**Department of Computing**

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

**Lab 03: Implementing Rasterization**

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

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

**Date: 11th Feb 2025**

**Time: 02:00 PM – 04:50 PM**

# Instructor: Dr. Sidra Sutana

# Lab Engineer: Mr. Aftab Farooq

**Name:** Ahmed Mohiuddin Shah

**CMS ID:** 415216

**Section:** BSCS-12-A

**Lab:** 3

**Lab 03: Implementing Rasterization**

### **Lab Objective:**

* This lab manual will guide you through the implementation of algorithms for drawing geometric shapes such as circles, ellipses, and parabolas.
* You will also implement different polygon filling techniques, including Boundary Fill, Flood Fill, and Scanline Fill.

## 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++

**Task 1: Drawing a Circle (Midpoint Circle Algorithm):**  
 - Implement the **Midpoint Circle Algorithm** to draw a circle.  
 - Given a center and a radius, compute the circle points.

### **Algorithm Explanation:** The Midpoint Circle Algorithm is an efficient method for drawing a circle by using symmetry. It calculates the next point using a decision parameter based on the midpoint between two possible points.

### **Steps:**

1. Initialize the center of the circle (xc, yc) and the radius (r).

2. Set the initial point at (0, r).

3. Use the decision parameter to determine the next point.

4. Plot the points in all octants using symmetry.

**Solution:**

**Code:**

#include <GL/freeglut.h>

*void* drawCirclePoints(*int* *x*, *int* *y*, *int* *center\_x*, *int* *center\_y*)

{

    // We used the 8 point symmetry of the circle to save on number of calculations

    glBegin(GL\_POINTS);

    glVertex2i(*center\_x* + *x*, *center\_y* + *y*);

    glVertex2i(*center\_x* - *x*, *center\_y* + *y*);

    glVertex2i(*center\_x* + *x*, *center\_y* - *y*);

    glVertex2i(*center\_x* - *x*, *center\_y* - *y*);

    glVertex2i(*center\_x* + *y*, *center\_y* + *x*);

    glVertex2i(*center\_x* - *y*, *center\_y* + *x*);

    glVertex2i(*center\_x* + *y*, *center\_y* - *x*);

    glVertex2i(*center\_x* - *y*, *center\_y* - *x*);

    glEnd();

}

*void* drawCircle(*int* *center\_x*, *int* *center\_y*, *int* *radius*)

{

*int* x = 0, y = *radius*;

*int* p = 1 - *radius*; // Midpoint decision parameter

    drawCirclePoints(x, y, *center\_x*, *center\_y*);

    while (x < y)

    {

        x++;

        if (p < 0)

            p += 2 \* x + 1;

        else

        {

            y--;

            p += 2 \* x - 2 \* y + 1;

        }

        drawCirclePoints(x, y, *center\_x*, *center\_y*);

    }

    glFlush();

}

*void* display()

{

    glClear(GL\_COLOR\_BUFFER\_BIT);

    // Drawing red Circles

    glColor3f(1.0f, 0.0f, 0.0f);

    drawCircle(280, 280, 100);

    glColor3f(1.0f, 1.0f, 0.0f);

    drawCircle(120, 120, 80);

}

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

{

    glutInit(&*argc*, *argv*);

    glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

    glutInitWindowSize(500, 500); // Here we set the window size

    glutCreateWindow("Task 1");

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Set background color to black

    glMatrixMode(GL\_PROJECTION);          // Switch to projection matrix

    glLoadIdentity();                     // Reset the matrix

    gluOrtho2D(0, 500, 500, 0);           // Set orthographic projection

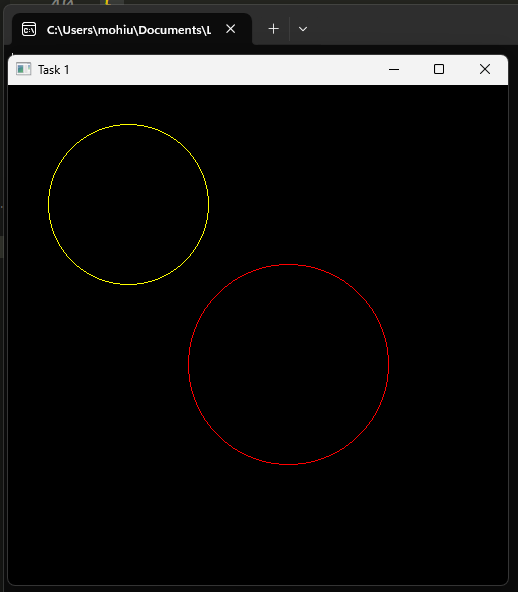
    glutDisplayFunc(display);

    glutMainLoop();

    return 0;

}

**Output:**

****

**Task 2: Drawing an Ellipse (Midpoint Ellipse Algorithm):**  
- Implement the **Midpoint Ellipse Algorithm** to draw an ellipse with given axes.

### **Algorithm Explanation:** The Midpoint Ellipse Algorithm divides the ellipse into two regions. The decision parameter is calculated for each region separately, using symmetry to plot points in all quadrants.

**Algorithm Steps:**

1. Initialize the center (xc, yc), semi-major axis (a), and semi-minor axis (b).

2. Calculate the initial points and use the decision parameter to determine the next points.

3. Plot points in all quadrants.

**Solution:**

**Code:**

#include <GL/freeglut.h>

*void* drawEllipsePoints(*int* *x*, *int* *y*, *int* *xc*, *int* *yc*)

{

    // We used the 4 point symmetry of the ellipse to save on number of calculations

    glBegin(GL\_POINTS);

    glVertex2i(*x* + *xc*, *y* + *yc*);

    glVertex2i(-*x* + *xc*, *y* + *yc*);

    glVertex2i(*x* + *xc*, -*y* + *yc*);

    glVertex2i(-*x* + *xc*, -*y* + *yc*);

    glEnd();

}

*void* drawEllipse(*int* *xc*, *int* *yc*, *int* *major*, *int* *minor*, *bool* *show\_diff\_color* = false)

{

*float* dx, dy, d1, d2, x, y;

    x = 0;

    y = *minor*;

    // Initial decision parameter of region 1

    d1 = (*minor* \* *minor*) - (*major* \* *major* \* *minor*) +

         (0.25 \* *major* \* *major*);

    dx = 2 \* *minor* \* *minor* \* x;

    dy = 2 \* *major* \* *major* \* y;

    if (*show\_diff\_color*)

    {

        glColor3f(0.0f, 1.0f, 1.0f);

    }

    // For region 1

    while (dx < dy)

    {

        // Print points based on 4-way symmetry

        drawEllipsePoints(x, y, *xc*, *yc*);

        // Checking and updating value of

        // decision parameter based on algorithm

        if (d1 < 0)

        {

            x++;

            dx = dx + (2 \* *minor* \* *minor*);

            d1 = d1 + dx + (*minor* \* *minor*);

        }

        else

        {

            x++;

            y--;

            dx = dx + (2 \* *minor* \* *minor*);

            dy = dy - (2 \* *major* \* *major*);

            d1 = d1 + dx - dy + (*minor* \* *minor*);

        }

    }

    if (*show\_diff\_color*)

    {

        glColor3f(1.0f, 1.0f, 0.0f);

    }

    // Decision parameter of region 2

    d2 = ((*minor* \* *minor*) \* ((x + 0.5) \* (x + 0.5))) +

         ((*major* \* *major*) \* ((y - 1) \* (y - 1))) -

         (*major* \* *major* \* *minor* \* *minor*);

    // Plotting points of region 2

    while (y >= 0)

    {

        // Print points based on 4-way symmetry

        drawEllipsePoints(x, y, *xc*, *yc*);

        // Checking and updating parameter

        // value based on algorithm

        if (d2 > 0)

        {

            y--;

            dy = dy - (2 \* *major* \* *major*);

            d2 = d2 + (*major* \* *major*) - dy;

        }

        else

        {

            y--;

            x++;

            dx = dx + (2 \* *minor* \* *minor*);

            dy = dy - (2 \* *major* \* *major*);

            d2 = d2 + dx - dy + (*major* \* *major*);

        }

    }

    glFlush();

}

*void* display()

{

    glClear(GL\_COLOR\_BUFFER\_BIT);

    // Drawing Ellipses

    glColor3f(1.0f, 0.0f, 0.0f);

    drawEllipse(100, 200, 100, 70);

    drawEllipse(300, 400, 90, 40, true);

}

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

{

    glutInit(&*argc*, *argv*);

    glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

    glutInitWindowSize(500, 500); // Here we set the window size

    glutCreateWindow("Task 1");

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Set background color to black

    glMatrixMode(GL\_PROJECTION);          // Switch to projection matrix

    glLoadIdentity();                     // Reset the matrix

    gluOrtho2D(0, 500, 500, 0);           // Set orthographic projection

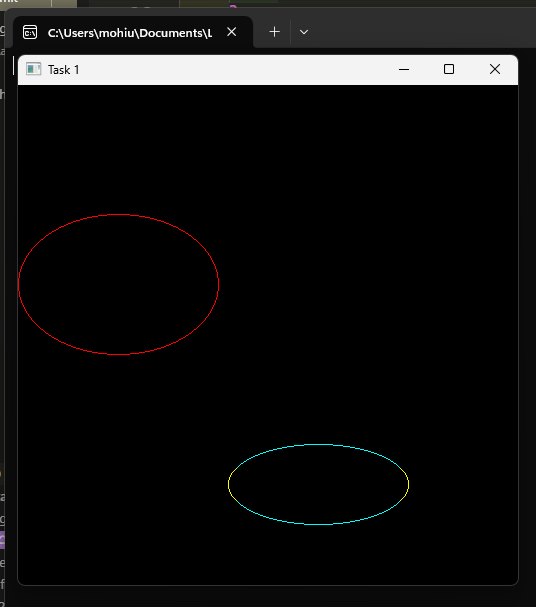
    glutDisplayFunc(display);

    glutMainLoop();

    return 0;

}

**Output:**

****

**Task 3: Drawing a Parabola**

- Implement the **Midpoint Parabola Algorithm** to draw a parabola.

### **Algorithm Explanation:** The Midpoint Parabola Algorithm draws a parabola by using incremental calculations of the midpoint and plotting the curve symmetrically.

**Algorithm Steps:**1. Define the vertex of the parabola (h, k) and the focus.

2. Calculate points using the parabola equation.  
3. Plot the points.

**Solution:**

**Code:**

#include <GL/glut.h>

*void* drawParabolaPoints(*int* *x*, *int* *y*, *int* *xc*, *int* *yc*)

{

    glBegin(GL\_POINTS);

    glVertex2i(*xc* + *x*, *yc* + *y*);

    glVertex2i(*xc* + *x*, *yc* - *y*);

    glEnd();

}

*void* drawParabola(*int* *xc*, *int* *yc*, *int* *a*, *bool* *show\_diff\_color* = false)

{

*int* x = 0, y = 0;

*int* p = 1 - 2 \* *a*;

    if (*show\_diff\_color*)

    {

        glColor3f(1.0f, 1.0f, 0.0f);

    }

    // Region 1: |dy/dx| <= 1

    while (y < 2 \* *a*)

    {

        drawParabolaPoints(x, y, *xc*, *yc*);

        if (p < 0)

        {

            y++;

            p += 2 \* y + 1;

        }

        else

        {

            x++;

            y++;

            p += 2 \* y + 1 - 4 \* *a*;

        }

    }

    if (*show\_diff\_color*)

    {

        glColor3f(0.0f, 1.0f, 1.0f);

    }

    // Region 2: |dy/dx| > 1

    p = (y + 0.5) \* (y + 0.5) - 4 \* *a* \* (x + 1);

    while (x < 200)

    {

        drawParabolaPoints(x, y, *xc*, *yc*);

        if (p > 0)

        {

            x++;

            p += 1 - 4 \* *a*;

        }

        else

        {

            x++;

            y++;

            p += 2 \* y + 1 - 4 \* *a*;

        }

    }

    glFlush();

}

*void* display()

{

    glClear(GL\_COLOR\_BUFFER\_BIT);

    // Drawing Parabola

    glColor3f(1.0f, 0.0f, 0.0f);

    drawParabola(10, 200, 70);

    drawParabola(50, 200, 40, true);

}

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

{

    glutInit(&*argc*, *argv*);

    glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

    glutInitWindowSize(500, 500); // Here we set the window size

    glutCreateWindow("Task 2");

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Set background color to black

    glMatrixMode(GL\_PROJECTION);          // Switch to projection matrix

    glLoadIdentity();                     // Reset the matrix

    gluOrtho2D(0, 500, 500, 0);           // Set orthographic projection

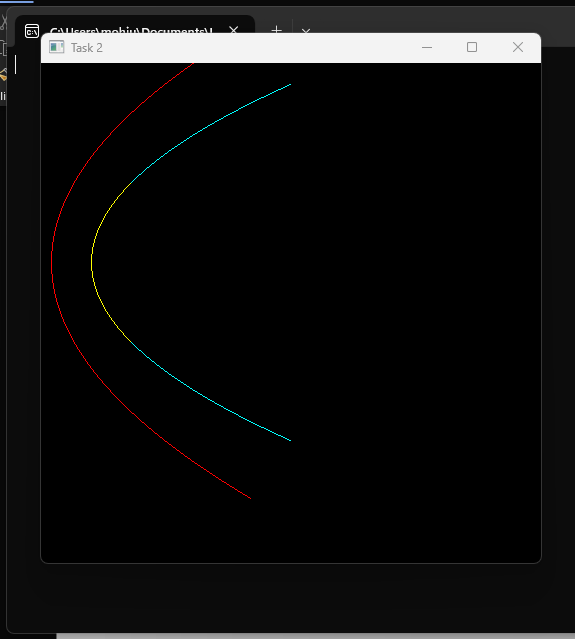
    glutDisplayFunc(display);

    glutMainLoop();

    return 0;

}

**Output:**

****

**Task 4: Polygon Filling Algorithms**

**Subtask 1: Boundary Fill Algorithm:**

- Implement the **Boundary Fill Algorithm** to fill a polygon with a color until the boundary is reached.

**Algorithm Explanation:**

The Boundary Fill Algorithm starts from a seed point inside the polygon and fills the area with the given color until the boundary color is encountered.

**Algorithm Steps:**

1. Start from a seed point inside the polygon.

2. Check the color of the current pixel.

3. If it matches the boundary color, return; otherwise, change the color and call the function recursively for neighboring pixels.

**Solution:**

**Code:**

#include <GL/glut.h>

#include <iostream>

// Define boundary color (black) and fill color (red)

*float* boundaryColor[3] = {0.0f, 0.0f, 0.0f};

*float* fillColor[3] = {1.0f, 0.0f, 0.0f};

// Function to get pixel color at a given coordinate

*void* getPixelColor(*int* *x*, *int* *y*, *float* *color*[3])

{

    glReadPixels(*x*, *y*, 1, 1, GL\_RGB, GL\_FLOAT, *color*);

}

// Function to set pixel color

*void* setPixelColor(*int* *x*, *int* *y*, *float* *color*[3])

{

    glBegin(GL\_POINTS);

    glColor3fv(*color*);

    glVertex2i(*x*, *y*);

    glEnd();

    glFlush();

}

// Recursive Boundary Fill Algorithm

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

{

*float* currentColor[3];

    getPixelColor(*x*, 500 - *y*, currentColor);

    // Check if current pixel is not boundary and not already filled

    if ((currentColor[0] != boundaryColor[0] || currentColor[1] != boundaryColor[1] || currentColor[2] != boundaryColor[2]) &&

        (currentColor[0] != fillColor[0] || currentColor[1] != fillColor[1] || currentColor[2] != fillColor[2]))

    {

        setPixelColor(*x*, *y*, fillColor);

        // Recursive calls for 4-connected neighbors

        boundaryFill(*x* + 1, *y*);

        boundaryFill(*x* - 1, *y*);

        boundaryFill(*x*, *y* + 1);

        boundaryFill(*x*, *y* - 1);

    }

}

// Function to draw a polygon

*void* drawPolygon(*int* *x*, *int* *y*, *int* *width*, *int* *height*)

{

    glColor3f(1.0, 0.0, 0.0); // Black boundary color

    glBegin(GL\_LINE\_LOOP);

    glVertex2i(*x*, *y*);

    glVertex2i(*x* + *width*, *y*);

    glVertex2i(*x* + *width*, *y* + *height*);

    glVertex2i(*x*, *y* + *height*);

    glEnd();

    glFlush();

}

// Display function

*void* display()

{

    glClear(GL\_COLOR\_BUFFER\_BIT);

    drawPolygon(50, 50, 60, 60);

    boundaryFill(60, 70); // Convert OpenGL coordinates

}

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

{

    glutInit(&*argc*, *argv*);

    glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

    glutInitWindowSize(500, 500); // Here we set the window size

    glutCreateWindow("Task 4");

    glClearColor(0.0f, 0.0f, 0.05f, 1.0f); // Set background color to black

    glMatrixMode(GL\_PROJECTION);           // Switch to projection matrix

    glLoadIdentity();                      // Reset the matrix

    gluOrtho2D(0, 500, 500, 0);            // Set orthographic projection

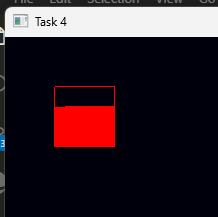
    glutDisplayFunc(display);

    glutMainLoop();

    return 0;

}

**Output:**

 A screenshot of a computer

AI-generated content may be incorrect.

**Subtask 2: Flood Fill Algorithm**

- Implement the **Flood Fill Algorithm** to fill a polygon using recursive or iterative approaches.

**Algorithm Explanation:**

The **Flood Fill Algorithm** starts from a seed point inside the polygon and fills all neighboring pixels until the entire area is colored. Unlike the boundary fill, it continues until all pixels are filled without considering boundaries but focusing on the existing color.

**Algorithm Steps:**1. Start from a seed point.

2. Change the color of the current pixel.

3. Recursively call the function for neighboring pixels.

**Solution:**

**Code:**

#include <GL/freeglut.h>

#include <iostream>

// Define old color (white) and fill color (red)

*float* oldColor[3] = {1.0f, 1.0f, 1.0f};

*float* fillColor[3] = {1.0f, 0.0f, 0.0f};

// Function to get pixel color at a given coordinate

*void* getPixelColor(*int* *x*, *int* *y*, *float* *color*[3])

{

    glReadPixels(*x*, *y*, 1, 1, GL\_RGB, GL\_FLOAT, *color*);

}

// Function to set pixel color

*void* setPixelColor(*int* *x*, *int* *y*, *float* *color*[3])

{

    glBegin(GL\_POINTS);

    glColor3fv(*color*);

    glVertex2i(*x*, *y*);

    glEnd();

    glFlush();

}

// Recursive Flood Fill Algorithm

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

{

*float* currentColor[3];

    getPixelColor(*x*, 500 - *y*, currentColor);

    // Check if pixel is not already filled

    if (currentColor[0] == oldColor[0] && currentColor[1] == oldColor[1] && currentColor[2] == oldColor[2])

    {

        setPixelColor(*x*, *y*, fillColor);

        // Recursive calls for 4-connected neighbors

        floodFill(*x* + 1, *y*);

        floodFill(*x* - 1, *y*);

        floodFill(*x*, *y* + 1);

        floodFill(*x*, *y* - 1);

    }

}

// Function to draw a polygon

*void* drawPolygon(*int* *x*, *int* *y*, *int* *width*, *int* *height*)

{

    glColor3f(1.0, 0.0, 0.0); // Black boundary color

    glBegin(GL\_LINE\_LOOP);

    glVertex2i(*x*, *y*);

    glVertex2i(*x* + *width*, *y*);

    glVertex2i(*x* + *width*, *y* + *height*);

    glVertex2i(*x*, *y* + *height*);

    glEnd();

    glFlush();

}

// Display function

*void* display()

{

    glClear(GL\_COLOR\_BUFFER\_BIT);

    drawPolygon(50, 50, 60, 60);

    floodFill(60, 70); // Convert OpenGL coordinates

}

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

{

    glutInit(&*argc*, *argv*);

    glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

    glutInitWindowSize(500, 500); // Here we set the window size

    glutCreateWindow("Task 4.2");

    glClearColor(1.0f, 1.0f, 1.05f, 1.0f); // Set background color to black

    glMatrixMode(GL\_PROJECTION);           // Switch to projection matrix

    glLoadIdentity();                      // Reset the matrix

    gluOrtho2D(0, 500, 500, 0);            // Set orthographic projection

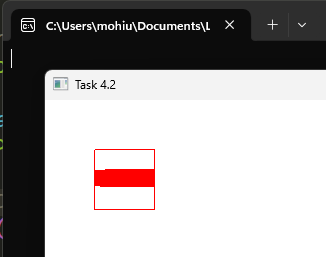
    glutDisplayFunc(display);

    glutMainLoop();

    return 0;

}

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Subtask 3: Scanline Fill Algorithm**

- Implement the **Scanline Polygon Fill Algorithm** to fill a polygon by processing horizontal scanlines.

**Algorithm Explanation:**

The **Scanline Fill Algorithm** works by processing each horizontal line (scanline) of the polygon. For each scanline, it calculates the intersection points of the scanline with the polygon edges, then fills between pairs of intersections.

**Algorithm Steps:**1. Identify the edges of the polygon.

2. For each scanline, find the intersection points of the scanline with the polygon's edges.

3. Sort the intersection points, and fill between each pair of intersections. **Solution:**

**Code:**

#include <GL/glut.h>

#include <vector>

#include <algorithm>

using *namespace* std;

*struct* Point

{

*int* x, y;

};

// Global variables

vector<Point> polygon = {{100, 150}, {200, 50}, {300, 150}, {250, 300}, {150, 300}}; // Example polygon

*void* drawPolygon()

{

    glColor3f(0, 0, 0); // Black color for boundary

    glBegin(GL\_LINE\_LOOP);

    for (*auto* &p : polygon)

    {

        glVertex2i(p.x, p.y);

    }

    glEnd();

    glFlush();

}

*void* scanlineFill()

{

*int* ymin = polygon[0].y, ymax = polygon[0].y;

    // Find ymin and ymax

    for (*auto* &p : polygon)

    {

        if (p.y < ymin)

            ymin = p.y;

        if (p.y > ymax)

            ymax = p.y;

    }

    for (*int* y = ymin; y <= ymax; y++)

    {

        vector<*int*> intersections;

        for (size\_t i = 0; i < polygon.size(); i++)

        {

            Point p1 = polygon[i];

            Point p2 = polygon[(i + 1) % polygon.size()];

            if (p1.y != p2.y)

            { // Ignore horizontal edges

                if ((y >= p1.y && y < p2.y) || (y >= p2.y && y < p1.y))

                {

*int* x = p1.x + (y - p1.y) \* (p2.x - p1.x) / (p2.y - p1.y);

                    intersections.push\_back(x);

                }

            }

        }

        sort(intersections.begin(), intersections.end());

        glColor3f(1, 0, 0); // Fill color (red)

        glBegin(GL\_LINES);

        for (size\_t i = 0; i < intersections.size(); i += 2)

        {

            glVertex2i(intersections[i], y);

            glVertex2i(intersections[i + 1], y);

        }

        glEnd();

        glFlush();

    }

}

*void* display()

{

    glClear(GL\_COLOR\_BUFFER\_BIT);

    drawPolygon();

    scanlineFill();

}

*void* init()

{

    glClearColor(1, 1, 1, 1);

    gluOrtho2D(0, 500, 0, 500);

}

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

{

    glutInit(&*argc*, *argv*);

    glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

    glutInitWindowSize(500, 500);

    glutCreateWindow("Task 4.3");

    init();

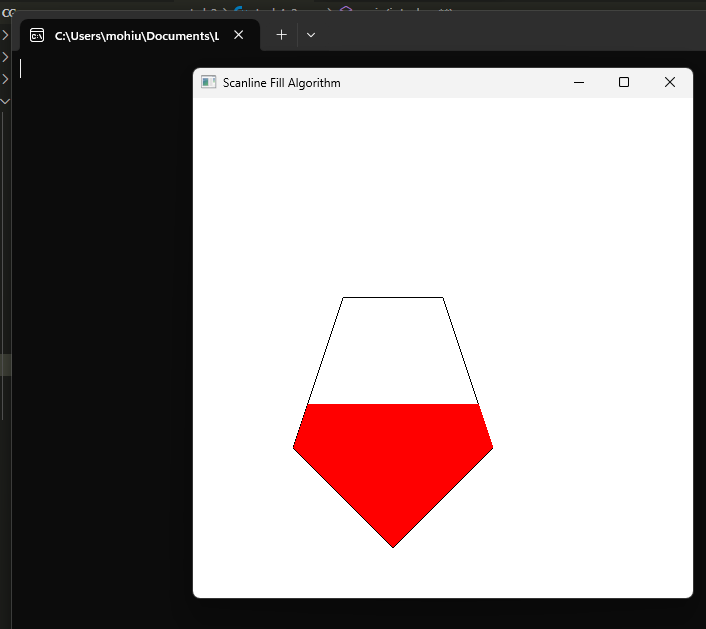
    glutDisplayFunc(display);

    glutMainLoop();

    return 0;

}

**Output:**



A screenshot of a computer

AI-generated content may be incorrect.

### **Deliverables:**

 Compile a single word document by filling in the solution part and submit this Word file on LMS

 Submit your code files (.cpp) for both the DDA and Bresenham’s algorithms.

 Include screenshots of the program outputs.

 Submit your Lab Word File and code files seperately on submission link.

# Lab Rubrics

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Lab Rubrics for Lab 3 (Implementing Rasterization) | | | | | |
|  | | | | | |
| **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. |