Міністерство освіти і науки України

Національний університет «Запорізька політехніка»

кафедра програмних засобів

ЗВІТ

з лабораторної роботи № 8

з дисципліни «Комп'ютерна графіка та обробка зображень» на тему:

«ТОЧКОВІ ТА ПРОЖЕКТОРНІ СВІТИЛЬНИКИ»

Виконав:

ст. гр. КНТ-113сп Іван ЩЕДРОВСЬКИЙ

Прийняв:

Асистент Артем ТУЛЕНКОВ

2025

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 бібіліотеки GLEW, GLFW, GLM.

2 Завдання до лабораторної роботи

2.1 Застосуйте різноманітне освітлення до об’єкта, змініть форму об’єкта.

2.2 Наведіть алгоритм встановлення освітлення.

2.3 Наведіть коментарі у коді

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.

Під час виконання лабораторної роботи було створено fragment шейдер який може працювати одночасно з декількома джерелами світла. Шейдер реалізує направлений(directional), точковий(point) та прожекторний(spot) світильники. Кожен з цих світильників працює на основі Phong lighting model, яка включає в себе ambient, diffuse та specular освітлення

Направлене світло відрізняється тим, що немає чіткої позиції в просторі. Ми думаємо, що воно є нескінченно віддаленим, а тому всі лучі падають від одним кутом. Реалізація направленого світла показана на рисунку 1.

```
vec3 calculateDirectionalLight(DirectionalLight light, vec3 normal, vec3 viewDir) {
    vec3 lightDir = normalize(-light.direction);

    float diff = max(dot(normal, lightDir), 0.0);
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);

    vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TextureCoord));
    vec3 ambient = light.ambient * vec3(texture(material.diffuse, TextureCoord));
    vec3 specular = light.specular * spec * vec3(texture(material.specular, TextureCoord));

    return ambient + diffuse + specular;
}
```

Рисунок 1 – Реалізація направленого світла

До точкового світла була додана логіка згасання з відстанню. Тепер кожен точковий світильник може мати відстань, на яку він може світити. Реалізація точкового світла показана на рисунку 2.

```
vec3 calculatePointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir) {
    vec3 lightDir = normalize(light.position - fragPos);

    float diff = max(dot(normal, lightDir), 0.0);

    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);

    float distance = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic * (distance * distance));

    vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TextureCoord));
    vec3 ambient = light.ambient * vec3(texture(material.diffuse, TextureCoord));
    vec3 specular = light.specular * spec * vec3(texture(material.specular, TextureCoord));

    ambient *= attenuation;
    diffuse *= attenuation;
    specular *= attenuation;

    return ambient + diffuse + specular;
}
```

Рисунок 2 - Реалізація точкового світла

Прожекторне світло відрізняється тим, що світить тільки в одному напрямку. Ми можемо представити ліхтарик який світить тільки в одну сторону та має деякий конус світла. А щоб краї світла не були жорсткими використовується цілих два конуси, або ж кута, один для внутрішнього світла, а другий для зовнішнього. Реалізація прожекторного світла показана на рисунку 3.

Для реалізації правильного відображення світла було додано матеріал та дві мапи текстур, одна для навколишнього та дифузного світла, а одна для дзеркального. Обидві мапи текстур показані на рисунках 4 та 5.

Рисунок 4 – Базова мапа текстур

Рисунок 5 – Мапа текстур для дзеркального світла

Для створення красивої сцени було реалізовано функціонал відображення десяти кубів та чотирьох точок освітлення, кожна з яких має свій колір. Це показано на рисунках 6, 7 та 8.

```
glm::vec3 cubePositions[] = {
    glm::vec3(0.0f,  0.0f,  0.0f),
    glm::vec3(2.0f,  5.0f, -15.0f),
    glm::vec3(-1.5f, -2.2f, -2.5f),
    glm::vec3(-3.8f, -2.0f, -12.3f),
    glm::vec3(2.4f, -0.4f, -3.5f),
    glm::vec3(-1.7f,  3.0f, -7.5f),
    glm::vec3(1.3f, -2.0f, -2.5f),
    glm::vec3(1.5f,  2.0f, -2.5f),
    glm::vec3(1.5f,  0.2f, -1.5f),
    glm::vec3(-1.3f,  1.0f, -1.5f)
};

glm::vec3 pointLightPositions[] = {
    glm::vec3(0.7f,  0.2f,  2.0f),
    glm::vec3(2.3f, -3.3f, -4.0f),
    glm::vec3(-4.0f,  2.0f, -12.0f),
    glm::vec3(0.0f,  0.0f, -2.0f)
};

glm::vec3 pointLightColors [] = {
    glm::vec3(1.0f,  0.0f,  0.0f), // Red
    glm::vec3(0.0f,  0.0f,  1.0f), // Blue
    glm::vec3(0.0f,  1.0f,  0.0f), // Green
    glm::vec3(1.0f,  1.0f,  0.0f) // Yellow
};
```

Рисунок 6 – Позиції кубів та точкових джерел освітлення

```
cubeShader.use();
cubeShader.setInt("ourTexture", 0);
cubeShader.setVec3("viewPos", cameraPos);

cubeShader.setInt("material.diffuse", 0);
cubeShader.setInt("material.specular", 1);
cubeShader.setFloat("material.shininess", 32.0f);

// Directional light
cubeShader.setVec3("dirLight.ambient", 0.1f, 0.1f, 0.1f);
cubeShader.setVec3("dirLight.diffuse", 1.0f, 1.0f, 1.0f);
cubeShader.setVec3("dirLight.specular", 1.0f, 1.0f, 1.0f);
cubeShader.setVec3("dirLight.direction", -0.2f, -1.0f, -0.3f);

for (unsigned int i = 0; i < sizeof(pointLightPositions) / sizeof(pointLightPositions[0]); i++) {
    std::string index = "pointLights[" + std::to_string(i) + "].";

    cubeShader.setVec3(index + "position", pointLightPositions[i]);
    cubeShader.setVec3(index + "ambient", pointLightColors[i] * 0.05f);
    cubeShader.setVec3(index + "diffuse", pointLightColors[i] * 0.5f);
    cubeShader.setVec3(index + "specular", pointLightColors[i]);
    cubeShader.setFloat(index + "constant", 1.0f);
    cubeShader.setFloat(index + "linear", 0.045f);
    cubeShader.setFloat(index + "quadratic", 0.0075f);
}

// Spot light
cubeShader.setVec3("spotLight.position", cameraPos);
cubeShader.setVec3("spotLight.direction", cameraFront);
cubeShader.setVec3("spotLight.ambient", 0.0f, 0.0f, 0.0f);
cubeShader.setVec3("spotLight.diffuse", 1.0f, 1.0f, 1.0f);
cubeShader.setVec3("spotLight.specular", 1.0f, 1.0f, 1.0f);
cubeShader.setFloat("spotLight.constant", 1.0f);
cubeShader.setFloat("spotLight.linear", 0.09f);
cubeShader.setFloat("spotLight.quadratic", 0.032f);
cubeShader.setFloat("spotLight.cutOff", glm::cos(glm::radians(3.0f)));
cubeShader.setFloat("spotLight.outerCutOff", glm::cos(glm::radians(5.0f)));
```

Рисунок 7 – Логіка використання шейдеру для робити зі всіма видами освітлення

```
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);

cubeShader.setVec2("u_resolution", WIDTH, HEIGHT);
cubeShader.setMat4("view", view);
cubeShader.setMat4("projection", projection);
glBindVertexArray(VAO2);

for (unsigned int i = 0; i < sizeof(cubePositions)/sizeof(cubePositions[0]); i++) {
    float angle = 20.0f * i;

    model = glm::mat4(1.0f);
    model = glm::translate(model, cubePositions[i]);
    model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));

    cubeShader.setMat4("model", model);
    glDrawArrays(GL_TRIANGLES, 0, 36);
}

lightSource.use();
lightSource.setMat4("projection", projection);
lightSource.setMat4("view", view);
glBindVertexArray(lightVAO);

for (unsigned int i = 0; i < sizeof(pointLightPositions)/sizeof(pointLightPositions[0]); i++) {
    model = glm::mat4(1.0f);
    model = glm::translate(model, pointLightPositions[i]);
    model = glm::scale(model, glm::vec3(0.2f));
    lightSource.setMat4("model", model);
    lightSource.setVec3("aColor", pointLightColors[i]);

    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

Рисунок 8 – Логіка відображення кубів та візуалізації точкових джерел освітлення

Вигляд фінальної реалізації показаний на рисунках 9, 10, 11 та 12

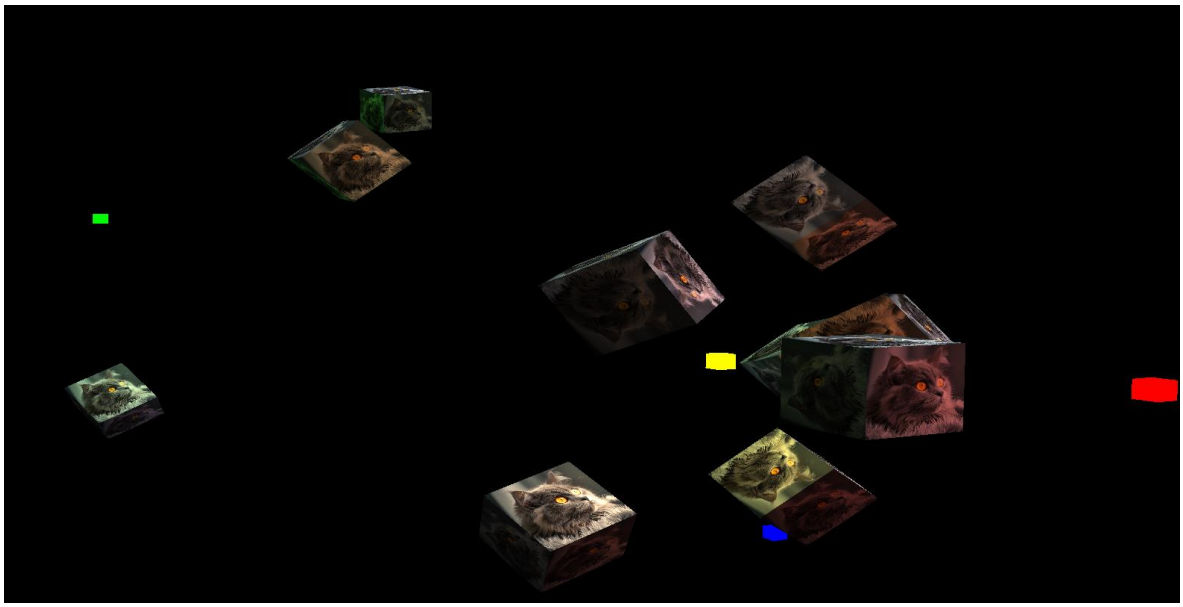

Рисунок 9 – Загальний вигляд сцени

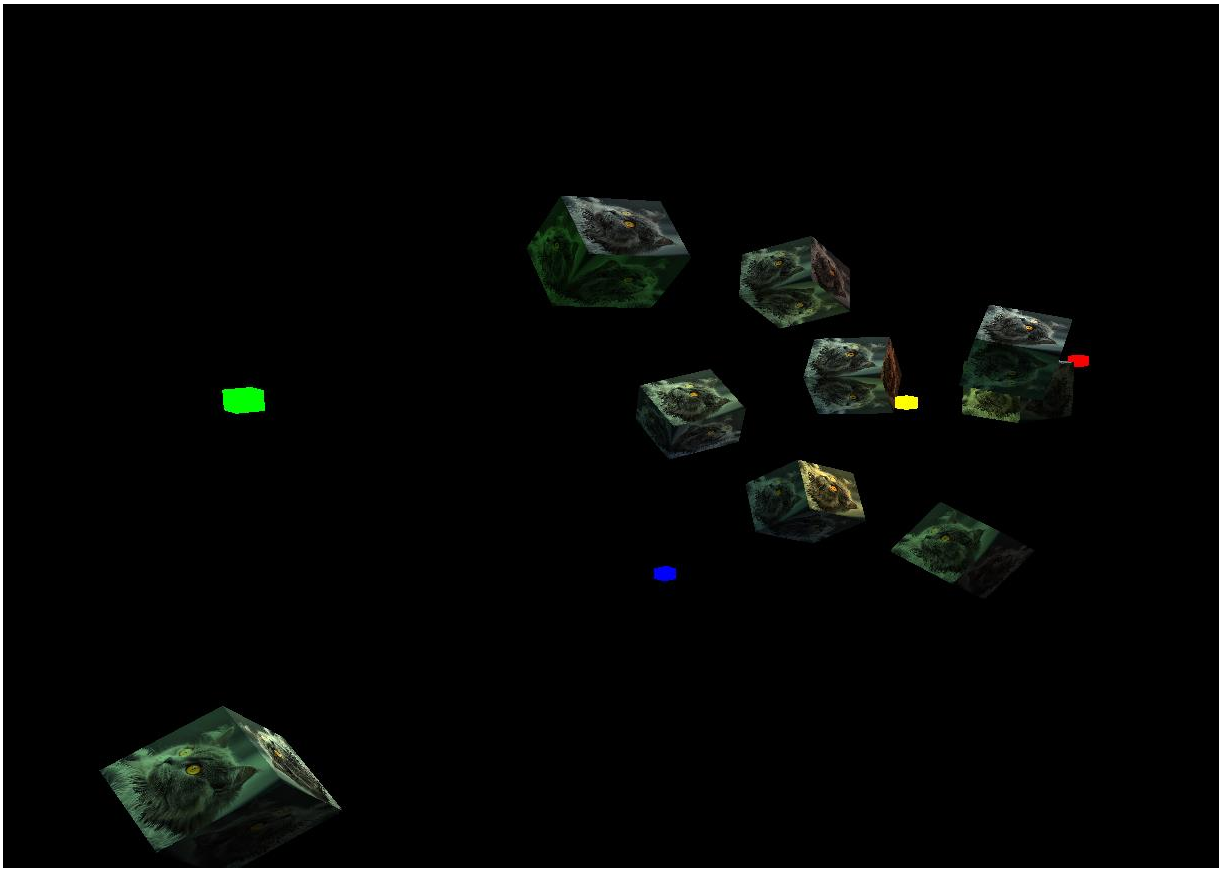

Рисунок 10 – Загальний вигляд сцени з іншого ракурсу

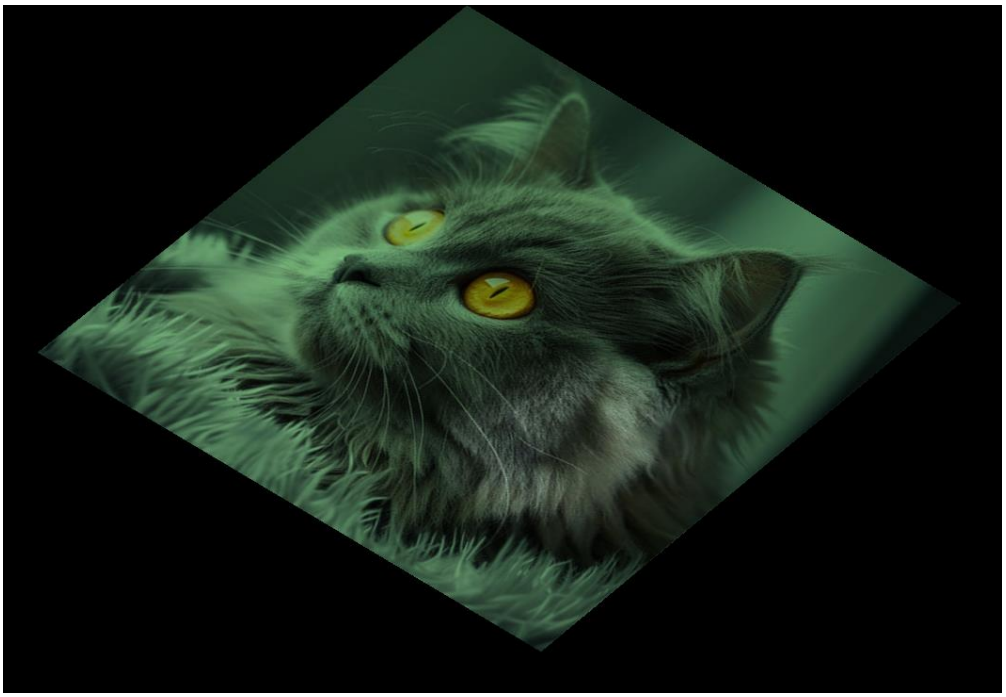

Рисунок 11 – Демонстрація роботи прожекторного світла на не дзеркальній поверхні

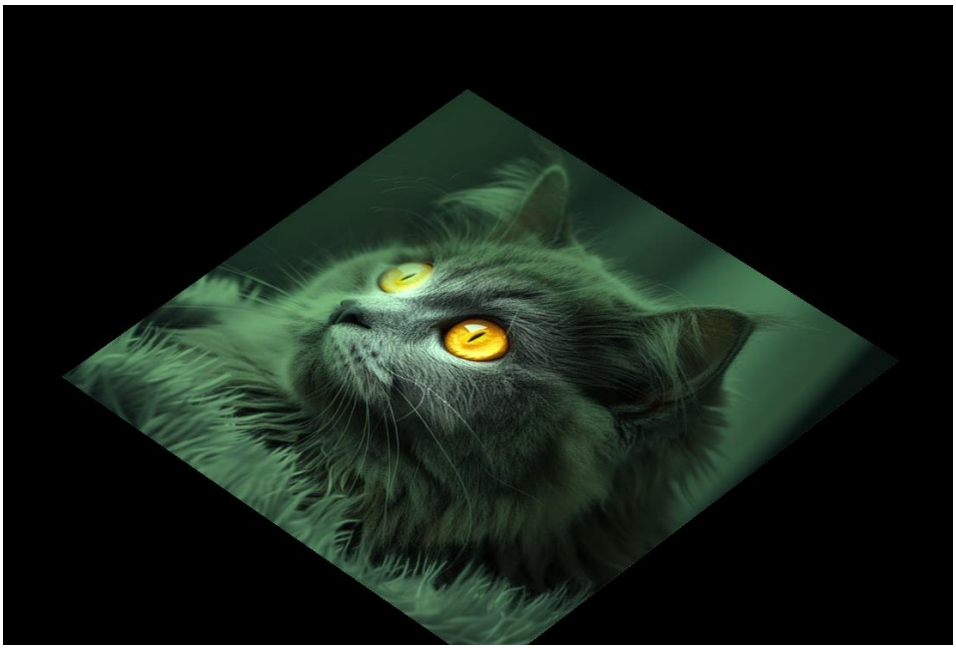

Рисунок 12 - Демонстрація роботи прожекторного світла на дзеркальній поверхні

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;
	void setFloat(const std::string& name, float 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);
}

void Shader::setFloat(const std::string& name, float number) const {
	glUniform1f(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(const char* path);
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
};

glm::vec3 cubePositions[] = {
        glm::vec3(0.0f,  0.0f,  0.0f),
        glm::vec3(2.0f,  5.0f, -15.0f),
        glm::vec3(-1.5f, -2.2f, -2.5f),
        glm::vec3(-3.8f, -2.0f, -12.3f),
        glm::vec3(2.4f, -0.4f, -3.5f),
```

```
        glm::vec3(-1.7f,  3.0f, -7.5f),
        glm::vec3(1.3f, -2.0f, -2.5f),
        glm::vec3(1.5f,  2.0f, -2.5f),
        glm::vec3(1.5f,  0.2f, -1.5f),
        glm::vec3(-1.3f,  1.0f, -1.5f)
};

glm::vec3 pointLightPositions[] = {
        glm::vec3(0.7f,  0.2f,  2.0f),
        glm::vec3(2.3f, -3.3f, -4.0f),
        glm::vec3(-4.0f,  2.0f, -12.0f),
        glm::vec3(0.0f,  0.0f, -2.0f)
};

glm::vec3 pointLightColors [] = {
        glm::vec3(1.0f,  0.0f,  0.0f), // Red
        glm::vec3(0.0f,  0.0f,  1.0f), // Blue
        glm::vec3(0.0f,  1.0f,  0.0f), // Green
        glm::vec3(1.0f,  1.0f,  0.0f) // Yellow
};

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

GLuint VAO1 = initVAO(vertices, sizeof(vertices));
GLuint VAO2 = initVAO(vertices, sizeof(vertices));
GLuint lightVAO = initLightVAO(vertices, sizeof(vertices));

GLuint texture = initTexture("cat.jpg");
GLuint textureSpecular = initTexture("cat-specular.jpg");

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

        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

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

        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, texture);
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, textureSpecular);

        cubeShader.use();
        cubeShader.setInt("ourTexture", 0);
        cubeShader.setVec3("viewPos", cameraPos);

        cubeShader.setInt("material.diffuse", 0);
        cubeShader.setInt("material.specular", 1);
        cubeShader.setFloat("material.shininess", 32.0f);

        // Directional light
        cubeShader.setVec3("dirLight.ambient", 0.1f, 0.1f, 0.1f);
        cubeShader.setVec3("dirLight.diffuse", 1.0f, 1.0f, 1.0f);
        cubeShader.setVec3("dirLight.specular", 1.0f, 1.0f, 1.0f);
        cubeShader.setVec3("dirLight.direction", -0.2f, -1.0f, -0.3f);
```

```
for (unsigned int i = 0; i < sizeof(pointLightPositions) / sizeof(pointLightPositions[0]); i++) {
	std::string index = "pointLights[" + std::to_string(i) + "].";

	cubeShader.setVec3(index + "position", pointLightPositions[i]);
	cubeShader.setVec3(index + "ambient", pointLightColors[i] * 0.05f);
	cubeShader.setVec3(index + "diffuse", pointLightColors[i] * 0.5f);
	cubeShader.setVec3(index + "specular", pointLightColors[i]);
	cubeShader.setFloat(index + "constant", 1.0f);
	cubeShader.setFloat(index + "linear", 0.045f);
	cubeShader.setFloat(index + "quadratic", 0.0075f);
}

// Spot light
cubeShader.setVec3("spotLight.position", cameraPos);
cubeShader.setVec3("spotLight.direction", cameraFront);
cubeShader.setVec3("spotLight.ambient", 0.0f, 0.0f, 0.0f);
cubeShader.setVec3("spotLight.diffuse", 1.0f, 1.0f, 1.0f);
cubeShader.setVec3("spotLight.specular", 1.0f, 1.0f, 1.0f);
cubeShader.setFloat("spotLight.constant", 1.0f);
cubeShader.setFloat("spotLight.linear", 0.09f);
cubeShader.setFloat("spotLight.quadratic", 0.032f);
cubeShader.setFloat("spotLight.cutOff", glm::cos(glm::radians(3.0f)));
cubeShader.setFloat("spotLight.outerCutOff", glm::cos(glm::radians(5.0f)));

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);

cubeShader.setVec2("u_resolution", WIDTH, HEIGHT);
cubeShader.setMat4("view", view);
cubeShader.setMat4("projection", projection);
glBindVertexArray(VAO2);

for (unsigned int i = 0; i < sizeof(cubePositions)/sizeof(cubePositions[0]); i++) {
	float angle = 20.0f * i;

	model = glm::mat4(1.0f);
	model = glm::translate(model, cubePositions[i]);
	model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));

	cubeShader.setMat4("model", model);
	glDrawArrays(GL_TRIANGLES, 0, 36);
}

lightSource.use();
lightSource.setMat4("projection", projection);
lightSource.setMat4("view", view);
glBindVertexArray(lightVAO);

for (unsigned int i = 0; i < sizeof(pointLightPositions)/sizeof(pointLightPositions[0]); i++) {
	model = glm::mat4(1.0f);
	model = glm::translate(model, pointLightPositions[i]);
	model = glm::scale(model, glm::vec3(0.2f));
	lightSource.setMat4("model", model);
	lightSource.setVec3("aColor", pointLightColors[i]);
```

```
                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(const char* path) {
        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(path, &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 aroud 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.frag
```

```
#version 330 core

struct Material {
	sampler2D diffuse;
	sampler2D specular;
	float shininess;
};

struct DirectionalLight {
	vec3 direction;

	vec3 ambient;
	vec3 diffuse;
	vec3 specular;
};

struct PointLight {
	vec3 position;

	vec3 ambient;
	vec3 diffuse;
	vec3 specular;

	float constant;
	float linear;
	float quadratic;
};

struct SpotLight {
	vec3 position;
	vec3 direction;

	vec3 ambient;
	vec3 diffuse;
	vec3 specular;

	float constant;
	float linear;
	float quadratic;

	float cutOff;
	float outerCutOff;
};

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

out vec4 FragColor;

#define NR_POINT_LIGHTS 4

uniform Material material;
uniform DirectionalLight dirLight;
uniform PointLight pointLights[NR_POINT_LIGHTS];
uniform SpotLight spotLight;
uniform vec3 viewPos;

vec3 calculateDirectionalLight(DirectionalLight light, vec3 normal, vec3 viewDir);
```

```
vec3 calculatePointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir);
vec3 calculateSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir);

void main() {
	vec3 normal = normalize(Normal);
	vec3 viewDir = normalize(viewPos - FragmentPos);

	vec3 result = calculateDirectionalLight(dirLight, normal, viewDir);

	for (int i = 0; i < NR_POINT_LIGHTS; i++) {
		result += calculatePointLight(pointLights[i], normal, FragmentPos, viewDir);
	}

	result += calculateSpotLight(spotLight, normal, FragmentPos, viewDir);

	FragColor = vec4(result, 1.0);
}

vec3 calculateSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir) {
	vec3 lightDir = normalize(light.position - fragPos);

	float diff = max(dot(normal, lightDir), 0.0);
	vec3 reflectDir = reflect(-lightDir, normal);
	float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);

	float distance = length(light.position - fragPos);
  float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic * (distance * distance));

	float theta = dot(lightDir, normalize(-light.direction));
	float epsilon = light.cutOff - light.outerCutOff;
	float intensity = clamp((theta - light.outerCutOff) / epsilon, 0.0, 1.0);

	vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TextureCoord));
	vec3 ambient = light.ambient * vec3(texture(material.diffuse, TextureCoord));
	vec3 specular = light.specular * spec * vec3(texture(material.specular, TextureCoord));

	ambient *= attenuation * intensity;
  diffuse *= attenuation * intensity;
  specular *= attenuation * intensity;

	return ambient + diffuse + specular;
}

vec3 calculatePointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir) {
	vec3 lightDir = normalize(light.position - fragPos);

	float diff = max(dot(normal, lightDir), 0.0);

	vec3 reflectDir = reflect(-lightDir, normal);
	float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);

	float distance = length(light.position - fragPos);
	float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic * (distance * distance));

	vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TextureCoord));
	vec3 ambient = light.ambient * vec3(texture(material.diffuse, TextureCoord));
	vec3 specular = light.specular * spec * vec3(texture(material.specular, TextureCoord));

	ambient *= attenuation;
```

```
        diffuse *= attenuation;
        specular *= attenuation;

        return ambient + diffuse + specular;
}

vec3 calculateDirectionalLight(DirectionalLight light, vec3 normal, vec3 viewDir) {
        vec3 lightDir = normalize(-light.direction);

        float diff = max(dot(normal, lightDir), 0.0);
        vec3 reflectDir = reflect(-lightDir, normal);
        float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);

        vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TextureCoord));
        vec3 ambient = light.ambient * vec3(texture(material.diffuse, TextureCoord));
        vec3 specular = light.specular * spec * vec3(texture(material.specular, TextureCoord));

        return ambient + diffuse + specular;
}
```

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, роботи з багатьма джерелами освітлення та роботи з бібліотекою OpenGL, використовуючи мову програмування C++