Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Inżynierii Metali i Informatyki Przemysłowej

Sprawozdanie z Laboratorium: Ładowanie modeli z plików .obj

Przedmiot: Wizualizacja Danych

Kierunek: Inżynieria Obliczeniowa

Autor: Filip Rak

Prowadzący ćwiczenia: dr inż. Marynowski Przemysław

Data: 2 Grudnia 2024

Numer lekcji: 8

Grupa laboratoryjna: 4

Cel Ćwiczenia

Zapoznanie z zagadnieniami parsowania plików .obj.

Przebieg Ćwiczenia

Zadaniem było wczytanie dwóch modeli, krzesła i stołu, w formacie .obj, utworzonych w zewnętrznym oprogramowaniu.

Zadanie zostało zrealizowane w następujących krokach:

- Usunięcie poprzednich statycznie zadeklarowanych danych opisujących sześcian.
- Dodanie funkcji load_obj do wczytywania modeli format .obj.
- Utworzenie struktury Model reprezentującej pojedynczy model wyświetlany w scenie.
- Dodanie funkcji split_model_horizontally do dzielenia modelu na kilka mniejszych.
- Dostosowanie reszty kodu do wielu różnych modeli.

Fragment struktury Model.

```
struct Model
{
    std::string name;
    std::vector<GLfloat> vertices;
    std::vector<GLuint> indices;
    GLuint vao;                         // Vertex Array Object
    GLuint vbo;                         // Vertex Buffer Object
    GLuint ebo;                         // Element Buffer Object
    glm::mat4 model_matrix;
    glm::vec3 color;    // Model's rendering colour

    // Constructor
    // (...)

    // Destructor
    // (...)

    // Methods
    // (...)
};
```

Definicja funkcji load_obj.

```
bool load_obj(const std::string& filePath, std::vector<GLfloat>& vertices, std::vector<GLuint>& indices)
{
    std::ifstream file(filePath);
    if (!file.is_open())
    {
        std::cerr << "Error: Cannot open file " << filePath << "\n";
        return false;
    }

    std::vector<glm::vec3> temp_positions;
    std::vector<GLuint> temp_indices;

    std::string line;
    while (std::getline(file, line))
    {
        std::istringstream ss(line);
        std::string prefix;
        ss >> prefix;

        if (prefix == "v")
        {
            glm::vec3 position;
            ss >> position.x >> position.y >> position.z;
            temp_positions.push_back(position);
        }
        else if (prefix == "f")
        {
            std::string vertexStr;
            for (int i = 0; i < 3; ++i)
            {
                ss >> vertexStr;
                std::istringstream vertexSS(vertexStr);
                std::string indexStr;
                std::getline(vertexSS, indexStr, '/');
                if (indexStr.empty())
                {
                    std::cerr << "Error: Invalid face format in file " << filePath << "\n";
                    return false;
                }
                GLuint vertexIndex = std::stoi(indexStr);
                temp_indices.push_back(vertexIndex - 1);
            }
        }
    }

    // Move positions to vertices vector
    for (const auto& pos : temp_positions)
    {
        vertices.push_back(pos.x);
        vertices.push_back(pos.y);
        vertices.push_back(pos.z);
    }

    indices = temp_indices;
    file.close();
    return true;
}
```

Wyjaśnienie działania metody split_model_horizontally.

```
void split_model_horizontally(std::vector<Model*>& models, int model_id, float threshold,
GLuint shader_program, std::string base_name, std::string top_name, glm::vec3 base_color,
glm::vec3 top_color)
{
    // Skopiuj wierzchołki i indeksy ze wczytanego modelu
    // (...)

    // Podziel indeksy na górne i dolne na podstawie ich średniej wartości Y
    // (...)

    // Zmapuj wierchołki do odpowiednich indeksów
    // (...)

    // Utwórz nowe modele z uzyskanego podziału
    // (...)

    // Zamień model bazowy na model dolny w tablicy i dodaj model górny
    // (...)
}
```

Pełny kod źródłowy

```
// Headers for OpenGL and SFML
// #include "stdafx.h"  // This line might be needed in some IDEs

#pragma once
#include <GL/glew.h>
#include <SFML/Window.hpp>
#include <SFML/System/Time.hpp>
#include <glm.hpp>
#include <gtc/matrix_transform.hpp>
#include <gtc/type_ptr.hpp>
#include <iostream>
#include <time.h>
#include <string.h>
#include <cmath>
#include <fstream>
#include <sstream>
#include <vector>
#include <map>
#include <string>

// Constants
// --------------------

// Flags
const bool enable_keyboard_movement = true;
const bool enable_mouse_movement = true;

const double PI = 3.14159265358979323846;
const float WINDOW_WIDTH = 800.0f;
const float WINDOW_HEIGHT = 600.0f;

// Camera
const float MAX_CAMERA_PITCH = 89.0f;
const float MIN_CAMERA_PITCH = -89.0f;
const float MAX_CAMERA_YAW = 360.0f;
const float MIN_CAMERA_YAW = 0.0f;
const float CAMERA_BASIC_SPEED = 3.0f;
const float CAMERA_FAST_SPEED = 9.0f;

// Strings
const std::string WINDOW_TITLE = "OpenGL";
const std::string SEPARATOR = std::string(45, '-') + "\n";

// Shaders
// --------------------

// Vertex shader takes care of positioning on the screen
const GLchar* vertex_source = R"glsl(
#version 150 core

in vec3 position; // Input vertex position

// Uniforms for transformation matrices
uniform mat4 model_matrix;  // Model
uniform mat4 view_matrix;   // View (camera)
uniform mat4 proj_matrix;   // Projection

void main()
{
    // Set the position of the vertex
    gl_Position = proj_matrix * view_matrix * model_matrix * vec4(position, 1.0);
```

```
}
)glsl";

// Fragment shader's job is to figure out area between surfaces
const GLchar* fragment_source = R"glsl(
#version 150 core

uniform vec3 model_color;      // Color for the model
out vec4 outColor;             // Output color to the framebuffer

void main()
{
    outColor = vec4(model_color, 1.0);  // Set the fragment color with full opacity
}
)glsl";

// Validation functions
// ------------------
bool shader_compiled(GLuint shader, bool console_dump = true, std::string name_identifier = "")
{
    // Check for compilation error
    GLint success;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);

    if (!success && console_dump)
    {
        // Get error log length
        GLint log_length;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &log_length);

        // Allocate space for error message
        std::string error_msg(log_length, ' ');  // Initialize the string with spaces

        // Retrieve the error log
        glGetShaderInfoLog(shader, log_length, NULL, &error_msg[0]);

        // Print the error message
        std::cerr << "ERROR: " << name_identifier << " Shader Compilation Failed!:\n\t" << error_msg <<
"\n";
    }

    return success;
}

bool program_linked(GLuint program, bool console_dump = true, std::string name_identifier = "")
{
    GLint success;
    glGetProgramiv(program, GL_LINK_STATUS, &success);

    if (!success && console_dump)
    {
        // Get error log length
        GLint log_length;
        glGetProgramiv(program, GL_INFO_LOG_LENGTH, &log_length);

        // Allocate space for error message
        std::string error_msg(log_length, ' ');  // Initialize the string with spaces

        // Retrieve the error log
        glGetProgramInfoLog(program, log_length, NULL, &error_msg[0]);

        // Print the error message
        std::cerr << "ERROR: " << name_identifier << " Program Linking Failed!:\n\t" << error_msg << "\n";
```

```
    }

    return success;
}

void check_gl_error(const std::string& context)
{
    GLenum err;
    while ((err = glGetError()) != GL_NO_ERROR)
    {
        std::cerr << "OpenGL error in " << context << ": " << err << "\n";
    }
}

// Model Structure
// ------------------
struct Model
{
    std::string name;
    std::vector<GLfloat> vertices;
    std::vector<GLuint> indices;
    GLuint vao;                         // Vertex Array Object
    GLuint vbo;                         // Vertex Buffer Object
    GLuint ebo;                         // Element Buffer Object
    glm::mat4 model_matrix;
    glm::vec3 color;    // Model's rendering colour

    // Constructor
    Model(const std::string name, const std::vector<GLfloat>& verts, const std::vector<GLuint>& inds, cons
glm::vec3& col, const GLuint shader_prog)
        : name(name), vertices(verts), indices(inds), color(col), model_matrix(1.0f)
    {
        // VAO, VBO, EBO Initialization
        glGenVertexArrays(1, &vao);
        glGenBuffers(1, &vbo);
        glGenBuffers(1, &ebo);

        glBindVertexArray(vao);

        // Vertex Buffer
        glBindBuffer(GL_ARRAY_BUFFER, vbo);
        glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(GLfloat), vertices.data(), GL_STATIC_DRAW);
        check_gl_error("VBO Setup");

        // Element Buffer
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLuint), indices.data(),
GL_STATIC_DRAW);
        check_gl_error("EBO Setup");

        // Positional attribute
        GLint pos_attrib = glGetAttribLocation(shader_prog, "position");
        if (pos_attrib == -1)
        {
            std::cerr << "Attribute 'position' not found in shader.\n";
        }
        glEnableVertexAttribArray(pos_attrib);
        glVertexAttribPointer(pos_attrib, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (void*)0);
        check_gl_error("Vertex Attribute Setup");

        glBindVertexArray(0);
    }
```

```
    // Destructor
    ~Model()
    {
        glDeleteBuffers(1, &vbo);
        glDeleteBuffers(1, &ebo);
        glDeleteVertexArrays(1, &vao);
    }

    // Function to render the model
    void draw(GLuint shader_program)
    {
        // Setup model's matrix
        GLint uni_model = glGetUniformLocation(shader_program, "model_matrix");
        if (uni_model == -1)
        {
            std::cerr << "Uniform 'model_matrix' not found.\n";
        }
        glUniformMatrix4fv(uni_model, 1, GL_FALSE, glm::value_ptr(model_matrix));

        // Set up model's colour
        GLint uni_color = glGetUniformLocation(shader_program, "model_color");
        if (uni_color == -1)
        {
            std::cerr << "Uniform 'model_color' not found.\n";
        }
        glUniform3fv(uni_color, 1, glm::value_ptr(color));

        // Rendering
        glBindVertexArray(vao);
        glDrawElements(GL_TRIANGLES, static_cast<GLsizei>(indices.size()), GL_UNSIGNED_INT, 0);
        glBindVertexArray(0);
        check_gl_error("Drawing Model");
    }
};

// Shapes
// --------------------
bool load_obj(const std::string& filePath, std::vector<GLfloat>& vertices, std::vector<GLuint>& indices)
{
    std::ifstream file(filePath);
    if (!file.is_open())
    {
        std::cerr << "Error: Cannot open file " << filePath << "\n";
        return false;
    }

    std::vector<glm::vec3> temp_positions;
    std::vector<GLuint> temp_indices;

    std::string line;
    while (std::getline(file, line))
    {
        std::istringstream ss(line);
        std::string prefix;
        ss >> prefix;

        if (prefix == "v")
        {
            glm::vec3 position;
            ss >> position.x >> position.y >> position.z;
            temp_positions.push_back(position);
        }
        else if (prefix == "f")
```

```
        {
            std::string vertexStr;
            for (int i = 0; i < 3; ++i)
            {
                ss >> vertexStr;
                std::istringstream vertexSS(vertexStr);
                std::string indexStr;
                std::getline(vertexSS, indexStr, '/');
                if (indexStr.empty())
                {
                    std::cerr << "Error: Invalid face format in file " << filePath << "\n";
                    return false;
                }
                GLuint vertexIndex = std::stoi(indexStr);
                temp_indices.push_back(vertexIndex - 1);
            }
        }
    }

    // Move positions to vertices vector
    for (const auto& pos : temp_positions)
    {
        vertices.push_back(pos.x);
        vertices.push_back(pos.y);
        vertices.push_back(pos.z);
    }

    indices = temp_indices;
    file.close();
    return true;
}

void split_model_horizontally(std::vector<Model*>& models, int model_id, float threshold, GLuint
shader_program, std::string base_name, std::string top_name, glm::vec3 base_color, glm::vec3 top_color)
{
    if (models.empty())
        return;

    Model* original_model = models[model_id];

    // Vectors storing new vertices and indecies
    std::vector<GLfloat> org_vertices = original_model->vertices;
    std::vector<GLuint> org_indices = original_model->indices;

    std::vector<GLfloat> top_vertices;
    std::vector<GLuint> top_indices;
    std::vector<GLfloat> base_vertices;
    std::vector<GLuint> base_indices;

    // Vertice mapping
    std::map<GLuint, GLuint> top_vertex_mapping;
    std::map<GLuint, GLuint> base_vertex_mapping;

    GLuint current_top_index = 0;
    GLuint current_base_index = 0;

    // Iteration through indecies
    for (size_t i = 0; i < org_indices.size(); i += 3)
    {
        // Get all triangle vertices
        GLuint idx0 = org_indices[i];
        GLuint idx1 = org_indices[i + 1];
        GLuint idx2 = org_indices[i + 2];
```

```
        // Get average Y value for the triangle
        float y0 = org_vertices[idx0 * 3 + 1];
        float y1 = org_vertices[idx1 * 3 + 1];
        float y2 = org_vertices[idx2 * 3 + 1];
        float average_y = (y0 + y1 + y2) / 3.0f;

        if (average_y > threshold)
        {
            // Assign to top
            for (int j = 0; j < 3; ++j)
            {
                GLuint original_idx = org_indices[i + j];
                if (top_vertex_mapping.find(original_idx) == top_vertex_mapping.end())
                {
                    top_vertex_mapping[original_idx] = current_top_index++;
                    top_vertices.push_back(org_vertices[original_idx * 3]);
                    top_vertices.push_back(org_vertices[original_idx * 3 + 1]);
                    top_vertices.push_back(org_vertices[original_idx * 3 + 2]);
                }
                top_indices.push_back(top_vertex_mapping[original_idx]);
            }
        }
        else
        {
            // Assign to base
            for (int j = 0; j < 3; ++j)
            {
                GLuint original_idx = org_indices[i + j];
                if (base_vertex_mapping.find(original_idx) == base_vertex_mapping.end())
                {
                    base_vertex_mapping[original_idx] = current_base_index++;
                    base_vertices.push_back(org_vertices[original_idx * 3]);
                    base_vertices.push_back(org_vertices[original_idx * 3 + 1]);
                    base_vertices.push_back(org_vertices[original_idx * 3 + 2]);
                }
                base_indices.push_back(base_vertex_mapping[original_idx]);
            }
        }
    }

    // Create new models
    Model* base = new Model(base_name, base_vertices, base_indices, base_color, shader_program);
    Model* top = new Model(top_name, top_vertices, top_indices, top_color, shader_program);

    // Replace original model with new ones
    delete models[model_id];
    models[model_id] = base;
    models.push_back(top);
}
// Paths
// --------------------
const std::string ASSETS_PATH = "assets/";
const std::string MODELS_PATH = ASSETS_PATH + "models/";

// Main function
// --------------------
int main()
{
    // OpenGL's context settings
    sf::ContextSettings settings;
    settings.depthBits = 24;     // Bits for depth buffer
    settings.stencilBits = 8;    // Bits for stencil buffer
```

```
    settings.majorVersion = 3;   // OpenGL major version
    settings.minorVersion = 3;   // OpenGL minor version
    settings.attributeFlags = sf::ContextSettings::Core;

    // Create window with OpenGL context settings
    sf::Window window(sf::VideoMode(WINDOW_WIDTH, WINDOW_HEIGHT, 32), WINDOW_TITLE, sf::Style::Titlebar |
sf::Style::Close, settings);

    window.setMouseCursorGrabbed(true);
    window.setMouseCursorVisible(false);

    // Enable Z-buffer
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    // Initialize GLEW (must be done after creating the window and OpenGL context)
    glewExperimental = GL_TRUE;
    if (glewInit() != GLEW_OK)
    {
        std::cerr << "Error initializing GLEW!\n";
        return -1;
    }
    check_gl_error("GLEW Initialization");

    // Debug info of OpenGL and GPU versions
    const GLubyte* renderer = glGetString(GL_RENDERER);  // GPU name
    const GLubyte* version = glGetString(GL_VERSION);    // OpenGL version
    const GLubyte* vendor = glGetString(GL_VENDOR);      // GPU vendor
    const GLubyte* shading_version = glGetString(GL_SHADING_LANGUAGE_VERSION); // GLSL version

    std::cout << SEPARATOR;
    std::cout << "GPU: " << renderer << "\n";
    std::cout << "GPU vendor: " << vendor << "\n";
    std::cout << "OpenGL version: " << version << "\n";
    std::cout << "GLSL version: " << shading_version << "\n";

    // Create and compile the vertex shader
    GLuint vertex_shader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex_shader, 1, &vertex_source, NULL);
    glCompileShader(vertex_shader);
    check_gl_error("Vertex Shader Compilation");

    // Create and compile the fragment shader
    GLuint fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragment_shader, 1, &fragment_source, NULL);
    glCompileShader(fragment_shader);
    check_gl_error("Fragment Shader Compilation");

    // Check for shader compilation
    if (!shader_compiled(vertex_shader, true, "Vertex") || !shader_compiled(fragment_shader, true,
"Fragment"))
    {
        // Cleanup: delete shaders, buffers, and close the window
        glDeleteShader(fragment_shader);
        glDeleteShader(vertex_shader);

        window.close();  // Close the rendering window
        return -1;
    }

    // Link both shaders into a single shader program
    GLuint shader_program = glCreateProgram();
    glAttachShader(shader_program, vertex_shader);
```

```
    glAttachShader(shader_program, fragment_shader);
    glBindFragDataLocation(shader_program, 0, "outColor");  // Bind fragment output
    glLinkProgram(shader_program);

    // Check program linking
    if (!program_linked(shader_program, true, "Shader"))
    {
        // Cleanup: delete shaders, buffers, and close the window
        glDeleteProgram(shader_program);
        glDeleteShader(fragment_shader);
        glDeleteShader(vertex_shader);

        window.close();  // Close the rendering window
        return -2;
    }

    // Use shader program
    glUseProgram(shader_program);
    check_gl_error("Using Shader Program");

    // Declare and set projection matrix
    glm::mat4 proj_matrix = glm::perspective(glm::radians(45.0f), WINDOW_WIDTH / WINDOW_HEIGHT, 0.01f,
100.0f);
    GLint uni_proj = glGetUniformLocation(shader_program, "proj_matrix");
    if (uni_proj == -1)
    {
        std::cerr << "Uniform 'proj_matrix' not found.\n";
    }
    glUniformMatrix4fv(uni_proj, 1, GL_FALSE, glm::value_ptr(proj_matrix));
    check_gl_error("Setting proj_matrix");

    // Declaration and setting of view matrix
    glm::vec3 camera_pos = glm::vec3(0.0f, 0.0f, 3.0f);
    glm::vec3 camera_front = glm::vec3(0.0f, 0.0f, -1.0f);
    glm::vec3 camera_up = glm::vec3(0.0f, 1.0f, 0.f);
    glm::mat4 view_matrix = glm::lookAt(camera_pos, camera_pos + camera_front, camera_up);
    GLint uni_view = glGetUniformLocation(shader_program, "view_matrix");
    if (uni_view == -1)
    {
        std::cerr << "Uniform 'view_matrix' not found.\n";
    }
    glUniformMatrix4fv(uni_view, 1, GL_FALSE, glm::value_ptr(view_matrix));
    check_gl_error("Setting view_matrix");

    // Vector of models
    std::vector<Model*> models;

    // Models to load
    std::vector<std::string> model_files = {
        "chair.obj",
        "table.obj"
    };

    // Set colors to each model
    std::vector<glm::vec3> model_colors = {
        glm::vec3(0.2f, 0.2f, 0.8f),
        glm::vec3(1.0f, 0.0f, 0.8f)
    };

    // Loading models
    for (size_t i = 0; i < model_files.size(); ++i)
    {
        std::vector<GLfloat> vertices;
```

```
        std::vector<GLuint> indices;

        if (!load_obj(MODELS_PATH + model_files[i], vertices, indices))
        {
            std::cerr << "Failed to load model: " << model_files[i] << "\n";
            continue; // Skip this model
        }

        // Assign set color or generate random
        srand(time(NULL));
        glm::vec3 color = (i < model_colors.size()) ? model_colors[i] : glm::vec3(static_cast<float>(rand()
/ RAND_MAX, static_cast<float>(rand()) / RAND_MAX, static_cast<float>(rand()) / RAND_MAX);

        // Create the moddel and add it to the list
        Model* new_model = new Model(model_files[i], vertices, indices, color, shader_program);

        // Adjust model's positiona and rotationl properties
        if (models.size() == 0) // First model (chair)
        {
            new_model->model_matrix = glm::translate(new_model->model_matrix, glm::vec3(0.f, 0.0f, 0.0f));
        }
        else if (models.size() == 1) // Second model (table)
        {
            new_model->model_matrix = glm::translate(new_model->model_matrix, glm::vec3(-2.f, 0.0f, -3.0f))
            new_model->model_matrix = glm::rotate(new_model->model_matrix, glm::radians(90.0f),
glm::vec3(0.0f, 1.0f, 0.0f));
        }

        models.push_back(new_model);
    }

    // Split models
    glm::vec3 chair_base_color(0.8f, 0.5f, 0.2f);   // Brown
    glm::vec3 chair_top_color(0.2f, 0.2f, 0.8f);    // Blue

    glm::vec3 table_base_color(1.0f, 0.0f, 0.8f);
    glm::vec3 table_top_color(0.8f, 1.f, 0.6f);

    split_model_horizontally(models, 0, 2.f, shader_program, "chair_base", "chair_backseat",
chair_base_color, chair_top_color);
    // split_model_horizontally(models, 1, 2.f, shader_program, "table_base", "table_backseat",
table_base_color, table_top_color);

    // Debug loaded models
    std::cout << SEPARATOR;
    std::cout << "Loaded " << models.size() << " models.\n";
    for (size_t i = 0; i < models.size(); ++i)
    {
        std::cout << models[i]->name << "\n";
        std::cout << "\tvertices=" << models[i]->vertices.size() / 3 << "\n";
        std::cout << "\tindices=" << models[i]->indices.size() << "\n";
        std::cout << "\tcolour=(" << models[i]->color.r << ", " << models[i]->color.g << ", " << models[i]
>color.b << ")\n";
    }

    // Print controls
    std::cout << SEPARATOR;
    std::cout << "Controls:\n";
    std::cout << "[W, S, A, D] = Camera Position.\n";
    std::cout << "[Q, E] = Camera Rotaion Y axis.\n";
    std::cout << "[Left Shift] = speed increase.\n";
    std::cout << "[Space, Left Control] = up, down.\n";
    std::cout << "[Mouse] = Camera Rotaion XYZ Axis.\n";
```

```cpp
// Main event loop
bool running = true;
GLenum used_primitive = GL_TRIANGLES;

// Camera
float camera_yaw = 270.0f;
float camera_pitch = 0.0f;
float camera_speed = CAMERA_BASIC_SPEED;
float camera_rotation_speed = 200.0f;
bool camera_pos_changed = false;    // Remove for damping implementation

// Mouse
double mouse_sensitivity = 0.05;

// Delta time
sf::Clock delta_clock;
float delta_time = 0.0f;
float update_interval = 0.2f;    // Timer for FPS update
float time_accumulator = 0.0f; // Time passed since last FPS update
int frame_count = 0;


while (running)
{
    // Update delta time
    delta_time = delta_clock.restart().asSeconds();

    // Accumulate time and count frames
    time_accumulator += delta_time;
    frame_count++;

    // Set the window title to current FPS
    if (time_accumulator >= update_interval)
    {
        // Get FPS from average time passed since last update
        int FPS = static_cast<int>(round(frame_count / time_accumulator));
        window.setTitle(WINDOW_TITLE + " - FPS: " + std::to_string(FPS));

        // Reset for next FPS update
        time_accumulator = 0.0f;
        frame_count = 0;
    }

    sf::Event window_event;
    while (window.pollEvent(window_event))
    {
        switch (window_event.type)
        {
        case sf::Event::Closed:
            running = false;
            break;

        case sf::Event::KeyPressed:
            // Exit condition
            if (window_event.key.code == sf::Keyboard::Escape)
            {
                running = false;
            }
            break;
        case sf::Event::MouseMoved:
            if (enable_mouse_movement)
            {
```

```
                    // Get the current mouse position and calculate the offset from the center
                    sf::Vector2i center_pos(static_cast<int>(WINDOW_WIDTH / 2),
static_cast<int>(WINDOW_HEIGHT / 2));
                    sf::Vector2i local_pos = sf::Mouse::getPosition(window);
                    double x_offset = static_cast<double>(local_pos.x - center_pos.x);
                    double y_offset = static_cast<double>(local_pos.y - center_pos.y);

                    // Apply the offset to yaw and pitch
                    camera_yaw += x_offset * mouse_sensitivity;
                    camera_pitch -= y_offset * mouse_sensitivity;

                    // Clamp pitch to prevent flipping
                    if (camera_pitch > MAX_CAMERA_PITCH) camera_pitch = MAX_CAMERA_PITCH;
                    else if (camera_pitch < MIN_CAMERA_PITCH) camera_pitch = MIN_CAMERA_PITCH;

                    // Normalize yaw
                    if (camera_yaw >= MAX_CAMERA_YAW) camera_yaw -= MAX_CAMERA_YAW;
                    else if (camera_yaw < MIN_CAMERA_YAW) camera_yaw += MAX_CAMERA_YAW;

                    // Set the flag to update view matrix
                    camera_pos_changed = true;

                    // Reset mouse position to the center of the window
                    sf::Mouse::setPosition(center_pos, window);
                }

                break;
            case sf::Event::Resized:
                // Update viewport
                glViewport(0, 0, window_event.size.width, window_event.size.height);

                // Update projection matrix
                proj_matrix = glm::perspective(glm::radians(45.0f),
static_cast<float>(window_event.size.width) / window_event.size.height, 0.01f, 100.0f);
                uni_proj = glGetUniformLocation(shader_program, "proj_matrix");
                if (uni_proj == -1)
                {
                    std::cerr << "Uniform 'proj_matrix' not found.\n";
                }
                glUniformMatrix4fv(uni_proj, 1, GL_FALSE, glm::value_ptr(proj_matrix));
                check_gl_error("Resized Event");

                break;
            }
        }

        if (enable_keyboard_movement)
        {
            std::string input_debug = "Input: ";
            bool input = false;

            // Check what camera speed to use
            if (sf::Keyboard::isKeyPressed(sf::Keyboard::LShift))
            {
                camera_speed = CAMERA_FAST_SPEED;
            }
            else
            {
                camera_speed = CAMERA_BASIC_SPEED;
            }

            // Check camera movement keys in real-time
            if (sf::Keyboard::isKeyPressed(sf::Keyboard::W))   // Forward
```

```
        {
            camera_pos += camera_speed * delta_time * camera_front;
            camera_pos_changed = true;
            input_debug += "W";
            input = true;
        }

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::S))   // Backwards
        {
            camera_pos -= camera_speed * delta_time * camera_front;
            camera_pos_changed = true;
            input_debug += "S";
            input = true;
        }

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::A))   // Move left
        {
            camera_pos -= glm::normalize(glm::cross(camera_front, camera_up)) * camera_speed *
delta_time;
            camera_pos_changed = true;
            input_debug += "A";
            input = true;
        }

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::D))   // Move right
        {
            camera_pos += glm::normalize(glm::cross(camera_front, camera_up)) * camera_speed *
delta_time;
            camera_pos_changed = true;
            input_debug += "D";
            input = true;
        }

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Q))   // Rotation left
        {
            camera_yaw -= camera_rotation_speed * delta_time;
            camera_pos_changed = true;
            input_debug += "Q";
            input = true;
        }

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::E))   // Rotation right
        {
            camera_yaw += camera_rotation_speed * delta_time;
            camera_pos_changed = true;
            input_debug += "E";
            input = true;
        }

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space))   // Move up
        {
            camera_pos += glm::vec3(0.0f, 1.0f, 0.0f) * camera_speed * delta_time;
            camera_pos_changed = true;
        }

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::LControl)) // Move down
        {
            camera_pos -= glm::vec3(0.0f, 1.0f, 0.0f) * camera_speed * delta_time;
            camera_pos_changed = true;
        }

        if (input && false)
            std::cout << input_debug << "\n";
```

```
        }

        if (camera_pos_changed)
        {
            // Update view matrix
            glm::vec3 new_front;
            new_front.x = cos(glm::radians(camera_yaw)) * cos(glm::radians(camera_pitch));
            new_front.y = sin(glm::radians(camera_pitch));
            new_front.z = sin(glm::radians(camera_yaw)) * cos(glm::radians(camera_pitch));
            camera_front = glm::normalize(new_front);

            view_matrix = glm::lookAt(camera_pos, camera_pos + camera_front, camera_up);
            glUniformMatrix4fv(uni_view, 1, GL_FALSE, glm::value_ptr(view_matrix));
            check_gl_error("Updating view_matrix");

            camera_pos_changed = false;
        }

        // Clear the screen to black
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        check_gl_error("Clearing Buffers");

        // Render models
        for (auto& model : models)
        {
            model->draw(shader_program);
        }

        // Swap the front and back buffers
        window.display();
    }

    // Cleanup: delete models, shaders, buffers etc. and close the window
    for (auto& model : models)
    {
        delete model;
    }

    models.clear();

    glDeleteProgram(shader_program);
    glDeleteShader(fragment_shader);
    glDeleteShader(vertex_shader);

    window.close();  // Close the rendering window
    return 0;
}
```