

Міністерство освіти і науки України
Національний університет «Запорізька політехніка»

кафедра програмних засобів

ЗВІТ

з лабораторної роботи № 7

з дисципліни «Комп'ютерна графіка та обробка зображень» на тему:
«НАВКОЛИШНЄ, ДИФУЗНЕ ТА ДЗЕРКАЛЬНЕ ОСВІТЛЕННЯ»

Виконав:

ст. гр. КНТ-113сп

Іван ЩЕДРОВСЬКИЙ

Прийняв:

Асистент

Артем ТУЛЕНКОВ

1 Мета роботи

Отримати практичні навички роботи з бібліотекою OpenGL, використовуючи мову програмування C++. Навчитися малювати 3D об'єкти, виконувати базові маніпуляції з ними, створювати вікна, керувати камерою, працювати з освітленням, створювати прості шейдери та зрозуміти як працюють бібліотеки GLEW, GLFW, GLM.

2 Завдання до лабораторної роботи

2.1 Змініть фігуру.

2.2 Змініть текстуру.

2.3 Змінити тип освітлення

3 Виконання лабораторної роботи

Для виконання лабораторної роботи було використано Visual Studio 17 2022

Лабораторна робота виконувалась, в тому числі, з взаємодією з книгою “Learn OpenGL – Graphics Programming” від Joey de Vries 2020 року, яку можна безкоштовно знайти на сайті learnopengl.com.

Для зручної роботи з нормлями було змінено фігуру з двох тривимірних трикутників на один куб. Це потрібно для зручної роботи з нормлями, оскільки їх розрахунок є досить складною операцією

Для виконання роботи було створено джерело світла, яке також представляє собою куб. Логіка створення, а також vertex й fragment шейдери для джерела світла показані на рисунках 1, 2 та 3.

```

GLuint initLightVAO(float vertices[], size_t verticesSize) {
    GLuint lightVAO;
    glGenVertexArrays(1, &lightVAO);
    glBindVertexArray(lightVAO);

    GLuint VBO;
    glGenBuffers(1, &VBO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, verticesSize, vertices, GL_STATIC_DRAW);

    // Position
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);

    return lightVAO;
}

```

Рисунок 1 – Логіка створення джерела освітлення

```

#version 330 core
layout (location = 0) in vec3 aPos;

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

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

```

Рисунок 2 – Vertex шейдер для джерела освітлення

```

#version 330 core
out vec4 FragColor;

void main() {
    FragColor = vec4(1.0);
}

```

Рисунок 3 – Fragment шейдер для джерела освітлення

Для реалізації різних видів освітлення було модифіковано vertex та fragment шейдери. Vertex тепер також передає координати фрагмента та нормаль, а fragment розраховує навколишнє, дифузне та дзеркальне освітлення. Це показано на рисунках 4 та 5.

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTextureCoord;

out vec2 TextureCoord;
out vec3 FragmentPos;
out vec3 Normal;

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

void main() {
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    FragmentPos = vec3(model * vec4(aPos, 1.0));
    Normal = mat3(transpose(inverse(model))) * aNormal;

    TextureCoord = aTextureCoord;
}
```

Рисунок 4 – Оновлений vertex шейдер

```

uniform vec3 viewPos;
uniform vec3 lightPos;
uniform vec3 lightColor;

float specularStrength = 0.5;
float ambientStrenght = 0.1;

✓ void main() {
    vec3 objectColor = vec3(texture(ourTexture, TextureCoord));

    vec3 ambient = ambientStrenght * lightColor;

    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragmentPos);

    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    vec3 viewDir = normalize(viewPos - FragmentPos);
    vec3 reflectDir = reflect(-lightDir, norm);

    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
    vec3 specular = specularStrength * spec * lightColor;

    vec3 result = (ambient + diffuse + specular) * objectColor;
    FragColor = vec4(result, 1.0);
}

```

Рисунок 5 – Оновлений fragment шейдер

Також було модифіковано код рендер-циклу, тепер він передає потрібні параметри в шейдери. Це показано на рисунку 6.

```

glm::mat4 view = glm::mat4(1.0f);
glm::mat4 projection = glm::mat4(1.0f);

view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);

projection = glm::perspective(glm::radians(fov), (float)WIDTH / (float)
    HEIGHT, 0.1f, 100.0f);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);

cubeShader.use();
cubeShader.setInt("ourTexture", 0);
cubeShader.setVec3("lightPos", lightPos);
cubeShader.setVec3("viewPos", cameraPos);
cubeShader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);

cubeShader.setVec2("u_resolution", WIDTH, HEIGHT);
cubeShader.setMat4("model", model);
cubeShader.setMat4("view", view);
cubeShader.setMat4("projection", projection);
glBindVertexArray(VAO2);
glDrawArrays(GL_TRIANGLES, 0, 36);

lightSource.use();
model = glm::mat4(1.0f);
model = glm::translate(model, lightPos);
model = glm::scale(model, glm::vec3(0.2f));

lightSource.setMat4("projection", projection);
lightSource.setMat4("view", view);
lightSource.setMat4("model", model);

glBindVertexArray(lightVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);

// Swap Buffers (front + back)
}

```

Рисунок 6 – Оновлена логіка в рендер-циклі

Приклади готової програми показані на рисунках 7, 8, 9 та 10



Рисунок 7 – Базовий приклад виконання програми

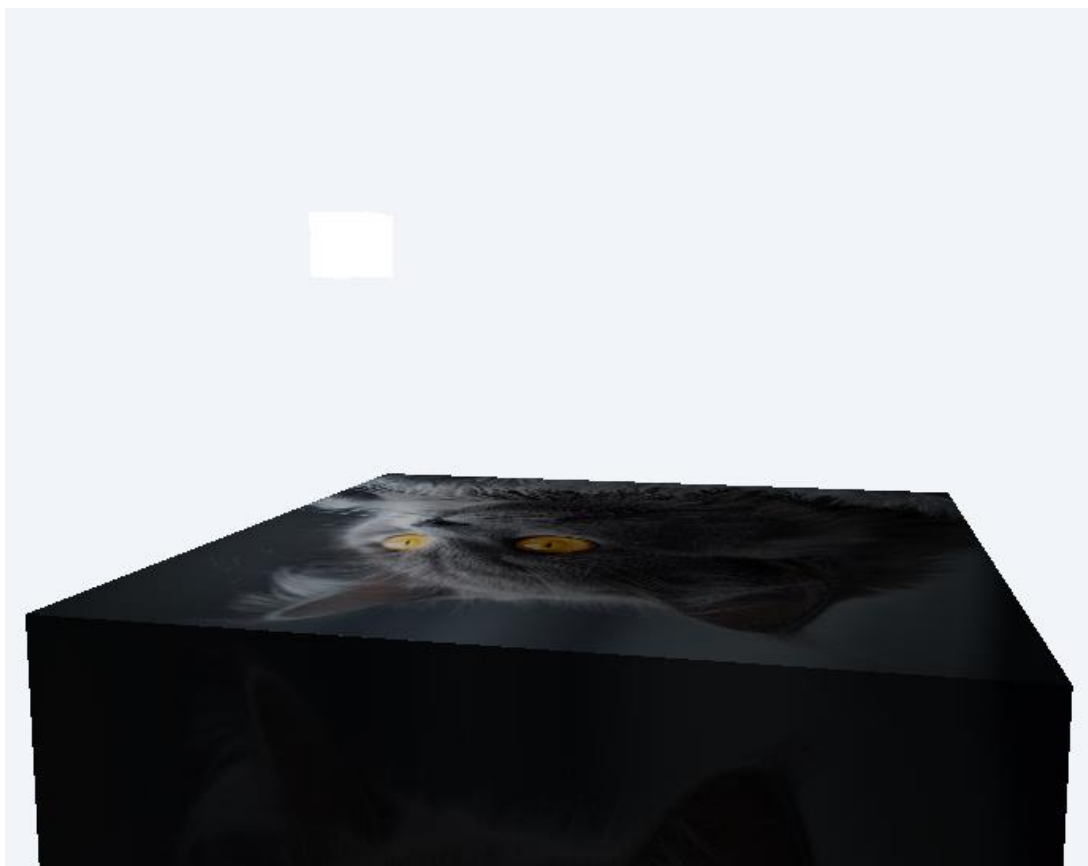


Рисунок 8 – Приклад роботи дзеркального освітлення

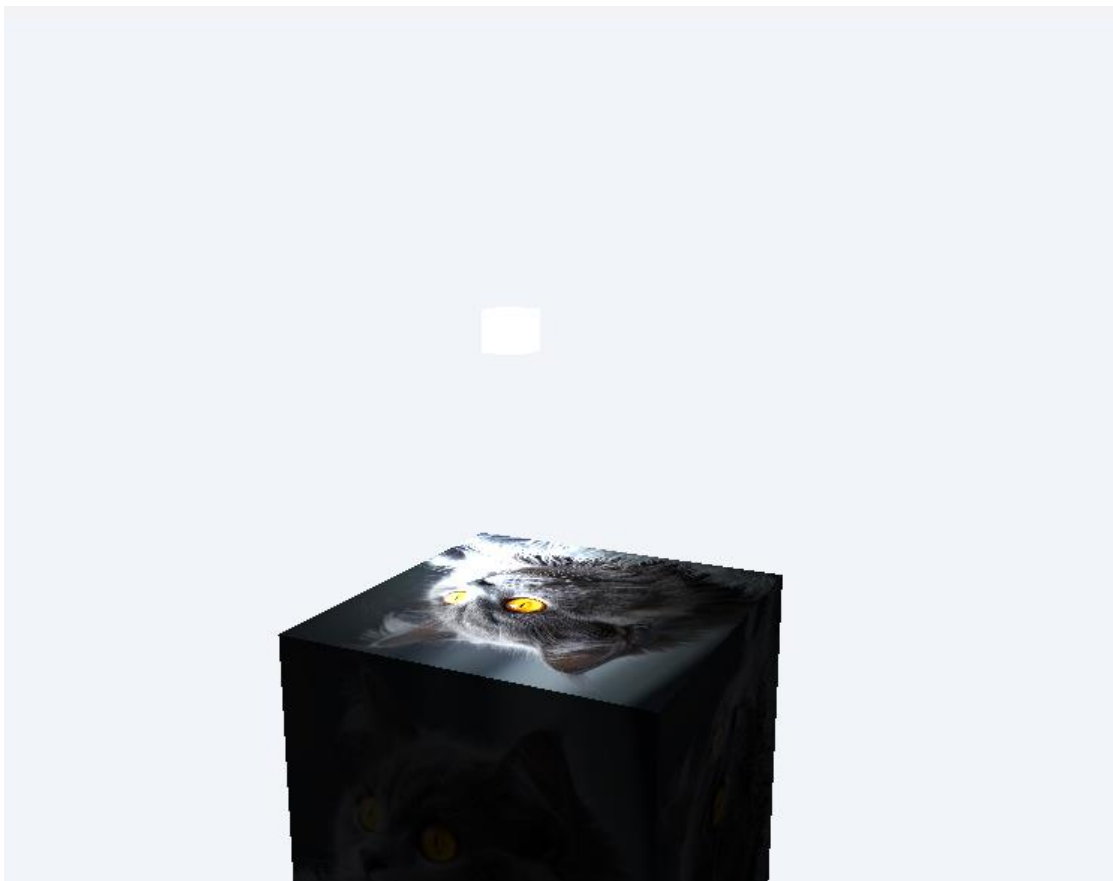


Рисунок 9 – Приклад роботи дзеркального освітлення зі зміненою силою



Рисунок 10 – Приклад роботи навколишнього освітлення з збільшеною силою

4 Тест розробленої програми

```
+ shader.hpp
#ifndef SHADER_H
#define SHADER_H

#include <string>
#include <glad/glad.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

/*!
 * OpenGL Shader program wrapper for work with GLSL shader files
 */
class Shader {
public:
    /*!
     * Create Shader program from two shader-files
     *
     * @param vertexPath - Path to vertex shader file in file system
     * @param fragmentPath - Path to vertex shader file in file system
     */
    Shader(const char* vertexPath, const char* fragmentPath);
```

```

    /*!
    * Use current shader
    */
    void use();

    /*!
    * Utils for set `uniform` variables in shaders
    */
    void setVec2(const std::string& name, float x, float y) const;
    void setVec3(const std::string& name, float x, float y, float z) const;
    void setVec3(const std::string& name, glm::vec3 vector) const;
    void setMat4(const std::string& name, glm::mat4 matrix) const;
    void setInt(const std::string& name, int number) const;
private:
    GLuint ID;

    enum Error {
        PROGRAM,
        VERTEX,
        FRAGMENT
    };

    const std::string readShaderFile(std::string path);
    void checkOnErrors(GLuint shader, Shader::Error type);
};
#endif

+ shader.cpp
#include "shader.hpp"
#include <glad/glad.h>

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

Shader::Shader(const char* vertexPath, const char* fragmentPath) {
    const std::string vertexCodeRaw = Shader::readShaderFile(vertexPath);
    const std::string fragmentCodeRaw = Shader::readShaderFile(fragmentPath);

    const char* vertexCode = vertexCodeRaw.c_str();
    const char* fragmentCode = fragmentCodeRaw.c_str();

    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexCode, NULL);
    glCompileShader(vertexShader);
    Shader::checkOnErrors(vertexShader, Shader::Error::VERTEX);

    GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &fragmentCode, NULL);
    glCompileShader(fragmentShader);
    Shader::checkOnErrors(fragmentShader, Shader::Error::FRAGMENT);

    Shader::ID = glCreateProgram();

    glAttachShader(Shader::ID, vertexShader);
    glAttachShader(Shader::ID, fragmentShader);
    glLinkProgram(Shader::ID);
    Shader::checkOnErrors(Shader::ID, Shader::Error::PROGRAM);

    glDeleteShader(vertexShader);

```

```

        glDeleteShader(fragmentShader);
    }

    void Shader::use() {
        glUseProgram(Shader::ID);
    }

    // TODO: Add GLM version?

    void Shader::setVec2(const std::string& name, float x, float y) const {
        glUniform2f(glGetUniformLocation(Shader::ID, name.c_str()), x, y);
    }

    void Shader::setVec3(const std::string& name, float x, float y, float z) const {
        glUniform3f(glGetUniformLocation(Shader::ID, name.c_str()), x, y, z);
    }

    void Shader::setVec3(const std::string& name, glm::vec3 vector) const {
        glUniform3f(glGetUniformLocation(Shader::ID, name.c_str()), vector.x, vector.y,
vector.z);
    }

    void Shader::setMat4(const std::string& name, glm::mat4 matrix) const {
        glUniformMatrix4fv(glGetUniformLocation(Shader::ID, name.c_str()), 1, GL_FALSE,
glm::value_ptr(matrix));
    }

    void Shader::setInt(const std::string& name, int number) const {
        glUniform1i(glGetUniformLocation(Shader::ID, name.c_str()), number);
    }

    const std::string Shader::readShaderFile(std::string path) {
        std::string code;
        std::ifstream file;

        file.exceptions(std::ifstream::failbit | std::ifstream::badbit);

        try {
            file.open(path);

            std::stringstream stream;
            stream << file.rdbuf();

            file.close();

            code = stream.str();
        } catch (std::ifstream::failure e) {
            std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" << std::endl;
        }

        return code;
    }

    void Shader::checkOnErrors(GLuint shader, Shader::Error type) {
        GLint success;
        GLchar infoLog[1024];

        if (type == Shader::Error::PROGRAM) {
            glGetProgramiv(shader, GL_LINK_STATUS, &success);

            if (!success) {
                glGetProgramInfoLog(shader, 512, NULL, infoLog);

                std::cout << "ERROR::SHADER::LINK_FAILED\n" <<
                    infoLog << std::endl;
            }
        }
    }

```

```

    }

    return;
}

glGetShaderiv(shader, GL_COMPILE_STATUS, &success);

if (!success) {
    glGetShaderInfoLog(shader, 512, NULL, infoLog);

    std::cout << "ERROR::SHADER::SHADER_COMPILATION_ERROR in type:" << type <<
"\n" <<
        infoLog << std::endl;
}
}

```

```

+ main.cpp
#include <glad/glad.h>
#include <glfw/glfw3.h>
#include <iostream>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include "shader.hpp"

#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"

const unsigned int WIDTH = 800;
const unsigned int HEIGHT = 600;

void framebuffer_size_callback(GLFWwindow* window, int width, int height);
void processInput(GLFWwindow* window);
GLuint initVAO(float vertices[], size_t verticesSize);
GLuint initTexture();
GLuint initLightVAO(float vertices[], size_t verticesSize);
void mouse_callback(GLFWwindow* window, double xPos, double yPos);
void scroll_callback(GLFWwindow* window, double xOffset, double yOffset);

glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

float deltaTime = 0.0f; // Time between current frame and last frame
float lastFrame = 0.0f; // Time of last frame

float lastX = WIDTH / 2;
float lastY = HEIGHT / 2;

// Camera up/down
float pitch = 0.0f;
// Camera right/left
float yaw = -90.0f;

bool firstMouse = true;
float fov = 45.0f;

glm::vec3 lightPos(1.2f, 1.0f, 2.0f);

int main() {
    if (!glfwInit()) {
        std::cout << "GLFW failed to start" << std::endl;
    }
}

```

```

        glfwTerminate();
        return -1;
    }

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

    GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "LearnOpenGL", NULL, NULL);

    if (window == NULL) {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }

    glfwMakeContextCurrent(window);

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

    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
    glfwSetScrollCallback(window, scroll_callback);
    glfwSetCursorPosCallback(window, mouse_callback);
    glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
    glEnable(GL_DEPTH_TEST);

    float vertices[] = {
        // positions            normals            texture coords
        -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,

        -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,

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

        -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,    1.0f, 1.0f,
        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, 0.0f,
        -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,    0.0f, 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,  1.0f, 1.0f,
        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, 0.0f,
        -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,  0.0f, 1.0f
};

Shader cubeShader("base.vert", "base.frag");
Shader lightSource("lightSource.vert", "lightSource.frag");

GLuint VA01 = initVAO(vertices, sizeof(vertices));
GLuint VA02 = initVAO(vertices, sizeof(vertices));
GLuint lightVAO = initLightVAO(vertices, sizeof(vertices));

GLuint texture = initTexture();

while (!glfwWindowShouldClose(window)) {
    processInput(window);

    glClearColor(0.945f, 0.961f, 0.976f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    float currentFrame = glfwGetTime();
    deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;

    glm::mat4 model = glm::mat4(1.0f);
    glm::mat4 view = glm::mat4(1.0f);
    glm::mat4 projection = glm::mat4(1.0f);

    view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);

    projection = glm::perspective(glm::radians(fov), (float)WIDTH / (float)HEIGHT,
0.1f, 100.0f);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture);

    cubeShader.use();
    cubeShader.setInt("ourTexture", 0);
    cubeShader.setVec3("lightPos", lightPos);
    cubeShader.setVec3("viewPos", cameraPos);
    cubeShader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);

    cubeShader.setVec2("u_resolution", WIDTH, HEIGHT);
    cubeShader.setMat4("model", model);
    cubeShader.setMat4("view", view);
    cubeShader.setMat4("projection", projection);
    glBindVertexArray(VA02);
    glDrawArrays(GL_TRIANGLES, 0, 36);

    lightSource.use();
    model = glm::mat4(1.0f);
    model = glm::translate(model, lightPos);
    model = glm::scale(model, glm::vec3(0.2f));

    lightSource.setMat4("projection", projection);
    lightSource.setMat4("view", view);
    lightSource.setMat4("model", model);

    glBindVertexArray(lightVAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);
}

```

```

        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwTerminate();
    return 0;
}

GLuint initLightVAO(float vertices[], size_t verticesSize) {
    GLuint lightVAO;
    glGenVertexArrays(1, &lightVAO);
    glBindVertexArray(lightVAO);

    GLuint VBO;
    glGenBuffers(1, &VBO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, verticesSize, vertices, GL_STATIC_DRAW);

    // Position
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);

    return lightVAO;
}

/*!
 * Create Vertex Array Object from all vertices and indices
 */
GLuint initVAO(float vertices[], size_t verticesSize) {
    GLuint VAO;
    glGenVertexArrays(1, &VAO);
    glBindVertexArray(VAO);

    GLuint VBO;
    glGenBuffers(1, &VBO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, verticesSize, vertices, GL_STATIC_DRAW);

    // Position
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);

    // Normal
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3 *
sizeof(float)));
    glEnableVertexAttribArray(1);

    // Texture
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 *
sizeof(float)));
    glEnableVertexAttribArray(2);

    return VAO;
}

GLuint initTexture() {
    GLuint texture;
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
}

```

```

    stbi_set_flip_vertically_on_load(true);

    int width, height, nrChannels;
    unsigned char* data = stbi_load("cat.jpg", &width, &height, &nrChannels, 0);

    if (data) {
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);
    } else {
        std::cout << "Failed to load texture" << std::endl;
    }

    stbi_image_free(data);

    return texture;
}

/*!
 * Process interaction with user
 */
void processInput(GLFWwindow* window) {
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS) {
        glfwSetWindowShouldClose(window, true);
    }

    const 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;
    }

    // Rotate around center
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS || glfwGetKey(window, GLFW_KEY_D) ==
GLFW_PRESS) {
        float difference = 1.0f;

        if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS) {
            difference *= -1;
        }

        glm::mat4 rotationMatrix(1.0f);
        rotationMatrix = glm::rotate(rotationMatrix, glm::radians(difference),
cameraUp);
        glm::vec4 rotatedRelativePos = rotationMatrix * glm::vec4(cameraPos, 1.0f);
        cameraPos = glm::vec3(rotatedRelativePos);

        yaw += difference * -1;

        glm::vec3 direction;

        direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
        direction.y = sin(glm::radians(pitch));
        direction.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));

        cameraFront = glm::normalize(direction);
    }
}

void mouse_callback(GLFWwindow* window, double xPos, double yPos) {
    if (firstMouse) {

```

```

        lastX = xPos;
        lastY = yPos;
        firstMouse = false;
    }

    const float sensitivity = 0.1f;

    float xOffset = xPos - lastX;
    // reversed, y ranges bottom to top
    float yOffset = lastY - yPos;

    lastX = xPos;
    lastY = yPos;

    xOffset *= sensitivity;
    yOffset *= sensitivity;

    yaw += xOffset;
    pitch += yOffset;

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

    glm::vec3 direction;

    direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    direction.y = sin(glm::radians(pitch));
    direction.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));

    cameraFront = glm::normalize(direction);
}

void scroll_callback(GLFWwindow* window, double xOffset, double yOffset) {
    fov -= (float)yOffset;

    if (fov < 1.0f) fov = 1.0f;
    if (fov > 45.0f) fov = 45.0f;
}

/*!
 * Auto update OpenGL viewport on resize
 */
void framebuffer_size_callback(GLFWwindow* window, int width, int height) {
    glViewport(0, 0, width, height);
}

+ base.vert
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTextureCoord;

out vec2 TextureCoord;
out vec3 FragmentPos;
out vec3 Normal;

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

void main() {
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    FragmentPos = vec3(model * vec4(aPos, 1.0));
    Normal = mat3(transpose(inverse(model))) * aNormal;
}

```

```

        TextureCoord = aTextureCoord;
    }

+ base.frag
#version 330 core
out vec4 FragColor;

in vec3 outColor;
in vec2 TextureCoord;
in vec3 FragmentPos;
in vec3 Normal;

uniform sampler2D ourTexture;
uniform vec3 viewPos;
uniform vec3 lightPos;
uniform vec3 lightColor;

float specularStrength = 0.5;
float ambientStrenght = 0.3;

void main() {
    vec3 objectColor = vec3(texture(ourTexture, TextureCoord));

    vec3 ambient = ambientStrenght * lightColor;

    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragmentPos);

    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    vec3 viewDir = normalize(viewPos - FragmentPos);
    vec3 reflectDir = reflect(-lightDir, norm);

    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
    vec3 specular = specularStrength * spec * lightColor;

    vec3 result = (ambient + diffuse + specular) * objectColor;
    FragColor = vec4(result, 1.0);
}

```

5 Висновки

Було отримано практичні навички створення шейдерів, роботи з матрицями, GLM, роботи з індексами та перспективою, роботи з камерою, роботи з текстурами, роботи з освітленням та роботи з бібліотекою OpenGL, використовуючи мову програмування C++