Лекція 3

Лабораторна робота 3

ШЕЙДЕРИ (Shaders)

Ми вже згадували шейдери у попередньому уроці. Шейдери - це невеликі програми, що виконуються на графічному прискорювачі (далі будемо використовувати більш поширену назву - GPU). Ці програми виконуються на кожній конкретній ділянці графічного конвеєра. Якщо описувати шейдери найбільш простим способом, то шейдери - це не більше ніж програми, що перетворюють входи у виходи. Шейдери зазвичай ізольовані одне від одного, і не мають механізмів комунікації між собою, крім згаданих вище входів і виходів.

У попередньому уроці ми коротко торкнулися теми "вершинних шейдерів" та того, як їх використовувати. У даному уроці ми розглянемо шейдери докладніше і зокрема шейдерну мову OpenGL (OpenGL Shading Language).

GLSL

Шейдери (як згадувалося вище, шейдери - це програми) програмуються С подібною мовою GLSL. Віна адаптована для використання у графіці та надає функції роботи з векторами та матрицями.

Опис шейдера починається із зазначення його версії, далі йдуть списки вхідних та вихідних змінних, глобальних змінних (ключове слово uniform), функції main. Функції main є початковою точкою шейдера. Усередині цієї функції можна проводити маніпуляції з вхідними даними, результат роботи шейдера міститься у вихідні змінні.

Нижче представлена узагальнена структура шейдера:

#version version_number

in type in_variable_name;

in type in_variable_name;

out type out_variable_name;

uniform type uniform_name;

```
void main()
{
  // process input(s) and do some weird graphics stuff
  ...
  // output processed stuff to output variable
  out_variable_name = weird_stuff_we_processed;
}
```

Вхідні змінні вершинного шейдера називаються вершинними атрибутами. Існує максимальна кількість вершин, які можна передати в шейдер, таке обмеження накладається обмеженими можливостями апаратного забезпечення. OpenGL гарантує можливості передачі щонайменше 16 4-х компонентних вершин, інакше кажучи в шейдер можна передати щонайменше 64 значення. Однак варто враховувати, що існують обчислювальні пристрої, що значно піднімає цю планку. Так чи інакше, дізнатися максимальну кількість вхідних змінних-вершин, що передаються в шейдер, можна, звернувшись до атрибуту *GL_MAX_VERTEX_ATTRIBS* .

```
GLint nrAttributes;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &nrAttributes);
std::cout << "Maximum nr of vertex attributes supported: " << nrAttributes
<< std::endl;
```

У результаті виконання цього коду, ми побачимо цифру >=16.

Типи

GLSL, як і будь-яка інша мова програмування, надає певний перелік типів змінних, до них належать такі примітивні типи: int, float, double, uint, bool. Також GLSL надає два типи-контейнери: vector та matrix.

Vector

Vector у GLSL – це контейнер, що містить від 1 до 4 значень будь-якого примітивного типу. Оголошення контейнера vector може мати такий вигляд (n — кількість елементів вектора):

vecn (наприклад vec4) — це стандартний vector, що містить у собі n значень типу float
bvecn (наприклад, bvec4) — це vector, що містить у собі n значень типу boolean
ivecn (наприклад, ivec4) — це vector, що містить у собі n значень типу integer

uvecn (наприклад, uvecn) — це vector, що містить n значень типу unsigned integer
dvecn (наприклад dvecn) — це vector, що містить n значень типу double.

У більшості випадків використовуватиметься стандартний vector vecn.

Для доступу до елементів контейнера vector ми будемо використовувати наступний синтаксис vec.x, vec.y, vec.z, vec.w (у даному випадку ми звернулися до всіх елементів по порядку від першого до останнього). Також можна ітеруватися RGBA, якщо вектор описує колір, або stpq якщо вектор описує координати текстури.

Допускається звернення одного вектора через XYZW, RGBA, STPQ. Не потрібно сприймати цю замітку як посібник до дії, будь ласка.

Взагалі ніхто не забороняє ітеруватися по вектору за допомогою індексу та оператора доступу за індексом [].
https://en.wikibooks.org/wiki/GLSL_Programming/Vector_and_Matrix_Operations#Components

З вектора, при зверненні до даних через точку, можна отримати не тільки одне значення, а й цілий вектор, використовуючи наступний синтаксис, який називається *swizzling*

```
vec2 someVec;

vec4 differentVec = someVec.xyxx;

vec3 anotherVec = differentVec.zyw;

vec4 otherVec = someVec.xxxx + anotherVec.yxzy;
```

Для створення нового вектора ви можете використовувати до 4 літералів одного типу або вектор, правило лише одне — у сумі потрібно отримати необхідну кількість елементів, наприклад: для створення вектора з 4 елементів ми можемо використовувати два вектори довжиною в 2 елементи, або один вектор довжиною у 3 елементи та один літерал. Також для створення вектор з елементів n допускається вказівка одного значення, в цьому випадку всі елементи вектора приймуть це значення. Допускається також використання змінних примітивних типів.

```
vec2 vect = vec2(0.5f, 0.7f);

vec4 result = vec4(vect, 0.0f, 0.0f);

vec4 otherResult = vec4(result.xyz, 1.0f);
```

Можна помітити, що вектор дуже гнучкий тип даних і його можна використовувати в ролі вхідних та вихідних змінних.

In та out змінні

Ми знаємо що шейдери - це маленькі програми, але в більшості випадків вони є частиною чогось більшого, тому в GLSL є in і out змінні, що дозволяють створити "інтерфейс" шейдера, що дозволяє отримати дані для обробки і передати результати стороні, що викликає. Таким чином, кожен шейдер може визначити для себе вхідні та вихідні змінні використовуючи ключові слова in і out.

Вершинний шейдер був би вкрай неефективним, якби він не приймав жодних вхідних даних. Сам по собі це шейдер відрізняється від інших шейдерів тим, що набуває вхідних значень безпосередньо з вершинних даних. Для того, щоб вказати OpenGL, як організовані аргументи, ми використовуємо метадані позиції для того, щоб ми могли налаштовувати атрибути на CPU. Ми вже бачили цей прийом раніше: layout (location = 0). Вершинний шейдер, у свою чергу, вимагає додаткових специфікацій для того, щоб ми могли зв'язатися аргументи з вершинними даними.

Можна опустити layout (location = 0), і використовувати виклик glGetAttributeLocation для отримання атрибутів вершин.

Ще одним винятком є те, що фрагментний шейдер (Fragment shader) повинен мати на виході vec4, інакше кажучи, фрагментний шейдер повинен надати у вигляді результату колір у форматі RGBA. Якщо цього не буде зроблено, то об'єкт буде змальований чорним або білим.

Таким чином, якщо перед нами стоїть завдання передачі інформації від одного шейдера до іншого, то необхідно визначити в передавальному шейдері out змінну такого типу як і in змінної в приймаючої шейдері. Таким чином, якщо типи та імена змінних будуть однакові з обох сторін, OpenGL з'єднає ці змінні разом, що дасть нам можливість обміну інформацією між шейдерами (це робиться на етапі компонування). Для демонстрації цього практично ми змінимо шейдери з попереднього уроку, таким чином, щоб вершинний шейдер надавав колір для фрагментного шейдера.

Vertex shader

```
#version 330 core

layout (location = 0) in vec3 position; // Встановлює позицію атрибута в 0


out vec4 vertexColor; // Передаємо колір у фрагментний шейдер


void main()
```

```
{
    gl_Position = vec4(position, 1.0); // Відразу передаємо vec3 в vec4
    vertexColor = vec4(0.5f, 0.0f, 0.0f, 1.0f); // Встановлюємо значення вихідної змінної темно-червоний колір.
}
```

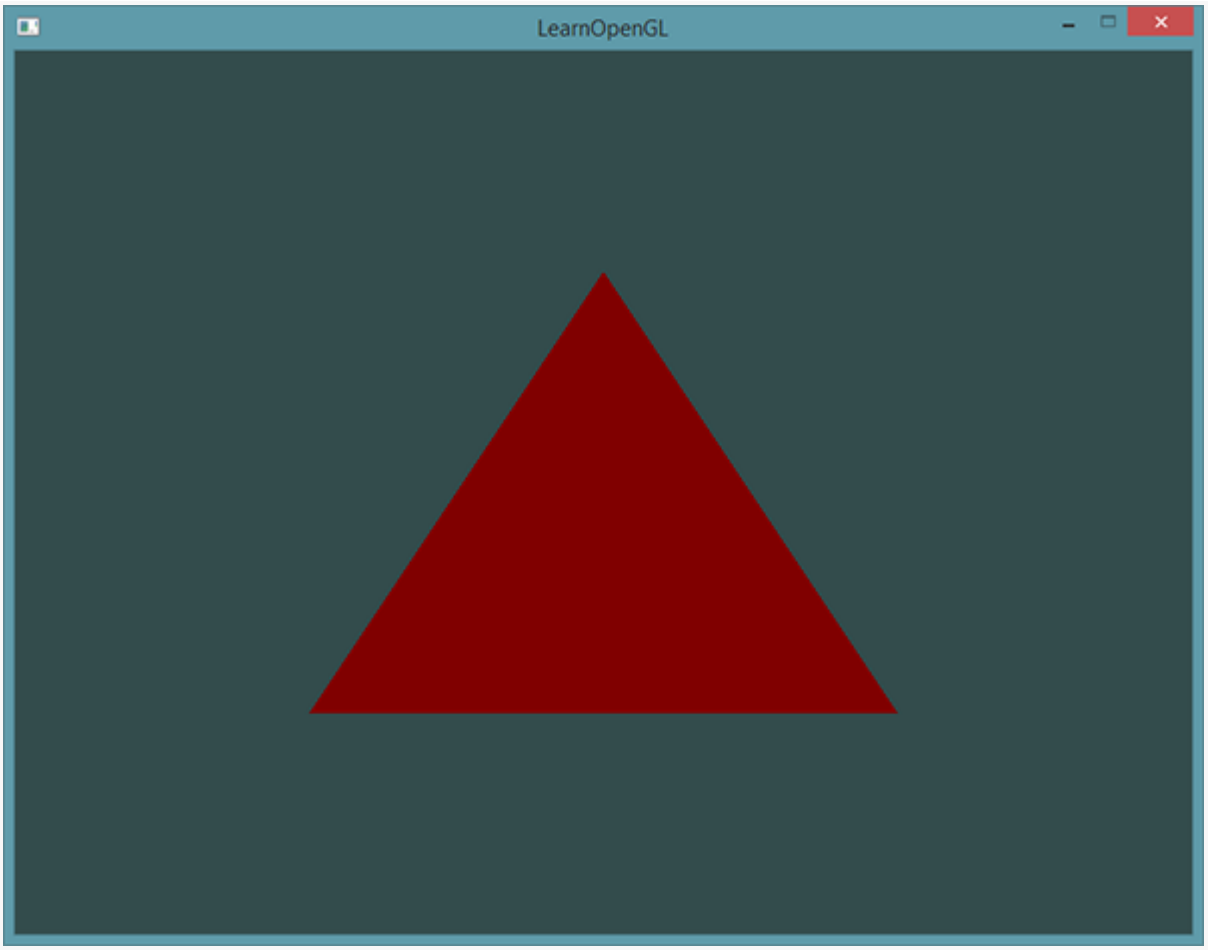
Fragment shader

```
#version 330 core
in vec4 vertexColor; // Вхідна змінна з вершинного шейдера (та сама назва і той же тип)


out vec4 color;


void main()
{
     color=vertexColor;
}
```

У цих прикладах ми оголосили вихідний вектор із 4 елементів з ім'ям vertexColor у вершинному шейдері та ідентичний вектор з назвою vertexColor, але тільки як вхідний у фрагментному шейдері. В результаті вихідний vertexColor з вершинного шейдера і вхідний vertexColor фрагментного шейдера були з'єднані. Т.к. ми встановили значення vertexColor у вершинному шейдері, що відповідає непрозорому бордовому (темно-червоному кольору), застосування шейдера до об'єкта робить його бордового кольору. Наступне зображення показує результат:



От і все. Ми зробили так, що значення вершинного шейдера було отримано фрагментним шейдером. Далі ми розглянемо спосіб передачі інформації шейдеру з нашої програми.

Uniforms

Uniforms (назвемо їх формами) — це ще один спосіб передачі інформації від нашого додатка, що працює на CPU, до шейдера, що працює на GPU. Форми трохи відрізняються від атрибутів вершин. Спочатку: форми є глобальними. Глобальна змінна для GLSL означає наступне: Глобальна змінна буде унікальною для кожної шейдерної програми, і доступ до неї має кожен шейдер на будь-якому етапі в цій програмі. Друге: значення форми зберігається доти, доки воно не буде скинуто або оновлено.

Для оголошення форми в GLSL використовується специфікатор змінної unifrom. Після оголошення форми його можна використовувати у шейдері. Давайте подивимося, як встановити колір нашого трикутника з використанням форми:

```
#version 330 core

out vec4 color;


uniform vec4 ourColor; // Ми встановлюємо значення цієї змінної у коді
OpenGL.


void main()

{

     color = ourColor;

}
```

Ми оголосили змінну форми outColor типу вектора із 4 елементів у фрагментному шейдері та використовуємо її для встановлення вихідного значення фрагментного шейдера. Т.к. Форма є глобальною змінною, то її оголошення можна проводити в будь-якому шейдері, а це означає, що нам не потрібно передавати щось з вершинного шейдера в фрагментний. Отже ми оголошуємо форму у вершинному шейдері, т.к. ми її не використовуємо.

Якщо ви оголошуєте форму і не використовуєте її в шейдерній програмі, то компілятор тишком видалить її, що може викликати деякі помилки, так що тримайте цю інформацію в голові.

На даний момент форма не містить у собі корисних даних, т.к. ми їх туди не помістили, то давайте ж зробимо це. Для початку нам потрібно дізнатися індекс, інакше кажучи місце розташування потрібного нам атрибуту форми в нашому шейдері. Отримавши значення індексу атрибута, ми зможемо вмістити туди необхідні дані. Щоб наочно продемонструвати працездатність цієї функції, ми змінюватимемо колір від часу:

```
GLfloat timeValue = glfwGetTime();
GLfloat greenValue = (sin(timeValue) / 2) + 0.5;
GLint vertexColorLocation = glGetUniformLocation(shaderProgram,
"ourColor");
glUseProgram(shaderProgram);
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

Спочатку ми отримуємо час роботи в секундах, викликавши glfwGetTime(). Після цього ми змінюємо значення від 0.0 до 1.0, використовуючи функцію sin і записуємо результат у змінну greenValue.

Після цього ми запитуємо індекс форми ourColor, використовуючи glGetUniformLocation. Ця функція приймає два аргументи: змінну програми-шейдера та назву форми, визначеної всередині цієї програми. Якщо glGetUniformLocation повернув -1, це означає, що такої форми з такою назвою не було знайдено. Останньою нашою дією є встановлення значення форми ourColor за допомогою функції glUniform4f. Зауважте, що пошук індексу форми

не вимагає попереднього виклику glUseProgram, але для оновлення значення форми спочатку необхідно викликати glUseProgram.

Так як OpenGL реалізований з використанням мови C, в якій немає перевантаження функцій, виклик функцій з різними аргументами неможливий, але OpenGL визначено функції для кожного типу даних, які визначаються постфіксом функції. Нижче наведено деякі постфікси:

f: функція приймає float аргумент;
i: функція приймає int аргумент;
ui: функція приймає unsigned int аргумент;
3f: функція приймає три аргументи типу float;
fv: функція приймає як аргумент вектор з float.

Таким чином, замість використання перевантажених функцій ми повинні використовувати функцію, реалізація якої призначена для певного набору аргументів, на що вказує постфікс функції. У наведеному вище прикладі ми використовували функцію glUniform…() спеціалізовану для обробки 4 аргументів типу float, таким чином повне ім'я функції було glUniform4f() (4f — чотири аргументи типу float).

Тепер, коли ми знаємо, як задавати значення формам, ми можемо використовувати їх у процесі рендерингу. Якщо ми хочемо змінювати колір з часом, то нам потрібно оновлювати значення форми кожну ітерацію циклу малювання (інакше кажучи, колір буде змінюватися на кожному кадрі), інакше наш трикутник буде одного кольору, якщо ми задамо колір тільки один раз. У наведеному нижче прикладі відбувається обчислення нового кольору трикутника та оновлення на кожній ітерації циклу рендерингу:

```
while(!glfwWindowShouldClose(window))
{
    // Обробляємо події
    glfwPollEvents();

    // Відмальовка
    // Очищаємо буфер кольору
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // Активуємо шейдерну програму
    glUseProgram(shaderProgram);

    // Оновлюємо колір форми
    GLfloat timeValue = glfwGetTime();
    GLfloat greenValue = (sin(timeValue) / 2) + 0.5;
    GLint vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
    glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);

    // Малюємо трикутник
    glBindVertexArray(VAO);
```

```
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glBindVertexArray(0);
}
```

Код дуже схожий на використовуваний раніше, але тепер ми виконуємо його всередині циклу, змінюючи значення форми кожної ітерації. Якщо все вірно, то ви побачите зміну кольору трикутника із зеленого на чорний та назад (якщо не зрозуміло, знайдіть зображення синусоїди).

http://learnopengl.com/video/getting-started/shaders.mp4

Повний вихідний код програми, яка творить такі дива можна подивитися тут .

Як ви вже помітили, форми – це дуже зручний спосіб обміну даними між шейдером та вашою програмою. Але що робити, якщо ми хочемо задати колір для кожної вершини? Для цього ми маємо оголосити стільки форм, скільки є вершин. Найбільш вдалим рішенням буде використання атрибутів вершин, що ми зараз продемонструємо.

Більше атрибутів богу атрибутів!

У попередньому уроці ми бачили, як заповнити VBO, налаштувати покажчики на атрибути вершин і як зберігати це все в VAO. Тепер нам потрібно додати інформацію про кольори до цих вершин. Для цього ми створимо вектор із трьох елементів float. Ми призначимо червоний, зелений, та синій кольори кожної з вершин трикутника відповідно:

```
GLfloat vertices[] = {
    // Позиції // Кольори
     0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 0.0f, // Нижній правий кут
    -0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, // Нижній лівий кут
     0.0f, 0.5f, 0.0f, 0.0f, 0.0f, 1.0f // Верхній кут
};
```

Тепер у нас є багато інформації для передачі її вершинному шейдеру, необхідно відредагувати шейдер так, щоб він отримував і вершини та кольори. Зверніть увагу на те, що ми встановлюємо розташування кольору в 1:

```
#version 330 core
layout (location = 0) in vec3 position; // Встановлюємо позицію змінної з координатами 0
layout (location = 1) in vec3 color; // А позицію змінної з кольором 1

out vec3 ourColor; // Передаємо колір у фрагментний шейдер

void main()
{
    gl_Position = vec4(position, 1.0);
```

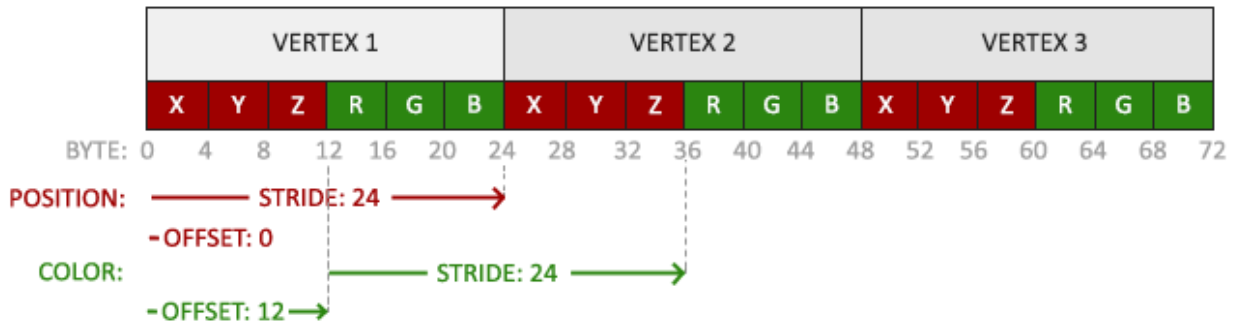
ourColor = color; // Встановлюємо значення кольору, отримане від вершинних даних
}

Зараз нам не потрібна форма ourColor, але вихідний параметр ourColor нам стане в нагоді для передачі значення фрагментному шейдеру:

```
#version 330 core
in vec3 ourColor;
out vec4 color;

void main()
{
    color = vec4(ourColor, 1.0f);
}
```

Т.к. ми додали новий параметр вершини і оновили VBO пам'ять, нам потрібно налаштувати покажчики атрибутів вершин. Оновлені дані в пам'яті VBO виглядають так:



Знаючи поточну схему ми можемо оновити формат вершин, використовуючи функцію *glVertexAttribPointer* :

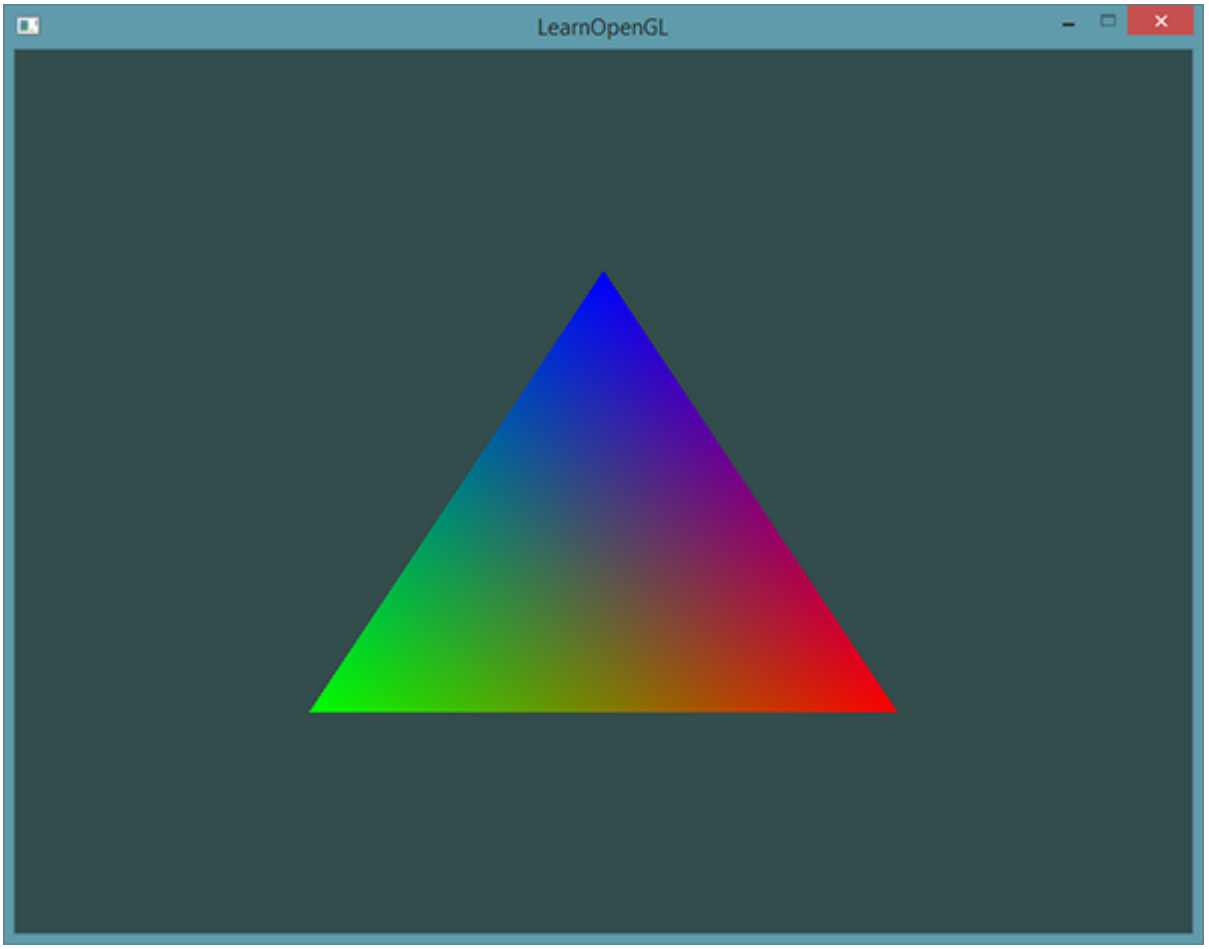
// Атрибут із координатами
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);
// Атрибут із кольором
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)(3* sizeof(GLfloat)));
glEnableVertexAttribArray(1);

Перші кілька атрибутів функції glVertexAttribPointer досить прості. У цьому прикладі ми використовуємо вершинний атрибут з позицією 1. Колір складається з трьох значень типу флоту і нам не потрібна нормалізація.

Т.к. тепер ми використовуємо два атрибути шейдерів, то слід перерахувати крок. Для доступу до наступного атрибуту шейдера (наступний x вектор вершин) нам потрібно переміститися на 6 елементів float вправо, на 3 для вектора вершин і на 3 для вектора кольору. Тобто. ми перемістимося 6 разів праворуч, тобто. на 24 байти праворуч.

Тепер ми розібралися зі зсувами. Першим йде вектор із координатами вершини. Вектор зі значенням RGB кольору йде після вектора з координатами, тобто. після 3* sizeof(GLfloat) = 12 байт.

Запустивши програму, ви можете побачити наступний результат:



Повний вихідний код, що творить це диво можна подивитися тут :

Може здатися, що результат не відповідає виконаній роботі, адже ми задали лише три кольори, а не палітру, яку ми бачимо в результаті. Такий результат дає фрагментна інтерполяція фрагментного шейдера. При малюванні трикутника, на етапі растеризації, виходить набагато більше областей, а не тільки вершини, які ми використовуємо як аргументи шейдера. Растеризатор визначає позиції цих областей з урахуванням їх становища на полігоні. З цієї позиції відбувається інтерполяція всіх аргументів фрагментного шейдера.

Припустимо, ми маємо просту лінію, з одного кінця вона зелена, а з іншого вона синя. Якщо фрагментний шейдер обробляє область, яка знаходиться приблизно посередині, то колір цієї області буде підібраний так, що зелений дорівнюватиме 50% від кольору, що використовується в лінії, і, відповідно, синій дорівнюватиме 50% відсотків від синього. Саме це відбувається на нашому трикутнику. Ми маємо три кольори, і три вершини, для кожної з яких встановлено один із кольорів. Якщо придивитися, то можна побачити, що червоний, при переході до синього спочатку стає фіолетовим, що цілком очікувано. Фрагментна інтерполяція застосовується до всіх атрибутів фрагментного шейдера.

ООП у маси! Робимо свій клас шейдера

Код, що описує шейдер, що робить його компіляцію, що дозволяє проводити налаштування шейдера може бути дуже громіздким. Так давайте ж зробимо наше життя трохи простіше, написавши клас, що зчитує наш шейдер з диска, що робить його компіляцію, лінковку, перевірку на помилки, ну і звичайно, має простий і приємний інтерфейс. Таким чином, ОВП допоможе нам інкапсулювати весь цей хаос усередині методів нашого класу.

Почнемо розробку нашого класу з оголошення його інтерфейсу та пропишемо до новоствореного заголовника всі необхідні include директиви. В результаті отримуємо щось подібне:

```
#ifndef SHADER_H
#define SHADER_H

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>

#include <GL/glew.h>; // Підключаємо glew для того, щоб отримати всі необхідні заголовні файли OpenGL

class Shader
{
public:
    // Ідентифікатор програми
    GLuint Program;
    // Конструктор зчитує та збирає шейдер
    Shader(const GLchar* vertexPath, const GLchar* fragmentPath);
    // Використання програми
    void Use();
};

#endif
```

Давайте будемо молодцями і використовувати директиви ifndef і define, щоб уникнути рекурсивного виконання директив include. Ця порада відноситься не до OpenGL, а до програмування на C++ загалом.

І так наш клас зберігатиме в собі свій ідентифікатор. Конструктор шейдера буде приймати в якості аргументів покажчики на масиви символів (іншими словами, текст, а в контексті класу, доречніше буде сказати - шлях до файлу з вихідним кодом нашого шейдера), що містять шлях до файлів, що містять вершинний і фрагментний шейдери, представлені звичайним текстом. Також додамо утилітарну функцію Use, що наочно демонструє переваги використання класів-шейдерів.

Зчитування файлу шейдера. Для зчитування будемо використовувати стандартні потоки C++, поміщаючи результат у рядки:

```
Shader(const GLchar* vertexPath, const GLchar* fragmentPath)
{
    // 1. Отримуємо вихідний код шейдера з filePath
    std::string vertexCode;
    std::string fragmentCode;
    std::ifstream vShaderFile;
    std::ifstream fShaderFile;
    // Переконаємося, що якщо stream об'єкти можуть викидати винятки
    vShaderFile.exceptions(std::ifstream::badbit);
    fShaderFile.exceptions(std::ifstream::badbit);
    try
    {
        // Відкриваємо файли
        vShaderFile.open(vertexPath);
        fShaderFile.open(fragmentPath);
        std::stringstream vShaderStream, fShaderStream;
        // Зчитуємо дані у потоки
        vShaderStream << vShaderFile.rdbuf();
        fShaderStream << fShaderFile.rdbuf();
        // Закриваємо файли
        vShaderFile.close();
        fShaderFile.close();
        // Перетворюємо потоки на масив GLchar
        vertexCode = vShaderStream.str();
        fragmentCode = fShaderStream.str();
    }
    catch(std::ifstream::failure e)
    {
        std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" <<
std::endl;
    }
    const GLchar* vShaderCode = vertexCode.c_str();
    const GLchar* fShaderCode = fragmentCode.c_str();
    [...]
```

Тепер нам потрібно скомпілювати і злінкувати наш шейдер (давайте робитимемо все добре, і зробимо функціонал, що повідомляє нам про помилку при компіляції та лінковки шейдера. Несподівано, але це дуже корисно в процесі налагодження):

```
// 2. Складання шейдерів
GLuint vertex, fragment;
GLint success;
GLchar infoLog[512];

// Вершинний шейдер
vertex = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex, 1, &vShaderCode, NULL);
glCompileShader(vertex);
// Якщо є помилки – вивести їх
glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
if(!success)
{
    glGetShaderInfoLog(vertex, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog
<< std::endl;
};

// Аналогічно для фрагментного шейдера
[...]

// Шейдерна програма
this->Program = glCreateProgram();
glAttachShader(this->Program, vertex);
glAttachShader(this->Program, fragment);
glLinkProgram(this->Program);
//Якщо є помилки - вивести їх
glGetProgramiv(this->Program, GL_LINK_STATUS, &success);
if(!success)
{
    glGetProgramInfoLog(this->Program, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog <<
std::endl;
}

// Видаляємо шейдери, оскільки вони вже у програму і нам більше не потрібні.
glDeleteShader(vertex);
glDeleteShader(fragment);
```

Ну а вишнею на торті буде реалізація методу Use:

```
void Use() { glUseProgram(this->Program); }
```

А ось і результат нашої роботи:

```
Shader ourShader("path/to/shaders/shader.vs", "path/to/shaders/shader.frag");
...
while(...)
{
    ourShader.Use();
    glUniform1f(glGetUniformLocation(ourShader.Program, "someUniform"), 1.0f);
    DrawStuff();
}
```

У цьому прикладі ми помістили наш вершинний шейдер у файл shader.vs, а фрагментний шейдер у файл shader.frag. В принципі, ви можете встановити свою конвенцію іменування файлів, особливо це не зіграє, але робіть це обдумано, зберігаючи семантику.

Вихідні коди програми з класом шейдера , класу шейдера , вершинного шейдера та фрагментного шейдера

Вправи:

1. Модифікуйте вершинний шейдер так, щоб у результаті трикутник перекинувся: рішення .

2. Передайте горизонтальне усунення за допомогою форми та перемістіть трикутник до правої сторони вікна за допомогою вершинного шейдера: рішення .

3. Передайте фрагментному шейдеру позицію вершини та встановіть значення кольору, що дорівнює значенню позиції (дивіться, як позиція вершини інтерполується по всьому трикутнику). Після того, як зробите це, постарайтеся відповісти на запитання, чому нижня ліва частина трикутника чорна ?