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 Hidden Surface Removal Algorithm in C++

Submitted by:
Name: Arun Mainali
Roll: 1307

Submitted to:
Binod Sitaula
Department of Computer Science

Submission Date: 03/21/2025

❖ LAB 9

OBJECTIVE:

Write a program in any high-level language to Hidden Surface Algorithm.

THEORY:

Hidden Surface Removal (HSR) is a technique used in computer graphics to determine which surfaces in a 3D scene should be visible to the viewer and which should be hidden or obscured by other surfaces. The primary goal of HSR is to render only the visible parts of a scene, thereby improving rendering efficiency and visual accuracy.

There are two approaches to implementing hidden surface removal:

1. **Object-Space Methods**: These algorithms work in the 3D world coordinate system and compare objects with each other to determine visibility.
2. **Image-Space Methods**: These algorithms work at the pixel level and determine visibility for each pixel on the screen.

The steps for hidden surface removal depend on the specific algorithm being used, but the general concept involves determining which surfaces or portions of surfaces are obscured from the viewer's perspective.

Z-Buffer (Depth Buffer) Algorithm:

The Z-Buffer algorithm is one of the most widely used image-space methods for hidden surface removal. It maintains a buffer (the z-buffer or depth buffer) with the same dimensions as the screen, storing the depth (z-value) of the closest pixel rendered so far at each position.

The algorithm works as follows:

- Initialize the z-buffer with the maximum depth value
- For each polygon in the scene:
    - For each pixel that the polygon covers:
        - Calculate the z-value (depth) of the polygon at that pixel
        - If this z-value is less than the value currently in the z-buffer (meaning the polygon is closer to the viewer):
            - Update the z-buffer with the new z-value
            - Update the frame buffer with the polygon's color at that pixel

Advantages and Disadvantages:

Z-Buffer Algorithm:

- **Advantages:**
    - Simple to implement
    - Works well for complex scenes

    - Hardware acceleration available in most GPUs
- **Disadvantages:**
    - Requires additional memory for the z-buffer
    - Limited precision can cause z-fighting artifacts
    - Does not handle transparency well

ALGORITHM:

A-Buffer Algorithm:

```
1. Initialize the frame buffer with background color values.
2. Initialize the z-buffer with maximum depth values.
3. For each polygon in the scene:
    1. Perform view transformation to convert to view coordinates.
    2. Apply projection transformation to get screen coordinates.
    3. For each pixel (x, y) covered by the polygon:
        1. Calculate the z-value of the polygon at pixel (x, y).
        2. If z-value < z-buffer[x, y]:
            1. Update z-buffer[x, y] = z-value.
            2. Update frame buffer[x, y] = polygon color at (x, y).
4. The frame buffer now contains the visible surface information.
```

Implementation of Flood Filling Algorithms in OpenGL:

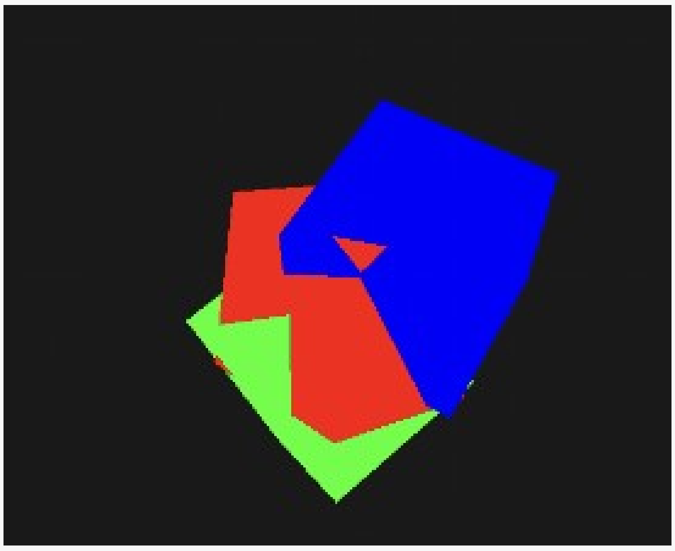
```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <iostream>
#include <vector>
#include <cmath>

// Vertex shader source
const char* vertexShaderSource = R"(
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;

out vec3 ourColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
   gl_Position = projection * view * model * vec4(aPos, 1.0);
   ourColor = aColor;
}
)";

// Fragment shader source
const char* fragmentShaderSource = R"(
#version 330 core
```

```
out vec4 FragColor;

in vec3 ourColor;

void main()
{
   FragColor = vec4(ourColor, 1.0);
}
)";

// Settings
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;

// Camera
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 5.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

// Timing
float deltaTime = 0.0f;
float lastFrame = 0.0f;

// Mouse
float yaw = -90.0f;
float pitch = 0.0f;
float lastX = SCR_WIDTH / 2.0;
float lastY = SCR_HEIGHT / 2.0;
bool firstMouse = true;

// Options
bool depthTestEnabled = true;
bool wireframeMode = false;

// Function prototypes
void framebuffer_size_callback(GLFWwindow* window, int width, int height);
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset);
void processInput(GLFWwindow* window);
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods);

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(SCR_WIDTH, SCR_HEIGHT, "Hidden Surface Removal -
OpenGL", NULL, NULL);
   if (!window)
   {
      std::cerr << "Failed to create GLFW window" << std::endl;
```

```
    glfwTerminate();
    return -1;
}

glfwMakeContextCurrent(window);
glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
glfwSetCursorPosCallback(window, mouse_callback);
glfwSetScrollCallback(window, scroll_callback);
glfwSetKeyCallback(window, key_callback);

// Tell GLFW to capture our mouse
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

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

// Enable depth test by default
glEnable(GL_DEPTH_TEST);

// 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::cerr << "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::cerr << "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::cerr << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
}
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);

// Set up vertex data for three intersecting cubes
float vertices[] = {
    // First cube positions          // colors
    -0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
     0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
     0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
     0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
    -0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
    -0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,

    -0.5f, -0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
     0.5f, -0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
    -0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
    -0.5f, -0.5f,  0.5f,  1.0f, 0.0f, 0.0f,

    -0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
    -0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
    -0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
    -0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
    -0.5f, -0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
    -0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 0.0f,

     0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
     0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
     0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
     0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
     0.5f, -0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 0.0f,

    -0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
     0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
     0.5f, -0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
     0.5f, -0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
    -0.5f, -0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
    -0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,

    -0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
     0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
    -0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 0.0f,
    -0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,

    // Second cube positions (green cube)
    -0.75f, -0.75f, -0.25f,  0.0f, 1.0f, 0.0f,
     0.25f, -0.75f, -0.25f,  0.0f, 1.0f, 0.0f,
     0.25f,  0.25f, -0.25f,  0.0f, 1.0f, 0.0f,
     0.25f,  0.25f, -0.25f,  0.0f, 1.0f, 0.0f,
    -0.75f,  0.25f, -0.25f,  0.0f, 1.0f, 0.0f,
    -0.75f, -0.75f, -0.25f,  0.0f, 1.0f, 0.0f,

    -0.75f, -0.75f,  0.75f,  0.0f, 1.0f, 0.0f,
     0.25f, -0.75f,  0.75f,  0.0f, 1.0f, 0.0f,
     0.25f,  0.25f,  0.75f,  0.0f, 1.0f, 0.0f,
     0.25f,  0.25f,  0.75f,  0.0f, 1.0f, 0.0f,
    -0.75f,  0.25f,  0.75f,  0.0f, 1.0f, 0.0f,
    -0.75f, -0.75f,  0.75f,  0.0f, 1.0f, 0.0f,

    -0.75f,  0.25f,  0.75f,  0.0f, 1.0f, 0.0f,
    -0.75f,  0.25f, -0.25f,  0.0f, 1.0f, 0.0f,
    -0.75f, -0.75f, -0.25f,  0.0f, 1.0f, 0.0f,
    -0.75f, -0.75f, -0.25f,  0.0f, 1.0f, 0.0f,
    -0.75f, -0.75f,  0.75f,  0.0f, 1.0f, 0.0f,
    -0.75f,  0.25f,  0.75f,  0.0f, 1.0f, 0.0f,

     0.25f,  0.25f,  0.75f,  0.0f, 1.0f, 0.0f,
     0.25f,  0.25f, -0.25f,  0.0f, 1.0f, 0.0f,
     0.25f, -0.75f, -0.25f,  0.0f, 1.0f, 0.0f,
     0.25f, -0.75f, -0.25f,  0.0f, 1.0f, 0.0f,
     0.25f, -0.75f,  0.75f,  0.0f, 1.0f, 0.0f,
     0.25f,  0.25f,  0.75f,  0.0f, 1.0f, 0.0f,

    -0.75f, -0.75f, -0.25f,  0.0f, 1.0f, 0.0f,
     0.25f, -0.75f, -0.25f,  0.0f, 1.0f, 0.0f,
     0.25f, -0.75f,  0.75f,  0.0f, 1.0f, 0.0f,
     0.25f, -0.75f,  0.75f,  0.0f, 1.0f, 0.0f,
    -0.75f, -0.75f,  0.75f,  0.0f, 1.0f, 0.0f,
    -0.75f, -0.75f, -0.25f,  0.0f, 1.0f, 0.0f,

    -0.75f,  0.25f, -0.25f,  0.0f, 1.0f, 0.0f,
     0.25f,  0.25f, -0.25f,  0.0f, 1.0f, 0.0f,
     0.25f,  0.25f,  0.75f,  0.0f, 1.0f, 0.0f,
     0.25f,  0.25f,  0.75f,  0.0f, 1.0f, 0.0f,
    -0.75f,  0.25f,  0.75f,  0.0f, 1.0f, 0.0f,
    -0.75f,  0.25f, -0.25f,  0.0f, 1.0f, 0.0f,

    // Third cube positions (blue cube)
    -0.25f, -0.25f, -0.75f,  0.0f, 0.0f, 1.0f,
     0.75f, -0.25f, -0.75f,  0.0f, 0.0f, 1.0f,
     0.75f,  0.75f, -0.75f,  0.0f, 0.0f, 1.0f,
     0.75f,  0.75f, -0.75f,  0.0f, 0.0f, 1.0f,
    -0.25f,  0.75f, -0.75f,  0.0f, 0.0f, 1.0f,
    -0.25f, -0.25f, -0.75f,  0.0f, 0.0f, 1.0f,

    -0.25f, -0.25f,  0.25f,  0.0f, 0.0f, 1.0f,
     0.75f, -0.25f,  0.25f,  0.0f, 0.0f, 1.0f,
     0.75f,  0.75f,  0.25f,  0.0f, 0.0f, 1.0f,
     0.75f,  0.75f,  0.25f,  0.0f, 0.0f, 1.0f,
    -0.25f,  0.75f,  0.25f,  0.0f, 0.0f, 1.0f,
    -0.25f, -0.25f,  0.25f,  0.0f, 0.0f, 1.0f,

    -0.25f,  0.75f,  0.25f,  0.0f, 0.0f, 1.0f,
    -0.25f,  0.75f, -0.75f,  0.0f, 0.0f, 1.0f,
    -0.25f, -0.25f, -0.75f,  0.0f, 0.0f, 1.0f,
    -0.25f, -0.25f, -0.75f,  0.0f, 0.0f, 1.0f,
    -0.25f, -0.25f,  0.25f,  0.0f, 0.0f, 1.0f,
    -0.25f,  0.75f,  0.25f,  0.0f, 0.0f, 1.0f,

     0.75f,  0.75f,  0.25f,  0.0f, 0.0f, 1.0f,
     0.75f,  0.75f, -0.75f,  0.0f, 0.0f, 1.0f,
     0.75f, -0.25f, -0.75f,  0.0f, 0.0f, 1.0f,
     0.75f, -0.25f, -0.75f,  0.0f, 0.0f, 1.0f,
     0.75f, -0.25f,  0.25f,  0.0f, 0.0f, 1.0f,
         0.75f,  0.75f,  0.25f,  0.0f, 0.0f, 1.0f,

        -0.25f, -0.25f, -0.75f,  0.0f, 0.0f, 1.0f,
         0.75f, -0.25f, -0.75f,  0.0f, 0.0f, 1.0f,
         0.75f, -0.25f,  0.25f,  0.0f, 0.0f, 1.0f,
         0.75f, -0.25f,  0.25f,  0.0f, 0.0f, 1.0f,
        -0.25f, -0.25f,  0.25f,  0.0f, 0.0f, 1.0f,
        -0.25f, -0.25f, -0.75f,  0.0f, 0.0f, 1.0f,

        -0.25f,  0.75f, -0.75f,  0.0f, 0.0f, 1.0f,
         0.75f,  0.75f, -0.75f,  0.0f, 0.0f, 1.0f,
         0.75f,  0.75f,  0.25f,  0.0f, 0.0f, 1.0f,
         0.75f,  0.75f,  0.25f,  0.0f, 0.0f, 1.0f,
        -0.25f,  0.75f,  0.25f,  0.0f, 0.0f, 1.0f,
        -0.25f,  0.75f, -0.75f,  0.0f, 0.0f, 1.0f
    };

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

    glBindVertexArray(VAO);

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

    // Position attribute
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    // Color attribute
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(1);

    // Initial console message
    std::cout << "\n=== Hidden Surface Removal Demo ===\n";
    std::cout << "Controls:\n";
    std::cout << "  W, A, S, D - Move camera\n";
    std::cout << "  Mouse      - Look around\n";
    std::cout << "  Z          - Toggle depth testing (currently ";
    std::cout << (depthTestEnabled ? "ON" : "OFF") << ")\n";
    std::cout << "  X          - Toggle wireframe mode (currently ";
    std::cout << (wireframeMode ? "ON" : "OFF") << ")\n";
    std::cout << "  ESC        - Exit\n\n";

    // Render loop
    while (!glfwWindowShouldClose(window))
    {
        // Per-frame time logic
        float currentFrame = static_cast<float>(glfwGetTime());
        deltaTime = currentFrame - lastFrame;
        lastFrame = currentFrame;

        // Input
        processInput(window);

        // Render
        glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
        // Clear both the color buffer and depth buffer
        if (depthTestEnabled)
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        else
            glClear(GL_COLOR_BUFFER_BIT);

        // Set wireframe mode if enabled
        if (wireframeMode)
            glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        else
            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

        // Activate shader
        glUseProgram(shaderProgram);

        // Create transformations
        glm::mat4 view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
        glm::mat4 projection = glm::perspective(glm::radians(45.0f), (float)SCR_WIDTH / (float)SCR_HEIGHT,
0.1f, 100.0f);

        // Retrieve the matrix uniform locations
        unsigned int viewLoc = glGetUniformLocation(shaderProgram, "view");
        unsigned int projLoc = glGetUniformLocation(shaderProgram, "projection");
        // Pass them to the shaders
        glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
        glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));

        // Render cubes
        glBindVertexArray(VAO);

        // Render each cube with different positions and rotations
        for (int i = 0; i < 3; i++) {
            // Calculate the model matrix for each object
            glm::mat4 model = glm::mat4(1.0f);

            // Apply rotation
            float angle = static_cast<float>(glfwGetTime()) * (i == 0 ? 50.0f : (i == 1 ? 25.0f : 75.0f));
            model = glm::rotate(model, glm::radians(angle), glm::vec3(0.5f, 1.0f, 0.0f));

            unsigned int modelLoc = glGetUniformLocation(shaderProgram, "model");
            glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

            // Draw the cube (36 vertices per cube)
            glDrawArrays(GL_TRIANGLES, i * 36, 36);
        }

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

    // De-allocate resources
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    glDeleteProgram(shaderProgram);

    glfwTerminate();
    return 0;
}

// Process all input
void processInput(GLFWwindow* window)
{
```

```cpp
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    float cameraSpeed = 2.5f * deltaTime;
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        cameraPos += cameraSpeed * cameraFront;
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        cameraPos -= cameraSpeed * cameraFront;
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
}

void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    if (key == GLFW_KEY_Z && action == GLFW_PRESS) {
        depthTestEnabled = !depthTestEnabled;
        if (depthTestEnabled) {
            glEnable(GL_DEPTH_TEST);
            std::cout << "Depth testing enabled - Hidden surfaces removed\n";
        }
        else {
            glDisable(GL_DEPTH_TEST);
            std::cout << "Depth testing disabled - All surfaces visible regardless of depth\n";
        }
    }

    if (key == GLFW_KEY_X && action == GLFW_PRESS) {
        wireframeMode = !wireframeMode;
        std::cout << "Wireframe mode " << (wireframeMode ? "enabled" : "disabled") << "\n";
    }
}

// GLFW: whenever the window size changed this callback function executes
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    // Make sure the viewport matches the new window dimensions
    glViewport(0, 0, width, height);
}

// GLFW: whenever the mouse moves, this callback is called
void mouse_callback(GLFWwindow* window, double xpos, double ypos)
{
    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos; // Reversed: y ranges bottom to top
    lastX = xpos;
    lastY = ypos;

    const float sensitivity = 0.1f;
    xoffset *= sensitivity;
    yoffset *= sensitivity;
    yaw += xoffset;
    pitch += yoffset;

    // Constrain pitch
    if (pitch > 89.0f)
        pitch = 89.0f;
    if (pitch < -89.0f)
        pitch = -89.0f;

    // Update camera front vector
    glm::vec3 front;
    front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    front.y = sin(glm::radians(pitch));
    front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
    cameraFront = glm::normalize(front);
}

// GLFW: whenever the mouse scroll wheel scrolls, this callback is called
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    // Can implement zoom functionality here if needed
}
```

OUTPUT:



CONCLUSION:

Hence in this lab, we were able to implement Hidden surface removal algorithm, especially Z-buffer Implementation in C++ using modern OpenGL.