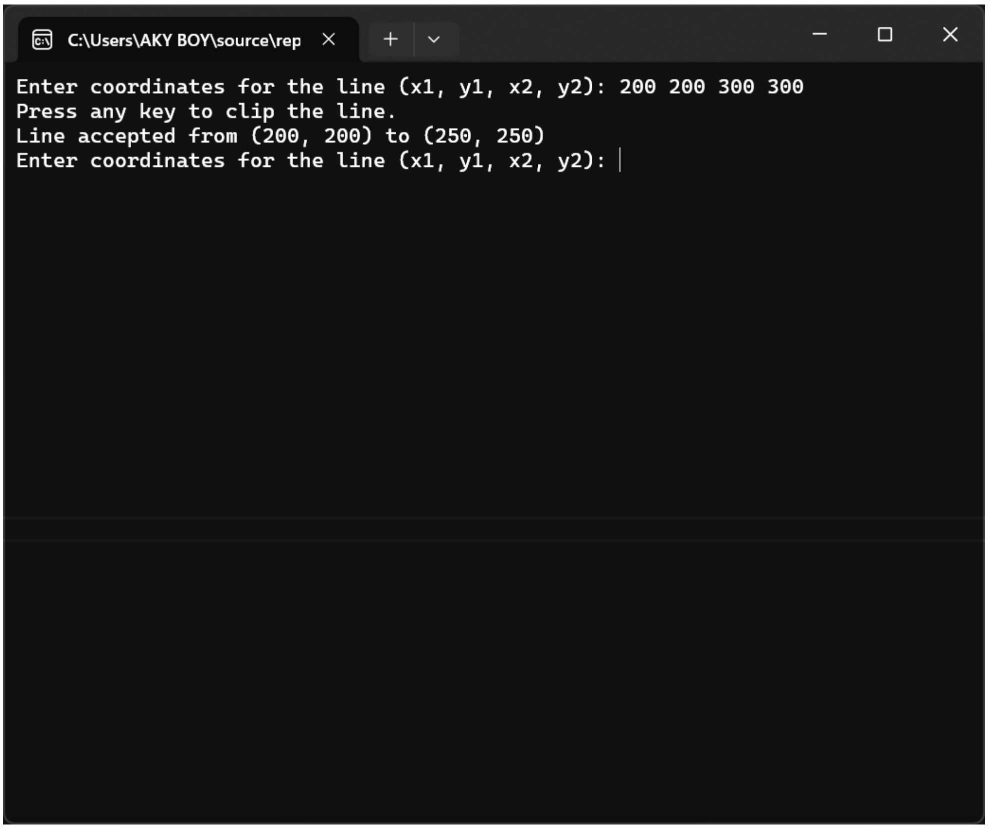
 # Take the window coordinates as input from the user, also take polygon coordinates as input.

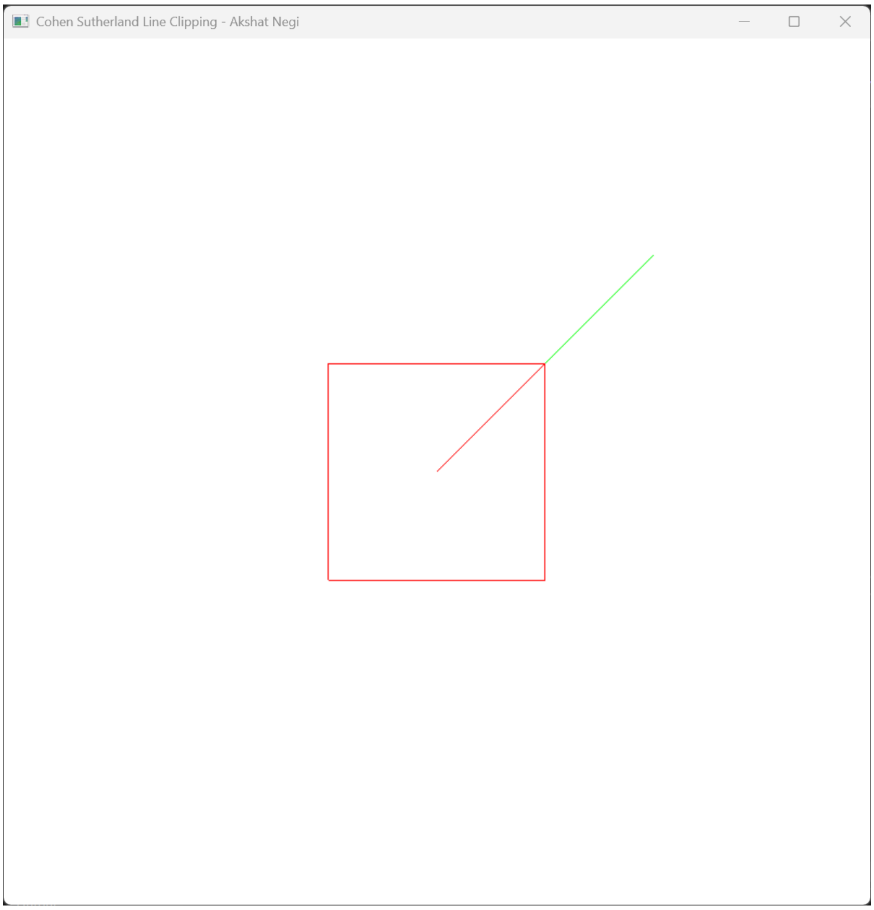
a. Write an interactive program for line clipping using Cohen Sutherland line clipping algorithm.

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

// Defining region codes
const int INSIDE = 0;  // 0000
const int LEFT = 1;    // 0001
const int RIGHT = 2;   // 0010
const int BOTTOM = 4;  // 0100
const int TOP = 8;     // 1000

// Defining x_max, y_max, x_min, y_min for clipping rectangle
const int x_max = 250;
const int y_max = 250;
const int x_min = 150;
const int y_min = 150;

// Function to compute region code for a point (x, y)
int computeCode(double x, double y) {
    int code = INSIDE;

    if (x < x_min) code |= LEFT;
    else if (x > x_max) code |= RIGHT;
    if (y < y_min) code |= BOTTOM;
    else if (y > y_max) code |= TOP;

    return code;
}

// Function to draw the clipping boundary
void drawBoundary() {
    glColor3f(1.0, 0.0, 0.0); // Red color for the boundary
    glBegin(GL_LINE_LOOP);
    glVertex2f(x_min, y_min);
    glVertex2f(x_max, y_min);
    glVertex2f(x_max, y_max);
    glVertex2f(x_min, y_max);
    glEnd();
}

// Function to draw a line with specific color
void drawLine(float x1, float y1, float x2, float y2, float r, float g, float b)
{
    glColor3f(r, g, b); // Set the color for the line
    glBegin(GL_LINES);
    glVertex2f(x1, y1);
```

```
        glVertex2f(x2, y2);
        glEnd();
    }

    // Cohen-Sutherland Line Clipping Algorithm
    void cohenSutherlandClip(double x1, double y1, double x2, double y2) {
        int code1 = computeCode(x1, y1);
        int code2 = computeCode(x2, y2);

        bool accept = false;

        while (true) {
            if ((code1 == 0) && (code2 == 0)) {
                // If both endpoints lie within the rectangle
                accept = true;
                break;
            }
            else if (code1 & code2) {
                // If both endpoints are outside the rectangle in the same region
                break;
            }
            else {
                // Some segment of the line lies within the rectangle
                int code_out;
                double x, y;

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

                // Find intersection point
                if (code_out & TOP) {
                    x = x1 + (x2 - x1) * (y_max - y1) / (y2 - y1);
                    y = y_max;
                }
                else if (code_out & BOTTOM) {
                    x = x1 + (x2 - x1) * (y_min - y1) / (y2 - y1);
                    y = y_min;
                }
                else if (code_out & RIGHT) {
                    y = y1 + (y2 - y1) * (x_max - x1) / (x2 - x1);
                    x = x_max;
                }
                else if (code_out & LEFT) {
                    y = y1 + (y2 - y1) * (x_min - x1) / (x2 - x1);
                    x = x_min;
                }

                // Replace the point outside the rectangle with the intersection
    point
                if (code_out == code1) {
                    x1 = x;
                    y1 = y;
                    code1 = computeCode(x1, y1);
                }
                else {
                    x2 = x;
                    y2 = y;
                    code2 = computeCode(x2, y2);
                }
            }
        }

        if (accept) {
```

```
        cout << "Line accepted from (" << x1 << ", " << y1 << ") to (" << x2 <<
", " << y2 << ")\n";
        drawLine(x1, y1, x2, y2, 1.0, 0.0, 0.0); // Draw the clipped line in red
    }
    else {
        cout << "Line rejected\n";
    }
}

// Function to display the content
void display() {
    glClear(GL_COLOR_BUFFER_BIT);  // Clear the screen

    drawBoundary();  // Draw the clipping boundary

    // Prompt the user for input coordinates
    double x1, y1, x2, y2;
    cout << "Enter coordinates for the line (x1, y1, x2, y2): ";
    cin >> x1 >> y1 >> x2 >> y2;

    drawLine(x1, y1, x2, y2, 0.0, 1.0, 0.0);  // Draw the original line in green

    cout << "Press any key to clip the line.\n";
    cohenSutherlandClip(x1, y1, x2, y2);  // Clip the line

    glFlush();  // Render now
}

// Function to set up OpenGL projection and modelview matrices
void init() {
    glClearColor(1.0, 1.0, 1.0, 1.0); // Set background color to white
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(0, 400, 0, 400);  // Define the 2D orthographic projection
}

int main(int argc, char** argv) {
    // Initialize GLUT
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(800, 800);  // Set the window size
    glutCreateWindow("Cohen Sutherland Line Clipping - Akshat Negi");  // Create
the window
    init();  // Initialize OpenGL state

    // Register display callback function
    glutDisplayFunc(display);

    // Enter the GLUT event processing loop
    glutMainLoop();

    return 0;
   }
```


C:\Users\AKY BOY\source\rep
Enter coordinates for the line (x1, y1, x2, y2): 200 200 300 300
Press any key to clip the line.
Line accepted from (200, 200) to (250, 250)
Enter coordinates for the line (x1, y1, x2, y2):

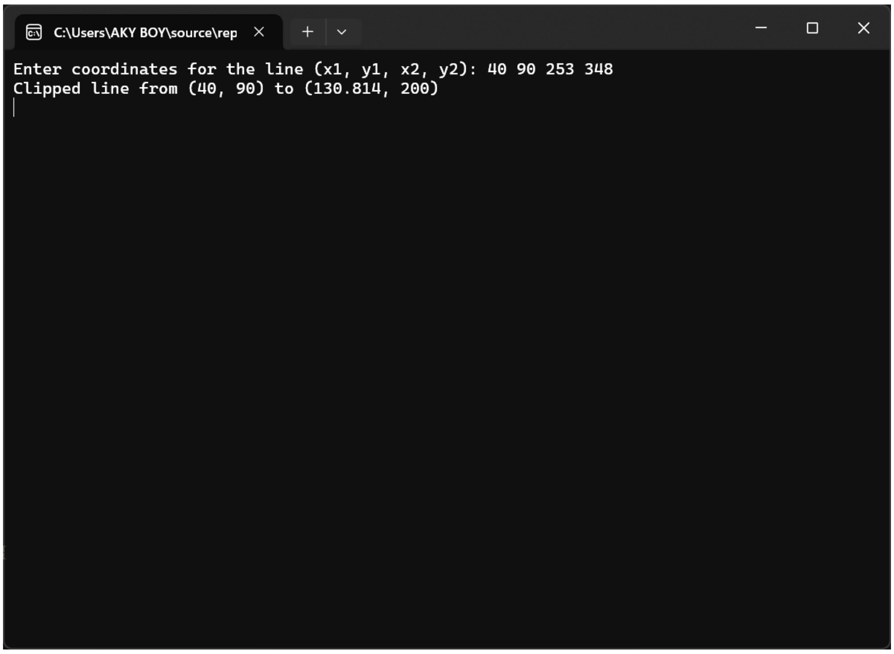

Cohen Sutherland Line Clipping - Akshat Negi

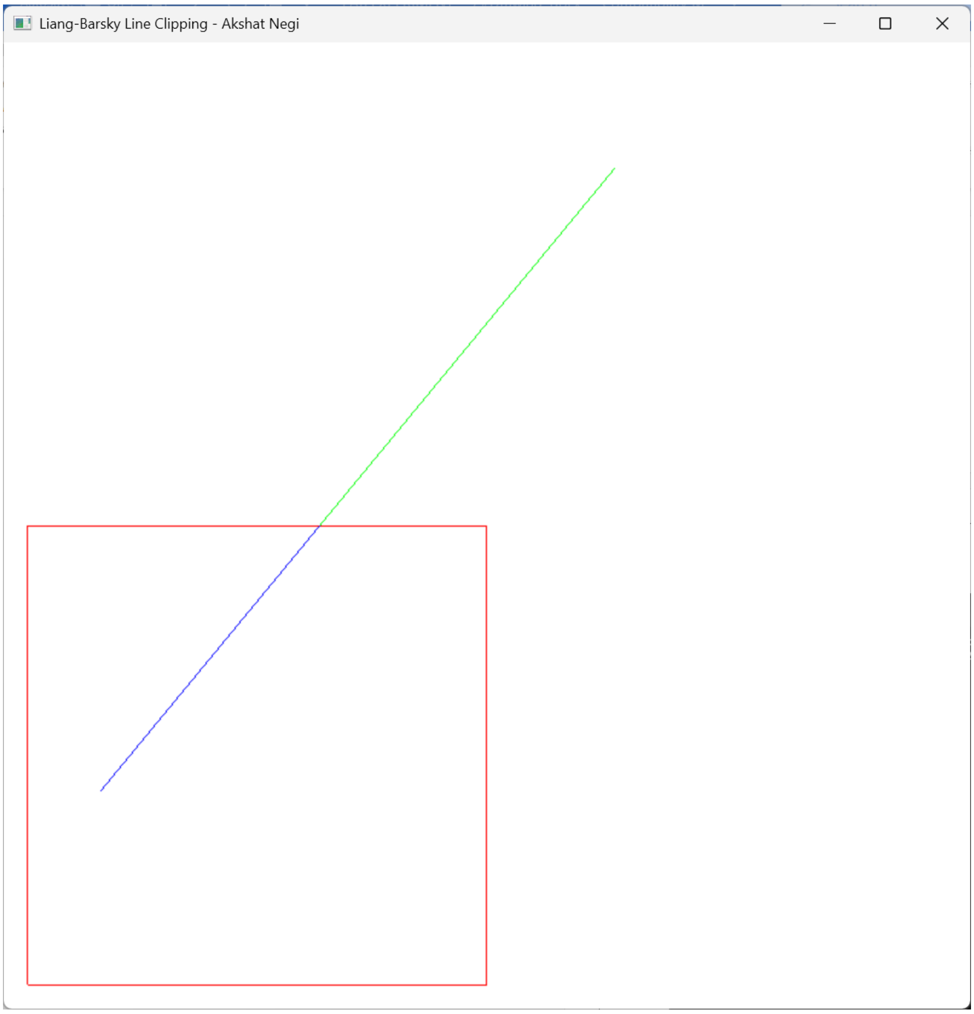
b. Write an interactive program for line clipping using Liang-Barsky line clipping algorithm.

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

// Defining the clipping window boundaries
const int x_min = 10;
const int x_max = 200;
const int y_min = 10;
const int y_max = 200;

// Function to draw a line with specified color
void drawLine(float x1, float y1, float x2, float y2, float r, float g, float b)
{
    glColor3f(r, g, b); // Set color for the line
    glBegin(GL_LINES);
    glVertex2f(x1, y1);
    glVertex2f(x2, y2);
    glEnd();
}

// Liang-Barsky Line Clipping Algorithm
bool liangBarskyClip(double& x1, double& y1, double& x2, double& y2) {
    double t0 = 0.0, t1 = 1.0;
    double dx = x2 - x1;
    double dy = y2 - y1;

    auto clipTest = [&](double p, double q) {
        if (p == 0) {
            if (q < 0) return false;
        }
        else {
            double t = q / p;
            if (p < 0) {
                if (t > t1) return false;
                if (t > t0) t0 = t;
            }
            else {
                if (t < t0) return false;
                if (t < t1) t1 = t;
            }
        }
        return true;
        };

    // Clip against each boundary
    if (!clipTest(-dx, x1 - x_min) || !clipTest(dx, x_max - x1) ||
        !clipTest(-dy, y1 - y_min) || !clipTest(dy, y_max - y1)) {
        return false;
    }

    // Update the points based on t0 and t1
    if (t1 < 1.0) {
        x2 = x1 + t1 * dx;
        y2 = y1 + t1 * dy;
    }
    if (t0 > 0.0) {
        x1 = x1 + t0 * dx;
        y1 = y1 + t0 * dy;
    }
```

```
    return true;
}

// Function to display the content
void display() {
    glClear(GL_COLOR_BUFFER_BIT);  // Clear the screen

    // Draw the clipping boundary (rectangle)
    glColor3f(1.0, 0.0, 0.0); // Red color for the boundary
    glBegin(GL_LINE_LOOP);
    glVertex2f(x_min, y_min);
    glVertex2f(x_max, y_min);
    glVertex2f(x_max, y_max);
    glVertex2f(x_min, y_max);
    glEnd();

    // Prompt the user for input coordinates
    double x1, y1, x2, y2;
    cout << "Enter coordinates for the line (x1, y1, x2, y2): ";
    cin >> x1 >> y1 >> x2 >> y2;

    // Draw the original line in green
    drawLine(x1, y1, x2, y2, 0.0, 1.0, 0.0);

    // Apply Liang-Barsky clipping
    bool isClipped = liangBarskyClip(x1, y1, x2, y2);

    if (isClipped) {
        // Draw the clipped line in blue and print clipped coordinates
        cout << "Clipped line from (" << x1 << ", " << y1 << ") to (" << x2 << ",
" << y2 << ")\n";
        drawLine(x1, y1, x2, y2, 0.0, 0.0, 1.0);  // Draw the clipped line in
blue
    }
    else {
        // Line is outside the window
        cout << "Line is outside the clipping window.\n";
    }

    glFlush();  // Render now
}

// Function to set up OpenGL projection and modelview matrices
void init() {
    glClearColor(1.0, 1.0, 1.0, 1.0); // Set background color to white
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(0, 400, 0, 400);  // Define the 2D orthographic projection
}

int main(int argc, char** argv) {
    // Initialize GLUT
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(800, 800);  // Set the window size
    glutCreateWindow("Liang-Barsky Line Clipping - Akshat Negi");  // Create the
window
    init();  // Initialize OpenGL state

    // Register display callback function
    glutDisplayFunc(display);

    // Enter the GLUT event processing loop
    glutMainLoop();
```

```
    return 0;
}
```


C:\Users\AKY BOY\source\rep
Enter coordinates for the line (x1, y1, x2, y2): 40 90 253 348
Clipped line from (40, 90) to (130.814, 200)

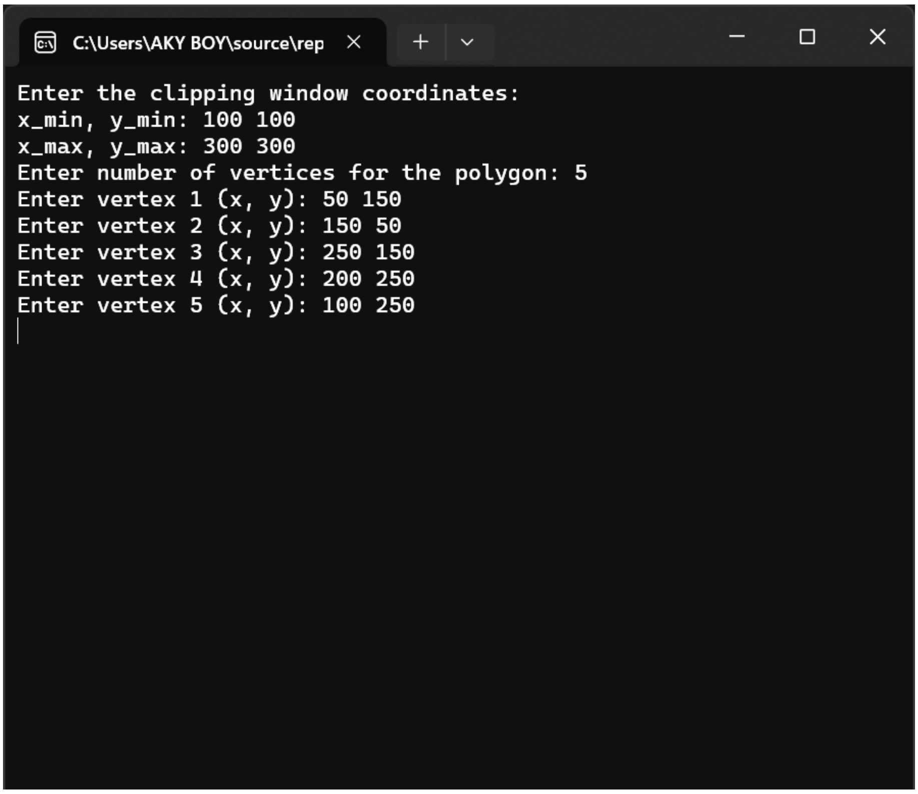

Liang-Barsky Line Clipping - Akshat Negi

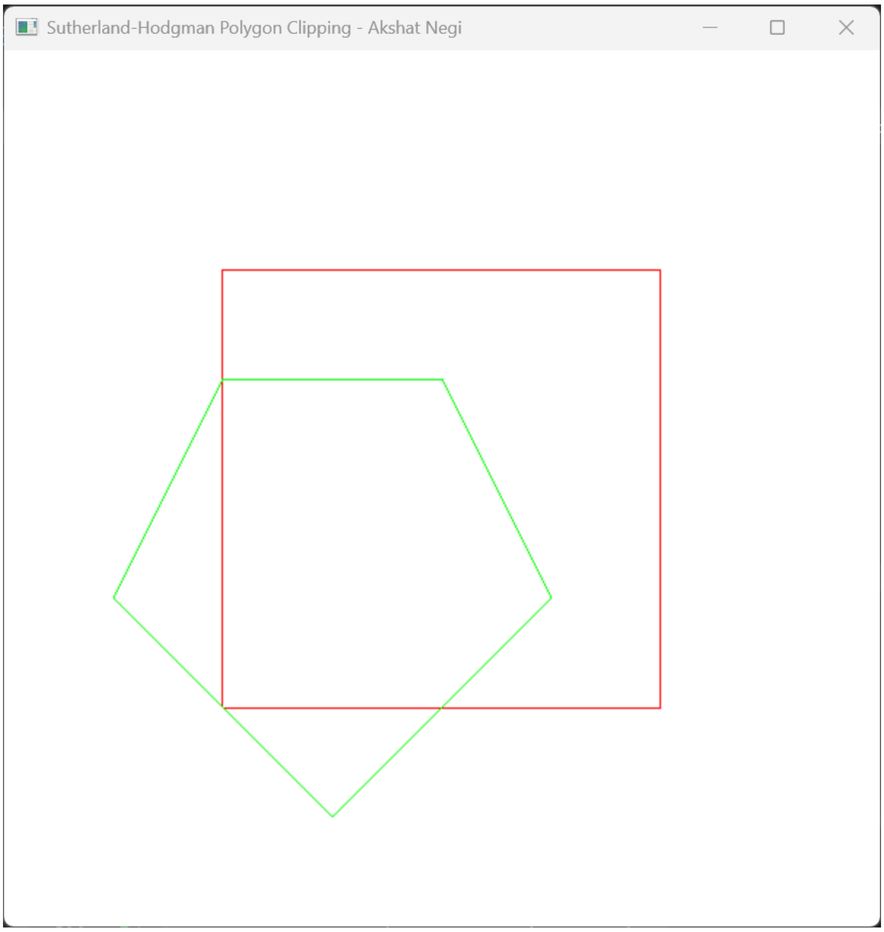
c. Write an interactive program for polygon clipping using Sutherland – Hodgeman polygon clipping algorithm.

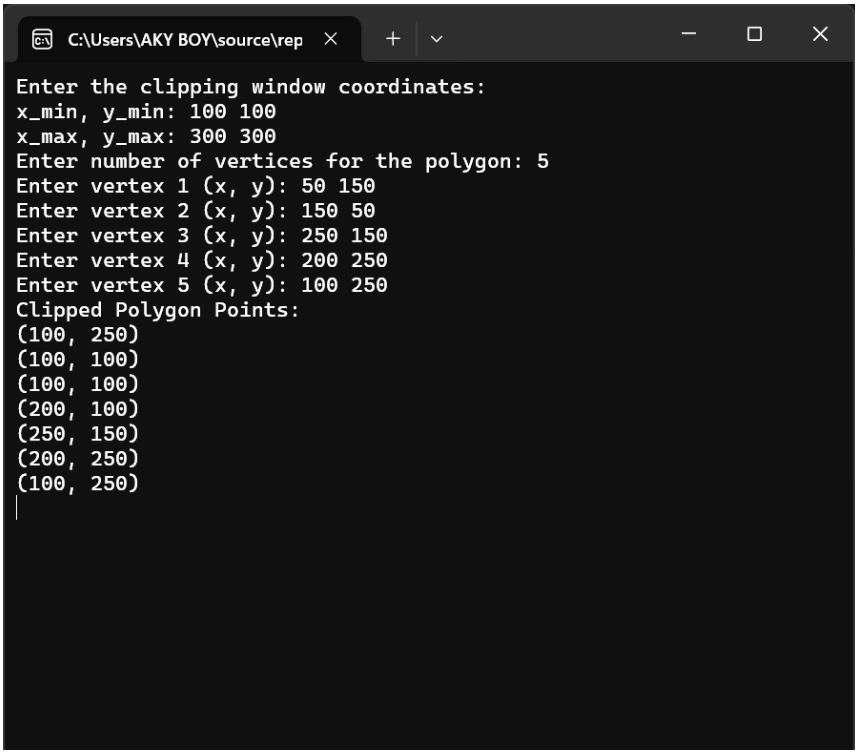
```
#include <GL/freeglut.h>
#include <iostream>
#include <vector>
using namespace std;

struct Point {
    float x, y;
};

// Global variables for the clipping window boundaries
int x_min, y_min, x_max, y_max;
vector<Point> polygon;
vector<Point> clippedPolygon;

// Function to draw a polygon
void drawPolygon(const vector<Point>& poly, float r, float g, float b) {
    glColor3f(r, g, b);
    glBegin(GL_LINE_LOOP);
    for (const auto& p : poly) {
        glVertex2f(p.x, p.y);
    }
    glEnd();
}

// Function to check if a point is inside the clipping boundary
bool inside(const Point& p, int edge) {
    switch (edge) {
    case 0: return p.x >= x_min;  // Left
    case 1: return p.x <= x_max;  // Right
    case 2: return p.y >= y_min;  // Bottom
    case 3: return p.y <= y_max;  // Top
    }
    return true;
}

// Function to compute the intersection point with a clipping edge
Point intersect(const Point& p1, const Point& p2, int edge) {
    Point intersection;
    float m;

    if (p2.x != p1.x)
        m = (p2.y - p1.y) / (p2.x - p1.x);  // Slope of the line

    switch (edge) {
    case 0: // Left edge
        intersection.x = x_min;
        intersection.y = p1.y + m * (x_min - p1.x);
        break;
    case 1: // Right edge
        intersection.x = x_max;
        intersection.y = p1.y + m * (x_max - p1.x);
        break;
    case 2: // Bottom edge
        intersection.y = y_min;
        if (p2.x != p1.x)
            intersection.x = p1.x + (y_min - p1.y) / m;
        else
            intersection.x = p1.x;
        break;
    case 3: // Top edge
```

```
        intersection.y = y_max;
        if (p2.x != p1.x)
            intersection.x = p1.x + (y_max - p1.y) / m;
        else
            intersection.x = p1.x;
        break;
    }
    return intersection;
}

// Sutherland-Hodgman Polygon Clipping Algorithm
vector<Point> sutherlandHodgmanClip(const vector<Point>& input, int edge) {
    vector<Point> output;
    Point s = input.back();  // Start with the last point

    for (const auto& e : input) {
        if (inside(e, edge)) { // Case 1: End point is inside
            if (!inside(s, edge)) // Case 1.1: Start point is outside
                output.push_back(intersect(s, e, edge));  // Add intersection
point
            output.push_back(e);  // Add end point
        }
        else if (inside(s, edge)) { // Case 2: End point is outside, start is
inside
            output.push_back(intersect(s, e, edge));  // Add intersection point
        }
        s = e;
    }
    return output;
}

// Clipping function to clip the polygon against all four edges
void clipPolygon() {
    clippedPolygon = polygon;
    for (int edge = 0; edge < 4; edge++) {
        clippedPolygon = sutherlandHodgmanClip(clippedPolygon, edge);
    }

    // Print the clipped polygon points to the console
    cout << "Clipped Polygon Points:\n";
    for (const auto& point : clippedPolygon) {
        cout << "(" << point.x << ", " << point.y << ")\n";
    }
}

// Display function to render the polygons
void display() {
    glClear(GL_COLOR_BUFFER_BIT);

    // Draw clipping window (rectangle)
    glColor3f(1.0, 0.0, 0.0);  // Red color for the boundary
    glBegin(GL_LINE_LOOP);
    glVertex2f(x_min, y_min);
    glVertex2f(x_max, y_min);
    glVertex2f(x_max, y_max);
    glVertex2f(x_min, y_max);
    glEnd();

    // Draw original polygon
    drawPolygon(polygon, 0.0f, 1.0f, 0.0f);  // Green color

    // Draw clipped polygon
    drawPolygon(clippedPolygon, 0.0f, 0.0f, 1.0f);  // Blue color
```

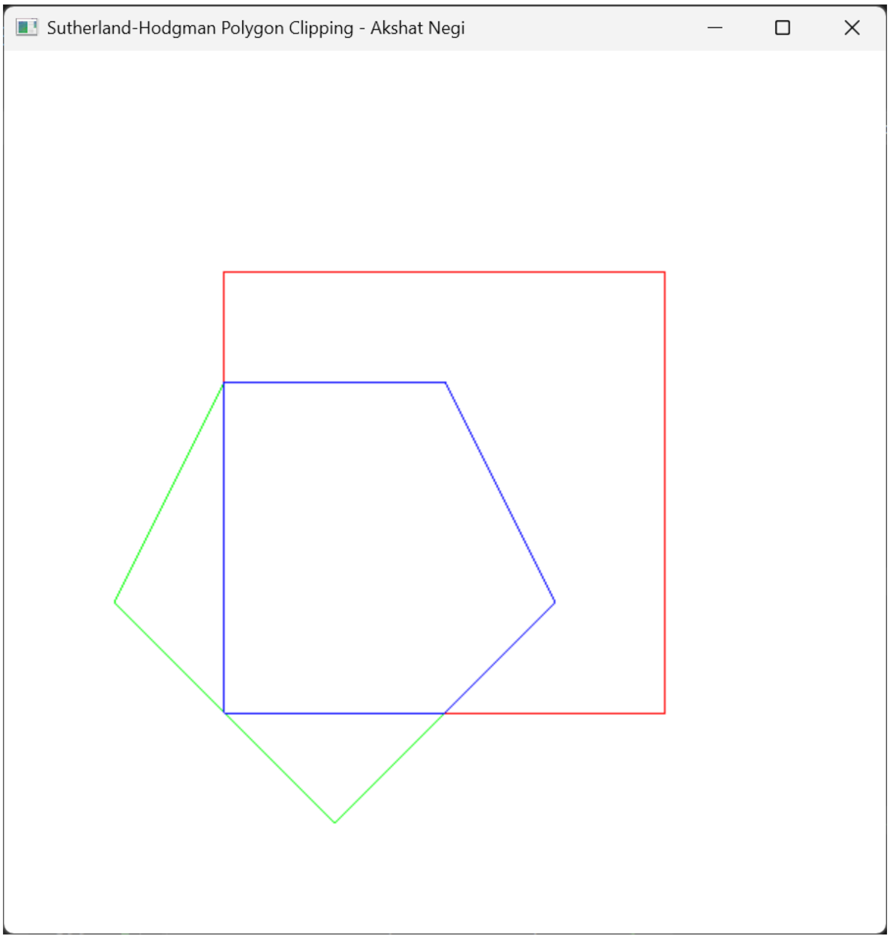
```cpp
    glFlush();
}

// Keyboard callback function
void handleKeypress(unsigned char key, int x, int y) {
    if (key == 'c') {
        clipPolygon();
        glutPostRedisplay();  // Request redisplay
    }
    else if (key == 27) {  // ESC key
        exit(0);
    }
}

// Setup OpenGL
void initGL() {
    glClearColor(1.0, 1.0, 1.0, 1.0); // Set background color to white
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(0, 400, 0, 400); // Define 2D orthographic projection
}

void inputPolygon() {
    int numVertices;
    cout << "Enter number of vertices for the polygon: ";
    cin >> numVertices;

    polygon.clear();
    for (int i = 0; i < numVertices; i++) {
        Point p;
        cout << "Enter vertex " << i + 1 << " (x, y): ";
        cin >> p.x >> p.y;
        polygon.push_back(p);
    }
}

void inputClippingWindow() {
    cout << "Enter the clipping window coordinates:\n";
    cout << "x_min, y_min: ";
    cin >> x_min >> y_min;
    cout << "x_max, y_max: ";
    cin >> x_max >> y_max;
}

int main(int argc, char** argv) {
    // User inputs
    inputClippingWindow();
    inputPolygon();

    // Initialize GLUT and display
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutCreateWindow("Sutherland-Hodgman Polygon Clipping - Akshat Negi");

    initGL();
    glutDisplayFunc(display);
    glutKeyboardFunc(handleKeypress);
    glutMainLoop();

    return 0;
}
```


C:\Users\AKY BOY\source\rep
Enter the clipping window coordinates:
x_min, y_min: 100 100
x_max, y_max: 300 300
Enter number of vertices for the polygon: 5
Enter vertex 1 (x, y): 50 150
Enter vertex 2 (x, y): 150 50
Enter vertex 3 (x, y): 250 150
Enter vertex 4 (x, y): 200 250
Enter vertex 5 (x, y): 100 250


Sutherland-Hodgman Polygon Clipping - Akshat Negi



```
Enter the clipping window coordinates:
x_min, y_min: 100 100
x_max, y_max: 300 300
Enter number of vertices for the polygon: 5
Enter vertex 1 (x, y): 50 150
Enter vertex 2 (x, y): 150 50
Enter vertex 3 (x, y): 250 150
Enter vertex 4 (x, y): 200 250
Enter vertex 5 (x, y): 100 250
Clipped Polygon Points:
(100, 250)
(100, 100)
(100, 100)
(200, 100)
(250, 150)
(200, 250)
(100, 250)
```



LAB EXPERIMENT – 6

Basic 2D & 3D Transformations

 # Perform all the experiment for 3-D transformation.

 # Take the following values as input from user: Theta (angle of rotation), translation factor, scaling factor and other values. Make necessary assumptions.

a. Write an interactive program for following basic transformation.
b. Translation
c. Rotation
d. Scaling
e. Reflection about axis.
f. Reflection about a line Y=mX+c and aX+bY+c=0.
g. Shear about an edge and about a vertex.

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

float angle = 0.0f;        // Rotation angle
float tx = 0.0f, ty = 0.0f; // Translation factors
float sx = 1.0f, sy = 1.0f; // Scaling factors
float shearX = 0.0f, shearY = 0.0f; // Shear factors
float reflectionX = 1.0f, reflectionY = 1.0f; // Reflection factors

void drawSquare() {
    glBegin(GL_LINE_LOOP);
    glVertex2f(-0.5f, -0.5f);
    glVertex2f(0.5f, -0.5f);
    glVertex2f(0.5f, 0.5f);
    glVertex2f(-0.5f, 0.5f);
    glEnd();
}

// Apply translation transformation
void translate(float x, float y) {
    glTranslatef(x, y, 0.0f);
}

// Apply rotation transformation
void rotate(float angle) {
    glRotatef(angle, 0.0f, 0.0f, 1.0f);
}

// Apply scaling transformation
void scale(float x, float y) {
    glScalef(x, y, 1.0f);
}

// Apply reflection
void reflect(bool x, bool y) {
    reflectionX = x ? -1.0f : 1.0f;
    reflectionY = y ? -1.0f : 1.0f;
    glScalef(reflectionX, reflectionY, 1.0f);
}
```

```
// Apply shear transformation
void shear(float shx, float shy) {
    GLfloat shearMatrix[16] = {
        1.0f, shx, 0.0f, 0.0f,
        shy, 1.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 1.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 1.0f
    };
    glMultMatrixf(shearMatrix);
}

void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();

    // Apply transformations in the order: translate, rotate, scale, reflect, shear
    translate(tx, ty);
    rotate(angle);
    scale(sx, sy);
    reflect(reflectionX < 0, reflectionY < 0);
    shear(shearX, shearY);

    drawSquare();
    glutSwapBuffers();
}

// Handle keyboard input for transformations
void keyboard(unsigned char key, int x, int y) {
    switch (key) {
    case 'r': // Rotate
        std::cout << "Enter rotation angle: ";
        std::cin >> angle;
        break;
    case 't': // Translate
        std::cout << "Enter translation factors (tx ty): ";
        std::cin >> tx >> ty;
        break;
    case 's': // Scale
        std::cout << "Enter scaling factors (sx sy): ";
        std::cin >> sx >> sy;
        break;
    case 'x': // Reflect about X-axis
        reflectionX = -reflectionX;
        break;
    case 'y': // Reflect about Y-axis
        reflectionY = -reflectionY;
        break;
    case 'h': // Shear
        std::cout << "Enter shear factors (shearX shearY): ";
        std::cin >> shearX >> shearY;
        break;
    case 27: // Escape key to exit
        exit(0);
    }
    glutPostRedisplay();
}

void init() {
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(-2.0, 2.0, -2.0, 2.0); // Set up a 2D orthogonal projection
    glMatrixMode(GL_MODELVIEW);
}

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

```
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutCreateWindow("2D Transformations - Akshat Negi");
    init();
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

Interaction:

Keyboard keys:

r: Prompts for rotation angle.

t: Prompts for translation factors (tx, ty).

s: Prompts for scaling factors (sx, sy).

x, y: Reflects about the X or Y axis, respectively.

h: Prompts for shear factors (shearX, shearY).

Esc: Exits the program.

SAMPLE INPUT

Enter rotation angle:

30

Enter translation factors (tx ty):

1.0 0.5

Enter scaling factors (sx sy):

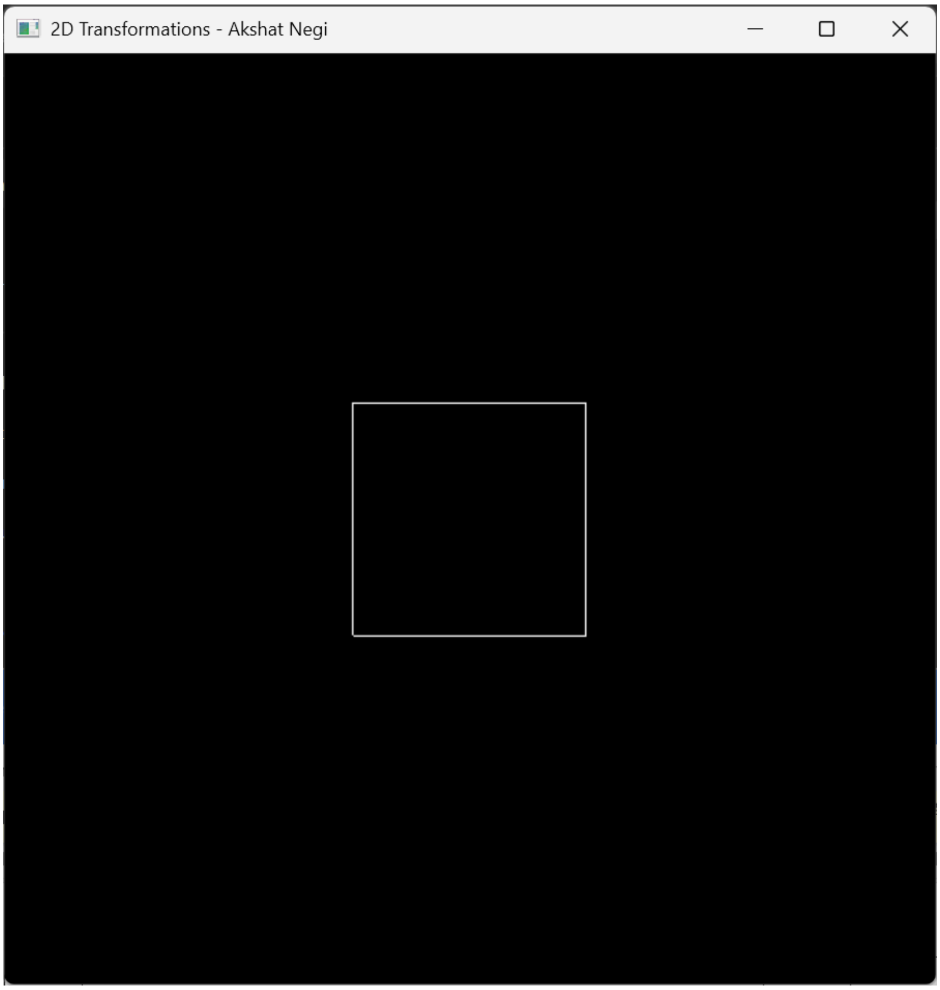
2.0 0.5

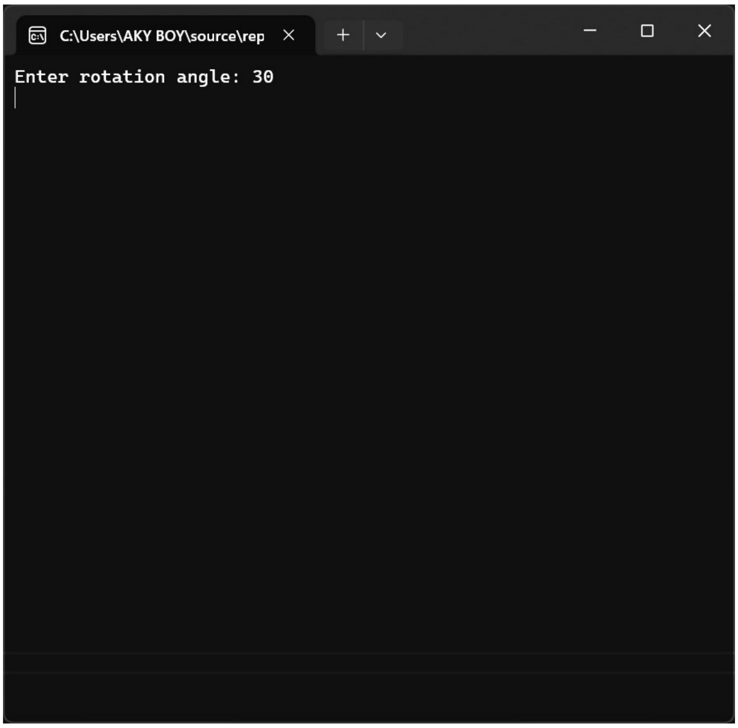
Press 'x' for reflection about X-axis

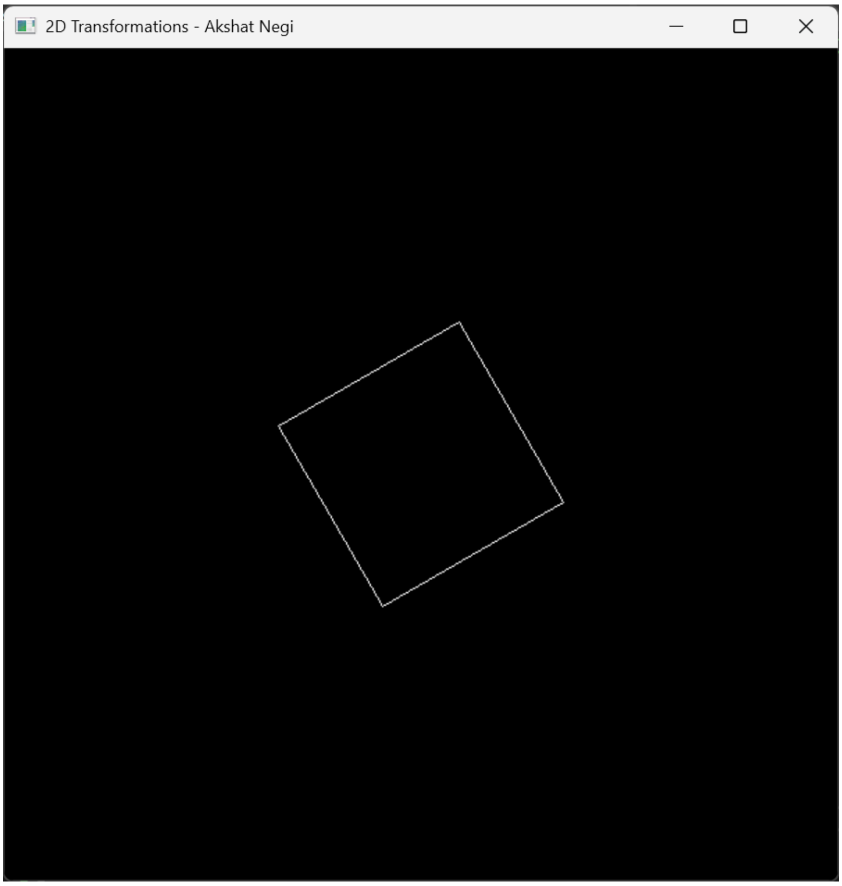
Press 'h' for shear factors:

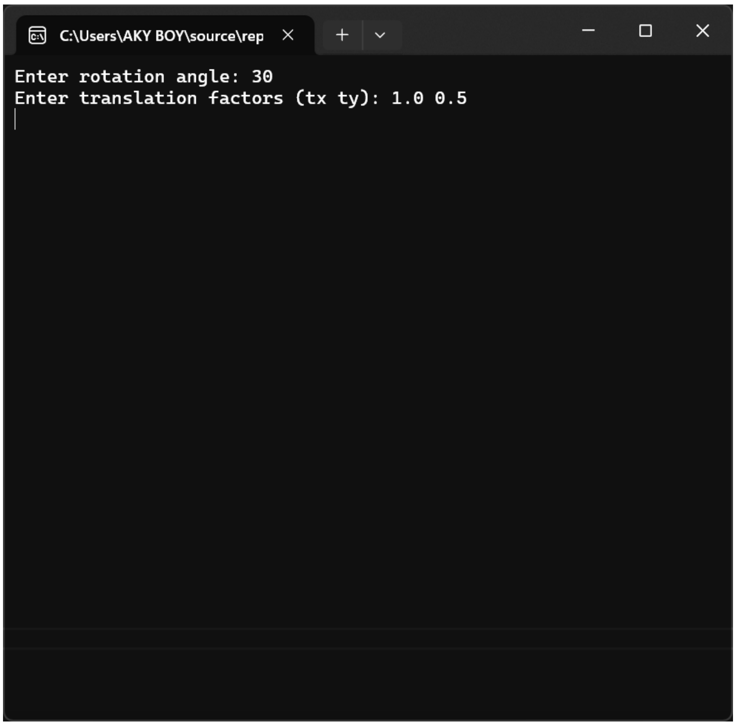
Enter shear factors (shearX shearY):

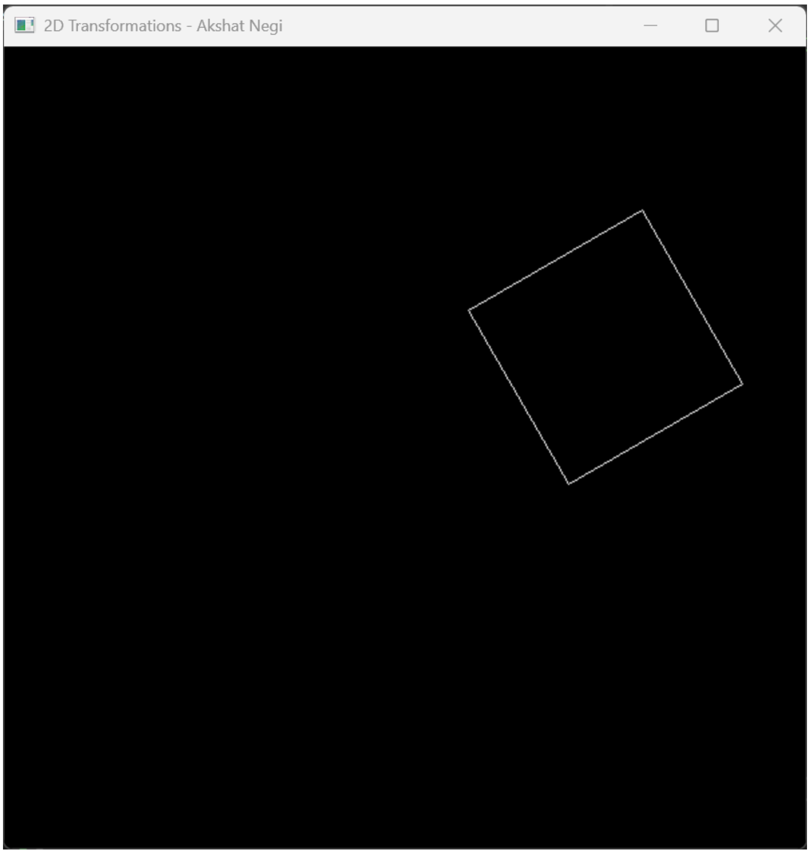
0.2 0.0

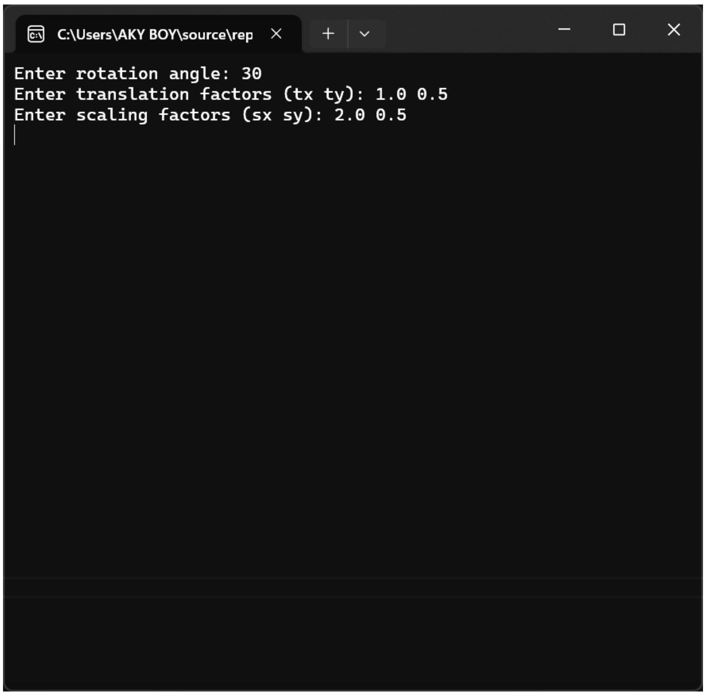

2D Transformations - Akshat Negi

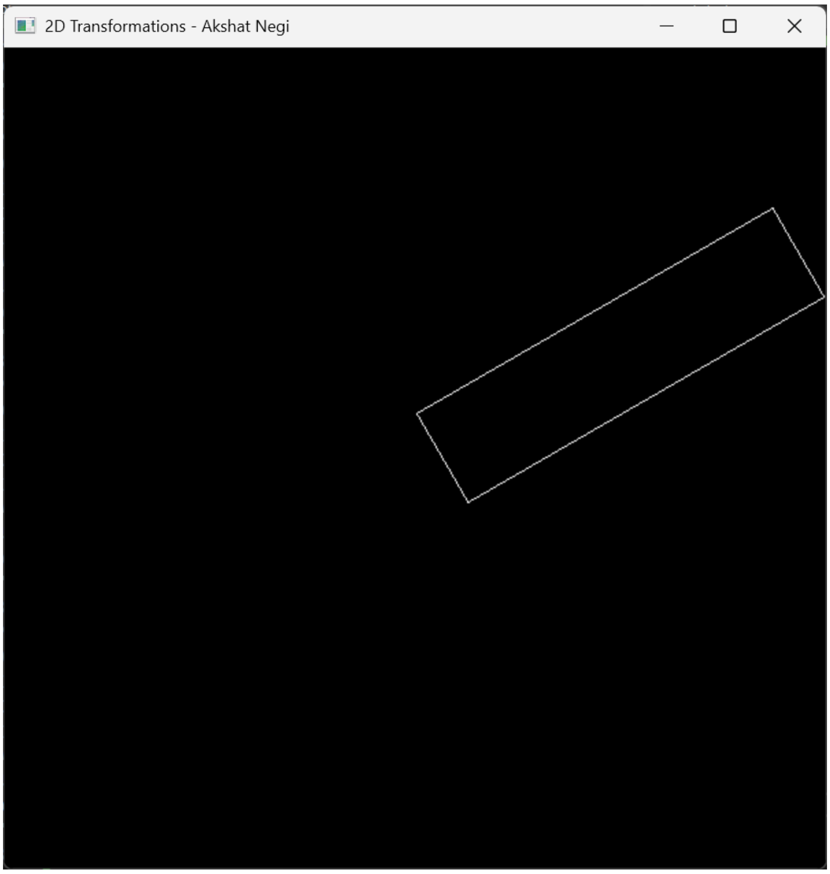

C:\Users\AKY BOY\source\rep
Enter rotation angle: 30

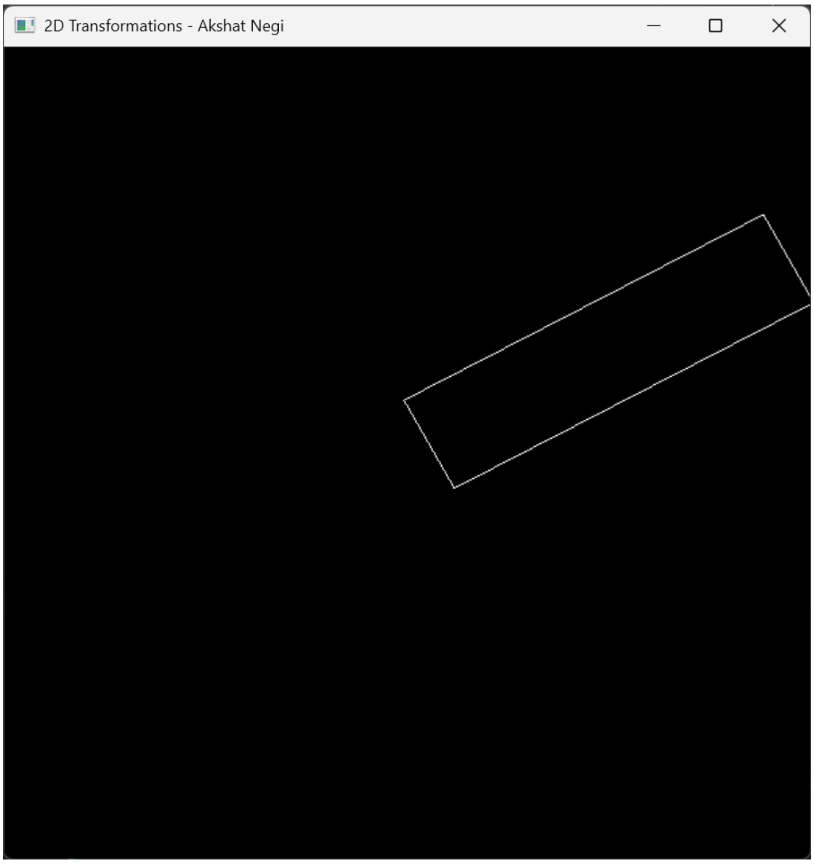

2D Transformations - Akshat Negi

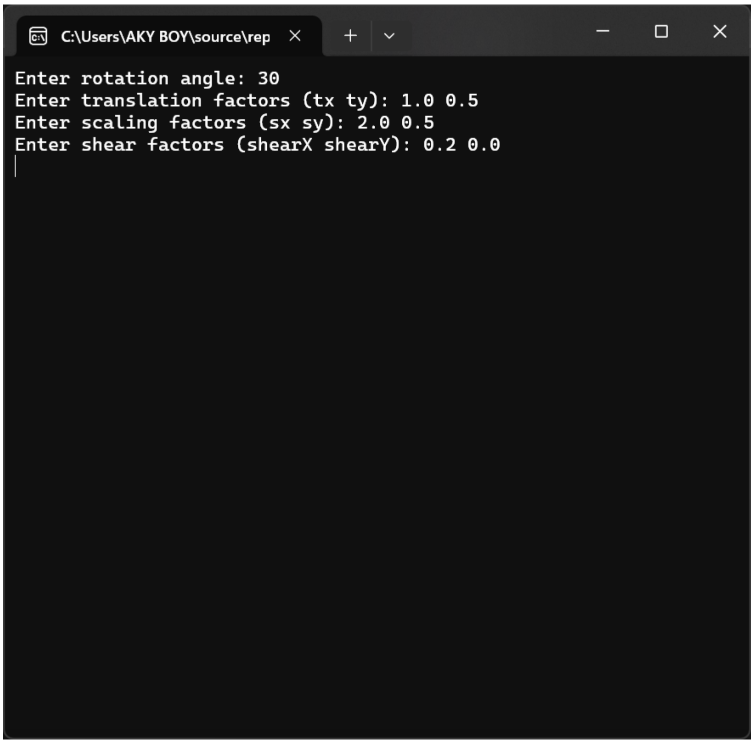

C:\Users\AKY BOY\source\rep
Enter rotation angle: 30
Enter translation factors (tx ty): 1.0 0.5

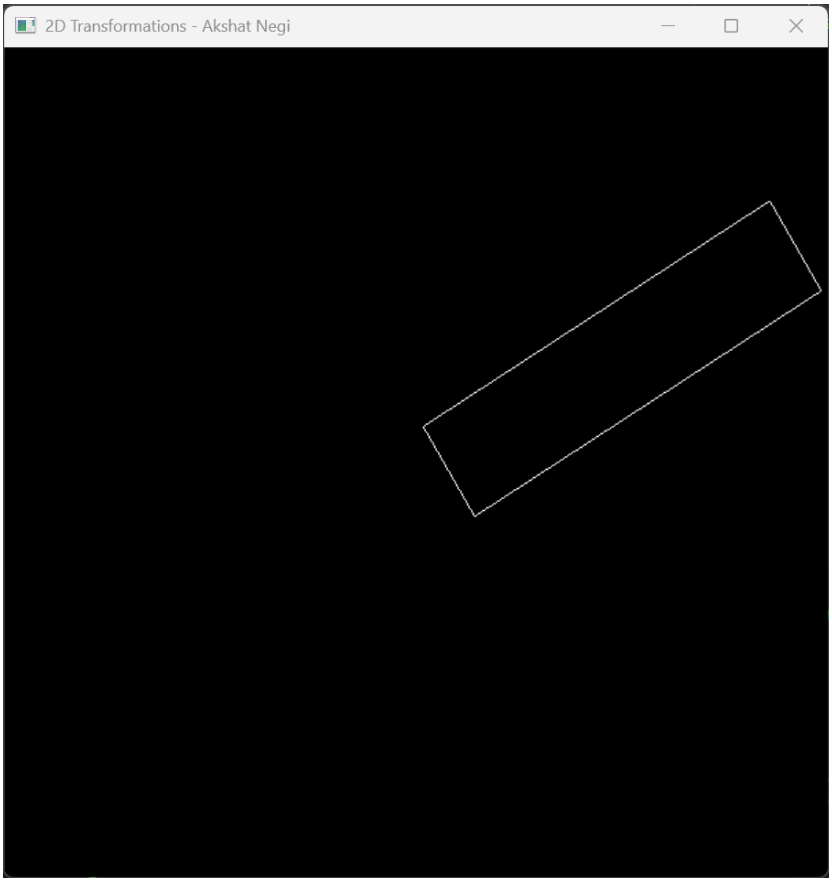

2D Transformations - Akshat Negi


C:\Users\AKY BOY\source\rep
Enter rotation angle: 30
Enter translation factors (tx ty): 1.0 0.5
Enter scaling factors (sx sy): 2.0 0.5


2D Transformations - Akshat Negi


2D Transformations - Akshat Negi


C:\Users\AKY BOY\source\rep
Enter rotation angle: 30
Enter translation factors (tx ty): 1.0 0.5
Enter scaling factors (sx sy): 2.0 0.5
Enter shear factors (shearX shearY): 0.2 0.0


2D Transformations - Akshat Negi

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

float theta = 0.0f;        // Rotation angle
float tx = 0.0f, ty = 0.0f, tz = 0.0f; // Translation factors
float sx = 1.0f, sy = 1.0f, sz = 1.0f; // Scaling factors
float shearX = 0.0f, shearY = 0.0f, shearZ = 0.0f; // Shear factors
float reflectionX = 1.0f, reflectionY = 1.0f, reflectionZ = 1.0f; // Reflection
factors

void drawCube() {
    glutWireCube(1.0); // Draw a unit cube
}

// Apply translation transformation
void translate(float x, float y, float z) {
    glTranslatef(x, y, z);
}

// Apply rotation transformation
void rotate(float angle, float x, float y, float z) {
    glRotatef(angle, x, y, z);
}

// Apply scaling transformation
void scale(float x, float y, float z) {
    glScalef(x, y, z);
}

// Apply reflection about axis
void reflect(bool x, bool y, bool z) {
    reflectionX = x ? -1.0f : 1.0f;
    reflectionY = y ? -1.0f : 1.0f;
    reflectionZ = z ? -1.0f : 1.0f;
```

```
    glScalef(reflectionX, reflectionY, reflectionZ);
}

// Apply shear transformation
void shear(float shx, float shy, float shz) {
    GLfloat shearMatrix[16] = {
        1.0f, shx, 0.0f, 0.0f,
        shy, 1.0f, 0.0f, 0.0f,
        0.0f, shz, 1.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 1.0f
    };
    glMultMatrixf(shearMatrix);
}

void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(0.0f, 0.0f, -5.0f); // Position the object

    // Apply transformations in the order: translate, rotate, scale, reflect, shear
    translate(tx, ty, tz);
    rotate(theta, 1.0f, 1.0f, 1.0f);
    scale(sx, sy, sz);
    reflect(reflectionX < 0, reflectionY < 0, reflectionZ < 0);
    shear(shearX, shearY, shearZ);

    drawCube();
    glutSwapBuffers();
}

// Handle keyboard input for transformations
void keyboard(unsigned char key, int x, int y) {
    switch (key) {
    case 'r': // Rotate
        std::cout << "Enter rotation angle: ";
        std::cin >> theta;
```

```
            break;
        case 't': // Translate
            std::cout << "Enter translation factors (tx ty tz): ";
            std::cin >> tx >> ty >> tz;
            break;
        case 's': // Scale
            std::cout << "Enter scaling factors (sx sy sz): ";
            std::cin >> sx >> sy >> sz;
            break;
        case 'x': // Reflect about X-axis
            reflectionX = -reflectionX;
            break;
        case 'y': // Reflect about Y-axis
            reflectionY = -reflectionY;
            break;
        case 'z': // Reflect about Z-axis
            reflectionZ = -reflectionZ;
            break;
        case 'h': // Shear
            std::cout << "Enter shear factors (shearX shearY shearZ): ";
            std::cin >> shearX >> shearY >> shearZ;
            break;
        case 27: // Escape key to exit
            exit(0);
    }
    glutPostRedisplay();
}

void init() {
    glEnable(GL_DEPTH_TEST);
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glMatrixMode(GL_PROJECTION);
    gluPerspective(45.0, 1.0, 1.0, 100.0);
    glMatrixMode(GL_MODELVIEW);
}
```

```
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(600, 600);
    glutCreateWindow("3D Transformations - Akshat Negi");
    init();
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

Interaction:

Keyboard keys:

r: Prompts for rotation angle.

t: Prompts for translation factors (tx, ty, tz).

s: Prompts for scaling factors (sx, sy, sz).

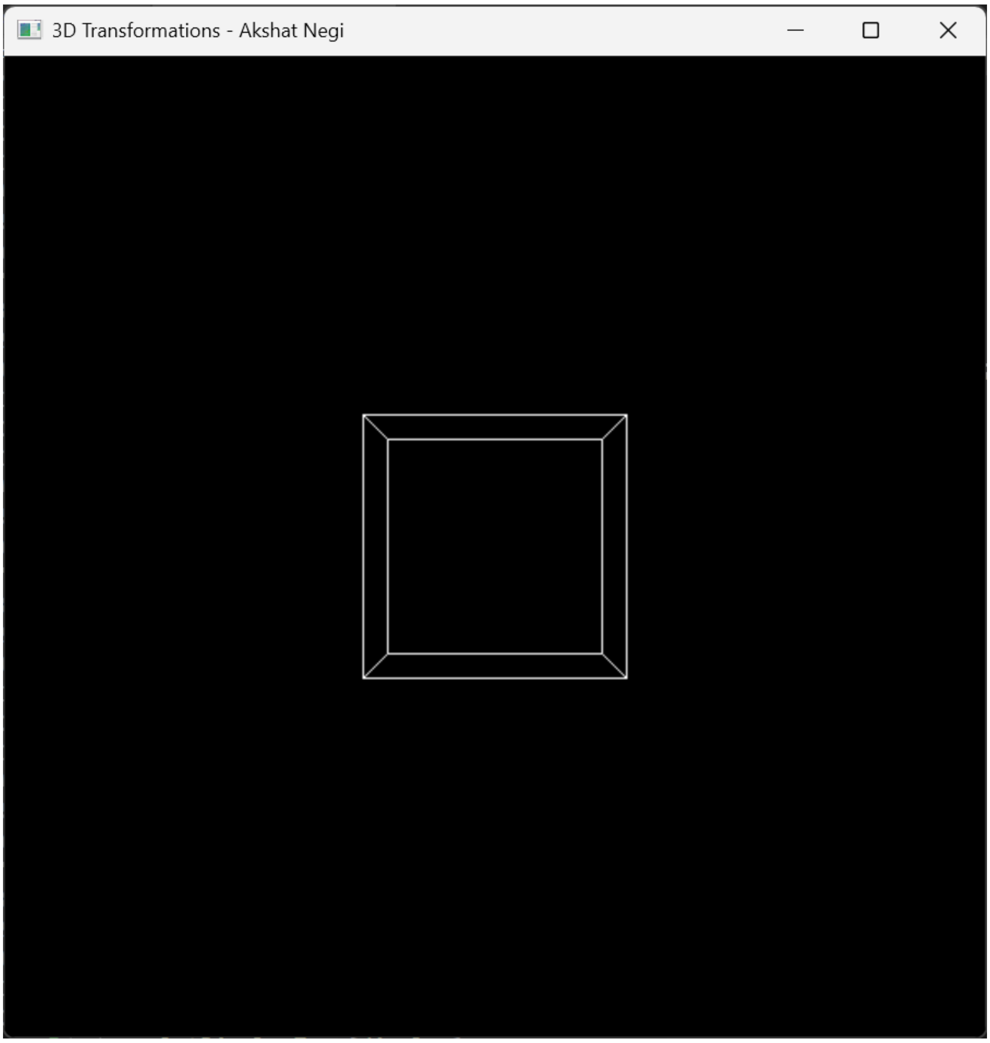
x, y, z: Reflects about X, Y, or Z axes.

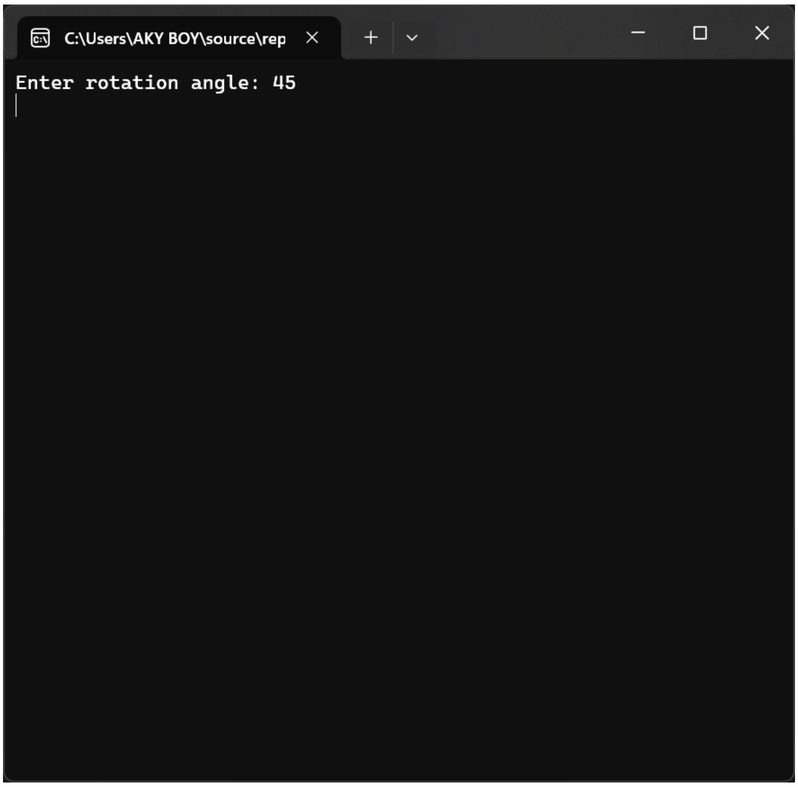
h: Prompts for shear factors (shearX, shearY, shearZ).

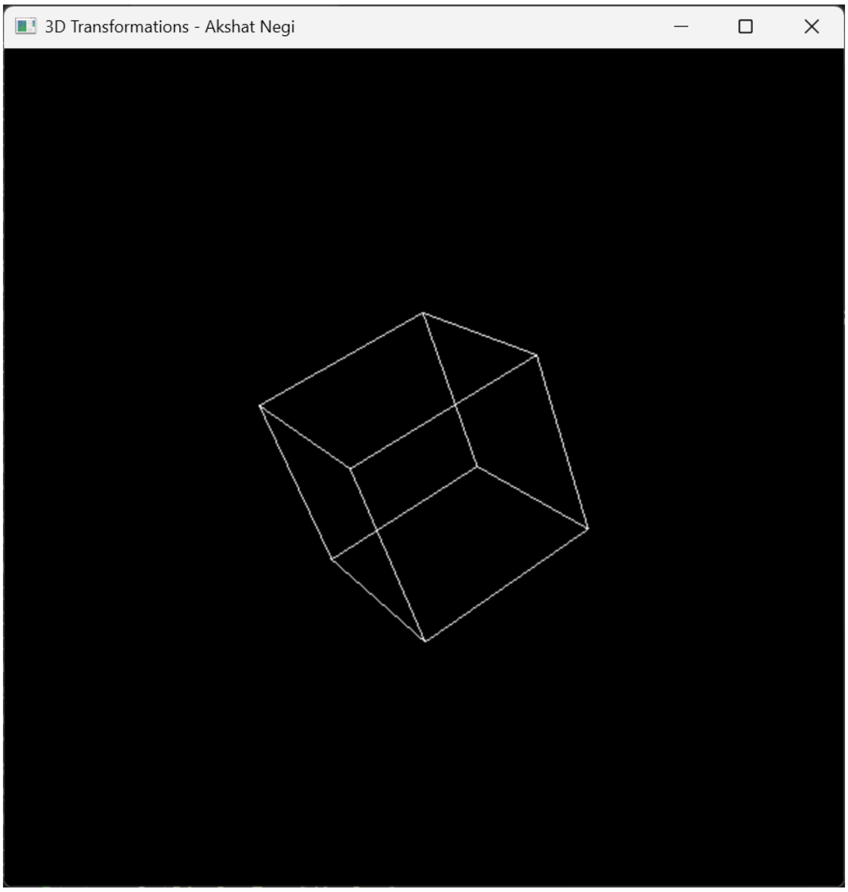
Esc: Exits the program.

SAMPLE INPUT

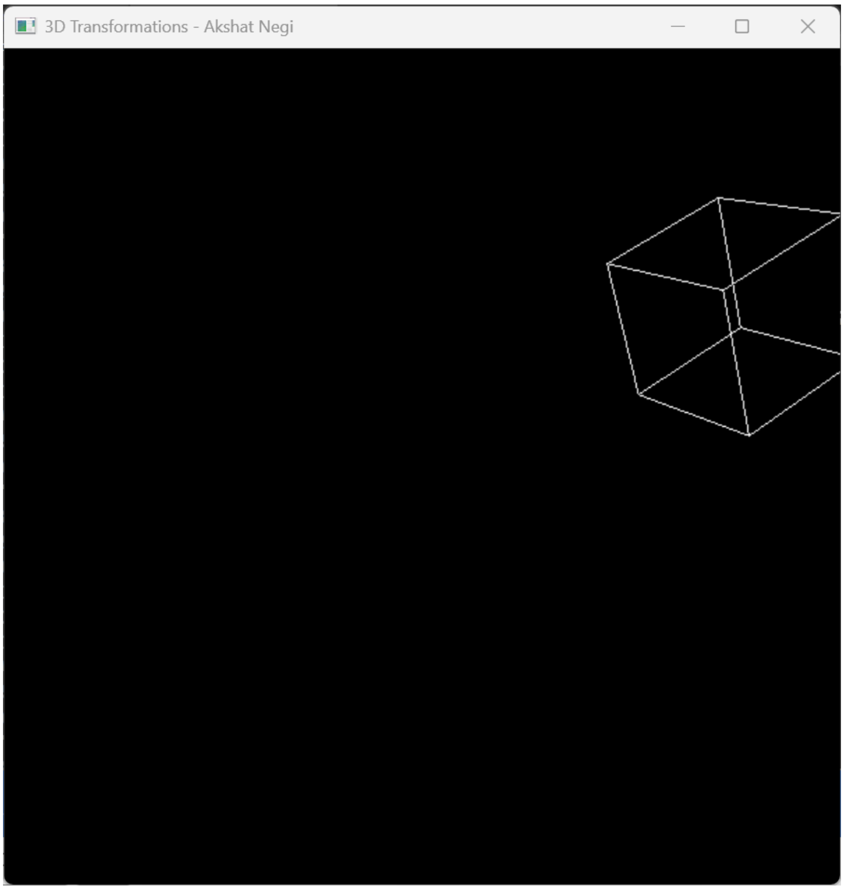
```
Enter rotation angle:
45
Enter translation factors (tx ty tz):
2.0 1.0 -1.5
Enter scaling factors (sx sy sz):
1.5 0.5 2.0
Press 'x' for reflection about X-axis
Press 'h' for shear factors:
Enter shear factors (shearX shearY shearZ):
0.5 0.0 1.0
```

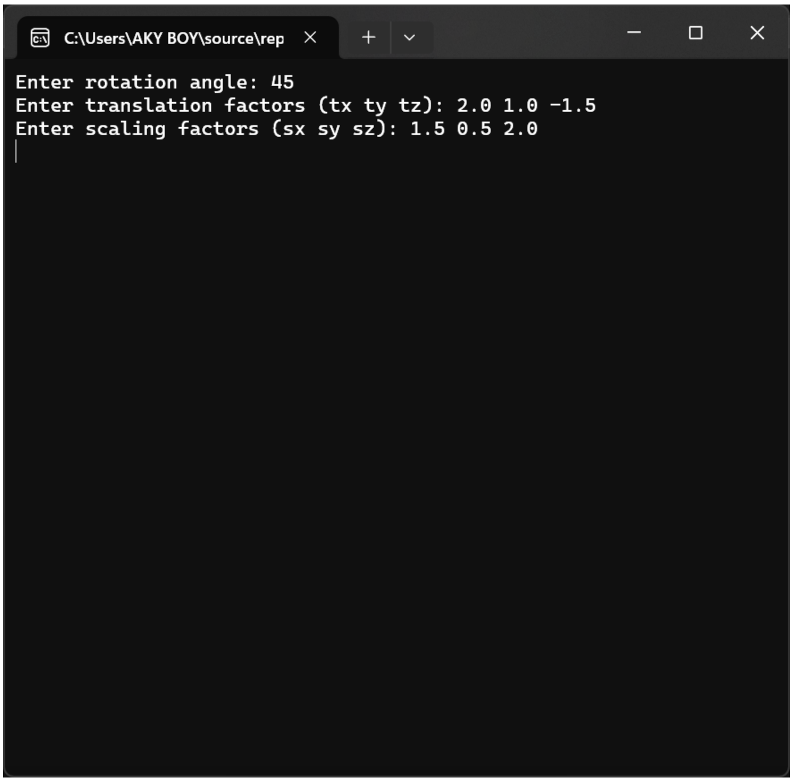

3D Transformations - Akshat Negi

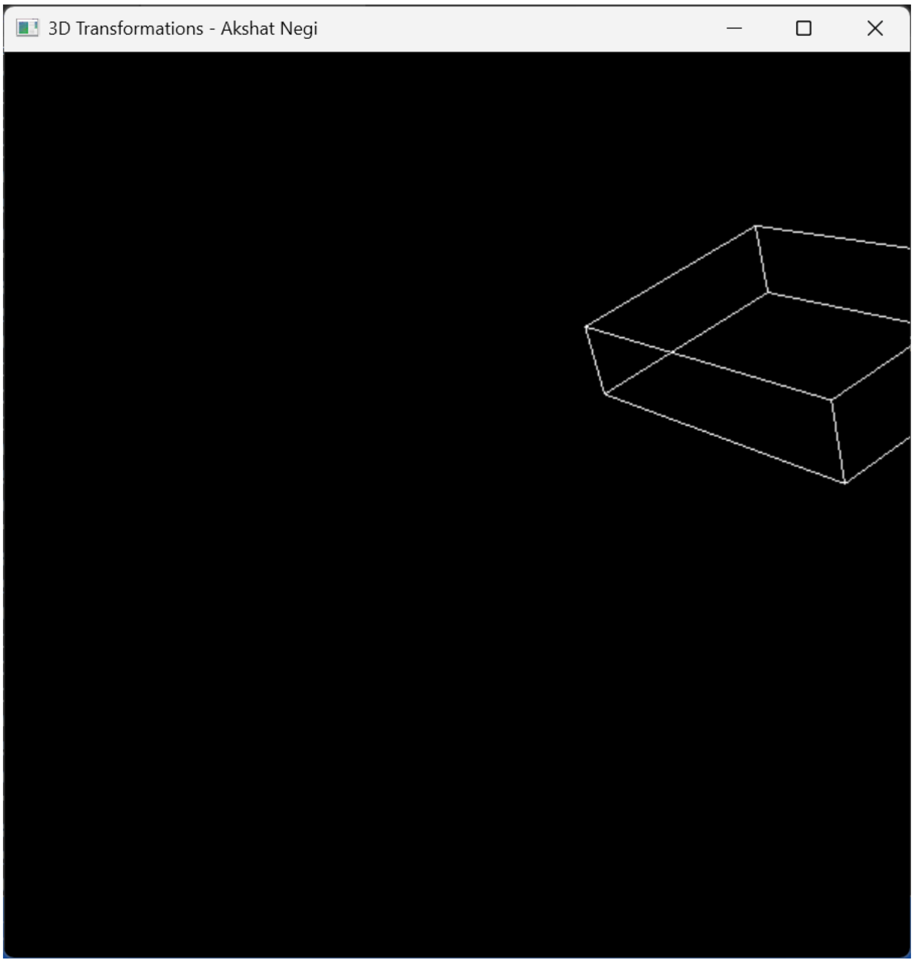

C:\Users\AKY BOY\source\rep
Enter rotation angle: 45

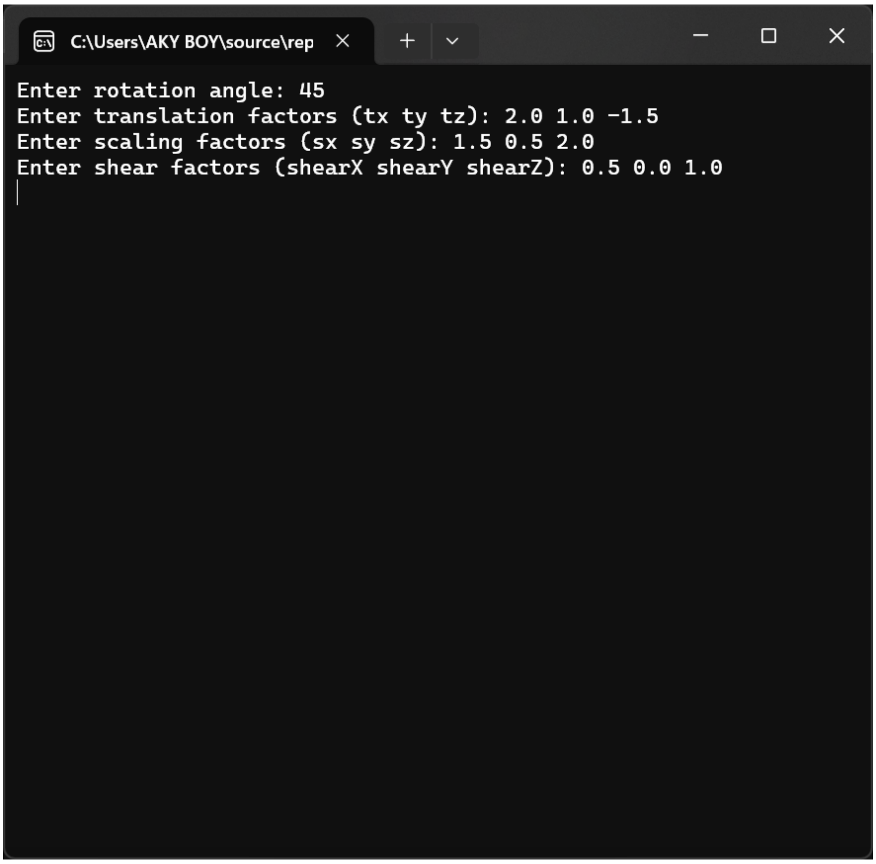

3D Transformations - Akshat Negi

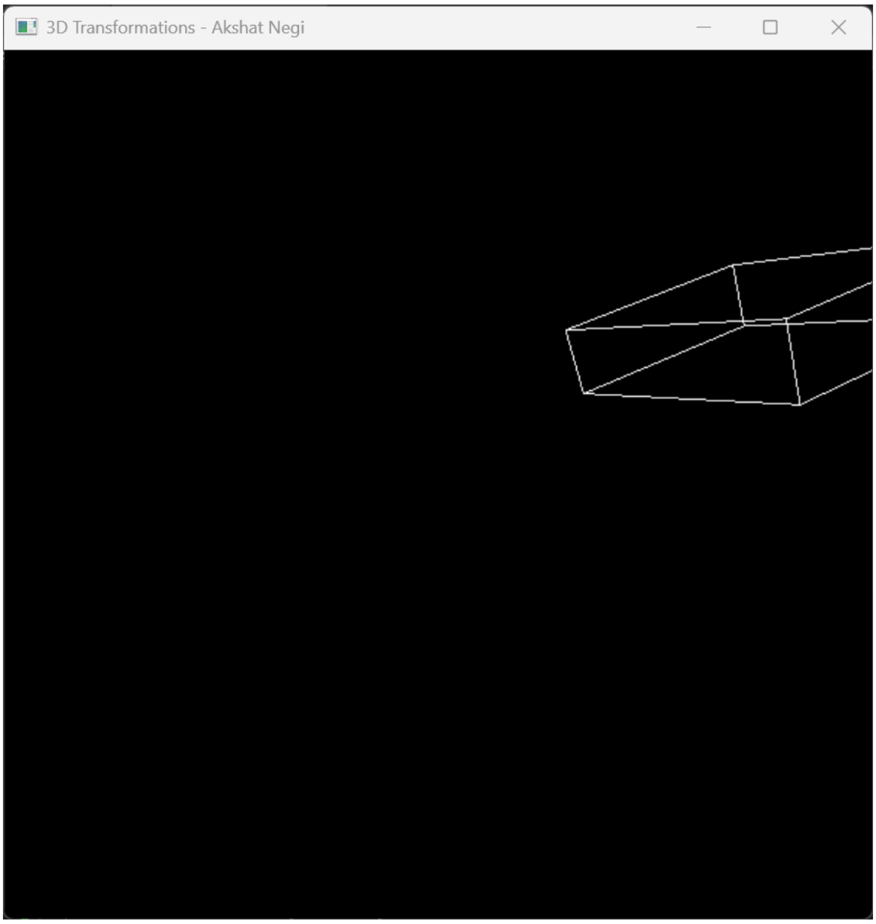

C:\Users\AKY BOY\source\rep
Enter rotation angle: 45
Enter translation factors (tx ty tz): 2.0 1.0 -1.5

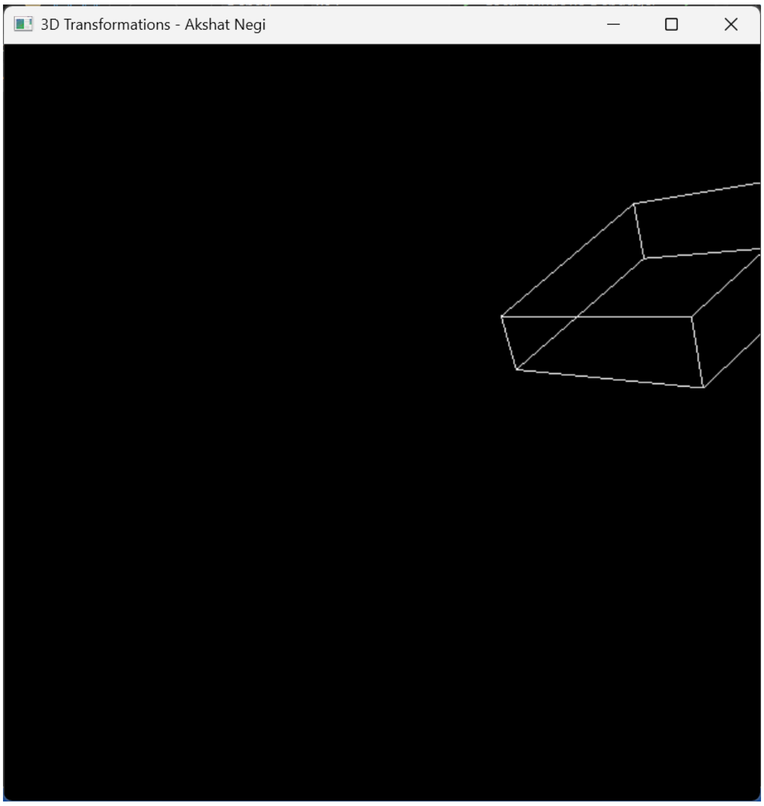

3D Transformations - Akshat Negi

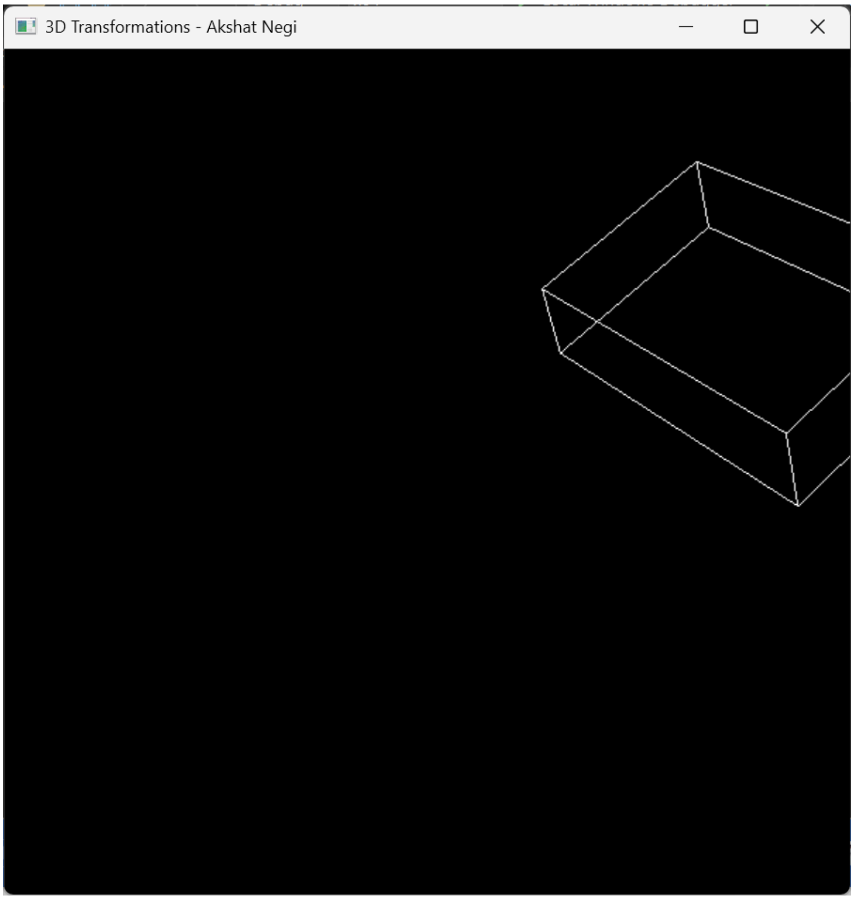

C:\Users\AKY BOY\source\rep
Enter rotation angle: 45
Enter translation factors (tx ty tz): 2.0 1.0 -1.5
Enter scaling factors (sx sy sz): 1.5 0.5 2.0

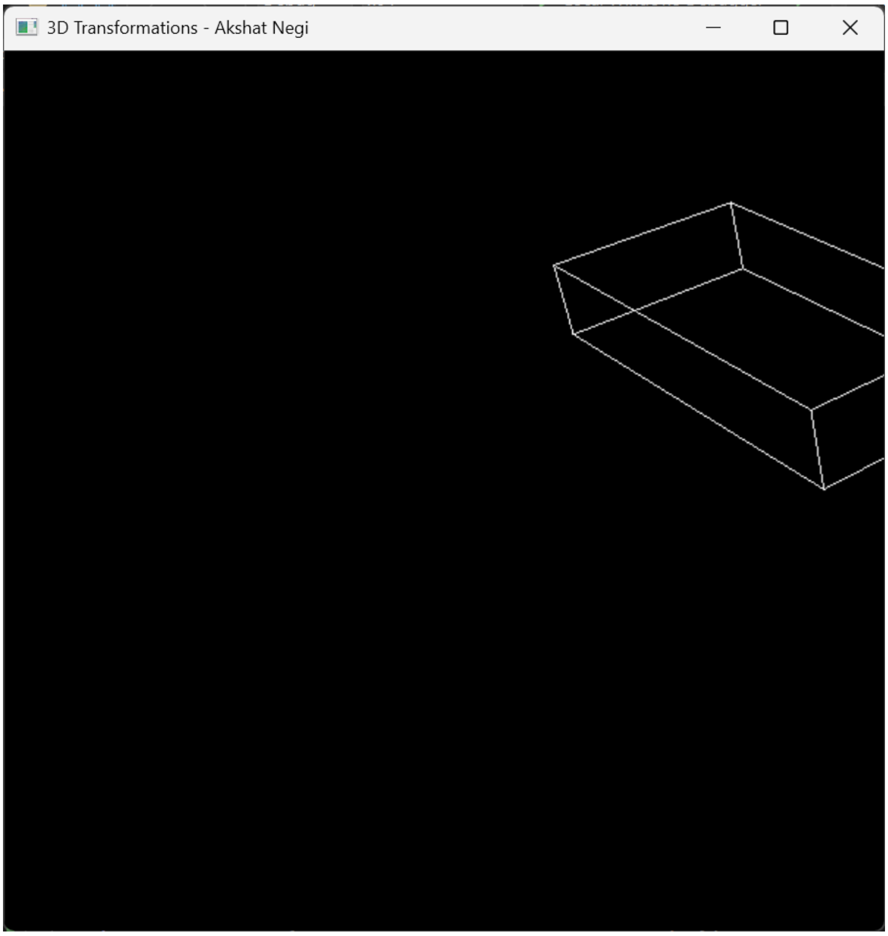

3D Transformations - Akshat Negi


C:\Users\AKY BOY\source\rep
Enter rotation angle: 45
Enter translation factors (tx ty tz): 2.0 1.0 -1.5
Enter scaling factors (sx sy sz): 1.5 0.5 2.0
Enter shear factors (shearX shearY shearZ): 0.5 0.0 1.0


3D Transformations - Akshat Negi


3D Transformations - Akshat Negi


3D Transformations - Akshat Negi


3D Transformations - Akshat Negi

LAB EXPERIMENT – 7

Drawing Bezier Curves

[Virtual GLUT based demonstration]

a. Write a program to draw a cubic spline.

```
#include <GL/freeglut.h>
#include <vector>
#include <iostream>
#include <cmath>

struct Point {
    float x, y;
};

// Define a vector to store control points for the cubic spline
std::vector<Point> controlPoints(4);

// Function to interpolate points for a cubic spline
Point cubicSpline(float t, Point p0, Point p1, Point p2, Point p3) {
    float a = (1 - t) * (1 - t) * (1 - t);
    float b = 3 * t * (1 - t) * (1 - t);
    float c = 3 * t * t * (1 - t);
    float d = t * t * t;

    return {
        a * p0.x + b * p1.x + c * p2.x + d * p3.x,
        a * p0.y + b * p1.y + c * p2.y + d * p3.y
    };
}

// Function to render the cubic spline
void renderSpline() {
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 0.0, 0.0); // Red color for the spline
    glBegin(GL_LINE_STRIP);

    for (float t = 0; t <= 1; t += 0.01) {
        Point p = cubicSpline(t, controlPoints[0], controlPoints[1], controlPoints[2],
controlPoints[3]);
        glVertex2f(p.x, p.y);
    }

    glEnd();
    glFlush();
}

void init() {
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glOrtho(0.0, 500.0, 0.0, 500.0, -1.0, 1.0);
}

int main(int argc, char** argv) {
    std::cout << "Enter 4 control points for the cubic spline (x y):\n";
    for (int i = 0; i < 4; ++i) {
        std::cout << "Point " << i + 1 << ": ";
        std::cin >> controlPoints[i].x >> controlPoints[i].y;
    }
```

```
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutCreateWindow("Cubic Spline - Akshat Negi");
    init();
    glutDisplayFunc(renderSpline);
    glutMainLoop();
    return 0;
}
```

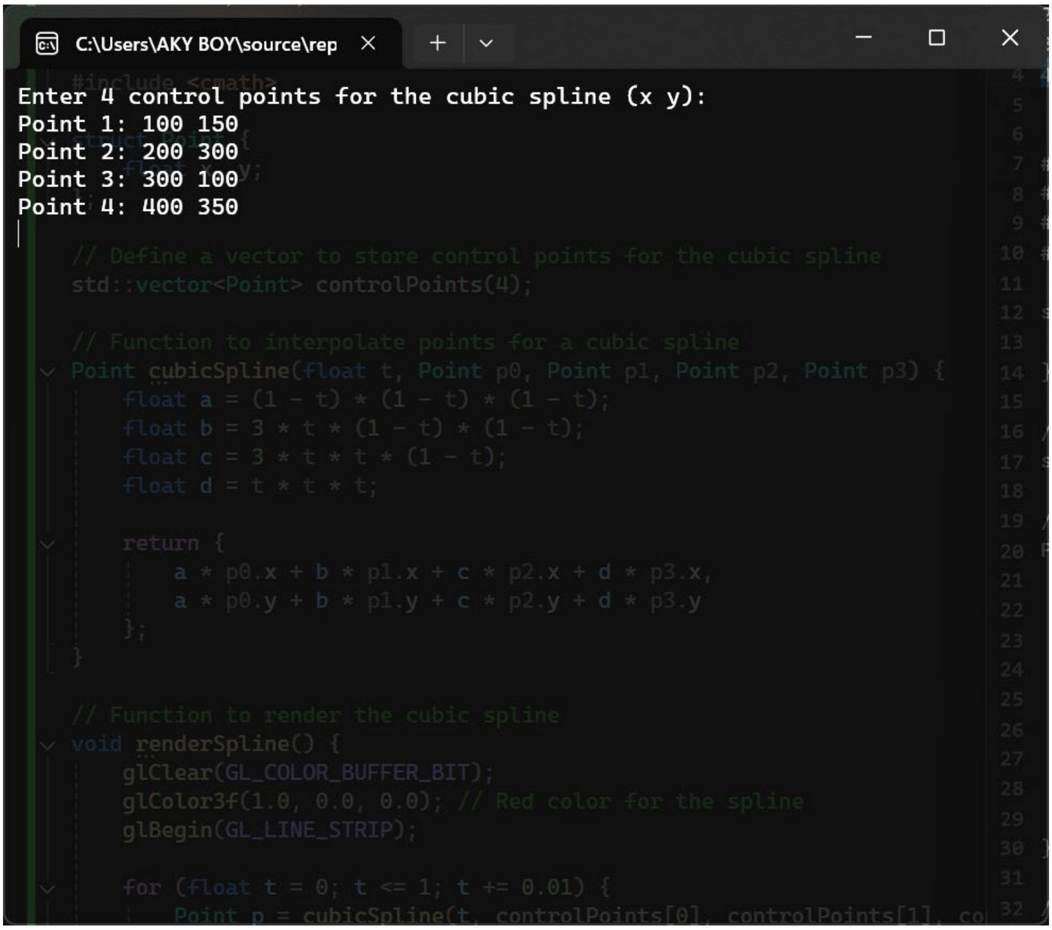
4 control points for the cubic spline (x y):

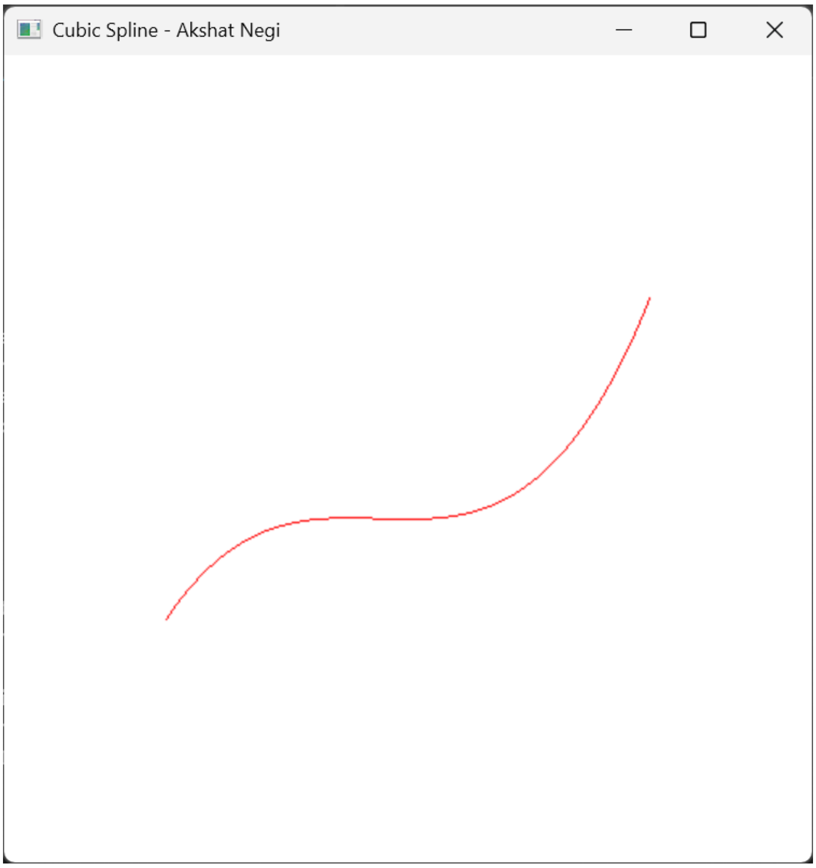
100 150

200 300

300 100

400 350





b. WAP to draw a Bezier curve.

 # Take necessary values as input from the user like degree of the Bezier curve.

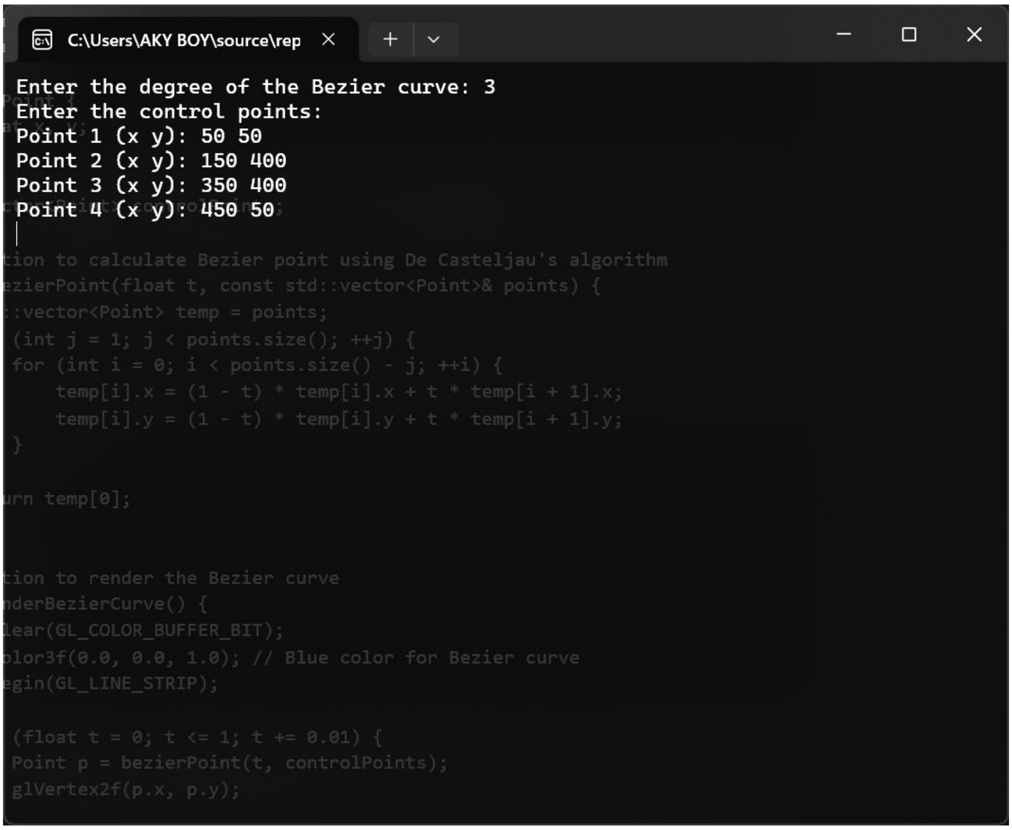
```
#include <GL/freeglut.h>
#include <iostream>
#include <vector>

struct Point {
    float x, y;
};

std::vector<Point> controlPoints;

// Function to calculate Bezier point using De Casteljau's algorithm
Point bezierPoint(float t, const std::vector<Point>& points) {
    std::vector<Point> temp = points;
    for (int j = 1; j < points.size(); ++j) {
        for (int i = 0; i < points.size() - j; ++i) {
            temp[i].x = (1 - t) * temp[i].x + t * temp[i + 1].x;
            temp[i].y = (1 - t) * temp[i].y + t * temp[i + 1].y;
        }
    }
    return temp[0];
}

// Function to render the Bezier curve
void renderBezierCurve() {
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 0.0, 1.0); // Blue color for Bezier curve
    glBegin(GL_LINE_STRIP);

    for (float t = 0; t <= 1; t += 0.01) {
```

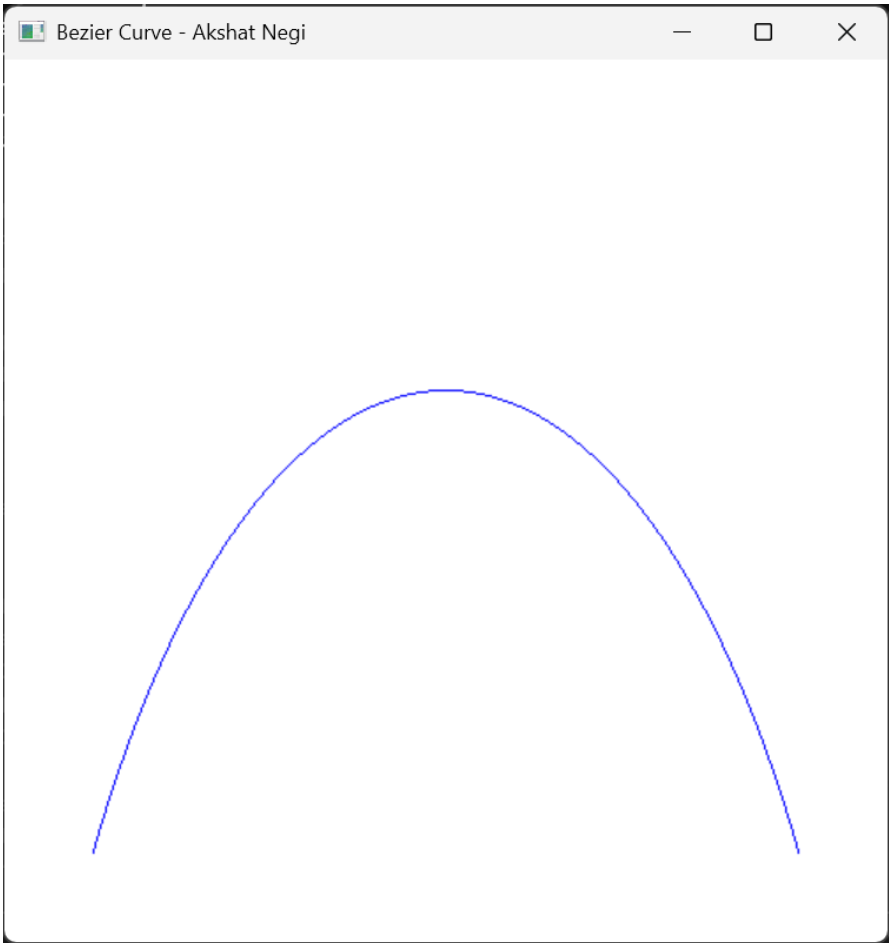
```
        Point p = bezierPoint(t, controlPoints);
        glVertex2f(p.x, p.y);
    }

    glEnd();
    glFlush();
}

void init() {
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glOrtho(0.0, 500.0, 0.0, 500.0, -1.0, 1.0);
}

int main(int argc, char** argv) {
    int degree;
    std::cout << "Enter the degree of the Bezier curve: ";
    std::cin >> degree;
    controlPoints.resize(degree + 1);

    std::cout << "Enter the control points:\n";
    for (int i = 0; i <= degree; i++) {
        std::cout << "Point " << i + 1 << " (x y): ";
        std::cin >> controlPoints[i].x >> controlPoints[i].y;
    }

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutCreateWindow("Bezier Curve - Akshat Negi");
    init();
    glutDisplayFunc(renderBezierCurve);
    glutMainLoop();
    return 0;
}
```

SAMPLE INPUTS

```
Enter the degree of the Bezier curve: 2

Enter the control points:

Point 1 (x y): 100 100

Point 2 (x y): 250 400

Point 3 (x y): 400 100


Enter the degree of the Bezier curve: 3

Enter the control points:

Point 1 (x y): 50 50

Point 2 (x y): 150 400

Point 3 (x y): 350 400

Point 4 (x y): 450 50


Enter the degree of the Bezier curve: 4

Enter the control points:

Point 1 (x y): 50 50
```

```
Point 2 (x y): 100 400
Point 3 (x y): 250 300
Point 4 (x y): 400 400
Point 5 (x y): 450 50
```




Bezier Curve - Akshat Negi

LAB EXPERIMENT – 8

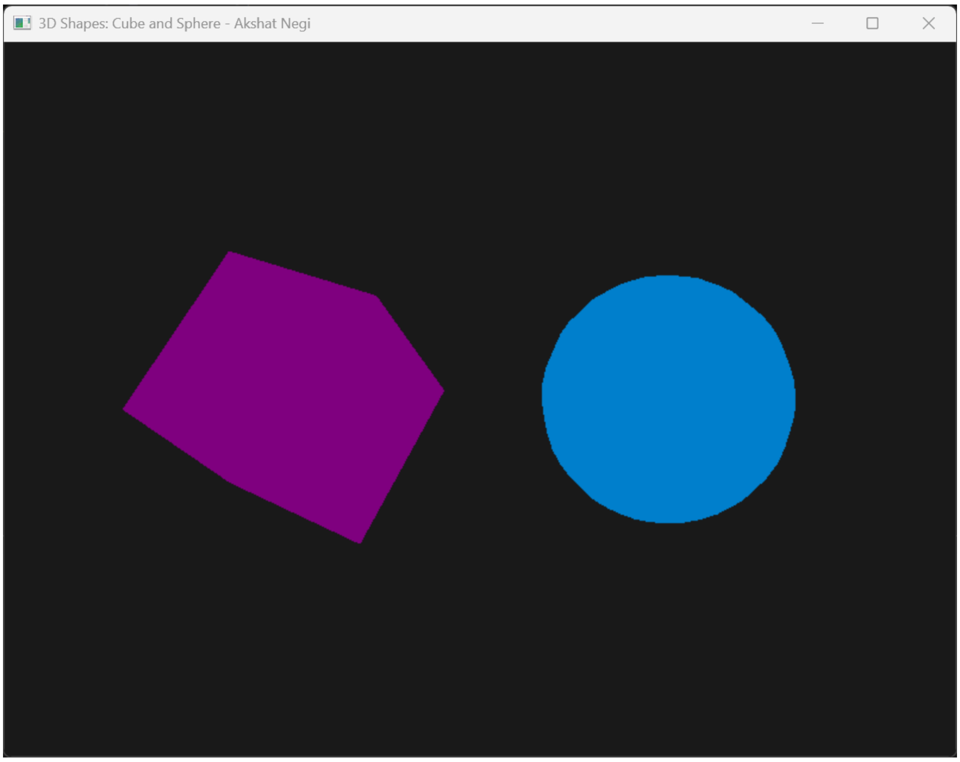
Event Handling

#Implement above with the help of animation.

a. Implement mouse input functionality.
b. Implement keypress functionality.
c. Implement another call back functions.

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

float angleCube = 0.0f;  // Rotation angle for the cube
float angleSphere = 0.0f;  // Rotation angle for the sphere
bool rotateCube = true;    // Toggle rotation for the cube
bool rotateSphere = true;  // Toggle rotation for the sphere

// Initialization of OpenGL settings
void initGL() {
    glEnable(GL_DEPTH_TEST);  // Enable depth testing for z-culling
    glClearColor(0.1f, 0.1f, 0.1f, 1.0f); // Set background color to dark gray
}

// Display function to render the shapes
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  // Clear the screen and
depth buffer
    glMatrixMode(GL_MODELVIEW);  // Switch to the drawing perspective

    // Draw Cube
    glLoadIdentity();
    glTranslatef(-1.5f, 0.0f, -7.0f); // Move left and into the screen
    glRotatef(angleCube, 1.0f, 1.0f, 1.0f); // Rotate the cube
    glColor3f(0.5f, 0.0f, 0.5f);  // Set color of the cube to purple
    glutSolidCube(1.5);  // Draw a cube with side length 1.5

    // Draw Sphere
    glLoadIdentity();
    glTranslatef(1.5f, 0.0f, -7.0f); // Move right and into the screen
    glRotatef(angleSphere, 1.0f, 0.0f, 0.0f); // Rotate the sphere
    glColor3f(0.0f, 0.5f, 0.8f);  // Set color of the sphere to cyan
    glutSolidSphere(1.0, 20, 20);  // Draw a sphere with radius 1.0 and detail
level 20

    glutSwapBuffers();  // Swap front and back buffers (double buffering)
}

// Timer function to update the rotation angles
void timer(int value) {
    if (rotateCube) {
        angleCube += 2.0f;
        if (angleCube > 360) angleCube -= 360;
    }
    if (rotateSphere) {
        angleSphere += 1.5f;
```

```
        if (angleSphere > 360) angleSphere -= 360;
    }
    glutPostRedisplay();        // Post a paint request to activate display()
    glutTimerFunc(16, timer, 0); // Call this function again after 16
milliseconds
}

// Keyboard input for rotation toggle
void handleKeypress(unsigned char key, int x, int y) {
    switch (key) {
    case 'c':   // Toggle rotation for the cube
        rotateCube = !rotateCube;
        break;
    case 's':   // Toggle rotation for the sphere
        rotateSphere = !rotateSphere;
        break;
    case 27:    // ESC key
        exit(0);
    }
}

// Reshape function to handle window resizing
void reshape(int width, int height) {
    if (height == 0) height = 1;  // Prevent divide by zero
    float aspect = (float)width / (float)height;

    glViewport(0, 0, width, height);

    // Set the perspective projection
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0f, aspect, 0.1f, 100.0f);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);  // Enable double
buffering and depth test
    glutInitWindowSize(800, 600);  // Set window size
    glutCreateWindow("3D Shapes: Cube and Sphere - Akshat Negi");  // Create
window with title

    initGL(); // Initialize OpenGL settings
    glutDisplayFunc(display); // Set display function
    glutReshapeFunc(reshape); // Set reshape function
    glutKeyboardFunc(handleKeypress); // Set keyboard input function
    glutTimerFunc(0, timer, 0); // Set timer function

    glutMainLoop(); // Enter the main event loop
    return 0;
}
```


3D Shapes: Cube and Sphere - Akshat Negi

LAB EXPERIMENT – 9

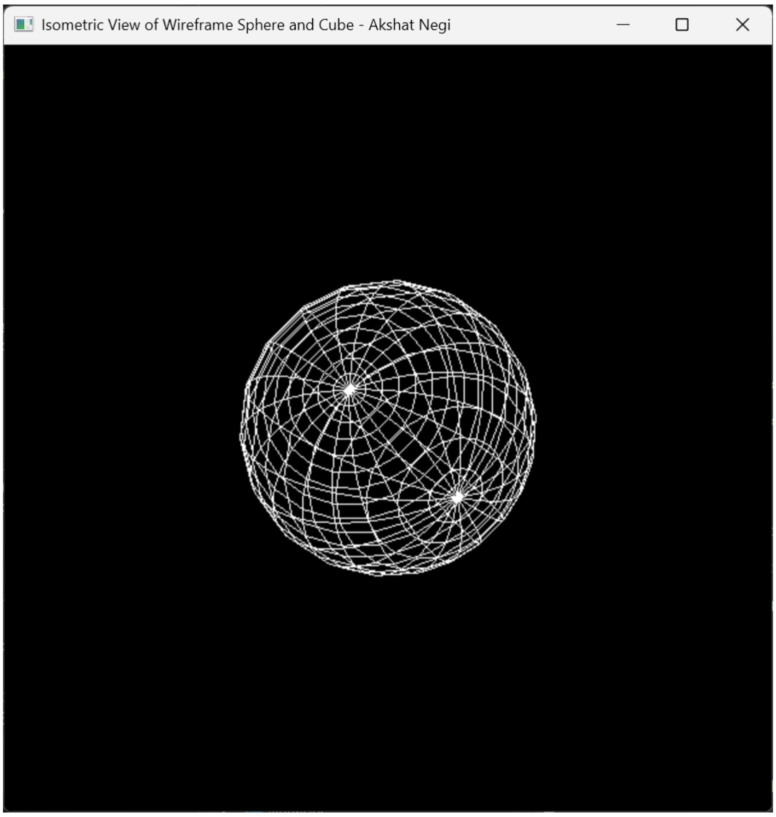
Creating 3D Shapes like Cube, Sphere and others.

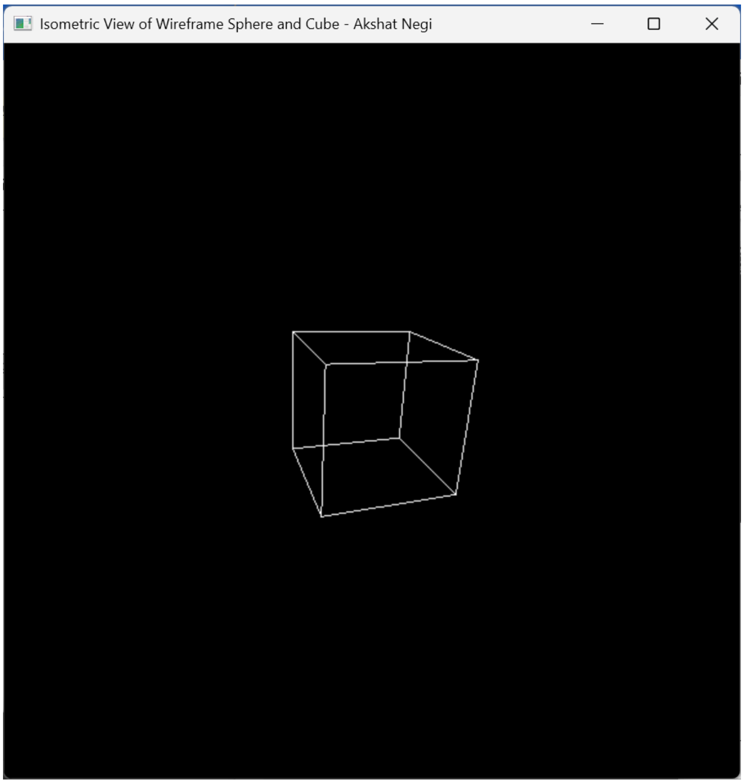
Creating 3D Shapes like Cube, Sphere and others.

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

bool showSphere = true; // Toggle between sphere and cube

// Function to draw a wireframe sphere
void drawWireframeSphere() {
    glColor3f(1.0f, 1.0f, 1.0f); // Set color to white
    glLineWidth(0.5f);            // Ensure thin lines
    glutWireSphere(1.0, 20, 20);  // Draw a wireframe sphere with radius 1 and 
resolution of 20 slices and stacks
}

// Function to draw a wireframe cube
void drawWireframeCube() {
    glColor3f(1.0f, 1.0f, 1.0f); // Set color to white
    glutWireCube(1.0);           // Draw a wireframe cube with side length 1
}

// Display function to render the isometric view of the chosen object
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  // Clear color and depth 
buffers

    // Set up the isometric view
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f, 0.0f, -3.0f);  // Position object further from the camera
    glRotatef(30, 1.0f, 1.0f, 0.0f);  // Rotate for isometric effect

    if (showSphere) {
        drawWireframeSphere();        // Draw the sphere
    }
    else {
        drawWireframeCube();          // Draw the cube
    }

    glutSwapBuffers();  // Swap buffers for double buffering
}

// Initialize OpenGL settings
void init() {
    glEnable(GL_DEPTH_TEST);             // Enable depth testing
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Set background to black

    // Set up projection
    glMatrixMode(GL_PROJECTION);
    gluPerspective(85.0, 1.0, 1.0, 100.0); // Perspective projection for depth
}
```

```
// Keyboard function to toggle between sphere and cube
void keyboard(unsigned char key, int x, int y) {
    if (key == 't') {
        showSphere = !showSphere; // Toggle object
        glutPostRedisplay();      // Request display update
    }
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH); // Double buffering
and depth
    glutInitWindowSize(600, 600);                            // Set window size
    glutCreateWindow("Isometric View of Wireframe Sphere and Cube - Akshat
Negi"); // Window title

    init();                    // Initialize OpenGL state
    glutDisplayFunc(display);  // Register display callback function
    glutKeyboardFunc(keyboard); // Register keyboard callback function

    glutMainLoop();
    return 0;
}
```

This program should display either the sphere or cube in an isometric view, allowing you to switch between them by pressing 't'.




Isometric View of Wireframe Sphere and Cube - Akshat Negi