Name – Dhanashri Dilip Mirajkar

Subject – Computer Graphics

Batch – S4

Roll_no – 23574

Assignment no – 2

Implement DDA and Bresenham's line drawing algorithm to draw:

1) Simple Line

2) Dotted Line

3) Dashed Line

4) Solid Line

Using mouse interface Divide the screen in four quadrants with center as (0, 0). The line should work for all the slopes positive as well as negative.

Program –

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

using namespace std;

int ch = 0;
int dx,dy;

void display(int x, int y)
```

```
{
    glColor3f(0,0,0);
    glPointSize(2);
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
}

void displaydotted(int x, int y)
{
    glColor3f(0,0,0);
    glPointSize(2);
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
}

void displaydashed(int x, int y)
{
    glColor3f(0,0,0);
    glPointSize(3);
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
}

void displaysolid(int x, int y)
{
    glColor3f(0,0,0);
    glPointSize(6);
    glBegin(GL_POINTS);
    glVertex2i(x, y);
```

```
    glEnd();
}

void SimpleLine(int x1, int y1, int x2, int y2){
       dx = x2 - x1;
       dy = y2 - y1;

      int steps;

      if(abs(dx) >= abs(dy)){
            steps = abs(dx);
      }
      else{
            steps = abs(dy);
      }

      float Xin = dx / (float) steps;
      float Yin = dy / (float) steps;
      float x = x1;
      float y = y1;

      for(int i=0; i<=steps; i++){
            display(x, y);
            x = x + Xin;
            y = y + Yin;
      }
      glFlush();
}

void DottedLine(int x1, int y1, int x2, int y2){
       dx = x2 - x1;
       dy = y2 - y1;
```

```
	int steps;

	if(abs(dx) >= abs(dy)){
		steps = abs(dx);
	}
	else{
		steps = abs(dy);
	}

	float Xin = dx / (float) steps;
	float Yin = dy / (float) steps;
	float x = x1;
	float y = y1;

	for(int i=0; i<=steps; i++){
		if(i % 6 == 0){
			displaydotted(x, y);
		}
		x = x + Xin;
		y = y + Yin;
	}
	glFlush();
}

void DashLine(int x1, int y1, int x2, int y2){
	 dx = x2 - x1;
	 dy = y2 - y1;

	int steps;

	if(abs(dx) >= abs(dy)){
```

```
            steps = abs(dx);
      }
      else{
            steps = abs(dy);
      }

      float Xin = dx / (float) steps;
      float Yin = dy / (float) steps;
      float x = x1;
      float y = y1;

      for(int i=0; i<=steps; i++){
            if(i % 8 > 4){
                  displaydashed(x, y);
            }
            x = x + Xin;
            y = y + Yin;
      }
      glFlush();
}

void SolidLine(int x1, int y1, int x2, int y2){
       dx = x2 - x1;
       dy = y2 - y1;

      int steps;

      if(abs(dx) >= abs(dy)){
            steps = abs(dx);
      }
      else{
            steps = abs(dy);
```

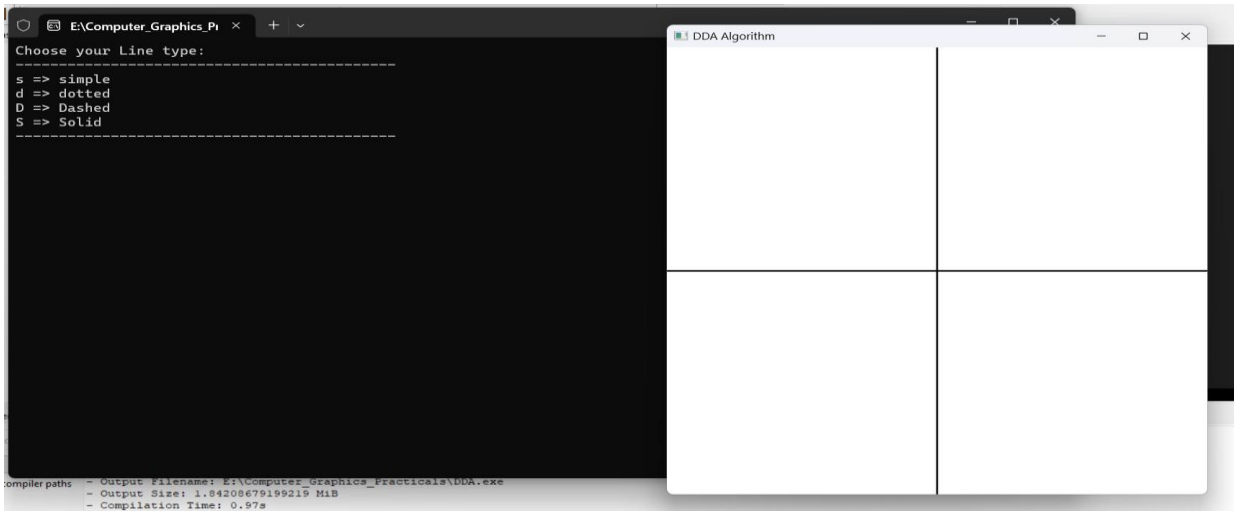
```
      }

      float Xin = dx / (float) steps;
      float Yin = dy / (float) steps;
      float x = x1;
      float y = y1;

      for(int i=0; i<=steps; i++){
            displaysolid(x, y);
            x = x + Xin;
            y = y + Yin;
      }
      glFlush();
}

void mouse(int button, int state, int x, int y)
{
    static int x1, y1, pt = 0;
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {
        if (pt == 0)
        {
            x1 = x;
            y1 = y;
            pt = pt + 1;
        }
        else
        {
            if (ch == 1)
            {
                SimpleLine(x1, y1, x, y);
            }
```

```
            else if (ch == 2)
            {
                DottedLine(x1, y1, x, y);
            }
            else if (ch == 3)
            {
                DashLine(x1, y1, x, y);
            }
            else if(ch == 4)
                {
                SolidLine(x1, y1, x, y);
            }
            x1 = x;
            y1 = y;
        }
    }
    else if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN){
      pt = 0;
      }
    glFlush();
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key)
    {
           case 's':
           {
                ch = 1;
                cout << "Simple Line is opted" << endl;
                glutMouseFunc(mouse);
                break;
```

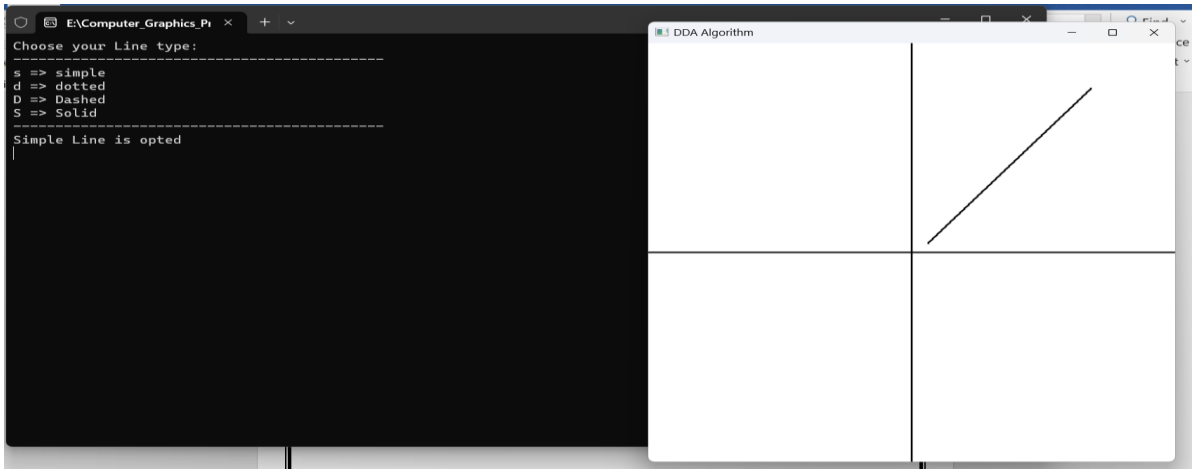
```
            }
            case 'd':
            {
                  ch = 2;
                  cout << "Dotted Line is opted" << endl;
                  glutMouseFunc(mouse);
                  break;
            }
            case 'D':
            {
                  ch = 3;
                  cout << "Dashed Line is opted" << endl;
                  glutMouseFunc(mouse);
                  break;
            }
            case 'S':
            {
                  ch = 4;
                  cout << "Solid Line is opted" << endl;
                  glutMouseFunc(mouse);
                  break;
            }
      }
      glutPostRedisplay();
}
void initialize()
{
      glClearColor(1.0, 1.0, 1.0, 1.0);
      glClear(GL_COLOR_BUFFER_BIT);
      gluOrtho2D(0, 600, 600, 0);
}
```

```
void initialaxis(){
    glColor3f(0,0,0);
    glLineWidth(2);
    glBegin(GL_LINES);
    glVertex2i(300, 0);
    glVertex2i(300, 600);
    glVertex2i(0, 300);
    glVertex2i(600, 300);
    glEnd();
     glFlush();
     glutKeyboardFunc(keyboard);
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(800, 100);
    glutCreateWindow("DDA Algorithm");
    initialize();
    cout << "Choose your Line type: " << endl;
    cout << "-----------------------------------------" << endl;
    cout << "s => simple" << endl;
    cout << "d => dotted" << endl;
    cout << "D => Dashed" << endl;
    cout << "S => Solid" << endl;
    cout << "-----------------------------------------" << endl;
    glutDisplayFunc(initialaxis);
    glutMainLoop();
    return 0;
}
```

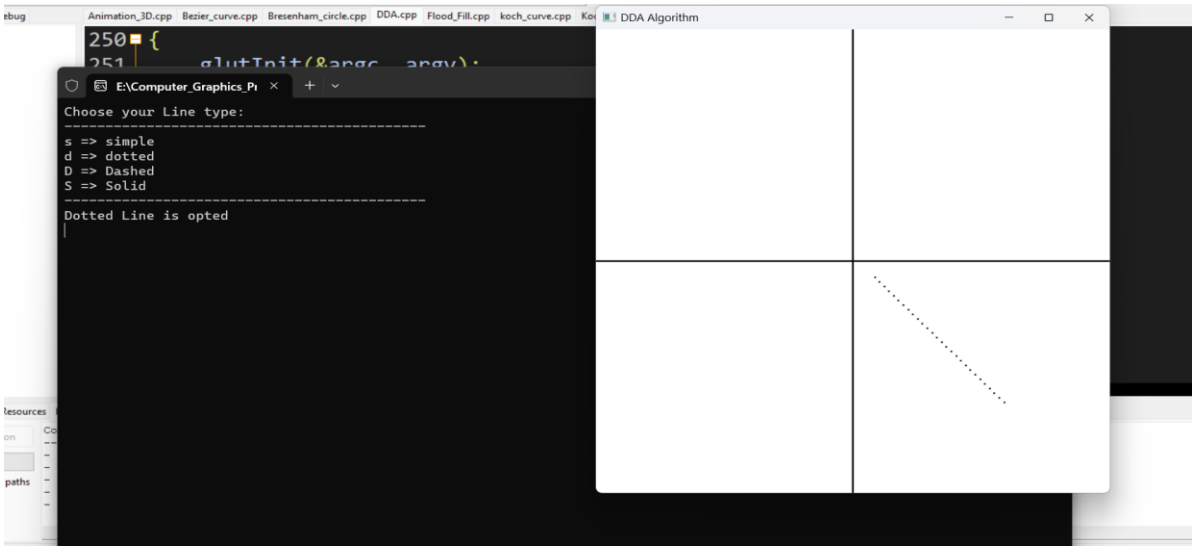
Output –



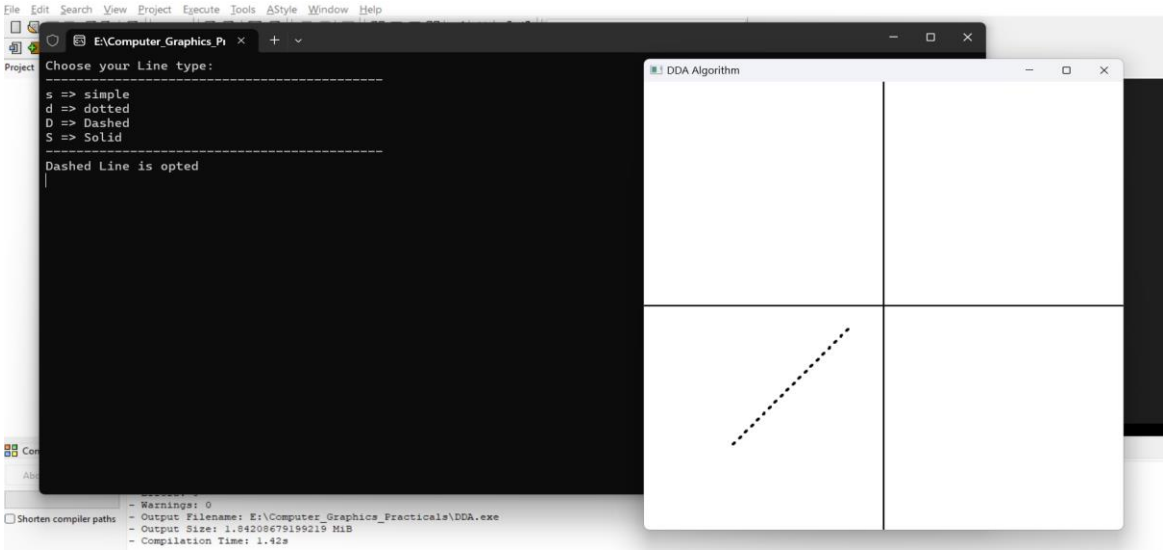
Simple Line –



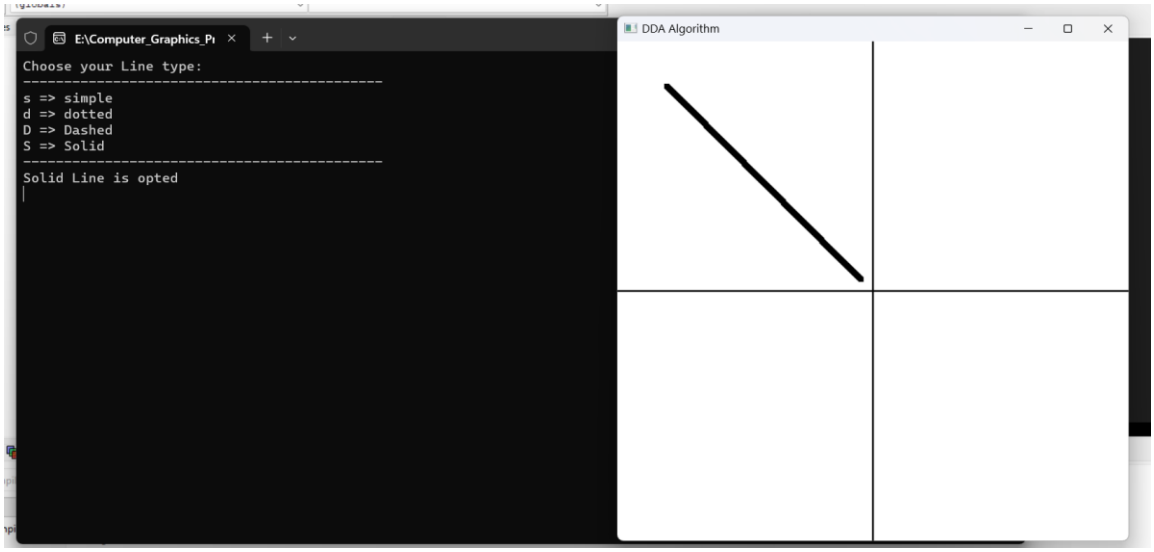
Dotted Line –



Dashed Line –



Solid Line –



ASSIGNMENT NO – 3

Implement Bresenham Circle Drawing Algorithm to draw any object. The object should display in all the quadrants with respect to center and radius.

Program –

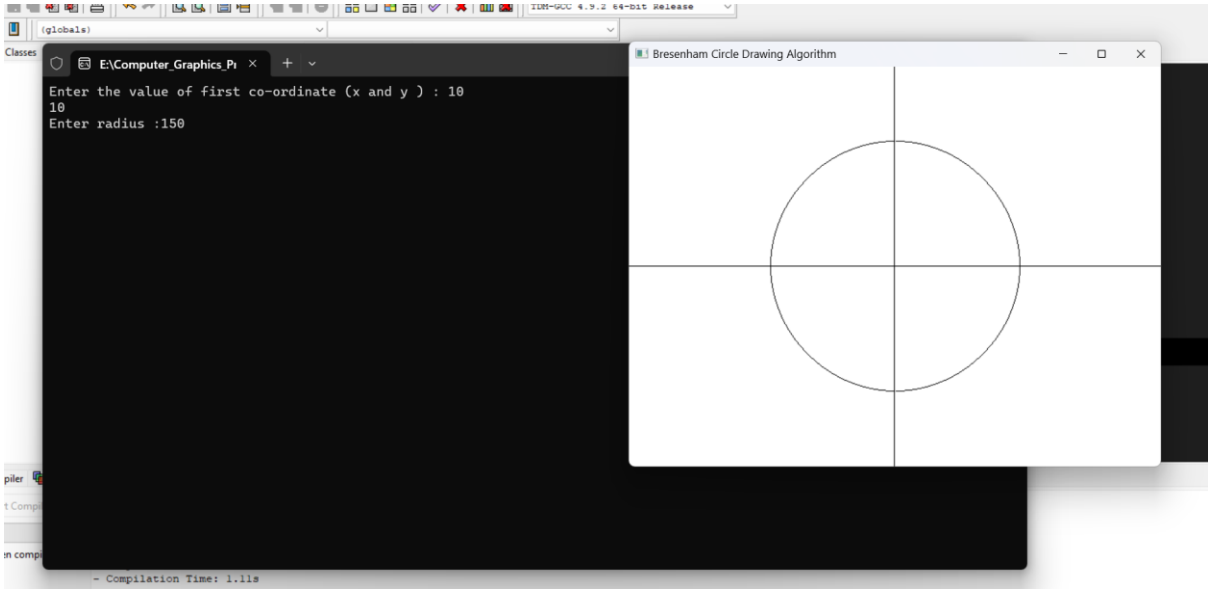
```
#include <iostream>
#include <GL/glut.h>
#include <GL/glu.h>
#include <math.h>

using namespace std;

int x, y, p, radius, xmax, ymax;
bool isCircle_Drawn = false;

int divide_quadrant()
{
    xmax = glutGet(GLUT_WINDOW_WIDTH);
    ymax = glutGet(GLUT_WINDOW_HEIGHT);

    glBegin(GL_LINES);
    glVertex2i(xmax / 2, 0);
    glVertex2i(xmax / 2, ymax);
    glVertex2i(0, ymax / 2);
    glVertex2i(xmax, ymax / 2);
    glEnd();
```

```
    return 0;
}

void putpixel(int x, int y)
{

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

void filling_circle(){
    xmax = glutGet(GLUT_WINDOW_WIDTH);
    ymax = glutGet(GLUT_WINDOW_HEIGHT);

    putpixel((xmax / 2 + x), (ymax / 2 - y));
    putpixel((xmax / 2 + y), (ymax / 2 - x));
    putpixel((xmax / 2 + y), (ymax / 2 + x));
    putpixel((xmax / 2 + x), (ymax / 2 + y));
    putpixel((xmax / 2 - x), (ymax / 2 + y));
    putpixel((xmax / 2 - y), (ymax / 2 + x));
    putpixel((xmax / 2 - y), (ymax / 2 - x));
    putpixel((xmax / 2 - x), (ymax / 2 - y));

    isCircle_Drawn = true;
}

int Bresenham_circle_drawing(){

    cout << "Enter the value of first co-ordinate (x and y ) : ";
```

```
    cin >> x >> y;

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

    x = 0;
    y = radius;

    p = 3 - 2*radius;

    while(x <= y){
        if(p < 0){
            x = 1 + x;
            y = y;
            p = p + (4*x)+6;
            filling_circle();
        }else{
            x = 1 + x;
            y = y - 1;
            p = p + (4*x - 4*y) +10;
            filling_circle();
        }
    }
    return 0;
}

void myDisplay()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3i(0.0f, 0.0f, 0.0f);
```

```
    divide_quadrant();
     if(!isCircle_Drawn){
           Bresenham_circle_drawing();
     }
    glFlush();
}

void init()
{
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 640.0, 0.0, 480.0);
    glFlush();
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(50, 50);
    glutCreateWindow("Bresenham Circle Drawing Algorithm");
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glutDisplayFunc(myDisplay);
    init();
    glutMainLoop();
    return 0;
}
```

Output –


E:\Computer_Graphics_P
Enter the value of first co-ordinate (x and y) : 10
10
Enter radius :150
Bresenham Circle Drawing Algorithm
- Compilation Time: 1.11s

ASSIGNEMENT NO – 4

Implement the following polygon filling methods:

1) flood fill / seed fill

2) Boundary fill; Using mouse click, keyboard interface and menu driven programming.

Program –

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

using namespace std;

bool isPolygonCreated = false;
vector<pair<int, int>> vec;

void setPixel(int x, int y) {
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
}

void floodFill(int x, int y, float fillColor[3], float oldColor[3])
{
    float currentColor[3];
    glReadPixels(x, y, 1, 1, GL_RGB, GL_FLOAT, currentColor);

    if (currentColor[0] != oldColor[0] || currentColor[1] !=
oldColor[1] || currentColor[2] != oldColor[2])
        return;
```

```
    setPixel(x, y);
    glFlush();

    floodFill(x + 1, y, fillColor, oldColor);
    floodFill(x - 1, y, fillColor, oldColor);
    floodFill(x, y + 1, fillColor, oldColor);
    floodFill(x, y - 1, fillColor, oldColor);
}

void boundaryFill(int x, int y, float fillColor[3], float
borderColor[3]) {
    float currentColor[3];
    glReadPixels(x, y, 1, 1, GL_RGB, GL_FLOAT, currentColor);

    if (currentColor[0] == borderColor[0] && currentColor[1] ==
borderColor[1] && currentColor[2] == borderColor[2])
        return;

    if (currentColor[0] != fillColor[0] || currentColor[1] !=
fillColor[1] || currentColor[2] != fillColor[2]) {
        glColor3fv(fillColor);
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL); // Set the fill
mode to GL_FILL
        setPixel(x, y);
        glFlush();

        boundaryFill(x + 1, y, fillColor, borderColor);
        boundaryFill(x - 1, y, fillColor, borderColor);
        boundaryFill(x, y + 1, fillColor, borderColor);
        boundaryFill(x, y - 1, fillColor, borderColor);
```

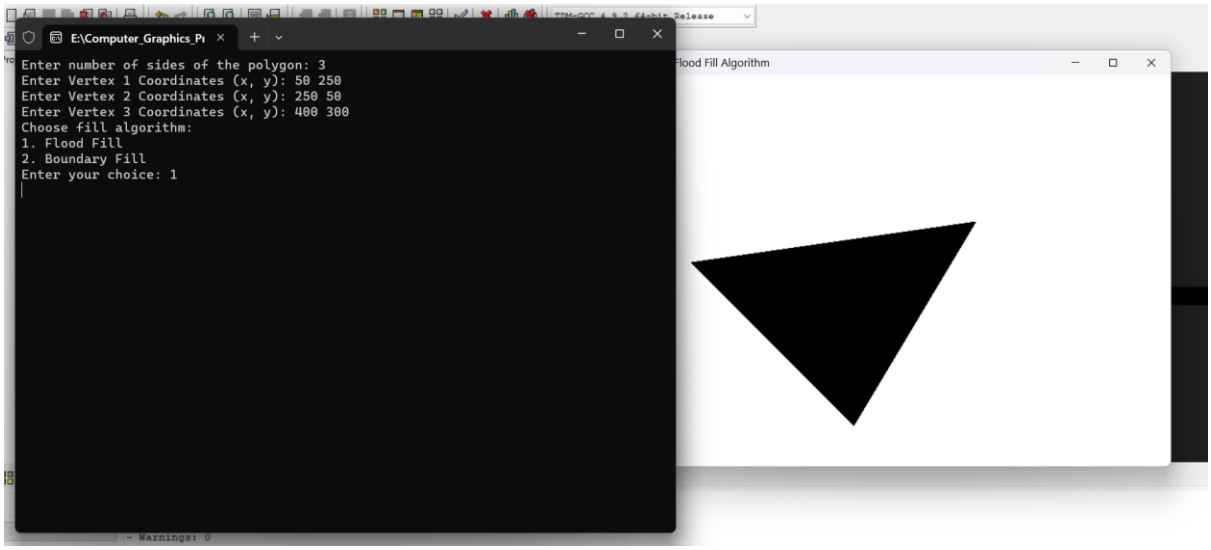
```
    }
}


int createPolygon() {
    int sides;
    cout << "Enter number of sides of the polygon: ";
    cin >> sides;

    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

    vec.clear();

    for (int i = 0; i < sides; i++) {
        int x, y;
        cout << "Enter Vertex " << i + 1 << " Coordinates (x, y): ";
        cin >> x >> y;

        vec.push_back(make_pair(x, y));
    }

    glBegin(GL_POLYGON);
    for (const auto& vertex : vec) {
        glVertex2i(vertex.first, vertex.second);
    }
    glEnd();

    isPolygonCreated = true;

    glFlush(); // Ensure polygon is rendered
```

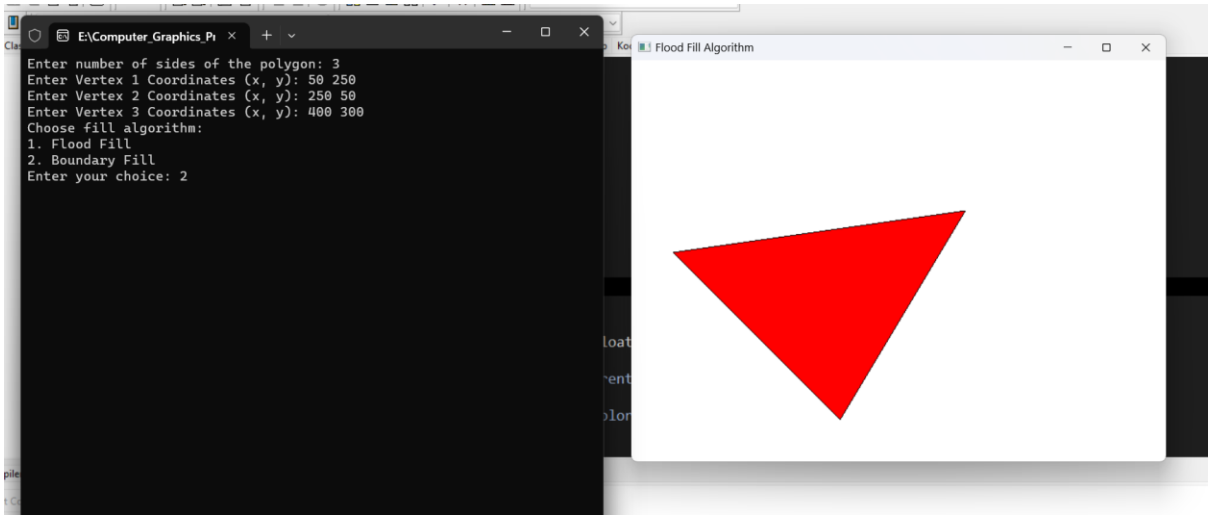
```
    return 0;
}

void mouseClick(int button, int state, int x, int y) {
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
        if (isPolygonCreated) {
            int choice;
            cout << "Choose fill algorithm:\n";
            cout << "1. Flood Fill\n";
            cout << "2. Boundary Fill\n";
            cout << "Enter your choice: ";
            cin >> choice;

            float fillColor[3] = {1.0f, 0.0f, 0.0f}; // Set fill
color to red

            switch (choice) {
                case 1: {
                    float oldColor[3] = {1.0f, 1.0f, 1.0f}; // Set
old color to white for flood fill
                    floodFill(x, glutGet(GLUT_WINDOW_HEIGHT) - y,
fillColor, oldColor);
                    break;
                }
                case 2: {
                    float borderColor[3] = {0.0f, 0.0f, 0.0f}; //
Set border color to black for boundary fill
                    boundaryFill(x, glutGet(GLUT_WINDOW_HEIGHT) - y,
fillColor, borderColor);
                    break;
                }
```

```
                default:
                    cout << "Invalid choice\n";
            }
        } else {
            cout << "Create a polygon first\n";
        }
    }
}
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);
    glColor3f(0.0f, 0.0f, 0.0f);

    if (isPolygonCreated == false) {
        createPolygon();
    }
    glFlush();
}
void init() {
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, glutGet(GLUT_WINDOW_WIDTH), 0.0,
glutGet(GLUT_WINDOW_HEIGHT));
}
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(800, 100);
```

```
    glutCreateWindow("Flood Fill Algorithm");
    glutDisplayFunc(display);
    glutMouseFunc(mouseClick);
    init();
    glutMainLoop();
    return 0;
}
```

Output –





ASSIGNEMENT NO – 5

Implement Cohen Sutherland Polygon clipping method to clip the polygon with respect the viewport and window. Use mouse click, keyboard interface.

Program –

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

using namespace std;

vector<pair<int, int>> vec;
vector<pair<int, int>> clippedVertices;

int sides;
bool isPolygonCreated = false, ismouseClicked = false;

const int INSIDE = 0;   // Inside the clipping region
const int LEFT = 1;     // Left of the clipping region
const int RIGHT = 2;    // Right of the clipping region
const int TOP = 3;      // Top of the clipping region
const int BOTTOM = 4;   // Bottom of the clipping region

// Constants for the clipping region
const int viewportXMin = 100;
const int viewportXMax = 500;
```

```
const int viewportYMin = 100;
const int viewportYMax = 500;

void createViewPort() {
    glColor3f(0.0f, 0.0f, 0.0f); // Set color to Black
    glBegin(GL_LINE_LOOP);
    glVertex2i(100, 100);    // Top-left corner
    glVertex2i(100, 300);  // Top-right corner
    glVertex2i(300, 300);  // Bottom-right corner
    glVertex2i(300, 100);    // Bottom-left corner
    glEnd();
    glFlush();
}

void text() {
      glColor3f(0.0f, 0.0f, 0.0f);
    glRasterPos2i(10, 450);

    string text_value = "Polygon Clipping using Cohen Sutherland
Algorithm";

    for (string::size_type i = 0; i < text_value.length(); ++i) {
        char c = text_value[i];
        glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, c);
    }
    glFlush();
}

int createPolygon() {
    glColor3f(1.0f, 0.0f, 0.0f);
```

```
    cout << "Enter number of sides of the polygon: ";
    cin >> sides;

    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

    vec.clear();

    for (int i = 0; i < sides; i++) {
        int x, y;
        cout << "Enter Vertex " << i + 1 << " Coordinates (x, y): ";
        cin >> x >> y;

       // Clip the coordinates to ensure they are within the
viewport
        int clippedX = max(viewportXMin, min(viewportXMax, x));
        int clippedY = max(viewportYMin, min(viewportYMax, y));

        vec.push_back(make_pair(x, y));
    }

    glBegin(GL_POLYGON);
    for (const auto& vertex : vec) {
        glVertex2i(vertex.first, vertex.second);
    }

     isPolygonCreated = true;

    glEnd();
    return 0;
```

```
}


int drawPolygon(vector<pair<int, int> > vec){

      glClear(GL_COLOR_BUFFER_BIT);
    createViewPort();
    text();
    glColor3f(1.0f, 0.0f, 0.0f);

    glBegin(GL_POLYGON);
    for (const auto& vertex : vec) {
        glVertex2i(vertex.first, vertex.second);
    }
    glEnd();

      return 0;
}

void clipLeft() {
    // Define the clipping region
    int xMin = 100;

    // Clear the clipped vertices
    clippedVertices.clear();

    // Iterate over each vertex of the polygon
    for (int i = 0; i < sides; i++) {
        int x1 = vec[i].first;
        int y1 = vec[i].second;
```

```
        int x2 = vec[(i + 1) % sides].first;
        int y2 = vec[(i + 1) % sides].second;

        int code1 = (x1 < xMin) ? LEFT : INSIDE;
        int code2 = (x2 < xMin) ? LEFT : INSIDE;

        if (code1 == INSIDE && code2 == INSIDE) {
            // Both endpoints are inside the clipping region
            clippedVertices.push_back(make_pair(x2, y2));
        } else if (code1 == INSIDE && code2 == LEFT) {
            // The first endpoint is inside, second endpoint is
outside
            int clippedX = xMin;
            int clippedY = y1 + (clippedX - x1) * (y2 - y1) / (x2 -
x1);
            clippedVertices.push_back(make_pair(clippedX,
clippedY));
        } else if (code1 == LEFT && code2 == INSIDE) {
            // The first endpoint is outside, second endpoint is
inside
            int clippedX = xMin;
            int clippedY = y1 + (clippedX - x1) * (y2 - y1) / (x2 -
x1);
            clippedVertices.push_back(make_pair(clippedX,
clippedY));
            clippedVertices.push_back(make_pair(x2, y2));
        }
    }

    // Update the polygon vertices with the clipped vertices
    vec = clippedVertices;
    sides = vec.size();
```

```
    drawPolygon(vec);
}

void clipRight() {
    // Define the clipping region
    int xMax = 300;

    // Clear the clipped vertices
    clippedVertices.clear();

    // Iterate over each vertex of the polygon
    for (int i = 0; i < sides; i++) {
        int x1 = vec[i].first;
        int y1 = vec[i].second;
        int x2 = vec[(i + 1) % sides].first;
        int y2 = vec[(i + 1) % sides].second;

        int code1 = (x1 > xMax) ? RIGHT : INSIDE;
        int code2 = (x2 > xMax) ? RIGHT : INSIDE;

        if (code1 == INSIDE && code2 == INSIDE) {
            // Both endpoints are inside the clipping region
            clippedVertices.push_back(make_pair(x2, y2));
        } else if (code1 == INSIDE && code2 == RIGHT) {
            // The first endpoint is inside, second endpoint is
outside
            int clippedX = xMax;
            int clippedY = y1 + (clippedX - x1) * (y2 - y1) / (x2 -
x1);
```

```
            clippedVertices.push_back(make_pair(clippedX,
clippedY));
        } else if (code1 == RIGHT && code2 == INSIDE) {
            // The first endpoint is outside, second endpoint is
inside
            int clippedX = xMax;
            int clippedY = y1 + (clippedX - x1) * (y2 - y1) / (x2 -
x1);
            clippedVertices.push_back(make_pair(clippedX,
clippedY));
            clippedVertices.push_back(make_pair(x2, y2));
        }
    }

    // Update the polygon vertices with the clipped vertices
    vec = clippedVertices;
    sides = vec.size();



    drawPolygon(vec);
}

void clipTop() {
    // Define the clipping region
    int yMax = 300;

    // Clear the clipped vertices
    clippedVertices.clear();

    // Iterate over each vertex of the polygon
    for (int i = 0; i < sides; i++) {
```

```
        int x1 = vec[i].first;
        int y1 = vec[i].second;
        int x2 = vec[(i + 1) % sides].first;
        int y2 = vec[(i + 1) % sides].second;

        int code1 = (y1 > yMax) ? TOP : INSIDE;
        int code2 = (y2 > yMax) ? TOP : INSIDE;

        if (code1 == INSIDE && code2 == INSIDE) {
            // Both endpoints are inside the clipping region
            clippedVertices.push_back(make_pair(x2, y2));
        } else if (code1 == INSIDE && code2 == TOP) {
            // The first endpoint is inside, second endpoint is
outside
            int clippedY = yMax;
            int clippedX = x1 + (clippedY - y1) * (x2 - x1) / (y2 -
y1);
            clippedVertices.push_back(make_pair(clippedX,
clippedY));
        } else if (code1 == TOP && code2 == INSIDE) {
            // The first endpoint is outside, second endpoint is
inside
            int clippedY = yMax;
            int clippedX = x1 + (clippedY - y1) * (x2 - x1) / (y2 -
y1);
            clippedVertices.push_back(make_pair(clippedX,
clippedY));
            clippedVertices.push_back(make_pair(x2, y2));
        }
    }

    // Update the polygon vertices with the clipped vertices
```

```
    vec = clippedVertices;
    sides = vec.size();



    drawPolygon(vec);
}

void clipBottom() {
    // Define the clipping region
    int yMin = 100;

    // Clear the clipped vertices
    clippedVertices.clear();

    // Iterate over each vertex of the polygon
    for (int i = 0; i < sides; i++) {
        int x1 = vec[i].first;
        int y1 = vec[i].second;
        int x2 = vec[(i + 1) % sides].first;
        int y2 = vec[(i + 1) % sides].second;

        int code1 = (y1 < yMin) ? BOTTOM : INSIDE;
        int code2 = (y2 < yMin) ? BOTTOM : INSIDE;

        if (code1 == INSIDE && code2 == INSIDE) {
            // Both endpoints are inside the clipping region
            clippedVertices.push_back(make_pair(x2, y2));
        } else if (code1 == INSIDE && code2 == BOTTOM) {
```

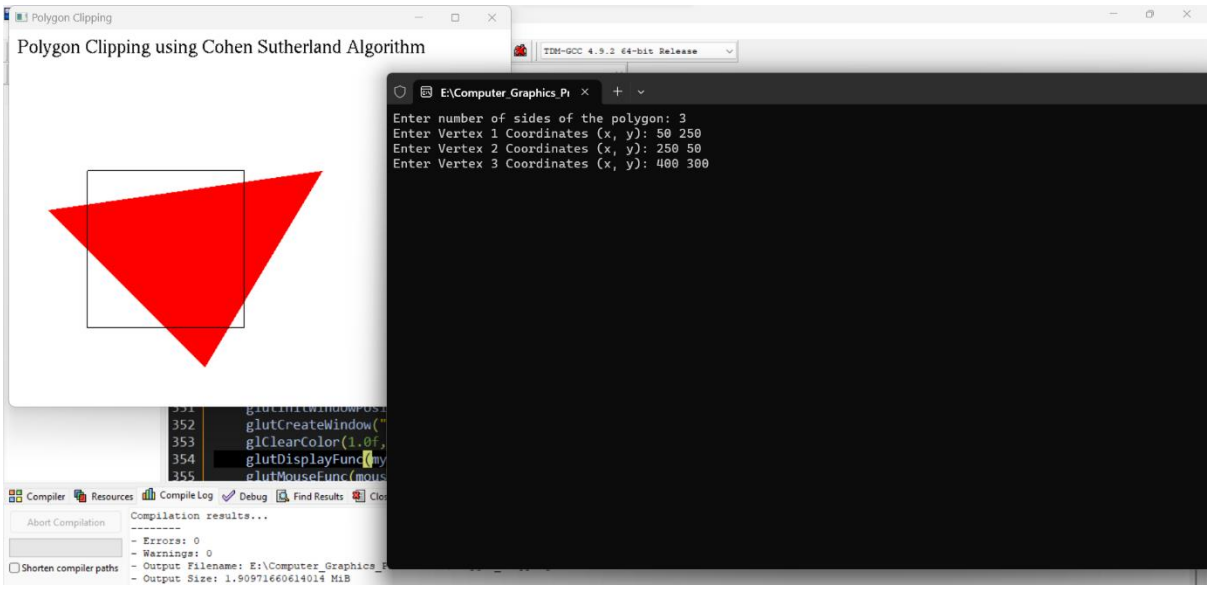
```
            // The first endpoint is inside, second endpoint is
outside
            int clippedY = yMin;
            int clippedX = x1 + (clippedY - y1) * (x2 - x1) / (y2 -
y1);
            clippedVertices.push_back(make_pair(clippedX,
clippedY));
        } else if (code1 == BOTTOM && code2 == INSIDE) {
            // The first endpoint is outside, second endpoint is
inside
            int clippedY = yMin;
            int clippedX = x1 + (clippedY - y1) * (x2 - x1) / (y2 -
y1);
            clippedVertices.push_back(make_pair(clippedX,
clippedY));
            clippedVertices.push_back(make_pair(x2, y2));
        }
    }

    // Update the polygon vertices with the clipped vertices
    vec = clippedVertices;
    sides = vec.size();



    drawPolygon(vec);
}



void menu(int id){
     switch(id){
           case 1:
                clipLeft();
```

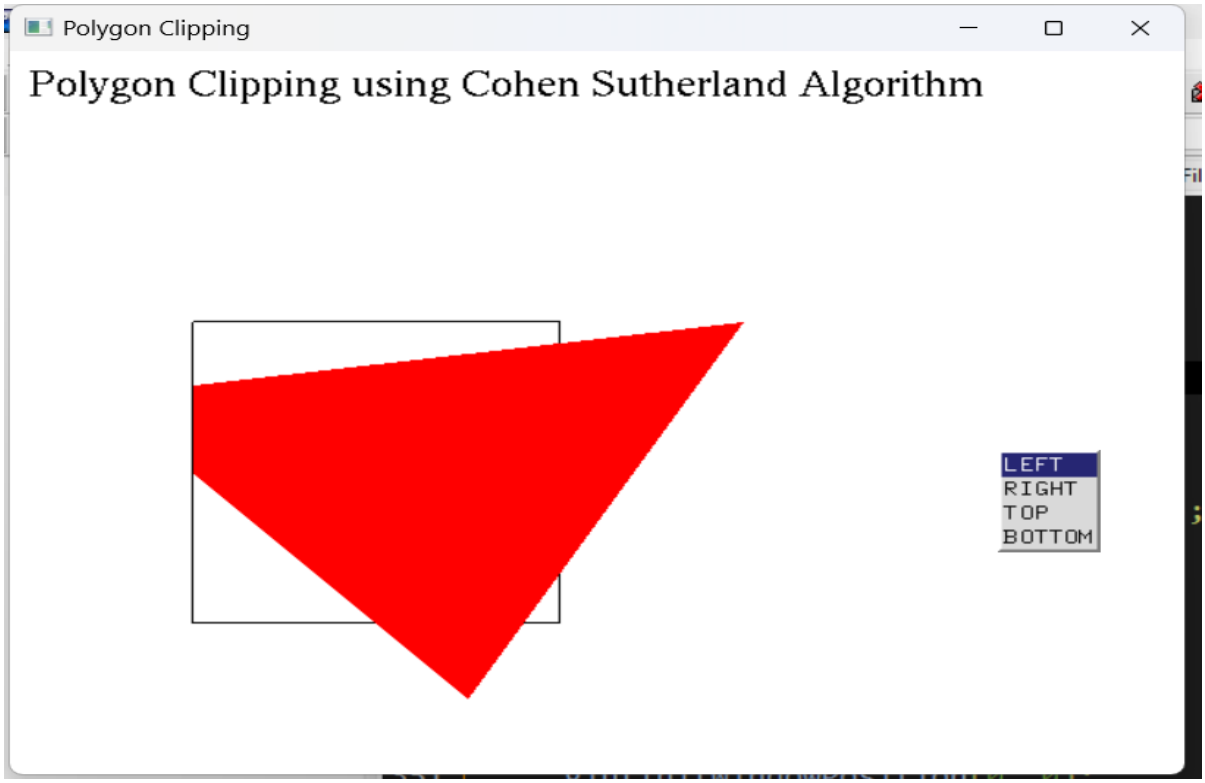
```
                break;

            case 2:
                clipRight();
                break;

            case 3:
                clipTop();
                break;

            case 4:
                clipBottom();
                break;

            default:
                cout<<"Please Enter a valid choice and Try it again
:("<<endl;
                break;
        }
}


void createMenu(){
      glutCreateMenu(menu);
      glutAddMenuEntry("LEFT", 1);
      glutAddMenuEntry("RIGHT", 2);
      glutAddMenuEntry("TOP", 3);
      glutAddMenuEntry("BOTTOM", 4);
      glutAttachMenu(GLUT_RIGHT_BUTTON);
      glFlush();
```

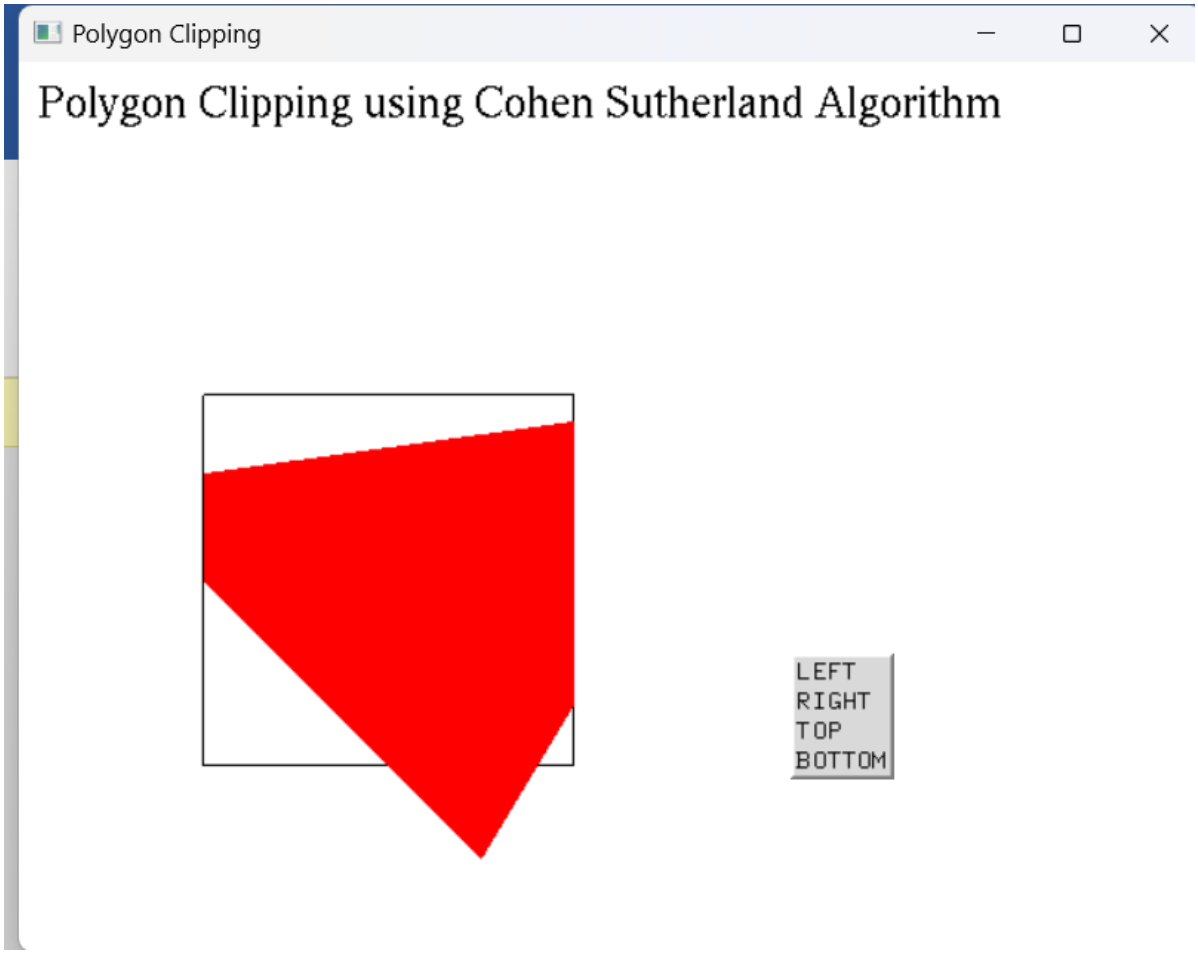
```
}

void mouseClick(int button, int state, int x, int y) {
    if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN) {
        if (!ismouseClicked) {
            vec.push_back(make_pair(x, 480 - y));
        }
        ismouseClicked = true;
        glutPostRedisplay();
    }
}

void myDisplay() {
      glClear(GL_COLOR_BUFFER_BIT);

      if(!isPolygonCreated){
            createPolygon();
      }
    text();
    createViewPort();
    createMenu();
    glFlush();
}
int myInit() {
      glColor3f(1.0f, 0.0f,0.0f);
      glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 640.0, 0.0, 480.0);
    glFlush();
```

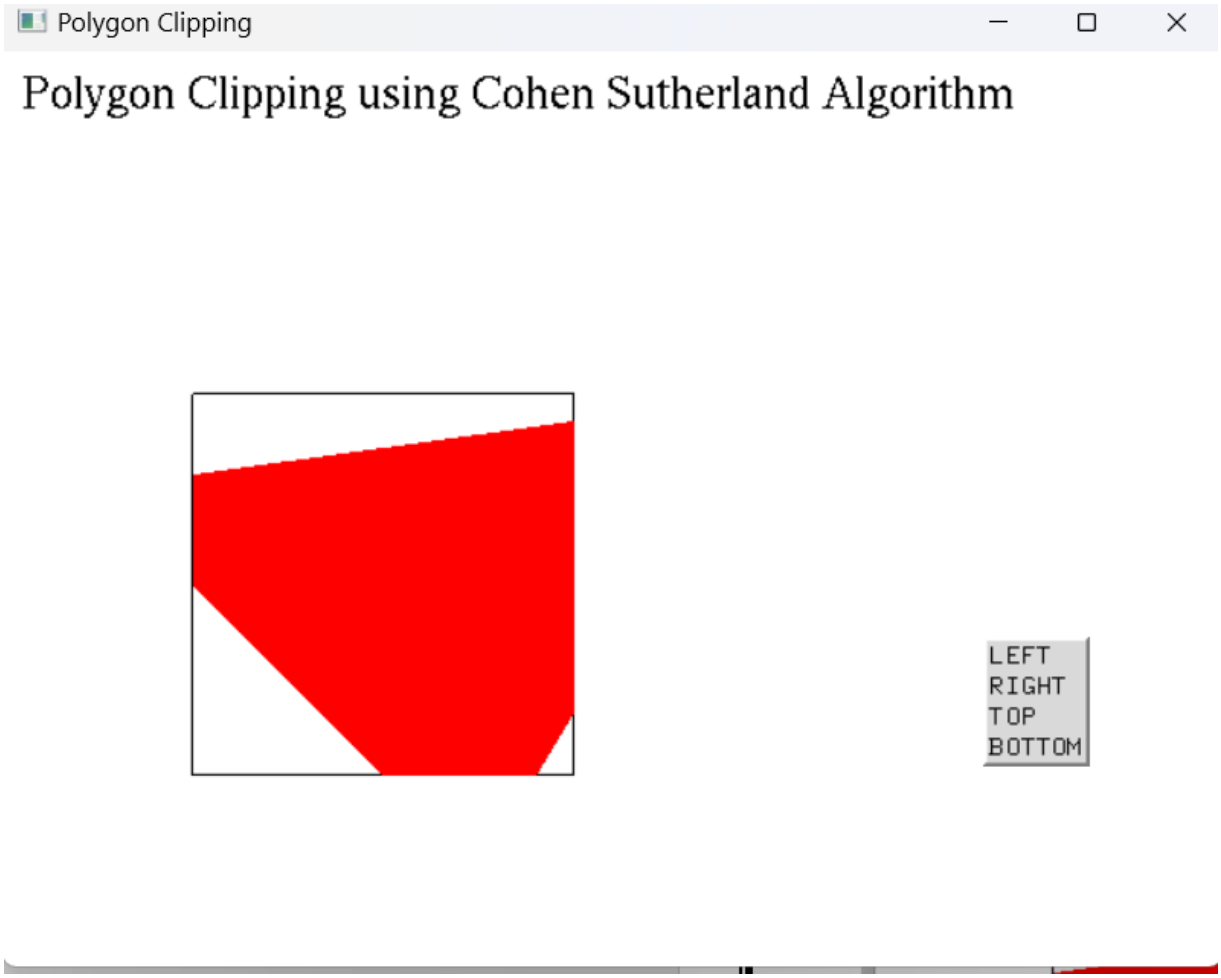
```
    return 0;
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Polygon Clipping");
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glutDisplayFunc(myDisplay);
    glutMouseFunc(mouseClick);
    myInit();
    glutMainLoop();
    return 0;
}
```

Output –




Polygon Clipping
Polygon Clipping using Cohen Sutherland Algorithm
LEFT
RIGHT
TOP
BOTTOM


Polygon Clipping
Polygon Clipping using Cohen Sutherland Algorithm
LEFT
RIGHT
TOP
BOTTOM


Polygon Clipping
Polygon Clipping using Cohen Sutherland Algorithm
LEFT
RIGHT
TOP
BOTTOM

ASSIGNEMENT NO – 6

Implement following 2D transformation on the object with respect to axis:

1) **Scaling**
2) **Rotation about arbitrary point.**
3) **Reflection**

Program –

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

#define M_PI 3.14

using namespace std;

int ch = 0;
vector<int> arr;
int ct = 0;
float colorarr[] = {1.0,0.0,0.0};
float flc[] = {};
float neg[] = {0.0,1.0,1.0};
void copyarr(float* arr1){
      for(int i=0; i<3; i++){
            flc[i] = arr1[i];
            if(arr1[i] == 0.0f){
                  neg[i] = 1.0f;
```

```
            }
            if(arr1[i] == 1.0f){
                  neg[i] = 0.0f;
            }
      }
}

void drawTriangle(int x1, int y1, int x2, int y2, int x3, int y3,
float* flc){
    glColor3f(flc[0],flc[1],flc[2]);
      glLineWidth(3);
    glBegin(GL_LINE_LOOP);
      glVertex2i(x1,y1);
      glVertex2i(x2,y2);
      glVertex2i(x3,y3);
    glEnd();
      glLineWidth(1);
      glFlush();
}

void drawCircle(int x1, int y1, int x2, int y2, float* flc){
      int r1 = abs(x2-x1);
      int r2 = abs(y2-y1);
      int r = sqrt(pow(r1,2)+pow(r2,2));
      float angle;
    glColor3f(flc[0],flc[1],flc[2]);
    glPointSize(4);
    glBegin(GL_POINTS);
        for (int i = 0; i < 360; i++) {
            angle = i * M_PI / 180;
```

```
            glVertex2f(x1 + r * cos(angle),
                       y1 + r * sin(angle));
        }
    glEnd();
    glPointSize(1);
     glFlush();
}

void drawQuadritaleral(int x1, int y1, int x2, int y2, int x3, int
y3, int x4, int y4, float* flc){
    glColor3f(flc[0],flc[1],flc[2]);
     glLineWidth(3);
    glBegin(GL_LINE_LOOP);
     glVertex2i(x1,y1);
     glVertex2i(x2,y2);
     glVertex2i(x3,y3);
     glVertex2i(x4,y4);
    glEnd();
     glLineWidth(1);
     glFlush();
}

void TrnsScaling(vector<int> arr){
     vector<int> sarr;
     int sz = arr.size();
     float Sx, Sy;
     cout << "Enter Sx scaling factor: ";
     cin >> Sx;
     cout << "Enter Sy scaling factor: ";
     cin >> Sy;
```

```
	for(int i=0; i<sz; i++){
		if(i % 2 == 0){
			sarr.push_back(arr.at(i)*Sx);
		}
		if(i % 2 == 1){
			sarr.push_back(arr.at(i)*Sy);
		}
	}
	if (sz == 6){

	drawTriangle(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),sarr.
at(4),sarr.at(5),neg);
	}
	else if(sz == 4){

	drawCircle(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),neg);
	}
	else if(sz == 8){

	drawQuadritaleral(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),
sarr.at(4),sarr.at(5),sarr.at(6),sarr.at(7),neg);
	}
	sarr.clear();
}


void TrnsReflectionX(vector<int> arr){
	vector<int> sarr;
	int sz = arr.size();
	for(int i=0; i<sz; i++){
		if(i % 2 == 0){
			sarr.push_back(arr.at(i)*(1));
		}
```

```
            if(i % 2 == 1){
                  sarr.push_back(arr.at(i)*(-1));
            }
      }
      if (sz == 6){

      drawTriangle(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),sarr.
at(4),sarr.at(5),neg);
      }
      else if(sz == 4){

      drawCircle(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),neg);
      }
      else if(sz == 8){

      drawQuadritaleral(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),
sarr.at(4),sarr.at(5),sarr.at(6),sarr.at(7),neg);
      }
      sarr.clear();
}


void TrnsReflectionY(vector<int> arr){
      vector<int> sarr;
      int sz = arr.size();
      for(int i=0; i<sz; i++){
            if(i % 2 == 0){
                  sarr.push_back(arr.at(i)*(-1));
            }
            if(i % 2 == 1){
                  sarr.push_back(arr.at(i)*(1));
            }
      }
```

```
	if (sz == 6){

	drawTriangle(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),sarr.
at(4),sarr.at(5),neg);
	}
	else if(sz == 4){

	drawCircle(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),neg);
	}
	else if(sz == 8){

	drawQuadritaleral(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),
sarr.at(4),sarr.at(5),sarr.at(6),sarr.at(7),neg);
	}
	sarr.clear();
}

void TrnsReflectionXY(vector<int> arr){
	vector<int> sarr;
	int sz = arr.size();
	for(int i=0; i<sz; i++){
		if(i % 2 == 0){
			sarr.push_back(arr.at(i)*(-1));
		}
		if(i % 2 == 1){
			sarr.push_back(arr.at(i)*(-1));
		}
	}
	if (sz == 6){

	drawTriangle(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),sarr.
at(4),sarr.at(5),neg);
```

```
	}
	else if(sz == 4){

	drawCircle(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),neg);
	}
	else if(sz == 8){

	drawQuadritaleral(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),
sarr.at(4),sarr.at(5),sarr.at(6),sarr.at(7),neg);
	}
	sarr.clear();
}


void TrnsReflectionXYLine(vector<int> arr){
	vector<int> sarr;
	int sz = arr.size();
	for(int i=0; i<sz; i++){
		if(i % 2 == 0){
			sarr.push_back(arr.at(i)*(-1));
		}
		if(i % 2 == 1){
			sarr.push_back(arr.at(i)*(-1));
		}
	}
	if (sz == 6){

	drawTriangle(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),sarr.
at(4),sarr.at(5),neg);
	}
	else if(sz == 4){

	drawCircle(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),neg);
```

```
      }
      else if(sz == 8){

      drawQuadritaleral(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),
sarr.at(4),sarr.at(5),sarr.at(6),sarr.at(7),neg);
      }
      sarr.clear();
}


void Rotationabtcen(vector<int> arr){
      int angle;
      cout << "Enter angle in degree's to rotate: ";
    cin >> angle;
    float theta = angle * M_PI / 180;
      vector<int> sarr;
      int sz = arr.size();
      for(int i=0; i<sz; i++){
            if(i % 2 == 0){
                  int nx = (arr.at(i)*cos(theta)) -
(arr.at(i+1)*sin(theta));
                  sarr.push_back(nx);
            }
            if(i % 2 == 1){
                  int ny = (arr.at(i-1)*sin(theta)) +
(arr.at(i)*cos(theta));
                  sarr.push_back(ny);
            }
      }
      if (sz == 6){

      drawTriangle(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),sarr.
at(4),sarr.at(5),neg);
```

```
      }
      else if(sz == 4){

      drawCircle(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),neg);
      }
      else if(sz == 8){

      drawQuadritaleral(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),
sarr.at(4),sarr.at(5),sarr.at(6),sarr.at(7),neg);
      }
      sarr.clear();
}


void Rotationabtpt(vector<int> arr, int x, int y){
      int xr = x;
      int yr = y;
      int angle;
      cout << "Enter angle in degree's to rotate: ";
    cin >> angle;
    float theta = angle * M_PI / 180;
      vector<int> sarr;
      int sz = arr.size();
      for(int i=0; i<sz; i++){
            if(i % 2 == 0){
                  int nx = xr + ((arr.at(i) - xr)*cos(theta)) -
((arr.at(i+1) - yr)*sin(theta));
                  sarr.push_back(nx);
            }
            if(i % 2 == 1){
                  int ny = yr + ((arr.at(i-1) - xr)*sin(theta)) +
((arr.at(i) - yr)*cos(theta));
                  sarr.push_back(ny);
```

```
            }
      }
      if (sz == 6){

      drawTriangle(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),sarr.
at(4),sarr.at(5),neg);
      }
      else if(sz == 4){

      drawCircle(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),neg);
      }
      else if(sz == 8){

      drawQuadritaleral(sarr.at(0),sarr.at(1),sarr.at(2),sarr.at(3),
sarr.at(4),sarr.at(5),sarr.at(6),sarr.at(7),neg);
      }
      sarr.clear();
}


void mouse(int button, int state, int x, int y)
{
      static int xx, yy;
      xx = x - 300;
      yy = 300 - y;
      int sz = arr.size();
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {
        if (ch == 1)
           {
                if (sz < 6){
                      arr.push_back(xx);
                      arr.push_back(yy);
```

```
                }
                sz = arr.size();
                if (sz == 6){

     drawTriangle(arr.at(0),arr.at(1),arr.at(2),arr.at(3),arr.at(4)
,arr.at(5),flc);
                }
        }
        if (ch == 2)
           {
                if (sz < 4){
                     arr.push_back(xx);
                     arr.push_back(yy);
                }
                sz = arr.size();
                if (sz == 4){

     drawCircle(arr.at(0),arr.at(1),arr.at(2),arr.at(3),flc);
                }
        }
        if (ch == 3)
           {
                if (sz < 8){
                     arr.push_back(xx);
                     arr.push_back(yy);
                }
                sz = arr.size();
                if (sz == 8){

     drawQuadritaleral(arr.at(0),arr.at(1),arr.at(2),arr.at(3),arr.
at(4),arr.at(5),arr.at(6),arr.at(7),flc);
                }
```

```
        }
        if (ch == 4)
           {
                 Rotationabtpt(arr, xx, yy);
        }
    }
    else if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
     {
          if(ct % 3 == 0){
                colorarr[0] = 1.0;
                colorarr[1] = 0.0;
                colorarr[2] = 0.0;
                cout << "Red color is choosen" << endl;
          }
          else if(ct % 3 == 1){
                colorarr[0] = 0.0;
                colorarr[1] = 1.0;
                colorarr[2] = 0.0;
                cout << "Green  color is choosen" << endl;
          }
          else if(ct % 3 == 2){
                colorarr[0] = 0.0;
                colorarr[1] = 0.0;
                colorarr[2] = 1.0;
                cout << "Blue  color is choosen" << endl;
          }
          ct++;
     }
    glFlush();
}
```

```
void keyboard(unsigned char key, int x, int y)
{
    switch (key)
    {
           case 't':
           {
                 ch = 1;
                 copyarr(colorarr);
                 cout << "Triangle is opted" << endl;
                 glutMouseFunc(mouse);
                 break;
           }
           case 'c':
           {
                 ch = 2;
                 copyarr(colorarr);
                 cout << "Circle is opted" << endl;
                 glutMouseFunc(mouse);
                 break;
           }
           case 'q':
           {
                 ch = 3;
                 copyarr(colorarr);
                 cout << "Quadrilateral is opted" << endl;
                 glutMouseFunc(mouse);
                 break;
           }
           case 's':
```

```
		{
			copyarr(colorarr);
			cout << "Scaling Transformation is opted" << endl;
			TrnsScaling(arr);
			break;
		}
		case 'r':
		{
			copyarr(colorarr);
			cout << "Rotation about center is opted" << endl;
			Rotationabtcen(arr);
			break;
		}
		case 'R':
		{
			ch = 4;
			copyarr(colorarr);
			cout << "Rotation about any arbitary point is opted"
<< endl;
			glutMouseFunc(mouse);
			cout << "Click on the arbitary point" << endl;
			break;
		}
		case 'h':
		{
			copyarr(colorarr);
			cout << "Reflection about X-axis is opted" << endl;
			TrnsReflectionX(arr);
			break;
		}
```

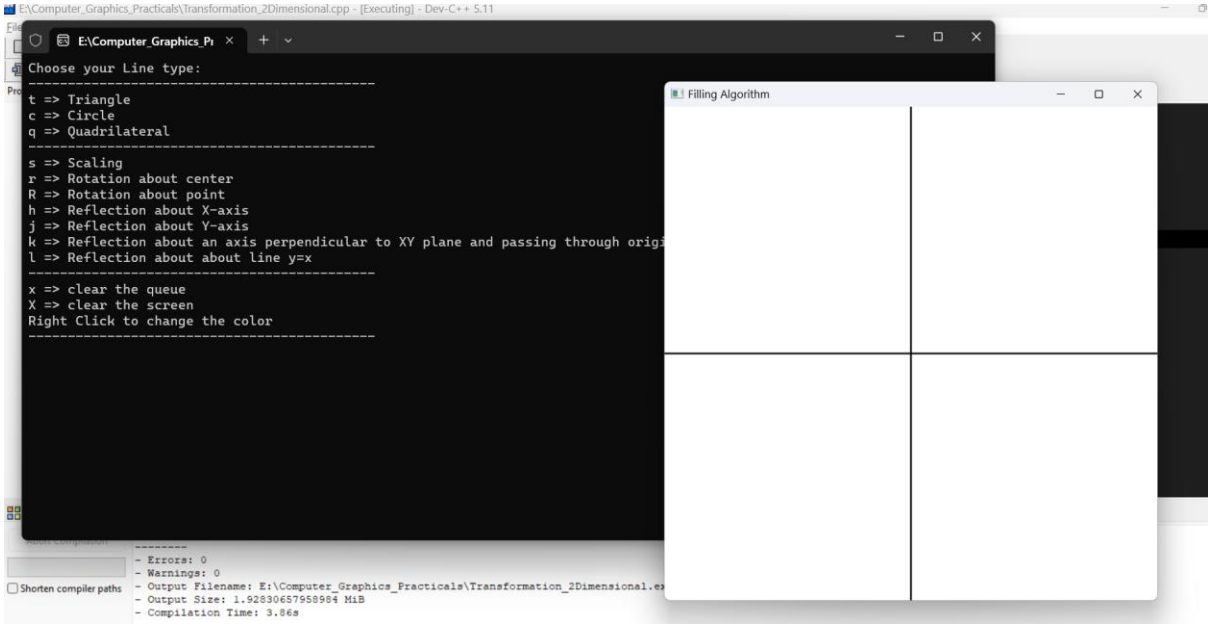
```
            case 'j':
            {
                  copyarr(colorarr);
                  cout << "Reflection about Y-axis is opted" << endl;
                  TrnsReflectionY(arr);
                  break;
            }
            case 'k':
            {
                  copyarr(colorarr);
                  cout << "Reflection about an axis perpendicular to
XY plane and passing through origin" << endl;
                  TrnsReflectionXY(arr);
                  break;
            }
            case 'l':
            {
                  copyarr(colorarr);
                  cout << "Reflection about about line y=x" << endl;
                  TrnsReflectionXYLine(arr);
                  break;
            }
            case 'x':
            {
                  arr.clear();
                  cout << "Queue is cleared" << endl;
                  break;
            }
            case 'X':
            {
```

```
                arr.clear();
                glClearColor(1.0, 1.0, 1.0, 1.0);
            glClear(GL_COLOR_BUFFER_BIT);
                cout << "Screen is cleared" << endl;
                break;
            }
    }
    glutPostRedisplay();
}

void initialize()
{
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    gluOrtho2D(-300, 300, -300, 300);
}

void initialaxis(){
     glColor3f(0,0,0);
     glLineWidth(2);
    glBegin(GL_LINES);
     glVertex2i(-300,0);
     glVertex2i(300,0);
     glVertex2i(0,-300);
     glVertex2i(0,300);
    glEnd();
     glFlush();
    glutKeyboardFunc(keyboard);
}
```

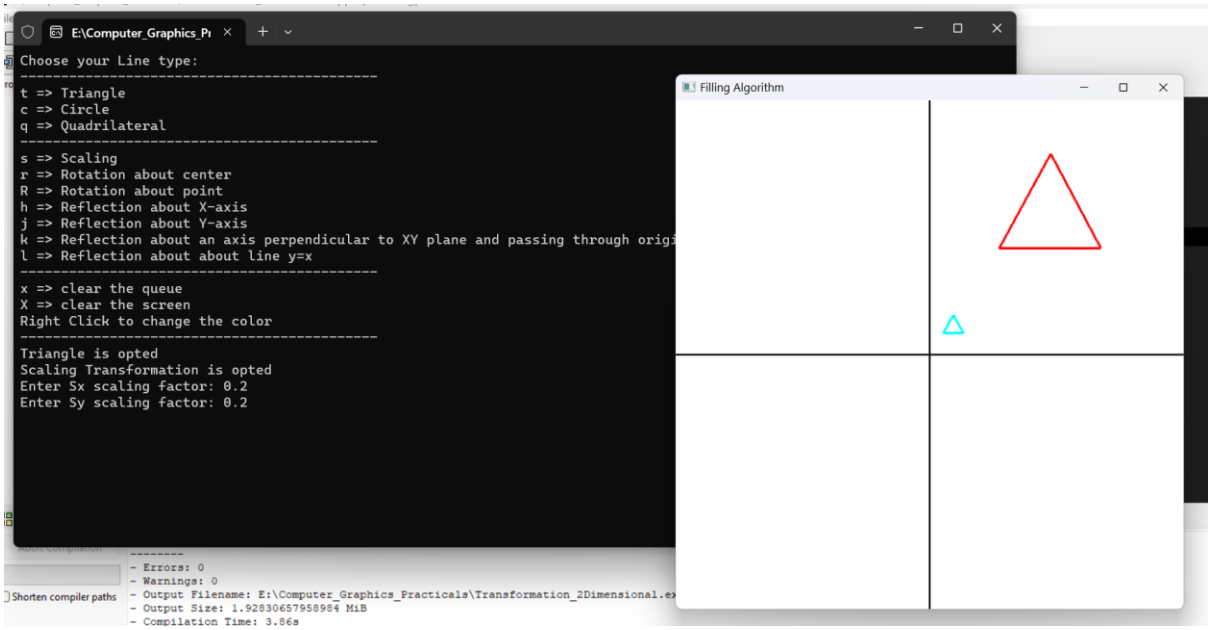
```
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(800, 100);
    glutCreateWindow("Filling Algorithm");
     initialize();
    cout << "Choose your Line type: " << endl;
    cout << "-----------------------------------------" << endl;
    cout << "t => Triangle" << endl;
    cout << "c => Circle" << endl;
    cout << "q => Quadrilateral" << endl;
    cout << "-----------------------------------------" << endl;
    cout << "s => Scaling" << endl;
    cout << "r => Rotation about center" << endl;
    cout << "R => Rotation about point" << endl;
    cout << "h => Reflection about X-axis" << endl;
    cout << "j => Reflection about Y-axis" << endl;
    cout << "k => Reflection about an axis perpendicular to XY plane
and passing through origin" << endl;
    cout << "l => Reflection about about line y=x" << endl;
     cout << "-----------------------------------------" <<
endl;
     cout << "x => clear the queue" << endl;
     cout << "X => clear the screen" << endl;
     cout << "Right Click to change the color" << endl;
    cout << "-----------------------------------------" << endl;
    glutDisplayFunc(initialaxis);
    glutMainLoop();
    return 0;
```

```
}
```

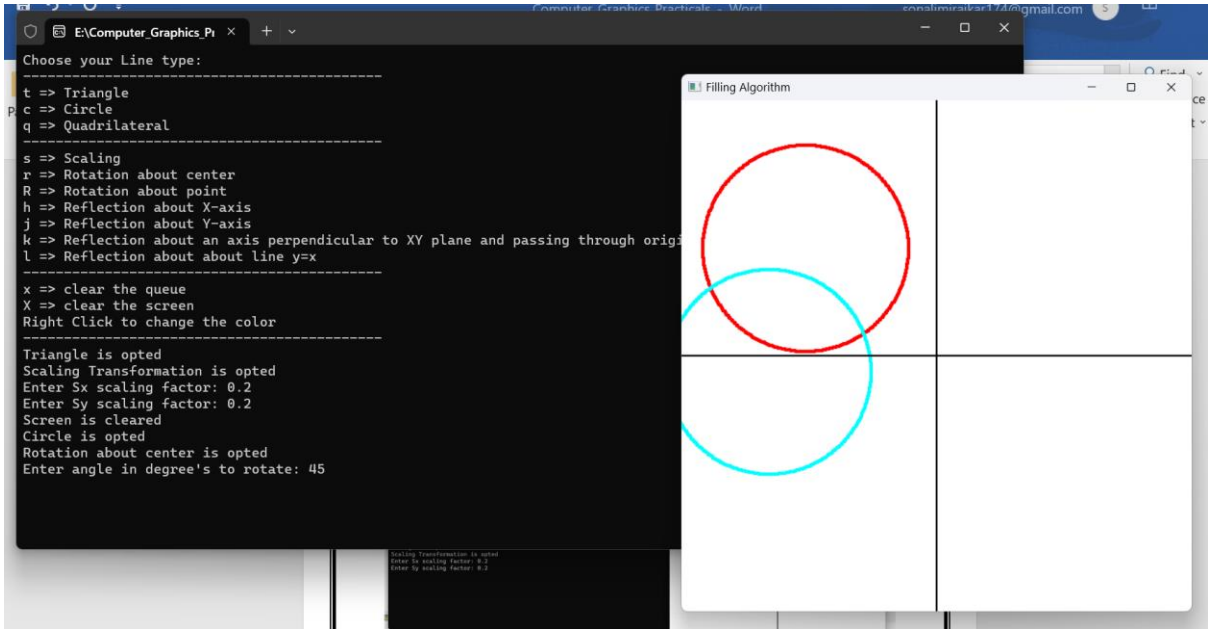
Output –



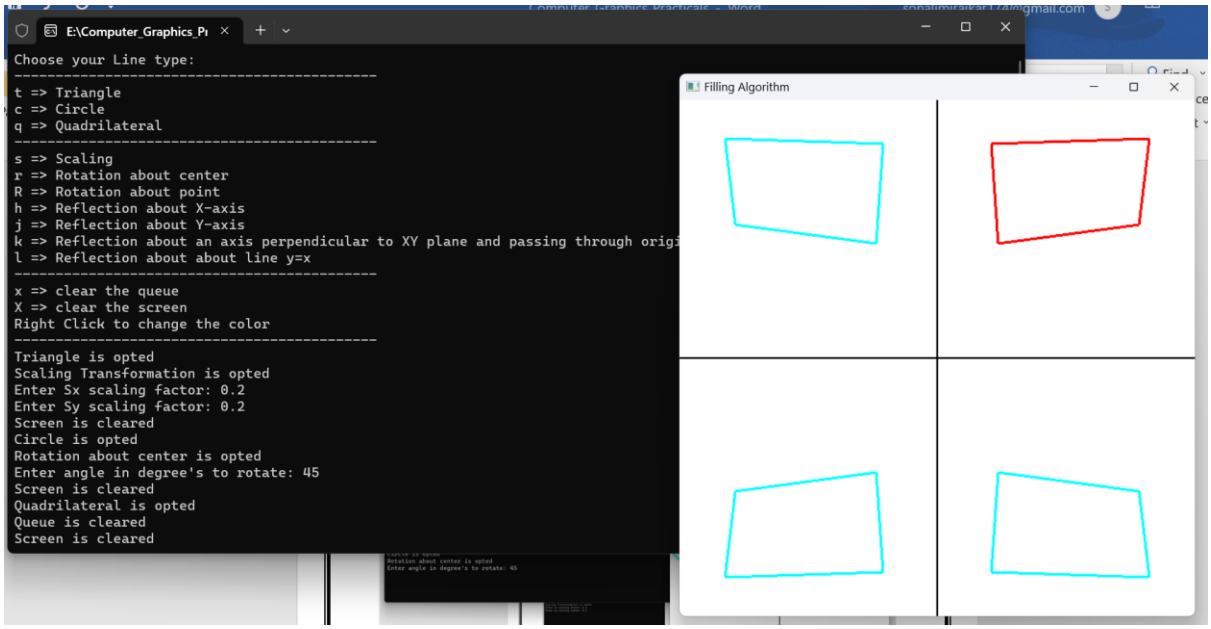
Scaling –



Rotation –



Reflection About All the axis –



ASSIGNEMENT NO – 7

Generate fractal patterns using:

1) Bezier
2) Koch Curve

Program –

Bezier Curve –

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

using namespace std;

int x[4], y[4];

void text() {
    glColor3f(0.0f, 0.0f, 0.0f);
    glRasterPos2i(10, 450);

    string text_value = "Bezier Curve in Computer Graphics";

    for (string::size_type i = 0; i < text_value.length(); ++i) {
        char c = text_value[i];
        glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, c);
    }
}

int bezier_curve() {
```

```
    cout << "Enter the coordinates of X and Y of the 4 coordinates:
";


    int windowHeight = glutGet(GLUT_WINDOW_HEIGHT); // Get the
height of the window


    for (int i = 0; i < 4; i++) {
        cin >> x[i] >> y[i];
        y[i] = windowHeight - y[i]; // Invert the y-coordinate
    }



    glBegin(GL_LINE_STRIP);


    for (float t = 0.0; t <= 1.0; t += 0.01) {
        float xt = pow(1 - t, 3) * x[0] + 3 * t * pow(1 - t, 2) *
x[1] + 3 * pow(t, 2) * (1 - t) * x[2] + pow(t, 3) * x[3];
        float yt = pow(1 - t, 3) * y[0] + 3 * t * pow(1 - t, 2) *
y[1] + 3 * pow(t, 2) * (1 - t) * y[2] + pow(t, 3) * y[3];


        glVertex2f(xt, yt);
    }


    glEnd();


    return 0;
}
void myDisplay() {
    glClear(GL_COLOR_BUFFER_BIT);


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

```
    bezier_curve();
     text();
    glFlush();
}

int myInit() {
    glClearColor(1.0, 1.0, 1.0, 0.0); // Set clear color to white
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 640.0, 0.0, 480.0);
    return 0;
}
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Bezier Curve");
    glutDisplayFunc(myDisplay);

    myInit();

    glutMainLoop();
    return 0;
}
```

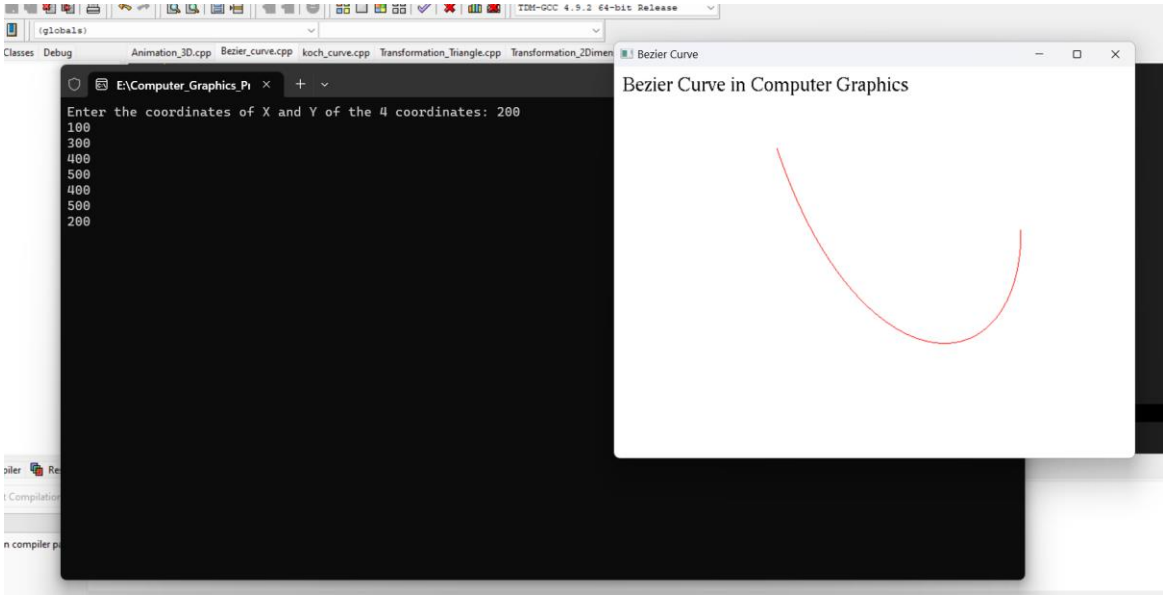
Koch Curve –

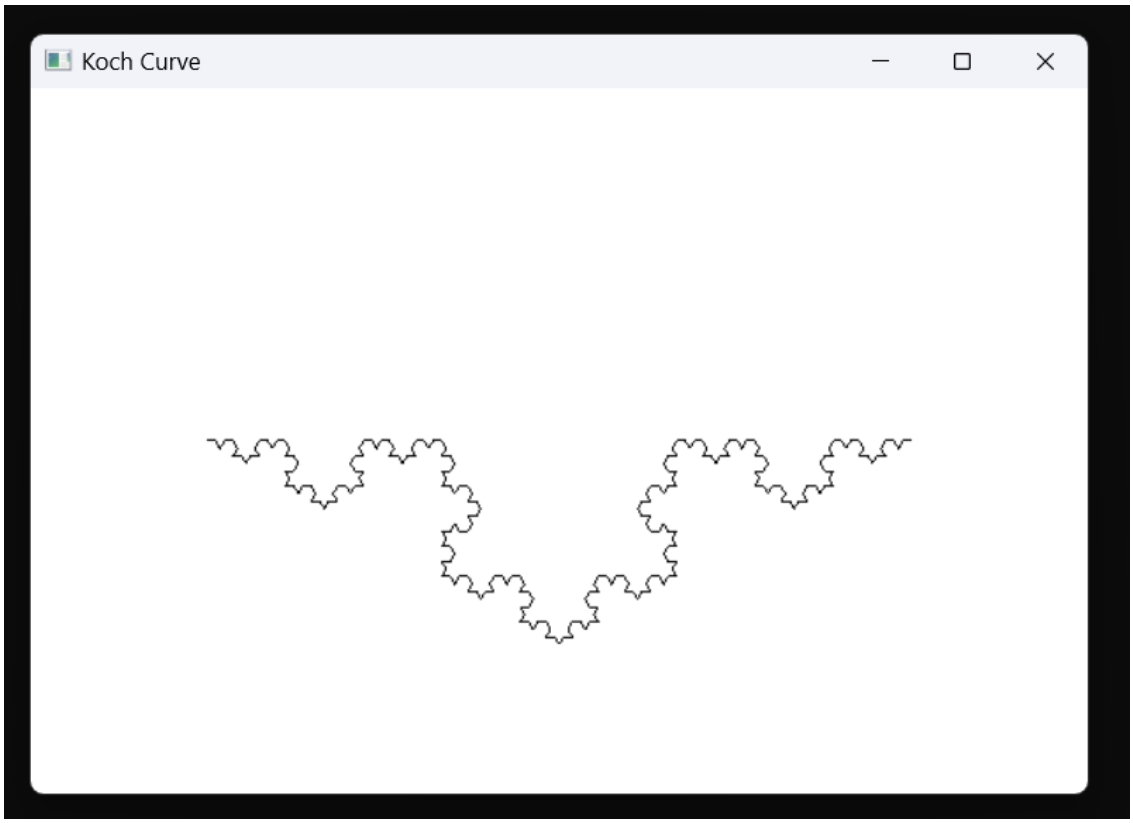
```
#include <GL/glut.h>
#include <cmath>
#include <iostream>

#define M_PI 3.14159265358979323846

using namespace std;

// Function to draw a line segment between two points
void drawLine(float x1, float y1, float x2, float y2) {
    glBegin(GL_LINES);
    glVertex2f(x1, y1);
    glVertex2f(x2, y2);
    glEnd();
}

// Function to recursively draw the Koch curve
void drawKochCurve(float x1, float y1, float x2, float y2, int
level) {
    if (level == 0) {
        drawLine(x1, y1, x2, y2);
    } else {
        float deltaX = x2 - x1;
        float deltaY = y2 - y1;

        // Calculate the coordinates of the four points of the Koch
curve
        float xA = x1 + deltaX / 3.0;
        float yA = y1 + deltaY / 3.0;
```

```
        float xC = x1 + 2.0 * deltaX / 3.0;
        float yC = y1 + 2.0 * deltaY / 3.0;

        float xB = xA + (xC - xA) * cos(M_PI / 3.0) + (yC - yA) *
sin(M_PI / 3.0);
        float yB = yA - (xC - xA) * sin(M_PI / 3.0) + (yC - yA) *
cos(M_PI / 3.0);

        // Recursively draw the four segments of the Koch curve
        drawKochCurve(x1, y1, xA, yA, level - 1);
        drawKochCurve(xA, yA, xB, yB, level - 1);
        drawKochCurve(xB, yB, xC, yC, level - 1);
        drawKochCurve(xC, yC, x2, y2, level - 1);
    }
}

void myDisplay() {
    glClear(GL_COLOR_BUFFER_BIT);

    // Set the color to black
    glColor3f(0.0, 0.0, 0.0);

    // Set the initial points of the Koch curve
    float x1 = 100.0;
    float y1 = 200.0;
    float x2 = 500.0;
    float y2 = 200.0;

    // Set the level of recursion for the Koch curve
    int level = 4;
```

```
    // Draw the Koch curve
    drawKochCurve(x1, y1, x2, y2, level);

    glFlush();
}

void myInit() {
    glClearColor(1.0, 1.0, 1.0, 0.0);
    glColor3f(0.0, 0.0, 0.0);
    glPointSize(2.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 600.0, 0.0, 400.0);
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(600, 400);
    glutCreateWindow("Koch Curve");
    glutDisplayFunc(myDisplay);
    myInit();
    glutMainLoop();
    return 0;
}
```

Output –


Bezier_curve.cpp
koch_curve.cpp
Transformation_Triangle.cpp
E:\Computer_Graphics_Pr
Enter the coordinates of X and Y of the 4 coordinates: 200
100
300
400
500
400
500
200
Bezier Curve
Bezier Curve in Computer Graphics


Koch Curve

ASSIGNEMENT NO – 8

Implement animation principles for any object.

Program –

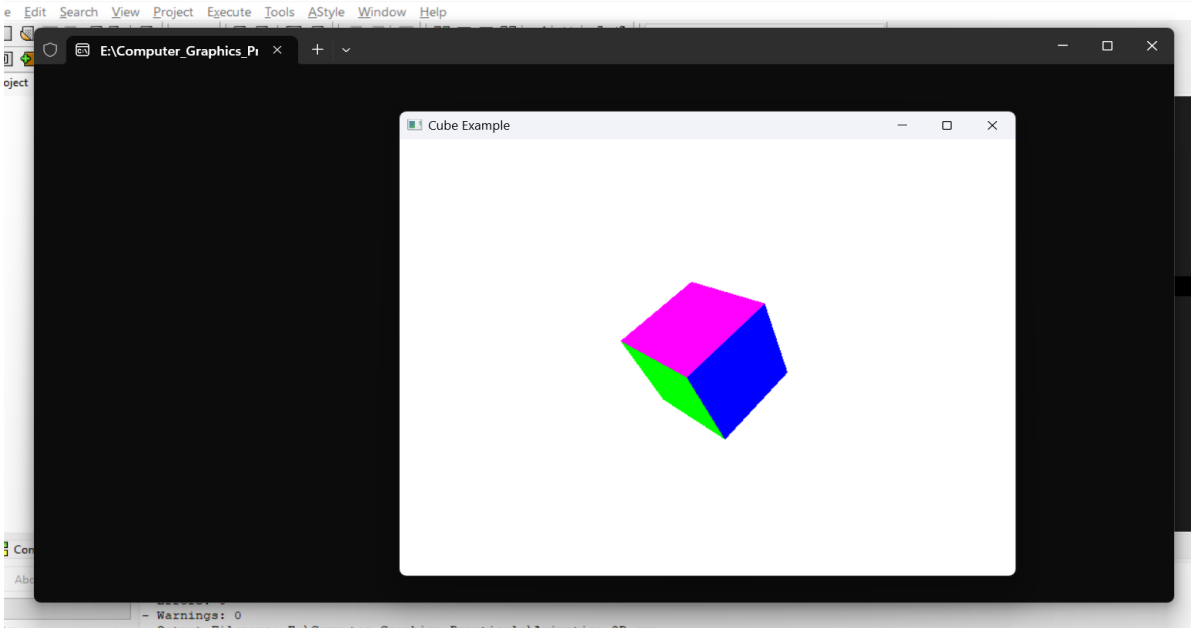
```
#include <GL/glut.h>
#include <iostream>
#include <math.h>

using namespace std;

float angle = 0.07;

void drawCube() {
    glBegin(GL_QUADS);

    // Front face
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex3f(-2.0f, -2.0f, 2.0f);
    glVertex3f(2.0f, -2.0f, 2.0f);
    glVertex3f(2.0f, 2.0f, 2.0f);
    glVertex3f(-2.0f, 2.0f, 2.0f);

    // Back face
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f(-2.0f, -2.0f, -2.0f);
    glVertex3f(-2.0f, 2.0f, -2.0f);
    glVertex3f(2.0f, 2.0f, -2.0f);
    glVertex3f(2.0f, -2.0f, -2.0f);
```

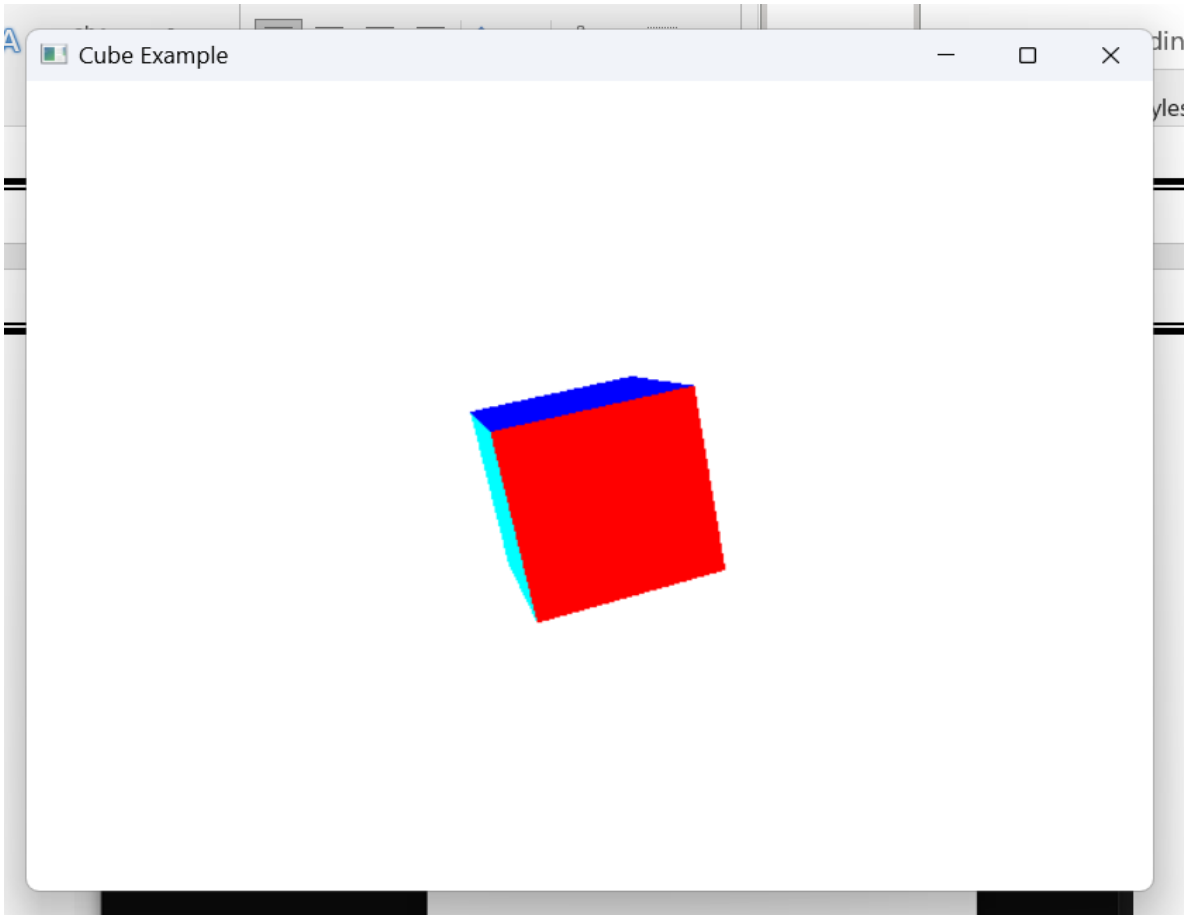
```
// Top face
glColor3f(0.0f, 0.0f, 1.0f);
glVertex3f(-2.0f, 2.0f, -2.0f);
glVertex3f(-2.0f, 2.0f, 2.0f);
glVertex3f(2.0f, 2.0f, 2.0f);
glVertex3f(2.0f, 2.0f, -2.0f);

// Bottom face
glColor3f(1.0f, 1.0f, 0.0f);
glVertex3f(-2.0f, -2.0f, -2.0f);
glVertex3f(2.0f, -2.0f, -2.0f);
glVertex3f(2.0f, -2.0f, 2.0f);
glVertex3f(-2.0f, -2.0f, 2.0f);

// Right face
glColor3f(1.0f, 0.0f, 1.0f);
glVertex3f(2.0f, -2.0f, -2.0f);
glVertex3f(2.0f, 2.0f, -2.0f);
glVertex3f(2.0f, 2.0f, 2.0f);
glVertex3f(2.0f, -2.0f, 2.0f);

// Left face
glColor3f(0.0f, 1.0f, 1.0f);
glVertex3f(-2.0f, -2.0f, -2.0f);
glVertex3f(-2.0f, -2.0f, 2.0f);
glVertex3f(-2.0f, 2.0f, 2.0f);
glVertex3f(-2.0f, 2.0f, -2.0f);

glEnd();
```

```
}

void myDisplay() {
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0, 0.0, 8.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

     glTranslatef(0.0, 0.0, -8.0);
    glRotatef(angle, 1.0, 1.0, 1.0); // Rotate the cube
    drawCube();

    glutSwapBuffers();
}

void reshape(int width, int height) {
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0f, static_cast<float>(width) / height, 1.0f,
100.0f);
}

void myInit() {
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glEnable(GL_DEPTH_TEST);
}
```

```
void timer(int) {
    glutPostRedisplay();
    glutTimerFunc(1000 / 60, timer, 0);

    angle += 0.8;

    if(angle > 360.0 ){
     angle = angle - 360.0;
     }
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(640, 480);
    glutCreateWindow("Cube Example");
    glutDisplayFunc(myDisplay);
    glutReshapeFunc(reshape);
    glutTimerFunc(0, timer, 0);
    myInit();
    glutMainLoop();
    return 0;
}
```

Output –


Edit Search View Project Execute Tools AStyle Window Help
E:\Computer_Graphics_Pr
Cube Example
Warnings: 0


Cube Example
