DEERWALK INSTITUTE OF TECHNOLOGY

Tribhuvan University

Faculties of Computer Science

Bachelors of Science in Computer Science and Information Technology (BSc. CSIT)

Course: Computer Graphics (CSC209)

Year/Semester: II/III

A Lab report on:

Implementation of Filling Algorithms in C++

Submitted by: Name: Arun Mainali Roll: 1307	Submitted to: Binod Sitaula Department of Computer Science

Submission Date: 03/21/2025

❖ LAB 8

OBJECTIVE:

Write a program in any high-level language to Filling Algorithms:

- Flood Fill Algorithm
- Boundary Filling Algorithm

THEORY:

Boundary Fill Algorithm:

The Boundary Fill algorithm works by starting from a point inside the polygon and then filling outward until a specified boundary color is encountered. This algorithm works for regions that are enclosed by a single color. The algorithm starts at a seed point (x, y) inside the region and colors all the connected pixels that are not of the boundary color.

There are two approaches to implementing the boundary fill algorithm:

1. **4-connected approach**: Each pixel is connected to 4 neighbors (left, right, top, bottom).
2. **8-connected approach**: Each pixel is connected to 8 neighbors (4 adjacent + 4 diagonal).

The steps for the boundary fill algorithm are as follows:

1. Start from the seed point (x, y).
2. Check if the current pixel is neither colored with the boundary color nor the fill color.
3. If the condition is true, color the current pixel with the fill color.
4. Recursively call the boundary fill function for the neighboring pixels.

The recursive implementation of the boundary fill algorithm can cause stack overflow for large regions, so iterative versions using a stack or queue are often preferred in practice.

Flood Fill Algorithm:

The Flood Fill algorithm is similar to the boundary fill algorithm, but instead of looking for a boundary color, it replaces a specific old color with a new fill color. This algorithm is suitable for filling a region consisting of similar colors.

The algorithm starts from a seed point (x, y) and colors all the connected pixels that have the same color as the seed pixel. Like Boundary Fill, Flood Fill can also be implemented using 4-connected or 8-connected approaches.

The steps for the flood fill algorithm are as follows:

1. Start from the seed point (x, y).
2. Check if the current pixel color is the same as the old color.
3. If the condition is true, color the current pixel with the new fill color.
4. Recursively call the flood fill function for the neighboring pixels.

Advantages and Disadvantages:

Boundary Fill:

- **Advantages:**
  - Simple to implement
  - Works well for simple shapes
- **Disadvantages:**
  - Stack overflow for large regions
  - Only works for regions with a single boundary color

Flood Fill:

- **Advantages:**
  - Flexible for regions with similar colors
  - Simple to implement
- **Disadvantages:**
  - Stack overflow for large regions
  - Not efficient for complex shapes

Algorithm:

Boundary Fill Algorithm:

1. Start with a seed point (x, y) inside the boundary.
2. Check the color of the current pixel:
   1. If the pixel is not the boundary color and not the fill color, proceed.
   2. Otherwise, return (base case for recursion).
3. Set the pixel color to the fill color.
4. Recursively apply the algorithm to its neighboring pixels (4-connected or 8-connected).
   1. 4-connected: Move to (x+1, y), (x-1, y), (x, y+1), (x, y-1).
   2. 8-connected: Move to (x+1, y), (x-1, y), (x, y+1), (x, y-1), (x+1, y+1), (x-1, y+1), (x+1, y-1), (x-1, y-1).
5. Repeat the process until all pixels inside the boundary are filled.

Flood Fill Algorithm:

1. Start with a seed point (x, y).

2. Get the current color of the pixel.

3. If the current color is already the fill color or is different from the original color, return.

4. Set the pixel color to the fill color.

5. Recursively apply the algorithm to its neighboring pixels (4-connected or 8-connected):

   1. 4-connected: (x+1, y), (x-1, y), (x, y+1), (x, y-1)

   2. 8-connected: (x+1, y+1), (x-1, y+1), (x+1, y-1), (x-1, y-1)

6. Repeat the process until the entire region is filled.

Implementation of Flood Filling Algorithms in OpenGL:

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <iostream>
#include <vector>
#include <queue>
#include <cmath>

// Window dimensions
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;

// Shader sources
const char* vertexShaderSource = R"(
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec2 aTexCoord;
out vec2 TexCoord;
void main()
{
    gl_Position = vec4(aPos, 0.0, 1.0);
    TexCoord = aTexCoord;
}
)";

const char* fragmentShaderSource = R"(
#version 330 core
out vec4 FragColor;
in vec2 TexCoord;
uniform sampler2D ourTexture;
void main()
{
    FragColor = texture(ourTexture, TexCoord);
}
)";

// Global variables
unsigned int texture;
std::vector<unsigned char> pixelData;
int textureWidth, textureHeight;
```

```
float fillColorR = 1.0f, fillColorG = 0.0f, fillColorB = 0.0f; // Default fill color
(red)
bool mousePressed = false;

// Function prototypes
void framebuffer_size_callback(GLFWwindow* window, int width, int height);
void mouse_button_callback(GLFWwindow* window, int button, int action, int
mods);
void key_callback(GLFWwindow* window, int key, int scancode, int action, int
mods);
void processInput(GLFWwindow* window);
void floodFill(int x, int y, unsigned char targetR, unsigned char targetG,
unsigned char targetB);
bool colorMatch(unsigned char r1, unsigned char g1, unsigned char b1,
    unsigned char r2, unsigned char g2, unsigned char b2, int tolerance = 10);
void updateTexture();

int main()
{
    // Initialize GLFW
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE);

    // Create a GLFW window
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH,
SCR_HEIGHT, "Flood Fill Algorithm", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
    glfwSetMouseButtonCallback(window, mouse_button_callback);
    glfwSetKeyCallback(window, key_callback);

    // Initialize GLAD
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return -1;
    }

    // Build and compile shaders
    // Vertex shader
    unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    glCompileShader(vertexShader);

    // Check for shader compile errors
    int success;
    char infoLog[512];
    glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
        glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
        std::cout <<
"ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog <<
std::endl;
    }

    // Fragment shader
    unsigned int fragmentShader =
glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
    glCompileShader(fragmentShader);

    // Check for shader compile errors
    glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
        glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
        std::cout <<
"ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog
<< std::endl;
    }

    // Link shaders
    unsigned int shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    glLinkProgram(shaderProgram);
    // Check for linking errors
    glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
    if (!success) {
        glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n"
<< infoLog << std::endl;
    }

    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);

    // Set up vertex data and buffers
    float vertices[] = {
        // positions         // texture coords
         1.0f,  1.0f,        1.0f, 1.0f,   // top right
         1.0f, -1.0f,        1.0f, 0.0f,   // bottom right
        -1.0f, -1.0f,        0.0f, 0.0f,   // bottom left
        -1.0f,  1.0f,        0.0f, 1.0f    // top left
    };
    unsigned int indices[] = {
        0, 1, 3,  // first triangle
        1, 2, 3   // second triangle
    };

    unsigned int VBO, VAO, EBO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);

    // Position attribute
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float),
(void*)0);
```

```
    glEnableVertexAttribArray(0);
    // Texture coord attribute
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float),
(void*)(2 * sizeof(float)));
    glEnableVertexAttribArray(1);

    // Create and bind texture
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);

    // Set texture parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);

    // Set texture dimensions
    textureWidth = 400;
    textureHeight = 300;

    // Initialize pixel data with a checkerboard pattern
    pixelData.resize(textureWidth * textureHeight * 4);
    for (int y = 0; y < textureHeight; y++) {
        for (int x = 0; x < textureWidth; x++) {
            int index = (y * textureWidth + x) * 4;
            if ((x / 40 + y / 40) % 2 == 0) {
                pixelData[index] = 200;     // R
                pixelData[index + 1] = 200; // G
                pixelData[index + 2] = 200; // B
            }
            else {
                pixelData[index] = 100;     // R
                pixelData[index + 1] = 100; // G
                pixelData[index + 2] = 100; // B
            }
            pixelData[index + 3] = 255;     // A
        }
    }
```

```
// Send texture data to GPU
updateTexture();

// Print instructions
std::cout << "Flood Fill Application\n"
    << "----------------------\n"
    << "Left-click: Perform flood fill\n"
    << "R: Set fill color to Red\n"
    << "G: Set fill color to Green\n"
    << "B: Set fill color to Blue\n"
    << "Y: Set fill color to Yellow\n"
    << "C: Clear to checkerboard pattern\n"
    << "ESC: Exit\n";

// Render loop
while (!glfwWindowShouldClose(window))
{
    // Input
    processInput(window);

    // Render
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // Draw textured quad
    glUseProgram(shaderProgram);
    glBindTexture(GL_TEXTURE_2D, texture);
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

    // Swap buffers and poll events
    glfwSwapBuffers(window);
    glfwPollEvents();
}

// Clean up
glDeleteVertexArrays(1, &VAO);
glDeleteBuffers(1, &VBO);
glDeleteBuffers(1, &EBO);
glDeleteProgram(shaderProgram);
glDeleteTextures(1, &texture);

glfwTerminate();
```

```
    return 0;
}

// Called when window is resized
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}

// Called when a mouse button is pressed/released
void mouse_button_callback(GLFWwindow* window, int button, int action, int mods)
{
    if (button == GLFW_MOUSE_BUTTON_LEFT) {
        if (action == GLFW_PRESS) {
            mousePressed = true;

            // Get cursor position
            double xpos, ypos;
            glfwGetCursorPos(window, &xpos, &ypos);

            // Convert screen coordinates to texture coordinates
            int width, height;
            glfwGetFramebufferSize(window, &width, &height);

            int texX = static_cast<int>((xpos / width) * textureWidth);
            int texY = static_cast<int>(((height - ypos) / height) * textureHeight);

            // Make sure coordinates are within texture bounds
            if (texX >= 0 && texX < textureWidth && texY >= 0 && texY < textureHeight) {
                // Get the color at the clicked position
                int index = (texY * textureWidth + texX) * 4;
                unsigned char targetR = pixelData[index];
                unsigned char targetG = pixelData[index + 1];
                unsigned char targetB = pixelData[index + 2];

                // Perform flood fill
                floodFill(texX, texY, targetR, targetG, targetB);

                // Update texture
                updateTexture();
            }
        }
        else if (action == GLFW_RELEASE) {
            mousePressed = false;
        }
    }
}

// Called when a key is pressed/released
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    if (action == GLFW_PRESS) {
        switch (key) {
        case GLFW_KEY_R:  // Red
            fillColorR = 1.0f;
            fillColorG = 0.0f;
            fillColorB = 0.0f;
            std::cout << "Fill color set to Red\n";
            break;
        case GLFW_KEY_G:  // Green
            fillColorR = 0.0f;
            fillColorG = 1.0f;
            fillColorB = 0.0f;
            std::cout << "Fill color set to Green\n";
            break;
        case GLFW_KEY_B:  // Blue
            fillColorR = 0.0f;
            fillColorG = 0.0f;
            fillColorB = 1.0f;
            std::cout << "Fill color set to Blue\n";
            break;
        case GLFW_KEY_Y:  // Yellow
            fillColorR = 1.0f;
            fillColorG = 1.0f;
            fillColorB = 0.0f;
            std::cout << "Fill color set to Yellow\n";
            break;
        case GLFW_KEY_C:  // Clear to checkerboard
            // Recreate checkerboard pattern
            for (int y = 0; y < textureHeight; y++) {
                for (int x = 0; x < textureWidth; x++) {
                    int index = (y * textureWidth + x) * 4;
                    if ((x / 40 + y / 40) % 2 == 0) {
                        pixelData[index] = 200;     // R
                        pixelData[index + 1] = 200; // G
                        pixelData[index + 2] = 200; // B
                    }
                    else {
                        pixelData[index] = 100;     // R
                        pixelData[index + 1] = 100; // G
                        pixelData[index + 2] = 100; // B
                    }
                    pixelData[index + 3] = 255;     // A
                }
            }
            updateTexture();
            std::cout << "Cleared to checkerboard pattern\n";
            break;
        }
    }
}

// Process keyboard input
void processInput(GLFWwindow* window)
{
    // Close window on ESC
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
}

// Flood fill algorithm using BFS (breadth-first search)
void floodFill(int startX, int startY, unsigned char targetR, unsigned char
targetG, unsigned char targetB)
{
    // If target color is already the fill color, do nothing
    if (colorMatch(targetR, targetG, targetB,
        static_cast<unsigned char>(fillColorR * 255),
        static_cast<unsigned char>(fillColorG * 255),
        static_cast<unsigned char>(fillColorB * 255)))
        return;

    // Create a queue for BFS
    std::queue<std::pair<int, int>> queue;

    // Push the starting pixel
    queue.push(std::make_pair(startX, startY));
    // Define 4-connected neighbors (up, right, down, left)
    const int dx[] = { 0, 1, 0, -1 };
    const int dy[] = { 1, 0, -1, 0 };

    // Process pixels until queue is empty
    while (!queue.empty()) {
        // Get the next pixel from queue
        int x = queue.front().first;
        int y = queue.front().second;
        queue.pop();

        // Check if this pixel is within bounds and has the target color
        if (x < 0 || x >= textureWidth || y < 0 || y >= textureHeight)
            continue;

        int index = (y * textureWidth + x) * 4;
        if (!colorMatch(pixelData[index], pixelData[index + 1], pixelData[index +
2],
            targetR, targetG, targetB))
            continue;

        // Change the pixel color
        pixelData[index] = static_cast<unsigned char>(fillColorR * 255);
        pixelData[index + 1] = static_cast<unsigned char>(fillColorG * 255);
        pixelData[index + 2] = static_cast<unsigned char>(fillColorB * 255);

        // Add 4-connected neighbors to the queue
        for (int i = 0; i < 4; i++) {
            int nx = x + dx[i];
            int ny = y + dy[i];
            queue.push(std::make_pair(nx, ny));
        }
    }
}

// Check if two colors are close enough (with tolerance)
bool colorMatch(unsigned char r1, unsigned char g1, unsigned char b1,
    unsigned char r2, unsigned char g2, unsigned char b2, int tolerance)
{
    return (abs(r1 - r2) <= tolerance &&
        abs(g1 - g2) <= tolerance &&
        abs(b1 - b2) <= tolerance);
```

```
}

// Update the OpenGL texture with the current pixel data
void updateTexture()
{
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, textureWidth,
textureHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, pixelData.data());
}
```

mplementation of Boundary Filling Algorithms in OpenGL:

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <iostream>
#include <vector>
#include <queue>
#include <cmath>

// Vertex shader source
const char* vertexShaderSource = R"(
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

void main()
{
    gl_Position = vec4(aPos, 0.0, 1.0);
    TexCoord = aTexCoord;
}
)";

// Fragment shader source
const char* fragmentShaderSource = R"(
#version 330 core
out vec4 FragColor;

in vec2 TexCoord;

uniform sampler2D canvasTexture;
```

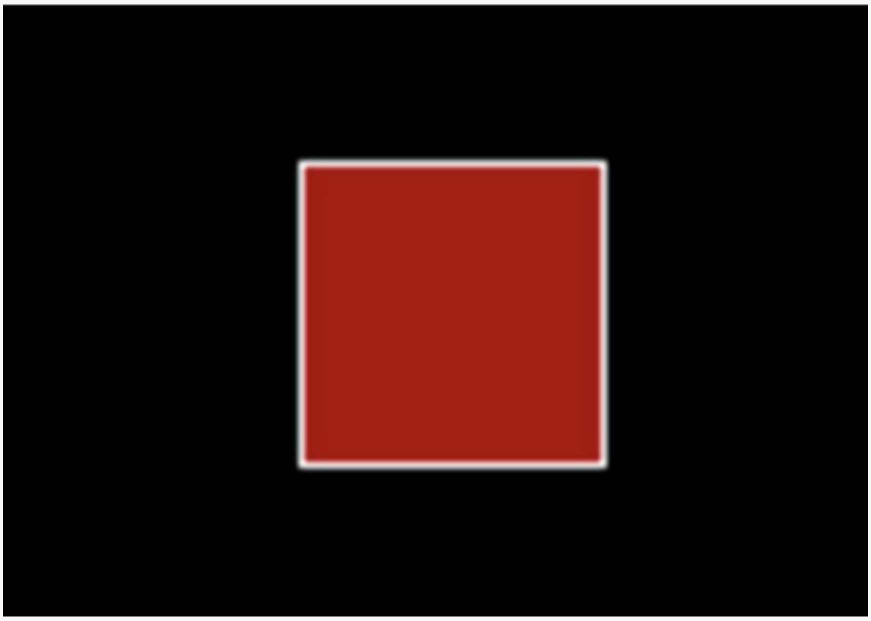
```
void main()
{
    FragColor = texture(canvasTexture, TexCoord);
}
)";

// Global variables
unsigned int canvasWidth = 800;
unsigned int canvasHeight = 600;
GLuint canvasTexture;
GLuint canvasFBO;
std::vector<unsigned char> pixelData;
bool mousePressed = false;
double mouseX, mouseY;
float fillColorR = 1.0f, fillColorG = 0.0f, fillColorB = 0.0f; // Default fill color (red)
float boundaryColorR = 0.0f, boundaryColorG = 0.0f, boundaryColorB = 0.0f; // Default boundary color (black)

// Function prototypes
void framebuffer_size_callback(GLFWwindow* window, int width, int height);
void mouse_button_callback(GLFWwindow* window, int button, int action, int mods);
void cursor_position_callback(GLFWwindow* window, double xpos, double ypos);
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods);
void boundaryFill(int x, int y, const unsigned char fillColor[3], const unsigned char boundaryColor[3]);
bool isBoundary(const unsigned char* color, const unsigned char* boundaryColor, int tolerance = 30);
bool isSameColor(const unsigned char* c1, const unsigned char* c2, int tolerance = 10);
void initCanvas();
void promptForColors();

int main()
{
    // Initialize GLFW
    if (!glfwInit())
    {
        std::cerr << "Failed to initialize GLFW" << std::endl;
        return -1;
    }

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE);

    // Create a window
    GLFWwindow* window = glfwCreateWindow(canvasWidth, canvasHeight,
"OpenGL Boundary Fill", NULL, NULL);
    if (!window)
    {
        std::cerr << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }

    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
    glfwSetMouseButtonCallback(window, mouse_button_callback);
    glfwSetCursorPosCallback(window, cursor_position_callback);
    glfwSetKeyCallback(window, key_callback);

    // Initialize GLAD
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cerr << "Failed to initialize GLAD" << std::endl;
        return -1;
    }

    // Compile and link shaders
    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    glCompileShader(vertexShader);

    GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
    glCompileShader(fragmentShader);

    GLuint shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    glLinkProgram(shaderProgram);
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);

    // Set up vertex data for a full-screen quad
    float vertices[] = {
        // positions        // texture coords
        -1.0f,  1.0f,       0.0f, 1.0f,   // top left
         1.0f,  1.0f,       1.0f, 1.0f,   // top right
         1.0f, -1.0f,       1.0f, 0.0f,   // bottom right
        -1.0f, -1.0f,       0.0f, 0.0f    // bottom left
    };

    unsigned int indices[] = {
        0, 1, 2,   // first triangle
        0, 2, 3    // second triangle
    };

    GLuint VBO, VAO, EBO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);

    // Position attribute
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float),
(void*)0);
    glEnableVertexAttribArray(0);

    // Texture coord attribute
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float),
(void*)(2 * sizeof(float)));
    glEnableVertexAttribArray(1);
    // Initialize the canvas texture and framebuffer
    initCanvas();

    // Draw an initial shape on the canvas (a rectangle in this case)
    // Boundary color is black
    const unsigned char boundaryColor[3] = { 0, 0, 0 };

    // Draw a rectangle
    for (int x = 200; x < 600; x++) {
        for (int y = 150; y < 450; y++) {
            if (x == 200 || x == 599 || y == 150 || y == 449) {
                int pixelIndex = (y * canvasWidth + x) * 4;
                pixelData[pixelIndex] = boundaryColor[0];     // R
                pixelData[pixelIndex + 1] = boundaryColor[1]; // G
                pixelData[pixelIndex + 2] = boundaryColor[2]; // B
                pixelData[pixelIndex + 3] = 255;              // A
            }
        }
    }

    // Prompt for fill and boundary colors
    promptForColors();

    // Main render loop
    while (!glfwWindowShouldClose(window))
    {
        // Update the texture with the current pixel data
        glBindTexture(GL_TEXTURE_2D, canvasTexture);
        glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, canvasWidth,
canvasHeight, GL_RGBA, GL_UNSIGNED_BYTE, pixelData.data());

        // Render the canvas
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        glUseProgram(shaderProgram);
        glBindTexture(GL_TEXTURE_2D, canvasTexture);
        glBindVertexArray(VAO);
        glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

        glfwSwapBuffers(window);
        glfwPollEvents();
    }
    // Clean up
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    glDeleteBuffers(1, &EBO);
    glDeleteProgram(shaderProgram);
    glDeleteTextures(1, &canvasTexture);
    glDeleteFramebuffers(1, &canvasFBO);

    glfwTerminate();
    return 0;
}

void initCanvas()
{
    // Initialize the pixel data (white background)
    pixelData.resize(canvasWidth * canvasHeight * 4, 255);

    // Create a texture for the canvas
    glGenTextures(1, &canvasTexture);
    glBindTexture(GL_TEXTURE_2D, canvasTexture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, canvasWidth,
canvasHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, pixelData.data());
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);

    // Create a framebuffer object for off-screen rendering
    glGenFramebuffers(1, &canvasFBO);
    glBindFramebuffer(GL_FRAMEBUFFER, canvasFBO);
    glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, canvasTexture, 0);

    // Check if framebuffer is complete
    if (glCheckFramebufferStatus(GL_FRAMEBUFFER) !=
GL_FRAMEBUFFER_COMPLETE) {
        std::cerr << "Framebuffer is not complete!" << std::endl;
    }

    // Bind back to default framebuffer
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}

void mouse_button_callback(GLFWwindow* window, int button, int action, int
mods)
{
    if (button == GLFW_MOUSE_BUTTON_LEFT && action ==
GLFW_PRESS) {
        mousePressed = true;

        // Convert screen coordinates to pixel coordinates
        int x = static_cast<int>(mouseX);
        int y = canvasHeight - static_cast<int>(mouseY) - 1; // Flip y-coordinate

        // Perform boundary fill if click is within window bounds
        if (x >= 0 && x < canvasWidth && y >= 0 && y < canvasHeight) {
            unsigned char fillColor[3] = {
                static_cast<unsigned char>(fillColorR * 255),
                static_cast<unsigned char>(fillColorG * 255),
                static_cast<unsigned char>(fillColorB * 255)
            };

            unsigned char boundaryColor[3] = {
                static_cast<unsigned char>(boundaryColorR * 255),
                static_cast<unsigned char>(boundaryColorG * 255),
                static_cast<unsigned char>(boundaryColorB * 255)
            };

            boundaryFill(x, y, fillColor, boundaryColor);
        }
    }
    else if (button == GLFW_MOUSE_BUTTON_LEFT && action ==
GLFW_RELEASE) {
        mousePressed = false;
    }
}

void cursor_position_callback(GLFWwindow* window, double xpos, double
ypos)
{
    mouseX = xpos;
    mouseY = ypos;
}

void key_callback(GLFWwindow* window, int key, int scancode, int action, int
mods)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS) {
        glfwSetWindowShouldClose(window, true);
    }
    else if (key == GLFW_KEY_C && action == GLFW_PRESS) {
        // Clear the canvas to white
        std::fill(pixelData.begin(), pixelData.end(), 255);
    }
    else if (key == GLFW_KEY_R && action == GLFW_PRESS) {
        // Reset to initial state with rectangle
        std::fill(pixelData.begin(), pixelData.end(), 255);

        // Draw a rectangle
        const unsigned char boundaryColor[3] = { 0, 0, 0 };
        for (int x = 200; x < 600; x++) {
            for (int y = 150; y < 450; y++) {
                if (x == 200 || x == 599 || y == 150 || y == 449) {
                    int pixelIndex = (y * canvasWidth + x) * 4;
                    pixelData[pixelIndex] = boundaryColor[0];     // R
                    pixelData[pixelIndex + 1] = boundaryColor[1]; // G
                    pixelData[pixelIndex + 2] = boundaryColor[2]; // B
                    pixelData[pixelIndex + 3] = 255;              // A
                }
            }
        }
    }
    else if (key == GLFW_KEY_P && action == GLFW_PRESS) {
        // Prompt for new fill and boundary colors
        promptForColors();
    }
}

void promptForColors()
{
    std::cout << "\n=== Color Settings ===" << std::endl;

    std::cout << "Enter fill color (R G B values between 0.0 and 1.0): ";
    std::cin >> fillColorR >> fillColorG >> fillColorB;

    std::cout << "Enter boundary color (R G B values between 0.0 and 1.0): ";
    std::cin >> boundaryColorR >> boundaryColorG >> boundaryColorB;

    // Validate inputs
    fillColorR = std::max(0.0f, std::min(1.0f, fillColorR));
    fillColorG = std::max(0.0f, std::min(1.0f, fillColorG));
    fillColorB = std::max(0.0f, std::min(1.0f, fillColorB));

    boundaryColorR = std::max(0.0f, std::min(1.0f, boundaryColorR));
    boundaryColorG = std::max(0.0f, std::min(1.0f, boundaryColorG));
    boundaryColorB = std::max(0.0f, std::min(1.0f, boundaryColorB));

    std::cout << "Fill color set to: (" << fillColorR << ", " << fillColorG << ", "
<< fillColorB << ")" << std::endl;
    std::cout << "Boundary color set to: (" << boundaryColorR << ", " <<
boundaryColorG << ", " << boundaryColorB << ")" << std::endl;
    std::cout << "Click inside a shape to fill it." << std::endl;
    std::cout << "Press 'C' to clear, 'R' to reset, 'P' to change colors, 'ESC' to exit."
<< std::endl;
}

void boundaryFill(int x, int y, const unsigned char fillColor[3], const unsigned
char boundaryColor[3])
{
    // Using a queue for breadth-first traversal
    std::queue<std::pair<int, int>> pixels;
    pixels.push(std::make_pair(x, y));

    while (!pixels.empty()) {
        // C++14 compatible way to get the front element
        int currentX = pixels.front().first;
        int currentY = pixels.front().second;
        pixels.pop();

        // Check if out of bounds
        if (currentX < 0 || currentX >= canvasWidth || currentY < 0 || currentY >=
canvasHeight)
            continue;

        // Get the current pixel color
        int pixelIndex = (currentY * canvasWidth + currentX) * 4;
        unsigned char currentColor[3] = {
            pixelData[pixelIndex],
            pixelData[pixelIndex + 1],
            pixelData[pixelIndex + 2]
        };

        // If pixel is a boundary or already filled, skip it
        if (isBoundary(currentColor, boundaryColor) ||
            isSameColor(currentColor, fillColor))
            continue;

        // Fill the current pixel
        pixelData[pixelIndex] = fillColor[0];
        pixelData[pixelIndex + 1] = fillColor[1];
        pixelData[pixelIndex + 2] = fillColor[2];
        pixelData[pixelIndex + 3] = 255; // Alpha

        // Add the adjacent pixels to the queue (4-connected)
        pixels.push(std::make_pair(currentX + 1, currentY));
        pixels.push(std::make_pair(currentX - 1, currentY));
        pixels.push(std::make_pair(currentX, currentY + 1));
        pixels.push(std::make_pair(currentX, currentY - 1));
    }
}

bool isBoundary(const unsigned char* color, const unsigned char*
boundaryColor, int tolerance)
{
    return std::abs(static_cast<int>(color[0]) -
static_cast<int>(boundaryColor[0])) <= tolerance &&
        std::abs(static_cast<int>(color[1]) - static_cast<int>(boundaryColor[1]))
<= tolerance &&
        std::abs(static_cast<int>(color[2]) - static_cast<int>(boundaryColor[2]))
<= tolerance;
}

bool isSameColor(const unsigned char* c1, const unsigned char* c2, int
tolerance)
{
    return std::abs(static_cast<int>(c1[0]) - static_cast<int>(c2[0])) <=
tolerance &&
        std::abs(static_cast<int>(c1[1]) - static_cast<int>(c2[1])) <= tolerance
&&
       std::abs(static_cast<int>(c1[2]) - static_cast<int>(c2[2])) <= tolerance;
}
```

Output

Before filling color:

After Color Fill:



CONCLUSION:

Hence in this lab, we were able to implement Filling Algorithms in C++ using modern OpenGL.