**Department of Computing**

**CS-361: Computer Graphics  
  
Class: BSCS-12ABC & SE12AB**

**Lab 07: Clipping (Line Clipping and Polygon Clipping)**

# CLO-02: Develop 2D and 3D graphical applications using programming libraries and tools.

# CLO-03: Implement algorithms for rendering, transformations, and animations.

**Date: 11th Mar 2025**

**Time: 12:00 PM – 14:30 PM**

# Instructor: Dr. Sidra Sutana

# Lab Engineer: Mr. Aftab Farooq

**Name:** Ahmed Mohiuddin Shah

**CMS ID:** 415216

**Section:** BSCS-12-A

**Lab:** 7

**Lab 07: Clipping (Line Clipping and Polygon Clipping)**

### **Introduction:** Clipping is a fundamental concept in computer graphics that involves determining which parts of geometric primitives (lines, polygons) should be displayed within a defined viewing window and which parts should be discarded. This lab focuses on implementing line and polygon clipping algorithms.

### **Lab Objective:**

The objective of this lab is to understand the concept of clipping in computer graphics, including line clipping and polygon clipping techniques. Students will implement various clipping algorithms and visualize the results.

## Tools/Software Requirement:

* **Operating System:**
  + Windows / macOS / Linux (Ubuntu recommended)
* **Development Environment:**
  + **Windows:** [Code::Blocks](http://www.codeblocks.org/) or [Visual Studio](https://visualstudio.microsoft.com/)
  + **macOS:** [Xcode](https://developer.apple.com/xcode/)
  + **Linux:** GCC and g++ compilers
* **Graphics Libraries:**
  + **OpenGL** (built-in on macOS and Linux, available in Windows IDEs)
  + **GLUT** (OpenGL Utility Toolkit)
  + **GLEW** (OpenGL Extension Wrangler Library)
* **Package Manager (for macOS/Linux):**
  + **Homebrew** (macOS): brew install freeglut glew
  + **APT** (Linux): sudo apt-get install freeglut3-dev glew-utils
* **Compilers:**
  + **Windows:** MinGW (for Code::Blocks) or Microsoft C++ Compiler (for Visual Studio)
  + **macOS/Linux:** GCC/G++
* A programming environment (e.g., Visual Studio, PyCharm, or any IDE of your choice).
* A graphics library (optional, e.g., OpenGL, SDL, or a simple image library for saving images).
* Basic knowledge of clipping.

## Prerequisites :

 Basic Basic understanding of **clipping concepts** in computer graphics.

 Familiarity with programming (preferably in Python, C++, or Java).  
  Knowledge of **geometric algorithms** (line equations, polygon representation, and intersections).  
  Knowledge Familiarity with **debugging and visualization tools** to test and display clipped outputs.

**Lab Tasks :**

**Lab Task-01 : Implement Line Clipping (Cohen-Sutherland Algorithm)**

1. Write a program to implement the Cohen-Sutherland algorithm.
2. The program should:
   * Accept user-defined coordinates for the viewport (clipping window).
   * Accept multiple line segments with their coordinates.
   * Display the original and clipped lines in different colors.

**Deliverable:** Submit the source code, a report explaining the logic, and screenshots showing sample outputs.

**Solution:**

**Code:**

#include <GL/glut.h>

#include <math.h>

#include "helpers.h"

Params params;

*int* screenWidth = 800;

*int* screenHeight = 800;

*bool* toggle = false;

*void* display()

{

    glClear(GL\_COLOR\_BUFFER\_BIT);

    drawViewPort(&params.viewPort, screenWidth, screenHeight);

    for (*auto* line : params.lines)

    {

        if (toggle) {

            glColor3f((rand() % 255) / 255.0, (rand() % 255) / 255.0, (rand() % 255) / 255.0);

            glBegin(GL\_LINES);

            glVertex2i(line.x1, line.y1);

            glVertex2i(line.x2, line.y2);

            glEnd();

            continue;

        }

        drawCohenSutherlandLineClip(line, params.viewPort);

    }

    glFlush();

}

*void* keyboard(*unsigned* *char* *key*, *int* *x*, *int* *y*)

{

    if (*key* == 27)

        exit(0);

    // w key

    if (*key* == 119)

    {

        toggle = !toggle;

    }

    // rerender

    glutPostRedisplay();

}

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

{

    params = getParams(true, false);

    glutInit(&*argc*, *argv*);

    glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

    glutInitWindowSize(screenWidth, screenHeight);

    glutCreateWindow("Task1 - Cohen Sutherland Line Clipping");

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    gluOrtho2D(0.0, 800.0, 0.0, 800.0);

    glutDisplayFunc(display);

    glutKeyboardFunc(keyboard);

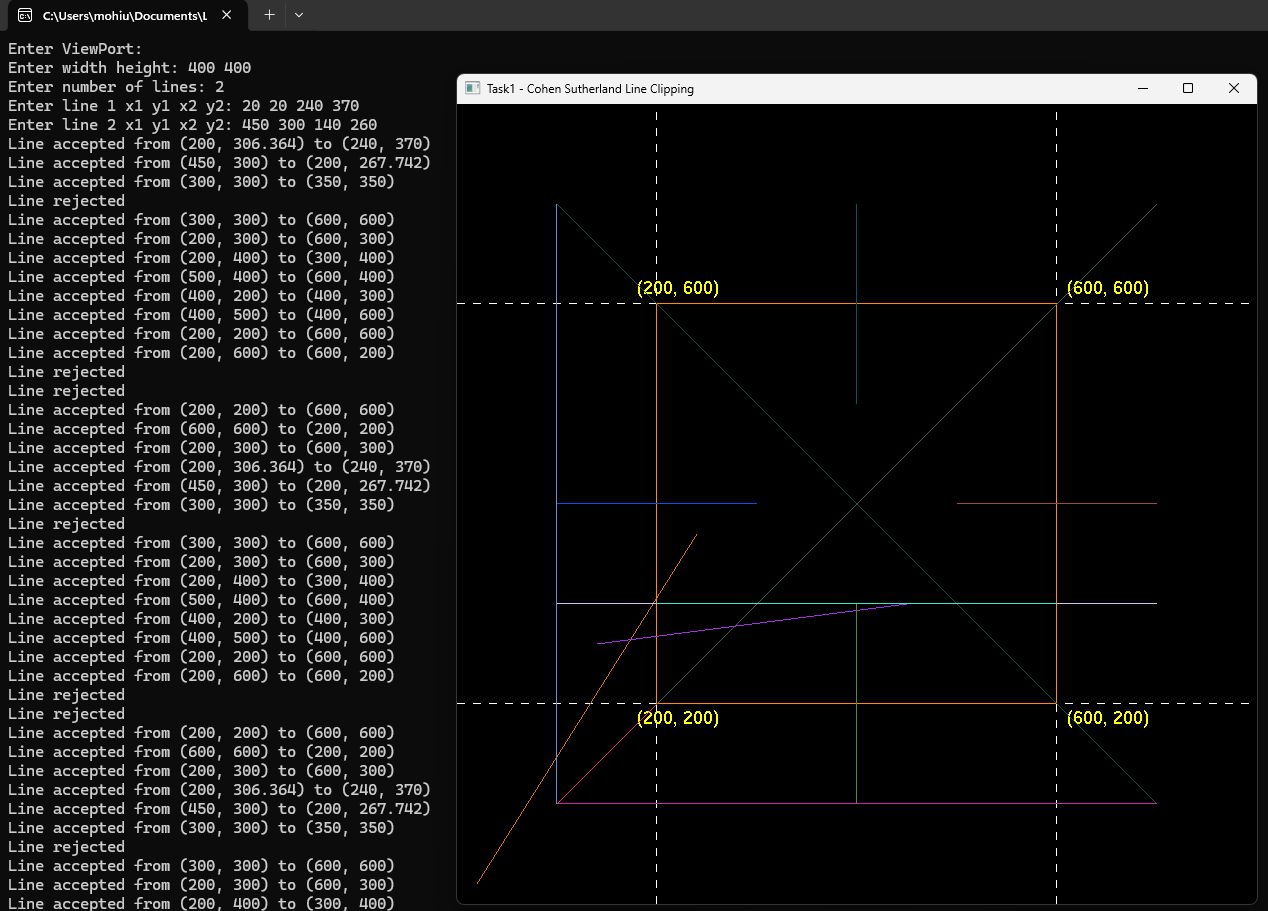
    glutMainLoop();

    return 0;

}

**Output:**

**Use ‘w’ to switch between clipped and non clipped lines.**

****

**Red Line is Trivially Rejected lines and the green lines are trivially accepted and clipped lines.A screen shot of a graph

AI-generated content may be incorrect.**

**Lab Task-02: Implement Polygon Clipping (Sutherland-Hodgman Algorithm)**

1. Write a program to implement the Sutherland-Hodgman algorithm.
2. The program should:
   * Accept user-defined viewport (clipping window) coordinates.
   * Accept polygon vertices from the user.
   * Display the original and clipped polygons in different colors.

**Deliverable:** Submit the source code, a report explaining the logic, and screenshots showing sample outputs.

**Solution:**

**Code:**

#include <GL/glut.h>

#include <math.h>

#include "helpers.h"

Params params;

*int* screenWidth = 800;

*int* screenHeight = 800;

*bool* toggle = false;

*void* display()

{

    glClear(GL\_COLOR\_BUFFER\_BIT);

    drawViewPort(&params.viewPort, screenWidth, screenHeight);

    for (*auto* polygon : params.polygons)

    {

        glColor3f(1.0, 0.0, 0.0);

        if (toggle) {

            for (*int* i = 0; i < polygon.points.size(); i++) {

                glBegin(GL\_LINES);

                glVertex2i(polygon.points[i].x, polygon.points[i].y);

                glVertex2i(polygon.points[(i + 1) % polygon.points.size()].x, polygon.points[(i + 1) % polygon.points.size()].y);

                glEnd();

                glRasterPos2i(polygon.points[i].x + 10, polygon.points[i].y - 20);

                glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, 'A' + i);

            }

            continue;

        }

        drawSutherHodgmanPolygonClip(&polygon, params.viewPort);

    }

    glFlush();

}

*void* keyboard(*unsigned* *char* *key*, *int* *x*, *int* *y*)

{

    if (*key* == 27)

        exit(0);

    // q key

    if (*key* == 113)

    {

    }

    // w key

    if (*key* == 119)

    {

        toggle = !toggle;

    }

    // rerender

    glutPostRedisplay();

}

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

{

    params = getParams(false, true);

    glutInit(&*argc*, *argv*);

    glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

    glutInitWindowSize(screenWidth, screenHeight);

    glutCreateWindow("Task2 - Sutherland Hodgman Polygon Clipping");

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    gluOrtho2D(0.0, 800.0, 0.0, 800.0);

    glutDisplayFunc(display);

    glutKeyboardFunc(keyboard);

    glutMainLoop();

    return 0;

}

**Output:**

**Use ‘w’ to switch between clipped and unclipped polygons**

**A screen shot of a computer

AI-generated content may be incorrect.**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Lab Task-03: Combines Cohen-Sutherland Line Clipping and Sutherland-Hodgman Polygon Clipping.**

1. Develop a graphical application that:
   * Combines Cohen-Sutherland Line Clipping and Sutherland-Hodgman Polygon Clipping.
2. Accepts multiple lines and polygons from the user.
3. Allows dynamic switching between clipping algorithms.
4. Provides visual representation of both original and clipped shapes.
5. Offers an interactive interface for drawing, resizing, and repositioning objects within the viewport.  
     
   **Deliverable:** Submit the source code, a report explaining the logic, and screenshots showing sample outputs.

**Solution:**

**Use ‘l’ to switch between line and polygon drawing algorithms**

**Use ‘w’ to switch between clipped and unclipped views**

**Use ‘q’ to clear the screen**

**Use left mouse button to draw lines or polygons.**

**Code:**

#include <GL/glut.h>

#include <math.h>

#include <vector>

#include <iostream>

#include "helpers.h"

// Global variables

Params params;

*int* screenWidth = 800;

*int* screenHeight = 800;

*bool* toggle = false;

*bool* isDrawing = false;

*bool* lineDrawingMode = false; // Tracks if we are in line-drawing mode

Point tempPoint;

Line tempLine;

PolygonShape tempPolygon;

// Function prototypes

*void* display();

*void* keyboard(*unsigned* *char* *key*, *int* *x*, *int* *y*);

*void* mouse(*int* *button*, *int* *state*, *int* *x*, *int* *y*);

*void* motion(*int* *x*, *int* *y*);

*void* drawViewPort(ViewPort \**viewPort*, *int* *screenWidth*, *int* *screenHeight*);

*void* drawCohenSutherlandLineClip(Line *line*, ViewPort *viewPort*);

*void* drawSutherHodgmanPolygonClip(PolygonShape \**polygon*, ViewPort *viewPort*);

// Display function

*void* display()

{

    glClear(GL\_COLOR\_BUFFER\_BIT);

    drawViewPort(&params.viewPort, screenWidth, screenHeight);

    // Draw lines

    for (*auto* line : params.lines)

    {

        glColor3f(1.0, 0.0, 0.0);

        if (toggle)

        {

            // Draw original line

            glBegin(GL\_LINES);

            glVertex2i(line.x1, line.y1);

            glVertex2i(line.x2, line.y2);

            glEnd();

            // Label endpoints

            glRasterPos2i(line.x1 + 10, line.y1 - 20);

            glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, 'A');

            glRasterPos2i(line.x2 + 10, line.y2 - 20);

            glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, 'B');

        }

        else

        {

            // Draw clipped line

            drawCohenSutherlandLineClip(line, params.viewPort);

        }

    }

    // Draw polygons

    for (*auto* polygon : params.polygons)

    {

        glColor3f(1.0, 0.0, 0.0);

        if (toggle)

        {

            // Draw original polygon

            for (*int* i = 0; i < polygon.points.size(); i++)

            {

                glBegin(GL\_LINES);

                glVertex2i(polygon.points[i].x, polygon.points[i].y);

                glVertex2i(polygon.points[(i + 1) % polygon.points.size()].x, polygon.points[(i + 1) % polygon.points.size()].y);

                glEnd();

                // Label vertices

                glRasterPos2i(polygon.points[i].x + 10, polygon.points[i].y - 20);

                glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, 'A' + i);

            }

        }

        else

        {

            // Draw clipped polygon

            drawSutherHodgmanPolygonClip(&polygon, params.viewPort);

        }

    }

    glFlush();

}

// Keyboard function

*void* keyboard(*unsigned* *char* *key*, *int* *x*, *int* *y*)

{

    if (*key* == 27) // ESC key

        exit(0);

    if (*key* == 113)

    {                            // 'q' key{

        params.polygons.clear(); // Clear polygons

        params.lines.clear();

    }                                       // Clear lines}

    if (*key* == 119)                         // 'w' key

        toggle = !toggle;                   // Toggle between original and clipped shapes

    if (*key* == 108)                         // 'l' key

        lineDrawingMode = !lineDrawingMode; // Toggle line-drawing mode

    glutPostRedisplay(); // Rerender

}

// Mouse function

*void* mouse(*int* *button*, *int* *state*, *int* *x*, *int* *y*)

{

*y* = screenHeight - *y*; // Convert y-coordinate to OpenGL coordinate system

    if (*button* == GLUT\_LEFT\_BUTTON && *state* == GLUT\_DOWN)

    {

        if (lineDrawingMode)

        {

            // Line-drawing mode: First click sets start point, second click sets end point

            if (!isDrawing)

            {

                tempLine = {*x*, *y*, *x*, *y*}; // Initialize line

                isDrawing = true;

            }

            else

            {

                tempLine.x2 = *x*;

                tempLine.y2 = *y*;

                params.lines.push\_back(tempLine); // Add line to the list

                isDrawing = false;

            }

        }

        else

        {

            // Polygon-drawing mode

            isDrawing = true;

            tempPoint = {*x*, *y*};

            tempPolygon.points.push\_back({*x*, *y*});

        }

    }

    if (*button* == GLUT\_LEFT\_BUTTON && *state* == GLUT\_UP && !lineDrawingMode)

    {

        isDrawing = false;

        if (tempPolygon.points.size() > 2)

        {

            params.polygons.push\_back(tempPolygon); // Add polygon to the list

        }

    }

}

// Motion function (for drawing)

*void* motion(*int* *x*, *int* *y*)

{

*y* = screenHeight - *y*; // Convert y-coordinate to OpenGL coordinate system

    if (isDrawing && !lineDrawingMode)

    {

        tempPolygon.points.push\_back({*x*, *y*});

        glutPostRedisplay();

    }

}

// Main function

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

{

    params = getParams(true, true);

    glutInit(&*argc*, *argv*);

    glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

    glutInitWindowSize(screenWidth, screenHeight);

    glutCreateWindow("Task3 - Combined Clipping");

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    gluOrtho2D(0.0, 800.0, 0.0, 800.0);

    glutDisplayFunc(display);

    glutKeyboardFunc(keyboard);

    glutMouseFunc(mouse);

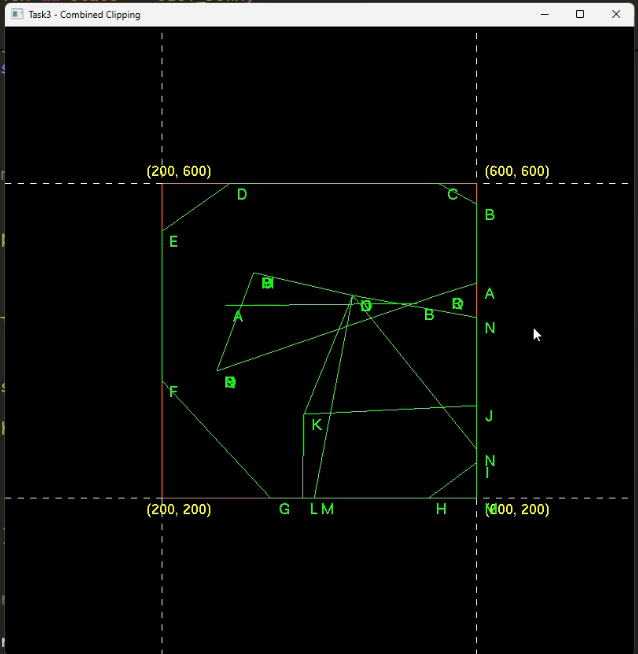
    glutMotionFunc(motion);

    glutMainLoop();

    return 0;

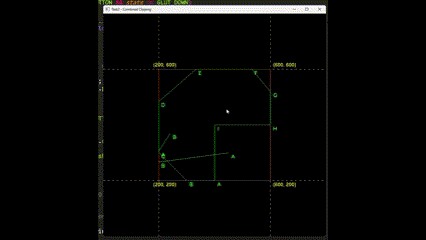
}

**Output:**



A screenshot of a computer

AI-generated content may be incorrect.



“helpers.h”

#pragma once

#include <GL/glut.h>

#include <iostream>

#include <vector>

#include <string>

#include <utility>

*struct* Point

{

*int* x, y;

};

*struct* Line

{

*int* x1, y1, x2, y2;

};

*struct* PolygonShape

{

    std::vector<Point> points;

};

*struct* Vertex

{

    Point A;

    Point B;

};

*struct* ViewPort

{

*int* x, y, width, height;

};

*struct* Params

{

    ViewPort viewPort;

    std::vector<Line> lines;

    std::vector<PolygonShape> polygons;

};

*void* drawViewPort(ViewPort \**viewPort*, *int* *screenWidth*, *int* *screenHeight*)

{

    // drawing viewport at the center of the screen

*int* x = (screenWidth - viewPort->width) / 2;

*int* y = (screenHeight - viewPort->height) / 2;

*int* width = viewPort->width;

*int* height = viewPort->height;

    viewPort->x = x;

    viewPort->y = y;

    // extend dotted lines to show the TOP\_VIEWPORT, BOTTOM\_VIEWPORT, left\_VIEWPORT and right\_VIEWPORT of the viewport

    glColor3f(1.0, 1.0, 1.0);

    glLineStipple(1, 0x00FF);

    glEnable(GL\_LINE\_STIPPLE);

    glBegin(GL\_LINES);

    glVertex2i(0, y);

    glVertex2i(screenWidth, y);

    glEnd();

    glBegin(GL\_LINES);

    glVertex2i(0, y + height);

    glVertex2i(screenWidth, y + height);

    glEnd();

    glBegin(GL\_LINES);

    glVertex2i(x, 0);

    glVertex2i(x, screenHeight);

    glEnd();

    glBegin(GL\_LINES);

    glVertex2i(x + width, 0);

    glVertex2i(x + width, screenHeight);

    glEnd();

    glDisable(GL\_LINE\_STIPPLE);

    glColor3f(1.0, 0.55, 0.0);

    glBegin(GL\_LINE\_LOOP);

    glVertex2i(x, y);

    glVertex2i(x + width, y);

    glVertex2i(x + width, y + height);

    glVertex2i(x, y + height);

    glEnd();

    // write the viewport coordinates at the corners of the viewport

    glColor3f(1.0, 1.0, 0.0);

    glRasterPos2i(x - 20, y - 20);

    std::string s\_x\_min = std::to\_string(viewPort->x);

    std::string s\_x\_max = std::to\_string(viewPort->x + viewPort->width);

    std::string s\_y\_min = std::to\_string(viewPort->y);

    std::string s\_y\_max = std::to\_string(viewPort->y + viewPort->height);

    std::string s = "(" + s\_x\_min + ", " + s\_y\_min + ")";

    for (*int* i = 0; i < s.length(); i++)

    {

        glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, s[i]);

    }

    glRasterPos2i(x + width + 10, y - 20);

    s = "(" + s\_x\_max + ", " + s\_y\_min + ")";

    for (*int* i = 0; i < s.length(); i++)

    {

        glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, s[i]);

    }

    glRasterPos2i(x - 20, y + height + 10);

    s = "(" + s\_x\_min + ", " + s\_y\_max + ")";

    for (*int* i = 0; i < s.length(); i++)

    {

        glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, s[i]);

    }

    glRasterPos2i(x + width + 10, y + height + 10);

    s = "(" + s\_x\_max + ", " + s\_y\_max + ")";

    for (*int* i = 0; i < s.length(); i++)

    {

        glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, s[i]);

    }

}

*class* CohenSutherLandAlgo

{

*private:*

*double* x\_max, y\_max, x\_min, y\_min;

    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

*public:*

*double* x1, y1, x2, y2;

    CohenSutherLandAlgo()

    {

        x1 = 0.0;

        x2 = 0.0;

        y1 = 0.0;

        y2 = 0.0;

    }

*void* getCoordinates(

*double* *x1*, *double* *y1*, *double* *x2*, *double* *y2*);

*void* getClippingRectangle(

*int* *x\_min*, *int* *y\_min*, *int* *x\_max*, *int* *y\_max*);

*int* generateCode(*double* *x*, *double* *y*);

*void* cohenSutherland();

};

*void* CohenSutherLandAlgo::getCoordinates(

*double* *x1*, *double* *y1*, *double* *x2*, *double* *y2*)

{

    this->x1 = x1;

    this->y1 = y1;

    this->x2 = x2;

    this->y2 = y2;

}

*void* CohenSutherLandAlgo::getClippingRectangle(*int* *x\_min*, *int* *y\_min*, *int* *x\_max*, *int* *y\_max*)

{

    this->x\_min = x\_min;

    this->y\_min = y\_min;

    this->x\_max = x\_max;

    this->y\_max = y\_max;

}

*int* CohenSutherLandAlgo::generateCode(*double* *x*, *double* *y*)

{

*int* code = INSIDE; // intially initializing as being inside

    if (x < x\_min)     // lies to the left of rectangle

        code |= LEFT;

    else if (x > x\_max) // lies to the right of rectangle

        code |= RIGHT;

    if (y < y\_min) // lies below the rectangle

        code |= BOTTOM;

    else if (y > y\_max) // lies above the rectangle

        code |= TOP;

    return code;

}

*void* CohenSutherLandAlgo::cohenSutherland()

{

*int* code1 = generateCode(x1, y1); // Compute region codes for P1.

*int* code2 = generateCode(x2, y2); // Compute region codes for P2.

*bool* accept = false;              // Initialize line as outside the rectangular window.

    while (true)

    {

        if ((code1 == 0) && (code2 == 0))

        {

            // If both endpoints lie within rectangle.

            accept = true;

            break;

        }

        else if (code1 & code2)

        {

            // If both endpoints are outside rectangle,in same region.

            break;

        }

        else

        {

            // Some segment of line lies within the rectangle.

*int* code\_out;

*double* x, y;

            // At least one endpoint lies outside the rectangle, pick it.

            if (code1 != 0)

                code\_out = code1;

            else

                code\_out = code2;

            /\*

             \* Find intersection point by using formulae :

             y = y1 + slope \* (x - x1)

             x = x1 + (1 / slope) \* (y - y1)

             \*/

            if (code\_out & TOP)

            {

                // point is above the clip rectangle

                x = x1 + (x2 - x1) \* (y\_max - y1) / (y2 - y1);

                y = y\_max;

            }

            else if (code\_out & BOTTOM)

            {

                // point is below the rectangle

                x = x1 + (x2 - x1) \* (y\_min - y1) / (y2 - y1);

                y = y\_min;

            }

            else if (code\_out & RIGHT)

            {

                // point is to the right of rectangle

                y = y1 + (y2 - y1) \* (x\_max - x1) / (x2 - x1);

                x = x\_max;

            }

            else if (code\_out & LEFT)

            {

                // point is to the left of rectangle

                y = y1 + (y2 - y1) \* (x\_min - x1) / (x2 - x1);

                x = x\_min;

            }

            // Intersection point x,y is found.

            // Replace point outside rectangle by intersection point.

            if (code\_out == code1)

            {

                x1 = x;

                y1 = y;

                code1 = generateCode(x1, y1);

            }

            else

            {

                x2 = x;

                y2 = y;

                code2 = generateCode(x2, y2);

            }

        }

    }

    if (accept)

    {

        std::cout << "Line accepted from " << "(" << x1 << ", "

                  << y1 << ")" << " to " << "(" << x2 << ", " << y2 << ")" << std::endl;

        glColor3f(0.0, 1.0, 0.0);

        glBegin(GL\_LINES);

        glVertex2i(x1, y1);

        glVertex2i(x2, y2);

        glEnd();

        glRasterPos2i(x1 + 10, y1 - 20);

        glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, 'A');

        glRasterPos2i(x2 + 10, y2 - 20);

        glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, 'B');

    }

    else

    {

        std::cout << "Line rejected" << std::endl;

        glColor3f(1.0, 0.0, 0.0);

        glBegin(GL\_LINES);

        glVertex2i(x1, y1);

        glVertex2i(x2, y2);

        glEnd();

        glRasterPos2i(x1 + 10, y1 - 20);

        glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, 'A');

        glRasterPos2i(x2 + 10, y2 - 20);

        glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, 'B');

    }

}

*void* drawCohenSutherlandLineClip(Line *line*, ViewPort *viewPort*)

{

    CohenSutherLandAlgo c;

    c.getCoordinates(line.x1, line.y1, line.x2, line.y2);

    c.getClippingRectangle(viewPort.x, viewPort.y, viewPort.x + viewPort.width, viewPort.y + viewPort.height);

    c.cohenSutherland();

}

*void* drawSutherHodgmanPolygonClip(PolygonShape \**polygon*, ViewPort *viewPort*)

{

    // getting vertex list

    std::vector<Vertex> vertexList;

    for (*int* i = 0; i < polygon->points.size(); i++)

    {

        Vertex vertex;

        vertex.A = polygon->points[i];

        vertex.B = polygon->points[(i + 1) % polygon->points.size()];

        vertexList.push\_back(vertex);

    }

    // clipping against TOP\_VIEWPORT boundary

    std::vector<Vertex> outputList;

    for (*int* i = 0; i < vertexList.size(); i++)

    {

        Vertex vertex = vertexList[i];

        Vertex nextVertex = vertexList[(i + 1) % vertexList.size()];

        if (vertex.A.y >= viewPort.y && nextVertex.A.y >= viewPort.y)

        {

            outputList.push\_back(nextVertex);

        }

        else if (vertex.A.y >= viewPort.y && nextVertex.A.y < viewPort.y)

        {

            Vertex newVertex;

            newVertex.A.x = vertex.A.x + (nextVertex.A.x - vertex.A.x) \* (viewPort.y - vertex.A.y) / (nextVertex.A.y - vertex.A.y);

            newVertex.A.y = viewPort.y;

            newVertex.B.x = nextVertex.A.x;

            newVertex.B.y = nextVertex.A.y;

            outputList.push\_back(newVertex);

        }

        else if (vertex.A.y < viewPort.y && nextVertex.A.y >= viewPort.y)

        {

            Vertex newVertex;

            newVertex.A.x = vertex.A.x + (nextVertex.A.x - vertex.A.x) \* (viewPort.y - vertex.A.y) / (nextVertex.A.y - vertex.A.y);

            newVertex.A.y = viewPort.y;

            newVertex.B.x = nextVertex.A.x;

            newVertex.B.y = nextVertex.A.y;

            outputList.push\_back(newVertex);

            outputList.push\_back(nextVertex);

        }

    }

    // clipping against right\_VIEWPORT boundary

    vertexList = outputList;

    outputList.clear();

    for (*int* i = 0; i < vertexList.size(); i++)

    {

        Vertex vertex = vertexList[i];

        Vertex nextVertex = vertexList[(i + 1) % vertexList.size()];

        if (vertex.A.x <= viewPort.x + viewPort.width && nextVertex.A.x <= viewPort.x + viewPort.width)

        {

            outputList.push\_back(nextVertex);

        }

        else if (vertex.A.x <= viewPort.x + viewPort.width && nextVertex.A.x > viewPort.x + viewPort.width)

        {

            Vertex newVertex;

            newVertex.A.x = viewPort.x + viewPort.width;

            newVertex.A.y = vertex.A.y + (nextVertex.A.y - vertex.A.y) \* (viewPort.x + viewPort.width - vertex.A.x) / (nextVertex.A.x - vertex.A.x);

            newVertex.B.x = nextVertex.A.x;

            newVertex.B.y = nextVertex.A.y;

            outputList.push\_back(newVertex);

        }

        else if (vertex.A.x > viewPort.x + viewPort.width && nextVertex.A.x <= viewPort.x + viewPort.width)

        {

            Vertex newVertex;

            newVertex.A.x = viewPort.x + viewPort.width;

            newVertex.A.y = vertex.A.y + (nextVertex.A.y - vertex.A.y) \* (viewPort.x + viewPort.width - vertex.A.x) / (nextVertex.A.x - vertex.A.x);

            newVertex.B.x = nextVertex.A.x;

            newVertex.B.y = nextVertex.A.y;

            outputList.push\_back(newVertex);

            outputList.push\_back(nextVertex);

        }

    }

    // clipping against BOTTOM\_VIEWPORT boundary

    vertexList = outputList;

    outputList.clear();

    for (*int* i = 0; i < vertexList.size(); i++)

    {

        Vertex vertex = vertexList[i];

        Vertex nextVertex = vertexList[(i + 1) % vertexList.size()];

        if (vertex.A.y <= viewPort.y + viewPort.height && nextVertex.A.y <= viewPort.y + viewPort.height)

        {

            outputList.push\_back(nextVertex);

        }

        else if (vertex.A.y <= viewPort.y + viewPort.height && nextVertex.A.y > viewPort.y + viewPort.height)

        {

            Vertex newVertex;

            newVertex.A.x = vertex.A.x + (nextVertex.A.x - vertex.A.x) \* (viewPort.y + viewPort.height - vertex.A.y) / (nextVertex.A.y - vertex.A.y);

            newVertex.A.y = viewPort.y + viewPort.height;

            newVertex.B.x = nextVertex.A.x;

            newVertex.B.y = nextVertex.A.y;

            outputList.push\_back(newVertex);

        }

        else if (vertex.A.y > viewPort.y + viewPort.height && nextVertex.A.y <= viewPort.y + viewPort.height)

        {

            Vertex newVertex;

            newVertex.A.x = vertex.A.x + (nextVertex.A.x - vertex.A.x) \* (viewPort.y + viewPort.height - vertex.A.y) / (nextVertex.A.y - vertex.A.y);

            newVertex.A.y = viewPort.y + viewPort.height;

            newVertex.B.x = nextVertex.A.x;

            newVertex.B.y = nextVertex.A.y;

            outputList.push\_back(newVertex);

            outputList.push\_back(nextVertex);

        }

    }

    // clipping against left\_VIEWPORT boundary

    vertexList = outputList;

    outputList.clear();

    for (*int* i = 0; i < vertexList.size(); i++)

    {

        Vertex vertex = vertexList[i];

        Vertex nextVertex = vertexList[(i + 1) % vertexList.size()];

        if (vertex.A.x >= viewPort.x && nextVertex.A.x >= viewPort.x)

        {

            outputList.push\_back(nextVertex);

        }

        else if (vertex.A.x >= viewPort.x && nextVertex.A.x < viewPort.x)

        {

            Vertex newVertex;

            newVertex.A.x = viewPort.x;

            newVertex.A.y = vertex.A.y + (nextVertex.A.y - vertex.A.y) \* (viewPort.x - vertex.A.x) / (nextVertex.A.x - vertex.A.x);

            newVertex.B.x = nextVertex.A.x;

            newVertex.B.y = nextVertex.A.y;

            outputList.push\_back(newVertex);

        }

        else if (vertex.A.x < viewPort.x && nextVertex.A.x >= viewPort.x)

        {

            Vertex newVertex;

            newVertex.A.x = viewPort.x;

            newVertex.A.y = vertex.A.y + (nextVertex.A.y - vertex.A.y) \* (viewPort.x - vertex.A.x) / (nextVertex.A.x - vertex.A.x);

            newVertex.B.x = nextVertex.A.x;

            newVertex.B.y = nextVertex.A.y;

            outputList.push\_back(newVertex);

            outputList.push\_back(nextVertex);

        }

    }

    // drawing clipped polygon

    glColor3f(0.0, 1.0, 0.0);

    glBegin(GL\_LINE\_LOOP);

    for (*int* i = 0; i < outputList.size(); i++)

    {

        // displaying vertex name

        glVertex2i(outputList[i].A.x, outputList[i].A.y);

    }

    glEnd();

    for (*int* i = 0; i < outputList.size(); i++)

    {

        // displaying vertex name

        glRasterPos2i(outputList[i].A.x + 10, outputList[i].A.y - 20);

        glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, 'A' + i);

    }

}

std::vector<Line> getLines()

{

    std::vector<Line> lines;

    // getting input from user

    std::cout << "Enter number of lines: ";

*int* n;

    std::cin >> n;

    for (*int* i = 0; i < n; i++)

    {

        Line line;

        std::cout << "Enter line " << i + 1 << " x1 y1 x2 y2: ";

        std::cin >> line.x1 >> line.y1 >> line.x2 >> line.y2;

        lines.push\_back(line);

    }

    // lines.push\_back({ 450, 300, 140, 260 });

    // lines.push\_back({ 20, 20, 240, 370 });

    // lines.push\_back({ 300, 300, 350, 350 });

    // lines.push\_back({ 100, 100, 150, 150 });

    // lines.push\_back({ 300, 300, 700, 700 });

    // lines.push\_back({ 100, 300, 700, 300 });

    // lines.push\_back({ 100, 400, 300, 400 });

    // lines.push\_back({ 500, 400, 700, 400 });

    // lines.push\_back({ 400, 100, 400, 300 });

    // lines.push\_back({ 400, 500, 400, 700 });

    // lines.push\_back({ 100, 100, 700, 700 });

    // lines.push\_back({ 100, 700, 700, 100 });

    // lines.push\_back({ 100, 100, 700, 100 });

    // lines.push\_back({ 100, 100, 100, 700 });

    // lines.push\_back({ 200, 200, 700, 700 });

    // lines.push\_back({ 700, 700, 200, 200 });

    // lines.push\_back({ 200, 300, 600, 300 });

    return lines;

}

std::vector<PolygonShape> getPolygons()

{

    // getting input from user

    std::vector<PolygonShape> polygons;

    std::cout << "Enter number of polygons: ";

*int* n;

    std::cin >> n;

    for (*int* i = 0; i < n; i++)

    {

        PolygonShape polygon;

        std::cout << "Enter polygon " << i + 1 << " number of points: ";

*int* m;

        std::cin >> m;

        for (*int* j = 0; j < m; j++)

        {

            Point point;

            std::cout << "Enter point " << j + 1 << " x y: ";

            std::cin >> point.x >> point.y;

            polygon.points.push\_back(point);

        }

        polygons.push\_back(polygon);

    }

    //  PolygonShape polygon;

    //  polygon.points.push\_back({ 450, 700 });

    //  polygon.points.push\_back({ 700, 400 });

    //  polygon.points.push\_back({ 400, 400 });

    //  polygon.points.push\_back({ 400, 100 });

    //  polygon.points.push\_back({ 100, 400 });

    //  polygons.push\_back(polygon);

    return polygons;

}

Params getParams(*bool* *lines* = false, *bool* *polygons* = false)

{

    Params params;

    ViewPort viewPort;

    std::cout << "Enter ViewPort: " << std::endl;

    std::cout << "Enter width height: ";

    std::cin >> viewPort.width >> viewPort.height;

    params.viewPort = viewPort;

    if (lines)

        params.lines = getLines();

    if (polygons)

        params.polygons = getPolygons();

    return params;

}

# Lab Rubrics

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Lab Rubrics for (Lab-07: Clipping (Line Clipping and Polygon Clipping)) | | | | | |
|  | | | | | |
| **Sr.**  **No.** | **Assessment** | **Unacceptable (0 Marks)** | **Does Not Meet Expectations (1/2 Marks)** | **Meets Expectations (3/4 Marks)** | **Exceeds Expectations (5 Marks)** |
| **1** | **Illustrating the basic understanding of semantics and syntax**  **(CLO3, PLO5)** | The student did not submit any work.  OR  The student plagiarized the solution and/or used unfair means. | The student is unable to demonstrate the understanding of syntax of C language and is unable to write an executable code.  The student is not able to understand the structure of a program at all. | The student demonstrates some understanding of syntax of C language and is able to write a code with few errors.  The student is able to understand the structure but still learning the syntax. | The student demonstrates good understanding of syntax of C language and is able to write executable code without help  The student is able to understand the structure and is able to identify problems in the code  when introduced |
| **2** | **Software Tool Usage**  **(CLO4-PLO3)** | The student demonstrates a lack of understanding of tool usage.  Implementation has syntax/semantic/runtime errors, and the student is unable to debug and correct the errors.  The code has inadequate comments and variable names and does not adhere to the coding standards.  No Error handling has been performed.  Documentation is poorly structured. | The student demonstrates some understanding of tool usage.  The codes are correct in terms of their syntax, however, the program output is not always correct in all test cases.  The code has limited comments and inconsistent variable names and may not adhere to the coding standards.  Some Error handling has been performed.  Documentation is adequately structured. | The student demonstrates a good understanding of tool usage.  Furthermore, his/her coding is complete and functional, and the program output is correct in all test cases.  The code has sufficient comments and consistent variable names and reasonably adhere to the coding standards.  Adequate Error handling has been performed.  Documentation is well structured. |