Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Inżynierii Metali i Informatyki Przemysłowej

Sprawozdanie z Laboratorium: Oświetlenie sceny OpenGL z użyciem shader'ów

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

Dodanie oświetlenia do wcześniej utworzonej sceny, wykorzystując różne rodzaje świateł, takie jak światło kierunkowe, punktowe i reflektor.

Przebieg Ćwiczenia

1. Dodano deklarację zmiennych w shaderach do obsługi oświetlenia:
    - W vertex shaderze dodano zmienną do przekazywania pozycji fragmentu
    - W fragment shaderze dodano zmienne do obliczania oświetlenia, takie jak
2. Zaimplementowano obsługę trzech typów światła (kierunkowe, punktowe, reflektor) w kodzie fragment shadera.
3. Dodano klawisze do zmiany rodzaju światła oraz włączania/wyłączania oświetlenia:
    - I - światło kierunkowe.
    - O - światło punktowe.
    - P - reflektor.
    - F - włączanie/wyłączanie światła.
4. Dodano obsługę intensywności światła sterowaną za pomocą klawiszy:
    - Strzałka w górę: zwiększenie intensywności.
    - Strzałka w dół: zmniejszenie intensywności.
5. Wprowadzono zmienne light_cut_off oraz light_outer_cut_off dla obsługi kąta reflektora.
6. Dodano kalkulację oświetlenia ambientowego i rozproszonego w shaderze fragmentu.
7. Zaktualizowano funkcję main_loop, main oraz globalne zmienne, aby obsługiwały zmienne oświetlenia i uaktualniała je w shaderze w każdej klatce.

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>

#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"


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

// Flags
const bool enable_keyboard_movement = true;
const bool enable_mouse_movement = true;
const bool enable_vert_manip = false;
const bool enable_primitve_manip = true;

// Primitives
const int primitives_num = 10;
const GLenum primitives[primitives_num] =
{
    GL_POINTS,        // 0
    GL_LINES,         // 1
    GL_LINE_LOOP,     // 2
    GL_LINE_STRIP,    // 3
    GL_TRIANGLES,     // 4
    GL_TRIANGLE_STRIP,// 5
    GL_TRIANGLE_FAN,  // 6
    GL_QUADS,         // 7
    GL_QUAD_STRIP,    // 8
    GL_POLYGON        // 9
};

const std::string primitives_names[primitives_num] =
{
    "GL_POINTS",        // 0
    "GL_LINES",         // 1
    "GL_LINE_LOOP",     // 2
    "GL_LINE_STRIP",    // 3
    "GL_TRIANGLES",     // 4
    "GL_TRIANGLE_STRIP",// 5
    "GL_TRIANGLE_FAN",  // 6
    "GL_QUADS",         // 7
    "GL_QUAD_STRIP",    // 8
    "GL_POLYGON",       // 9
};

const int DATA_PER_VERT = 8;
const int OFFSET_TO_TEX = 6;
```

```
const int OFFSET_TO_NORMAL = 3;
const int TEX_COORDS = 2;
const int NORMAL_COORDS = 3;
const double PI = 3.14159265358979323846;
const float WINDOW_WIDTH = 800.0;
const float WINDOW_HEIGHT = 600.0;
const int MIN_VERTS = 1;
const int MAX_VERTS = 36;

// Camera
const float MAX_CAMERA_PITCH = 89;
const float MIN_CAMERA_PITCH = -89;
const float MAX_CAMERA_YAW = 360;
const float MIN_CAMERA_YAW = 0;

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

// File paths
const std::string ASSETS_PATH = "assets/";
const std::string TEXTURES_PATH = ASSETS_PATH + "textures/";

// 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
in vec3 color;     // Input vertex color
in vec2 in_tex_coord;   // Input texture coord
in vec3 a_normal;

out vec3 frag_pos;
out vec3 normal;
out vec2 tex_coord;   // Input texture coord
out vec3 Color;    // Output color passed to the fragment shader

// Set outside the shader
uniform mat4 model_matrix;  // Model
uniform mat4 view_matrix;   // View (camera)
uniform mat4 proj_matrix;   // Projection

void main()
{
    // Pass the color to the fragment shader
    Color = color;

    // Pass the texture coord
    tex_coord = in_tex_coord;

    // Pass normals
    normal = a_normal;

    // Pass fragment position
    frag_pos = vec3(model_matrix * vec4(position, 1.0));

    // 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

// Texture sampler
uniform sampler2D texture1;

// Lighting parameters
uniform vec3 light_pos;          // Light position for point and spotlight
uniform vec3 light_dir;          // Light direction for directional and spotlight
uniform vec3 light_color;        // Light color
uniform vec3 view_pos;           // Camera/view position
uniform float light_intensity;  // Light intensity
uniform bool light_enabled;      // Enable/disable light
uniform int light_type;          // 0 = Directional, 1 = Point, 2 = Spotlight
uniform float light_cut_off;     // Spotlight inner cutoff (cosine of angle)
uniform float light_outer_cut_off; // Spotlight outer cutoff (cosine of angle)

// Inputs from the vertex shader
in vec3 normal;                  // Interpolated normal
in vec3 frag_pos;                // Fragment position in world space
in vec3 Color;                   // Color received from the vertex shader
in vec2 tex_coord;               // Texture coordinates received from vertex shader

// Output to the framebuffer
out vec4 outColor;

void main()
{
    vec3 norm = normalize(normal); // Normalize the normal vector

    vec3 light_dir_calc;
    float attenuation = 1.0;

    // If lighting is disabled, output black
    if (!light_enabled)
    {
        outColor = vec4(0.0);
        return;
    }

    // Directional light
    if (light_type == 0)
    {
        light_dir_calc = normalize(-light_dir);
    }

    // Point light and spotlight
    else if (light_type == 1 || light_type == 2)
    {
        light_dir_calc = normalize(light_pos - frag_pos);

        // Calculate attenuation based on distance
        float distance = length(light_pos - frag_pos);
        attenuation = 1.0 / (1.0 + 0.09 * distance + 0.032 * (distance * distance));

        if (light_type == 2)
        {
            // Spotlight calculations
            float theta = dot(light_dir_calc, normalize(-light_dir));

            // Spotlight only affects fragments within the cone
```

```
        if (theta > light_cut_off)
        {
            float epsilon = light_cut_off - light_outer_cut_off;
            float intensity = clamp((theta - light_outer_cut_off) / epsilon, 0.0, 1.0);
            attenuation *= intensity;
        }
        else
        {
            attenuation = 0.0; // Disable light outside the cone
        }
    }
}

if (dot(norm, light_dir_calc) < 0.0)
{
    norm = -norm; // Reverse the normal
}

// ** Visualization for the light position **
float light_marker_distance = length(light_pos - frag_pos);
if (light_marker_distance < 0.1) // Small radius for light marker
{
    outColor = vec4(1.0, 1.0, 0.0, 1.0); // Yellow color for the light position
    return;
}

// Ambient lighting
float ambient_strength = 0.1 * light_intensity;
vec3 ambient = ambient_strength * light_color;

// Diffuse lighting
float diff = max(dot(norm, light_dir_calc), 0.0);

vec3 diffuse = diff * light_color;

// Specular lighting
float specular_strength = 0.5;
vec3 view_dir = normalize(view_pos - frag_pos);
vec3 reflect_dir = reflect(-light_dir_calc, norm);
float spec = pow(max(dot(view_dir, reflect_dir), 0.0), 32);
vec3 specular = specular_strength * spec * light_color;

// Combine lighting components
vec3 lighting = ambient + attenuation * (diffuse + specular);

// Apply texture color
vec4 tex_color = texture(texture1, tex_coord);
outColor = vec4(lighting, 1.0) * tex_color;
}
)glsl";



// Shapes
GLfloat cube_vertices[] =
{
    // Front
    // x, y, z              nx, ny, nz          u,v
    -0.5f, -0.5f, -0.5f,    0.0f, 0.0f, -1.0f,  0.0f, 0.0f,
    0.5f, -0.5f, -0.5f,     0.0f, 0.0f, -1.0f,  1.0f, 0.0f,
    0.5f, 0.5f, -0.5f,      0.0f, 0.0f, -1.0f,  1.0f, 1.0f,
    0.5f, 0.5f, -0.5f,      0.0f, 0.0f, -1.0f,  1.0f, 1.0f,
    -0.5f, 0.5f, -0.5f,     0.0f, 0.0f, -1.0f,  0.0f, 1.0f,
```

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

    // Rear
    // x, y, z              nx, ny, nz          u,v
    -0.5f, -0.5f, 0.5f,     0.0f, 0.0f, 1.0f,   0.0f, 0.0f,
    0.5f, -0.5f, 0.5f,      0.0f, 0.0f, 1.0f,   1.0f, 0.0f,
    0.5f, 0.5f, 0.5f,       0.0f, 0.0f, 1.0f,   1.0f, 1.0f,
    0.5f, 0.5f, 0.5f,       0.0f, 0.0f, 1.0f,   1.0f, 1.0f,
    -0.5f, 0.5f, 0.5f,      0.0f, 0.0f, 1.0f,   0.0f, 1.0f,
    -0.5f, -0.5f, 0.5f,     0.0f, 0.0f, 1.0f,   0.0f, 0.0f,

    // Left
    // x, y, z              nx, ny, nz          u,v
    -0.5f, 0.5f, 0.5f,      -1.0f, 0.0f, 0.0f,   0.0f, 0.0f,
    -0.5f, 0.5f, -0.5f,     -1.0f, 0.0f, 0.0f,   1.0f, 0.0f,
    -0.5f, -0.5f, -0.5f,    -1.0f, 0.0f, 0.0f,   1.0f, 1.0f,
    -0.5f, -0.5f, -0.5f,    -1.0f, 0.0f, 0.0f,   1.0f, 1.0f,
    -0.5f, -0.5f, 0.5f,     -1.0f, 0.0f, 0.0f,   0.0f, 1.0f,
    -0.5f, 0.5f, 0.5f,      -1.0f, 0.0f, 0.0f,   0.0f, 0.0f,

    // Right
    // x, y, z              nx, ny, nz          u,v
    0.5f, 0.5f, 0.5f,       1.0f, 0.0f, 0.0f,   0.0f, 0.0f,
    0.5f, 0.5f, -0.5f,      1.0f, 0.0f, 0.0f,   1.0f, 0.0f,
    0.5f, -0.5f, -0.5f,     1.0f, 0.0f, 0.0f,   1.0f, 1.0f,
    0.5f, -0.5f, -0.5f,     1.0f, 0.0f, 0.0f,   1.0f, 1.0f,
    0.5f, -0.5f, 0.5f,      1.0f, 0.0f, 0.0f,   0.0f, 1.0f,
    0.5f, 0.5f, 0.5f,       1.0f, 0.0f, 0.0f,   0.0f, 0.0f,

    // Bottom
    // x, y, z              nx, ny, nz          u,v
    -0.5f, -0.5f, -0.5f,    0.0f, -1.0f, 0.0f,   0.0f, 0.0f,
    0.5f, -0.5f, -0.5f,     0.0f, -1.0f, 0.0f,   1.0f, 0.0f,
    0.5f, -0.5f, 0.5f,      0.0f, -1.0f, 0.0f,   1.0f, 1.0f,
    0.5f, -0.5f, 0.5f,      0.0f, -1.0f, 0.0f,   1.0f, 1.0f,
    -0.5f, -0.5f, 0.5f,     0.0f, -1.0f, 0.0f,   0.0f, 1.0f,
    -0.5f, -0.5f, -0.5f,    0.0f, -1.0f, 0.0f,   0.0f, 0.0f,

    // Top
    // x, y, z              nx, ny, nz          u,v
    -0.5f, 0.5f, -0.5f,     0.0f, 1.0f, 0.0f,  0.0f, 0.0f,
    0.5f, 0.5f, -0.5f,      0.0f, 1.0f, 0.0f,  1.0f, 0.0f,
    0.5f, 0.5f, 0.5f,       0.0f, 1.0f, 0.0f,  1.0f, 1.0f,
    0.5f, 0.5f, 0.5f,       0.0f, 1.0f, 0.0f,  1.0f, 1.0f,
    -0.5f, 0.5f, 0.5f,      0.0f, 1.0f, 0.0f,  0.0f, 1.0f,
    -0.5f, 0.5f, -0.5f,     0.0f, 1.0f, 0.0f,  0.0f, 0.0f
};

// Light globals
float light_intensity = 1.0f;
bool light_enabled = true;
glm::vec3 LIGHT_POS = glm::vec3(0.0f, 0.0f, 0.0f);

enum LightType
{
    Directional = 0,
    Point = 1,
    Spotlight = 2
};


// Main loop functions
// --------------------
```

```
void find_polygon_verts(GLfloat* vertices, int vert_num, float radius)
{
    // Starting angle and change of angles between every vert
    float start_angle = 0.0f;
    float angle_step = 2.0f * PI / vert_num;

    for (int i = 0; i < vert_num; i++)
    {
        // Angle of the current vert
        float angle = start_angle + i * angle_step;

        // Vertice coordinates
        vertices[i * DATA_PER_VERT] = radius * cos(angle);  // X
        vertices[i * DATA_PER_VERT + 1] = radius * sin(angle);  // Y
        vertices[i * DATA_PER_VERT + 2] = (float)rand() / RAND_MAX;  // Z

        // Colors
        vertices[i * DATA_PER_VERT + 3] = (float)rand() / RAND_MAX; // R
        vertices[i * DATA_PER_VERT + 4] = (float)rand() / RAND_MAX; // G
        vertices[i * DATA_PER_VERT + 5] = (float)rand() / RAND_MAX; // B
    }
}

int mouse_to_verts(float mouse_pos_y)
{
    // Normalize the mouse Y position (0 at the top, 1 at the bottom)
    float normalized_mouse_y = mouse_pos_y / WINDOW_HEIGHT;

    // Invert the Y position so it progresses from bottom (0) to top (1)
    float top_down_mouse_y = 1.0f - normalized_mouse_y;

    // Calculate the number of vertices based on the mouse position within the defined vertex range
    float vertex_range = MAX_VERTS - MIN_VERTS;
    float vertex_adj = vertex_range * top_down_mouse_y;

    // Set the vertex count by adjusting based on the mouse position
    int new_vert_num = (int)(MIN_VERTS + vertex_adj);

    return new_vert_num;
}

GLfloat* update_vertices(GLfloat* vertices, int vert_num, GLuint vbo)
{
    // Reallocate memory for the new number of vertices
    delete[] vertices;
    vertices = new GLfloat[vert_num * DATA_PER_VERT];

    // Update vertices based on the new vertex count
    find_polygon_verts(vertices, vert_num, 1.0f);

    // Upload the updated vertex data to the GPU
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, vert_num * DATA_PER_VERT * sizeof(GLfloat), vertices, GL_DYNAMIC_DRAW);

    return vertices;
}

void update_view_matrix(GLuint shader_program, const glm::vec3& camera_pos, glm::vec3& camera_front, const
glm::vec3& camera_up, float camera_yaw, float camera_pitch)
{
    // Get camera front based on yaw and pitch
    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));

    // Update camera front and normalize it
    camera_front = glm::normalize(new_front);

    // Update the view matrix
    glm::mat4 view_matrix = glm::lookAt(camera_pos, camera_pos + camera_front, camera_up);
    GLint uni_view = glGetUniformLocation(shader_program, "view_matrix");
    glUniformMatrix4fv(uni_view, 1, GL_FALSE, glm::value_ptr(view_matrix));
}

void main_loop(sf::Window& window, GLuint shader_program, GLuint vao, GLuint vbo, int vert_num, GLfloat*
vertices)
{
    bool running = true;
    GLenum used_primitive = GL_TRIANGLES;

    // Camera
    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);
    float camera_yaw = 270;
    float camera_pitch = 0;
    float camera_speed = 3;
    float camera_rotation_speed = 200;
    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;
    float update_interval = 0.2;    // Timer for FPS update
    float time_accumulator = 0; // Time passed since last FPS update
    int frame_count = 0;

    // Light
    glm::vec3 LIGHT_POS = glm::vec3(0.0f, 1.5f, 0.0f); // Light position above the cube
    glm::vec3 LIGHT_DIR = glm::vec3(0.0f, -1.0f, -.8f); // Light direction
    glm::vec3 LIGHT_COLOR = glm::vec3(1.0f, 1.0f, 1.0f); // Light color (white)
    glm::vec3 CAMERA_POS = glm::vec3(0.0f, 0.0f, 3.0f);  // Camera position
    int light_type = LightType::Point; // Default to directional light

    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 = round(frame_count / time_accumulator);
            window.setTitle(WINDOW_TITLE + " - FPS: " + std::to_string(FPS));

            // Reset for next FPS update
            time_accumulator = 0;
```

```
    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;
        }

        // Vertice number manipulation
        if (enable_vert_manip)
        {
            if (window_event.key.code == sf::Keyboard::Up)
            {
                int new_vert_num = vert_num + 1;
                if (new_vert_num > MAX_VERTS)
                    new_vert_num = MAX_VERTS;

                // Avoid unneccessary updates
                if (new_vert_num == vert_num)
                    break;

                // Update vert number
                vert_num = new_vert_num;
                std::cout << "Vertices: " << vert_num << "\n";

                // Update the display
                vertices = update_vertices(vertices, vert_num, vbo);
            }

            if (window_event.key.code == sf::Keyboard::Down)
            {
                int new_vert_num = vert_num - 1;
                if (new_vert_num < MIN_VERTS)
                    new_vert_num = MIN_VERTS;

                // Avoid unneccessary updates
                if (new_vert_num == vert_num)
                    break;

                // Update vert number
                vert_num = new_vert_num;
                std::cout << "Vertices: " << vert_num << "\n";

                // Update the display
                vertices = update_vertices(vertices, vert_num, vbo);
            }
        }

        if (enable_primitve_manip)
        {
            // Primitive manipulation
```

```
                    if (window_event.key.code >= sf::Keyboard::Num0 && window_event.key.code <=
sf::Keyboard::Num9)
                    {
                        // Save numerical key as an integer
                        int pressed_number = window_event.key.code - sf::Keyboard::Num0;
                        used_primitive = primitives[pressed_number % primitives_num];
                        std::cout << "Set primitive: " << primitives_names[used_primitive % primitives_num]
"\n";
                    }
                }

                if (window_event.key.code == sf::Keyboard::Up)
                {
                    light_intensity += 0.1f;
                    if (light_intensity > 2.0f) light_intensity = 2.0f;
                    std::cout << "Light intensity: " << light_intensity << "\n";
                }
                else if (window_event.key.code == sf::Keyboard::Down)
                {
                    light_intensity -= 0.1f;
                    if (light_intensity < 0.0f) light_intensity = 0.0f;
                    std::cout << "Light intensity: " << light_intensity << "\n";
                }
                else if (window_event.key.code == sf::Keyboard::F)
                {
                    light_enabled = !light_enabled;
                    std::cout << "Light enabled: " << (light_enabled ? "ON" : "OFF") << "\n";
                }

                else if (window_event.key.code == sf::Keyboard::I)
                {
                    light_type = LightType::Directional;
                    std::cout << "Light type: Directional\n";
                }
                else if (window_event.key.code == sf::Keyboard::O)
                {
                    light_type = LightType::Point;
                    std::cout << "Light type: Point\n";
                }
                else if (window_event.key.code == sf::Keyboard::P)
                {
                    light_type = LightType::Spotlight;
                    std::cout << "Light type: Spotlight\n";
                }

                break;
            case sf::Event::MouseMoved:

                if (enable_vert_manip)
                {
                    // Convert mouse pos to vertices
                    int new_vert_num = mouse_to_verts(window_event.mouseMove.y);
                    if (new_vert_num == vert_num)   // Avoid updates if unnecessary
                        break;

                    // Update vert number
                    vert_num = new_vert_num;
                    std::cout << "Vertices: " << vert_num << "\n";

                    // Update the display
                    vertices = update_vertices(vertices, vert_num, vbo);
                }
```

```
            if (enable_mouse_movement)
            {
                // Get the current mouse position and calculate the offset from the center
                sf::Vector2i center_pos(window.getSize().x / 2, window.getSize().y / 2);
                sf::Vector2i local_pos = sf::Mouse::getPosition(window);
                double x_offset = local_pos.x - center_pos.x;
                double y_offset = 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 += MIN_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;
        }
    }

    if (enable_keyboard_movement)
    {
        std::string input = "";

        // 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 += "W";
        }

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

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

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

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

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

            // if (input.size() > 0)
               // std::cout << "Input: " << input << "\n";
        }

        if (camera_pos_changed)
        {
            update_view_matrix(shader_program, camera_pos, camera_front, camera_up, camera_yaw, camera_pitc
            camera_pos_changed = false;
        }

        // Update uniforms
        glUniform3fv(glGetUniformLocation(shader_program, "light_pos"), 1, glm::value_ptr(LIGHT_POS));
        glUniform3fv(glGetUniformLocation(shader_program, "light_dir"), 1, glm::value_ptr(LIGHT_DIR));
        glUniform3fv(glGetUniformLocation(shader_program, "light_color"), 1, glm::value_ptr(LIGHT_COLOR));
        glUniform3fv(glGetUniformLocation(shader_program, "view_pos"), 1, glm::value_ptr(CAMERA_POS));
        glUniform1i(glGetUniformLocation(shader_program, "light_type"), light_type);
        glUniform1i(glGetUniformLocation(shader_program, "light_enabled"), light_enabled);
        glUniform1f(glGetUniformLocation(shader_program, "light_intensity"), light_intensity);
        glUniform1f(glGetUniformLocation(shader_program, "light_cut_off"), glm::cos(glm::radians(12.5f * 2)
        glUniform1f(glGetUniformLocation(shader_program, "light_outer_cut_off"), glm::cos(glm::radians(17.5
2)));

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

        // Draw the shape
        glDrawArrays(used_primitive, 0, vert_num);

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

// 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

        // Retreive 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;
}

// Main function
// --------------------
int main()
{
    // Init for radom number generation
    srand(time(NULL));

    // Setup OpenGL context settings
    sf::ContextSettings settings;
    settings.depthBits = 24;     // Bits for depth buffer
    settings.stencilBits = 8;    // Bits for stencil buffer

    // Create a rendering window with OpenGL context
    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);

    // window.setFramerateLimit(20);

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

    // Initialize GLEW (must be done after creating the window and OpenGL context)
```

```
    glewExperimental = GL_TRUE;
    glewInit();

    // Create and bind a Vertex Array Object (VAO) to store vertex state
    GLuint vao;
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    // Create a Vertex Buffer Object (VBO) and upload vertex data to it
    GLuint vbo;
    glGenBuffers(1, &vbo);

    // Vertex data: positions (x, y) and colors (r, g, b) for each vertex
    int vert_num = 3;
    GLfloat* vertices = new GLfloat[vert_num * DATA_PER_VERT];

    // Generate a polygon
    // find_polygon_verts(vertices, vert_num, 1.0f);

    // Generate a cube
    vert_num = 36;
    vertices = cube_vertices;

    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, vert_num * DATA_PER_VERT * sizeof(GLfloat), vertices, GL_STATIC_DRAW);

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

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

    // Check for shader compilation
    if (!shader_compiled(vertex_shader, true, "Vertex") || !shader_compiled(fragment_shader, true, "Fragmen
    {
        // Cleanup: delete shaders, buffers, and close the window
        glDeleteShader(fragment_shader);
        glDeleteShader(vertex_shader);
        glDeleteBuffers(1, &vbo);
        glDeleteVertexArrays(1, &vao);

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

    // Declare shader uniform data
    glm::mat4 model_matrix = glm::mat4(1.0f);
    model_matrix = glm::rotate(model_matrix, glm::radians(45.0f), glm::vec3(0.0f, 0.0f, 1.0f));
    glm::mat4 view_matrix = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f), glm::vec3(0.0f, 0.0f, 0.0f),
glm::vec3(0.0f, 1.0f, 0.0f));
    glm::mat4 proj_matrix = glm::perspective(glm::radians(45.0f), WINDOW_WIDTH / WINDOW_HEIGHT, 0.01f, 100.

    // 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);
```

```
    // Use the program if linking succeeded
    if (program_linked(shader_program, true, "Shader"))
    {
        // Debug info
        std::cout << SEPARATOR;
        std::cout << "Version:\t" << glGetString(GL_VERSION) << "\n";
        std::cout << "Running on:\t" << glGetString(GL_RENDERER) << "\n";
        std::cout << SEPARATOR;

        // Use the program
        glUseProgram(shader_program);

        // Add uniform data
        GLint uni_trans = glGetUniformLocation(shader_program, "model_matrix");
        glUniformMatrix4fv(uni_trans, 1, GL_FALSE, glm::value_ptr(model_matrix));

        GLint uni_view = glGetUniformLocation(shader_program, "view_matrix");
        glUniformMatrix4fv(uni_view, 1, GL_FALSE, glm::value_ptr(view_matrix));

        GLint uni_proj = glGetUniformLocation(shader_program, "proj_matrix");
        glUniformMatrix4fv(uni_proj, 1, GL_FALSE, glm::value_ptr(proj_matrix));

        GLint texture_location = glGetUniformLocation(shader_program, "texture1");
        glUniform1i(texture_location, 0); // Bind texture unit 0 to "texture1"

        // Add the texture
        GLint tex_coord = glGetAttribLocation(shader_program, "in_tex_coord");
        glEnableVertexAttribArray(tex_coord);
        glVertexAttribPointer(tex_coord, TEX_COORDS, GL_FLOAT, GL_FALSE, DATA_PER_VERT * sizeof(GLfloat),
(void*)(OFFSET_TO_TEX * sizeof(GLfloat)));

        // Add light
        GLint normal_attrib = glGetAttribLocation(shader_program, "a_normal");
        glEnableVertexAttribArray(normal_attrib);
        glVertexAttribPointer(normal_attrib, NORMAL_COORDS, GL_FLOAT, GL_FALSE, DATA_PER_VERT *
sizeof(GLfloat), (void*)(sizeof(GLfloat) * OFFSET_TO_NORMAL));

        GLint uni_light_pos = glGetUniformLocation(shader_program, "light_pos");
        glUniform3fv(uni_light_pos, 1, &LIGHT_POS[0]);

        GLint light_intensity_loc = glGetUniformLocation(shader_program, "light_intensity");
        glUniform1f(light_intensity_loc, light_intensity);

        GLint light_enabled_loc = glGetUniformLocation(shader_program, "light_enabled");
        glUniform1i(light_enabled_loc, light_enabled);
    }
    else
    {
        // Cleanup: delete shaders, buffers, and close the window
        glDeleteProgram(shader_program);
        glDeleteShader(fragment_shader);
        glDeleteShader(vertex_shader);
        glDeleteBuffers(1, &vbo);
        glDeleteVertexArrays(1, &vao);

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


    // Specify the layout of the vertex data
    GLint pos_attrib = glGetAttribLocation(shader_program, "position");
    glEnableVertexAttribArray(pos_attrib);
```

```
    glVertexAttribPointer(pos_attrib, 3, GL_FLOAT, GL_FALSE, DATA_PER_VERT * sizeof(GLfloat), 0);

    GLint col_attrib = glGetAttribLocation(shader_program, "color");
    glEnableVertexAttribArray(col_attrib);
    glVertexAttribPointer(col_attrib, 3, GL_FLOAT, GL_FALSE, DATA_PER_VERT * sizeof(GLfloat), (void*)(3 *
sizeof(GLfloat)));

    // Create texture
    unsigned int texture1;  // ID
    glGenTextures(1, &texture1);    // Generation
    glBindTexture(GL_TEXTURE_2D, texture1); // Binding

    // Setting texture parameters
    // Wrap-around
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);   // X axis
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);   // Y axis

    // Filtering
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);   // When texture is downsized
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);   // When texture is magnifieid

    int texture_width, texture_height, texture_channels;

    // Flip image on y-axis on load
    stbi_set_flip_vertically_on_load(true);

    // Load the texture
    std::string file_path = TEXTURES_PATH + "obanma.jpeg";
    unsigned char* data = stbi_load(file_path.c_str(), &texture_width, &texture_height, &texture_channels,
STBI_rgb);
    if (data)
    {

        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texture_width, texture_height, 0, GL_RGB, GL_UNSIGNED_BYTE,
data);
        glGenerateMipmap(GL_TEXTURE_2D);
    }
    else
        std::cout << "Failed to load texture!\n";

    stbi_image_free(data);


    // Main event loop
    main_loop(window, shader_program, vao, vbo, vert_num, vertices);

    // Cleanup: delete shaders, buffers, and close the window
    glDeleteProgram(shader_program);
    glDeleteShader(fragment_shader);
    glDeleteShader(vertex_shader);
    glDeleteBuffers(1, &vbo);
    glDeleteVertexArrays(1, &vao);

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