**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»**

**ІКНІ**

Кафедра **ПЗ**



**ЗВІТ**

до лабораторної роботи №3

**на тему:** *“Створення анімаційних зображень з використанням шейдерів вершини та фрагмента, мови GLSL та бібліотек GLFW і GLM”*

**з дисципліни** *“Засоби програмування комп'ютерної графіки”*

**Лектор:**

асис. каф. ПЗ

Журавчак Л.М.

**Виконав:**

ст. гр. ПЗ-33

Юшкевич. А.І.

**Прийняв:**

ст. викл. каф. ПЗ

Журавчак Л.М.

«\_\_» \_\_\_\_\_\_\_\_\_ 2024р.

∑=\_\_\_\_\_

Львів – 2024

**Тема:** Створення анімаційних зображень з використанням шейдерів вершини та фрагмента, мови GLSL та бібліотек GLFW і GLM.

**Мета:** Навчитися будувати двовимірні анімаційні зображення з використанням шейдерів вершини та фрагмента, мови GLSL та бібліотек GLFW і GLM.

**Теоретичні відомості**

Для створення анімаційних зображень в OpenGL використовують основні геометричні перетворення – перенесення, обертання і масштабування, реалізовані у бібліотеці GLM.

**Шейдер вершини** – це один з шейдерів, який може написати будь-яким програміст. Сучасний OpenGL вимагає, щоб ми налаштували, щонайменше, шейдери вершини і фрагмента, якщо плануємо виконати якийсь рендеринг.

Перше, що нам треба зробити – це написати шейдер вершини на мові шейдерів GLSL, а потім скомпілювати його, щоб використовувати в нашому застосунку.

Наприклад:

#version 330 core

layout (location = 0) in vec3 aPos;

void main() {

gl\_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0); }

Для того, щоб OpenGL міг використати шейдер, він має динамічно скомпілювати його зі свого вихідного коду під час виконання програми. Спочатку нам треба створити об’єкт шейдера з ідентифікатором, який посилається на нього. Для цього тип змінної шейдера вершини оголосимо як unsigned int і створимо сам шейдер за допомогою функції glCreateShader():

unsigned int vertexShader;

vertexShader = glCreateShader(GL\_VERTEX\_SHADER);

Далі передаємо тип шейдера, який ми хочемо створити, як аргумент функції glCreateShader() – у нашому випадку GL\_VERTEX\_SHADER. Потім прикріплюємо вихідний код шейдера до об’єкту шейдера і компілюємо його:

glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);

glCompileShader(vertexShader);

**Шейдер фрагмента** – другий і останній шейдер, який ми створимо для рендерингу фігур, він потрібний для обчислення вихідного кольору пікселів (у прикладі вибрано помаранчевий колір):

#version 330 core

out vec4 FragColor;

void main() {

FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);

}

Процес компіляції шейдера фрагмента аналогічний компіляції шейдера вершини, але цього разу як тип шейдера ми використовуємо константу GL\_FRAGMENT\_SHADER:

unsigned int fragmentShader;

fragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER);

glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);

glCompileShader(fragmentShader);

Нам залишається зв’язати два шейдерні об’єкти в шейдерну програму, яку ми зможемо використати для рендерингу. Створити об’єкт шейдерної програми дуже просто:

unsigned int shaderProgram;

shaderProgram = glCreateProgram();

Функція glCreateProgram() створює програму і повертає ідентифікатор посилання на новостворений об’єкт шейдерної програми. Тепер треба прикріпити раніше скомпільовані шейдери до об’єкта програми і з допомогою функції glLinkProgram() зв’язати їх:

glAttachShader(shaderProgram, vertexShader);

glAttachShader(shaderProgram, fragmentShader);

glLinkProgram(shaderProgram);

Як і раніше, ми можемо перевірити, чи нема помилок при зв’язуванні шейдерної програми, використавши такі функції:

glGetProgramiv(shaderProgram, GL\_LINK\_STATUS, &success);

if(!success) {

glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);

...}

Результатом є програмний об’єкт, який активуємо викликом функції glUseProgram() з новоствореним програмним об’єктом як аргументом:

glUseProgram(shaderProgram);

Після активації glUseProgram() кожен виклик шейдера і рендеринга буде використовувати цей програмний об’єкт (і, отже, шейдери). До речі, не забудьте видалити шейдерні об’єкти після зв’язування з програмним об’єктом, оскільки вони вам вже непотрібні:

glDeleteShader(vertexShader);

glDeleteShader(fragmentShader);

Більшість функцій GLM, які нам потрібні, знаходяться в трьох заголовних файлах, які підключають так:

#include <glm/glm.hpp>

#include <glm/gtc/matrix\_transform.hpp>

#include <glm/gtc/type\_ptr.hpp>

Приклади застосування функцій для паралельного перенесення, масштабування та обертання об’єкта :

glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);

glm::mat4 trans = glm::mat4(1.0f);

trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));

vec = trans \* vec;

std::cout << vec.x << vec.y << vec.z << std::endl;

glm::mat4 trans = glm::mat4(1.0f);

trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));

trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));

**Завдання**

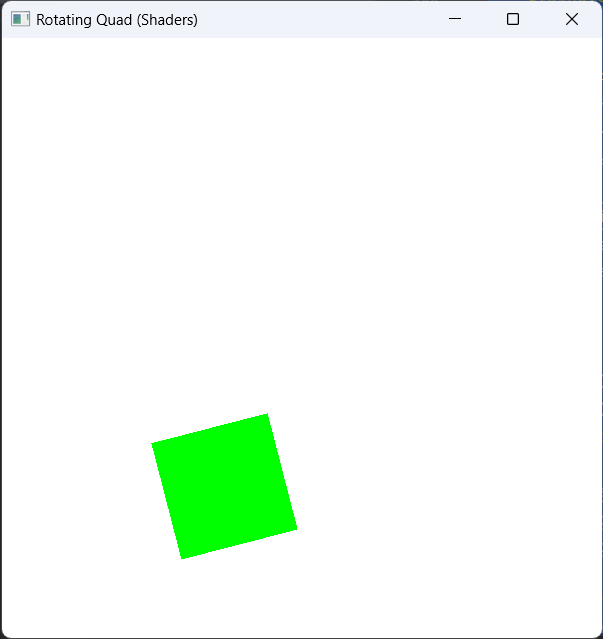
Базуючись на завданнях лабораторної роботи № 2, запрограмувати для їхньої реалізації використання шейдерів вершини та фрагмента, мови GLSL та бібліотек GLFW і GLM. Вивчіть матеріал тем 4 та 5 з розділу 1.

12. Задано **чотирикутник**. Реалізувати неперервний рух фігури за траєкторією ***y=ax***в обох напрямках з одночасним обертанням навколо заданої вершини.

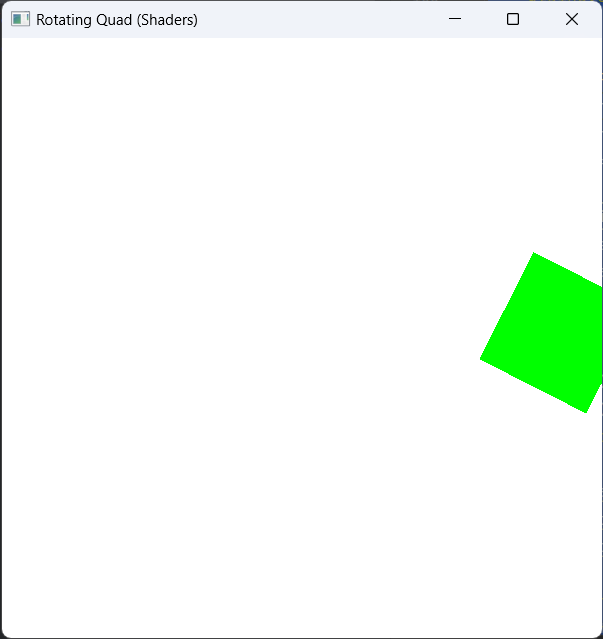
**Код програми**

#include <GL/glew.h>  
#include <GLFW/glfw3.h>  
#include <iostream>  
#include <cmath>  
#include <vector>  
  
// GLSL вершинний шейдер  
const char\* vertexShaderSource = R"(  
#version 330 core  
layout (location = 0) in vec2 aPos;  
uniform float angle;  
uniform vec2 translation;  
uniform vec2 pivot;  
void main() {  
 vec2 pos = aPos - pivot;  
 float s = sin(angle);  
 float c = cos(angle);  
 vec2 rotated = vec2(pos.x \* c - pos.y \* s, pos.x \* s + pos.y \* c) + pivot;  
 gl\_Position = vec4(rotated + translation, 0.0, 1.0);  
}  
)";  
  
// GLSL фрагментний шейдер  
const char\* fragmentShaderSource = R"(  
#version 330 core  
out vec4 FragColor;  
void main() {  
 FragColor = vec4(0.0, 1.0, 0.0, 1.0);  
}  
)";  
  
struct Vertex {  
 float x, y;  
};  
  
float angle = 0.f;  
float rotationSpeed = 0.05f;  
float speed = 0.0002f;  
float slope = 0.5f;  
float positionX = 0.0f, positionY = 0.0f;  
Vertex rotationVertex;  
GLuint VAO, VBO, EBO, shaderProgram;  
  
std::vector<Vertex> rectangle = {  
 {-0.2f, -0.2f},  
 { 0.2f, -0.2f},  
 { 0.2f, 0.2f},  
 {-0.2f, 0.2f},  
};  
  
GLuint indices[] = { 0, 1, 2, 2, 3, 0 };  
  
void compileShader(GLuint shader, const char\* source) {  
 glShaderSource(shader, 1, &source, nullptr);  
 glCompileShader(shader);  
 int success;  
 glGetShaderiv(shader, GL\_COMPILE\_STATUS, &success);  
 if (!success) {  
 char infoLog[512];  
 glGetShaderInfoLog(shader, 512, nullptr, infoLog);  
 std::cerr << "Shader Compilation Error:\n" << infoLog << std::endl;  
 }  
}  
  
void setupShaders() {  
 GLuint vertexShader = glCreateShader(GL\_VERTEX\_SHADER);  
 compileShader(vertexShader, vertexShaderSource);  
  
 GLuint fragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER);  
 compileShader(fragmentShader, fragmentShaderSource);  
  
 shaderProgram = glCreateProgram();  
 glAttachShader(shaderProgram, vertexShader);  
 glAttachShader(shaderProgram, fragmentShader);  
 glLinkProgram(shaderProgram);  
  
 int success;  
 glGetProgramiv(shaderProgram, GL\_LINK\_STATUS, &success);  
 if (!success) {  
 char infoLog[512];  
 glGetProgramInfoLog(shaderProgram, 512, nullptr, infoLog);  
 std::cerr << "Shader Program Linking Error:\n" << infoLog << std::endl;  
 }  
  
 glDeleteShader(vertexShader);  
 glDeleteShader(fragmentShader);  
}  
  
void setupBuffers() {  
 glGenVertexArrays(1, &VAO);  
 glGenBuffers(1, &VBO);  
 glGenBuffers(1, &EBO);  
  
 glBindVertexArray(VAO);  
 glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  
 glBufferData(GL\_ARRAY\_BUFFER, rectangle.size() \* sizeof(Vertex), rectangle.data(), GL\_STATIC\_DRAW);  
  
 glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO);  
 glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(indices), indices, GL\_STATIC\_DRAW);  
  
 glVertexAttribPointer(0, 2, GL\_FLOAT, GL\_FALSE, sizeof(Vertex), (void\*)0);  
 glEnableVertexAttribArray(0);  
  
 glBindBuffer(GL\_ARRAY\_BUFFER, 0);  
 glBindVertexArray(0);  
}  
  
void updatePosition() {  
 positionX += speed;  
 positionY = slope \* positionX;  
 angle += rotationSpeed;  
  
 if (positionX > 1.f || positionX < -0.8f) {  
 speed = -speed;  
 rotationSpeed = -rotationSpeed;  
 }  
}  
  
void render() {  
 glClear(GL\_COLOR\_BUFFER\_BIT);  
 glUseProgram(shaderProgram);  
  
 GLuint angleLoc = glGetUniformLocation(shaderProgram, "angle");  
 GLuint translationLoc = glGetUniformLocation(shaderProgram, "translation");  
 GLuint pivotLoc = glGetUniformLocation(shaderProgram, "pivot");  
  
 glUniform1f(angleLoc, angle \* M\_PI / 180.0f);  
 glUniform2f(translationLoc, positionX, positionY);  
 glUniform2f(pivotLoc, rotationVertex.x, rotationVertex.y);  
  
 glBindVertexArray(VAO);  
 glDrawElements(GL\_TRIANGLES, 6, GL\_UNSIGNED\_INT, 0);  
  
 glFlush();  
}  
  
int main() {  
 if (!glfwInit()) {  
 std::cerr << "Failed to initialize GLFW" << std::endl;  
 return -1;  
 }  
  
 GLFWwindow\* window = glfwCreateWindow(600, 600, "Rotating Quad (Shaders)", nullptr, nullptr);  
 if (!window) {  
 std::cerr << "Failed to create GLFW window" << std::endl;  
 glfwTerminate();  
 return -1;  
 }  
 glfwMakeContextCurrent(window);  
 glewInit();  
  
 glClearColor(1, 1, 1, 1);  
 rotationVertex = rectangle[0];  
  
 setupShaders();  
 setupBuffers();  
  
 while (!glfwWindowShouldClose(window)) {  
 updatePosition();  
 render();  
 glfwSwapBuffers(window);  
 glfwPollEvents();  
  
 if (glfwGetKey(window, GLFW\_KEY\_1) == GLFW\_PRESS) rotationVertex = rectangle[0];  
 if (glfwGetKey(window, GLFW\_KEY\_2) == GLFW\_PRESS) rotationVertex = rectangle[1];  
 if (glfwGetKey(window, GLFW\_KEY\_3) == GLFW\_PRESS) rotationVertex = rectangle[2];  
 if (glfwGetKey(window, GLFW\_KEY\_4) == GLFW\_PRESS) rotationVertex = rectangle[3];  
 }  
  
 glDeleteVertexArrays(1, &VAO);  
 glDeleteBuffers(1, &VBO);  
 glDeleteBuffers(1, &EBO);  
 glDeleteProgram(shaderProgram);  
  
 glfwDestroyWindow(window);  
 glfwTerminate();  
 return 0;  
}

**Результати виконання роботи**



*Рис. 1. Чотирикутник, що обертається навколо однієї зі своїх першин та рухається вздовж осі.*

****

*Рис. 2. Чотирикутник, що обертається навколо однієї зі своїх першин та рухається вздовж осі.*

**Висновки:** Під час виконання лабораторної роботи я навчився будувати двовимірні анімаційні зображення з використанням шейдерів вершини та фрагмента, мови GLSL та бібліотек GLFW і GLM.