**Лабораторная работа №4**

Основы программирования компьютерной графики.

Цель работы: Разработка приложений, реализующих вывод двумерной графики. Знакомство с программным интерфейсом OpenGL.

*Теоретические сведения*

Консорциум Khronos Group, включающий в себя более ста компаний, занимается созданием спецификаций открытых программных интерфейсов, одним из которых является OpenGL – интерфейс, предназначенный для применения аппаратного ускорения в компьютерной графике. OpenGL содержит определения и описания прототипов функций. Фирма-производитель видеокарт предоставляет программную реализацию функций спецификации в пакете драйверов оборудования.

Разработка графических ускорителей на этапе создания логических схем учитывает использование того или иного программного интерфейса.

Графический конвейер – абстрактная модель, описывающая шаги обработки входных данных для получения конечного результата в виде чередующихся изображений (визуализации в реальном времени) либо единичного изображения (при использовании сложных алгоритмов, расчет которых занимает длительное время). Графический конвейер реализуется на этапе проектирования оборудования. Для обработки входных данных на различных этапах конвейера используются программы – шейдеры. Байт-код скомпилированных шейдеров выполняется непосредственно графическим ускорителем. Использование шейдеров (также как использование программных интерфейсов) позволяет скрыть детали реализации оборудования, обеспечивая (чаще всего) одинаковое поведение разработанного программного обеспечения независимо от производителя графического ускорителя.

В цикле лабораторных работ будут рассмотрены вершинный (Vertex) и пиксельный (Fragment) шейдеры.

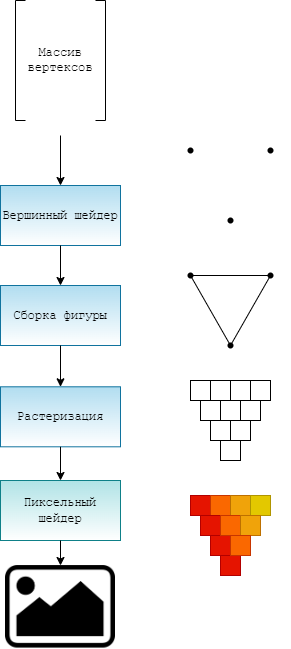


Рис. 1. Графический конвейер

OpenGL определен как конечный автомат. Состояние в OpenGL содержит конечный набор чисел. Число из набора может быть некоторой константой, определяющей результат визуализации, либо номером объекта (например буфера), содержащего данные. Чтобы изменить объект, созданный в OpenGL, необходимо чтобы его идентификатор находился в состоянии OpenGL.

На рис. 1 изображена модель графического конвейера. Вертекс – тип данных, содержащий параметры вершины, такие как координаты вершины, цвет и т.д. На вход графического конвейера передается массив вертексов.

Рассмотрим 4 этапа графического конвейера:

1. Вершинный шейдер. В вершинном шейдере происходит прием данных со входа графического конвейера. Для каждой вершины передаваемой на вход фигуры происходит вычисление геометрических преобразований, например перенос вершин.
2. Сборка фигуры. Переданные вершины формируют фигуру, состоящую из примитивов – треугольников. Сборку фигуры осуществляет драйвер.
3. Растеризация. Растеризация – проецирование фигуры (двумерной или трехмерной) на пиксели экрана (окна вывода). В результате растеризации получаем пиксели, которые покрывают видимую часть фигуры. Растеризацию осуществляет драйвер.
4. Пиксельный шейдер. Пиксельный шейдер производит вычисления связанные с определением цвета пикселей покрывающих видимую часть фигуры.

Перед разбором исходного кода рассмотрим математическую модель реализации геометрических преобразований объектов.

Рассмотрим матрицу переноса:

*=*

Вектор (x,y,z,1) содержит координаты переносимой точки (x,y,z) а также четвертый компонент w. При w = 1 -> (x,y,z) – точка, а при w = 0 (x,y,z) – вектор.

(Tx, Ty, Tz) – значения, на которые будет осуществлен перенос вершин.

Масштабирование осуществляется с помощью матрицы следующего вида:

=

Для вращения используется матрица вращения.

Пусть φ – величина угла, на которую необходимо повернуть точку.

Для поворота вокруг оси X используется операция:

*=*

Для поворота вокруг оси Y используется операция:

*=*

Для поворота вокруг оси Z используется операция:

*=*

Чтобы применить все геометрические трансформации используется произведение матрицы переноса, матриц вращения и матрицы масштабирования. Полученная матрица называется матрицей преобразований (model matrix).

*Методические указания*

Постановка задачи: разработать оконное приложение, визуализирующее двумерный объект с использованием программного интерфейса OpenGL.

Данную задачу можно разделить на 3 этапа:

1. Реализация контекста вывода.
2. Загрузка программных реализаций функций OpenGL из драйверов.
3. Использование интерфейса OpenGL для визуализации.

На первом этапе происходит разработка программы для создания оконного контекста вывода. Существует несколько способов осуществить обращение к ОС. В случае с ОС Windows можно использовать WinAPI, либо библиотеки, такие как: QT, GLFW, SDL и другие. Остановим выбор на библиотеке GLFW, так как она является кроссплатформенной и ее функционала достаточно для наших целей.

Для решения второй подзадачи воспользуемся библиотекой glad loader.

Проверим подключение библиотеки GLFW:

#include <GLFW/glfw3.h> //Библиотека GLFW

int main() {

glfwInit();//Инициализация GLFW

return 0;

}

Код для проверки подключения библиотеки GLFW

* Функция glfwInit() инициализирует библиотеку GLFW

При успешной компиляции и выполнении выше приведенного исходного кода можем констатировать факт, что библиотека GLFW подключена успешна.

Далее разберем код для создания окна с помощью библиотеки GLFW:

#include <GLFW/glfw3.h> //Библиотека GLFW

GLFWwindow\* window;

int main() {

glfwInit();//Инициализация GLFW

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);//Core Profile

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3); //Младшая версия GL

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3); //Старшая версия GL

//Параметры: ширина, высота, название окна

window = glfwCreateWindow(800, 600, "Window", nullptr, nullptr); //Создание окна

//Позволяет вызывать OpenGL функции для созданного окна

glfwMakeContextCurrent(window);

//Пока окно не закрыто

while (!glfwWindowShouldClose(window))

{

//Чередовать буферы с изображением

glfwSwapBuffers(window);

//Обработать события ОС

glfwPollEvents();

}

//Освободить ресурсы

glfwDestroyWindow(window);

glfwTerminate();

return 0;

}

Пример создания окна

* В первую очередь вызывается процедура glfwWindowHint(), которая выставляет константы библиотеки. Выставляется младшая и старшая версия OpenGL, а также режим работы Core Profile, позволяющий использовать шейдеры и другие объекты введенные в спецификацию начиная с версии OpenGL 3.3.
* glfwCreateWindow() - возвращает указатель на созданное окно GLFWwindow\*.
* glfwMakeContextCurrent() – вызов этой функции позволяет использовать контекст OpenGL для созданного окна.
* glfwWindowShouldClose() – возвращает булево значение, сигнализирующее о закрытии окна.
* glfwSwapBuffers() – чередует буферы изображений. Каждый раз, когда готовое изображение (front buffer) выводится на экран, следующее изображение (back buffer) рассчитывается. После того, как следующее изображение рассчитано, оно выводится (front и back buffer меняются).
* glfwPollEvents() – обрабатывает события ОС.
* glfwTerminate() и glfwDestroyWindow() используются для освобождения ресурсов.

Получим следующий результат:

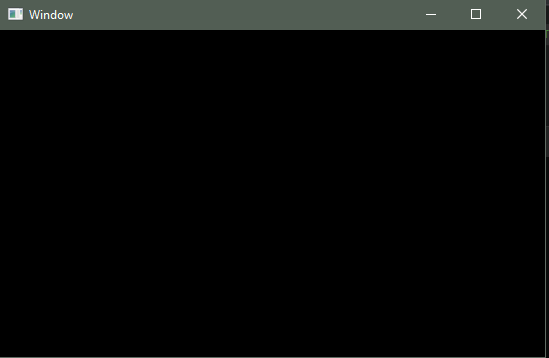


Рис. 2. Результат выполнения кода

После создания окна с помощью библиотеки GLFW воспользуемся библиотекой GLAD для загрузки программных реализаций функций OpenGL из драйверов.

#include <glad/glad.h>

#include <GLFW/glfw3.h> //Библиотека GLFW

GLFWwindow\* window;

int main() {

...

//Позволяет вызывать OpenGL функции для созданного окна

glfwMakeContextCurrent(window);

//Загрузка реализаций функций OpenGL из драйверов

gladLoadGLLoader((GLADloadproc)glfwGetProcAddress);

...

Загрузка программных реализаций функций OpenGL из драйверов

Библиотека GLAD должна быть включена до включения библиотеки GLFW. После выполнения функции glfwMakeContextCurrent(), вызывается функция gladLoadGLLoader(), загружающая имплементации функций OpenGL. Теперь мы можем вызывать функции OpenGL:

...

//Пока окно не должно быть закрытым

while (!glfwWindowShouldClose(window))

{

glClear(GL\_COLOR\_BUFFER\_BIT); //Очистить экран

glClearColor(0.3, 0.3, 0.3, 1.0f); //Цвет очистки экрана

...

Очистить экран и заполнить цветом

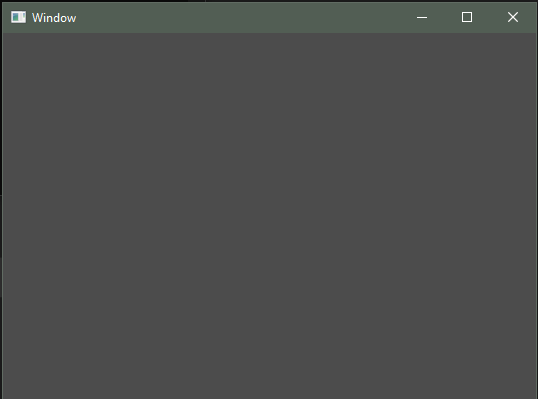


Рис. 3. Результат выполнения кода

Функция glClear() очищает экран (обнуляет буфер с пикселями), и заполняет его цветом, заданным с помощью функции glClearColor.

Для визуализации любых объектов необходимо разработать пиксельный и вершинный шейдеры c помощью языка GLSL (OpenGL Shading Language):

//Исходный код вершинного шейдера

std::string vShader = R"glsl(

#version 440 core //Версия GLSL

layout(location = 0) in vec2 vPos; //Координаты вершины примитива

void main() {

gl\_Position = vec4(vPos.x, vPos.y, 0.0f, 1.0f);

}

)glsl";

Вершинный шейдер

Строка vShader хранит исходный код вершинного шейдера. В первой строке объявляется версия GLSL. Так как мы передаем в вершинный шейдер обычный массив данных, который хранит несколько параметров для каждой вершины, необходимо задать как интерпретировать входные данные.

...

//Загрузка реализаций функций OpenGL из драйверов

gladLoadGLLoader((GLADloadproc)glfwGetProcAddress);

const float vertices[] = {

0.0f, 0.5f,

0.5f, -0.5f,

-0.5f, -0.5f

};

...

Массив вершин

В демонстрируемом массиве вершин имеем три точки: (0, 0.5), (0.5, -0.5), (-0.5, -0.5). Эти точки образуют треугольник. Чтобы сформировать фигуру, необходимо считывать из буфера по два элемента.

Вернемся к вершинному шейдеру.

#version 440 core //Версия GLSL

layout(location = 0) in vec2 vPos; //Координаты вершины примитива

void main() {

gl\_Position = vec4(vPos.x, vPos.y, 0.0f, 1.0f);

}

Вершинный шейдер

Начнем разбор второй строки начиная справа налево.

vPos – идентификатор переменной.

vec2 – тип данных: двумерный вектор.

in – обозначает что переменная является входной.

layout(location = 0) - считывается по идентификатору ноль.

Таким образом мы считываем из массива по два значения в переменную vPos. Предопределенная переменная gl\_Position определяет конечное положение вершины примитива.

Рассмотрим пиксельный шейдер:

//Исходный код пиксельного шейдера

std::string fShader = R"glsl(

#version 440 core //Версия GLSL

out vec4 fragColor; //Выходной цвет пикселя

void main() {

fragColor = vec4(1.0f, 0.0f, 0.0f, 1.0f); //Изменение значения выходного цвета пикселя

}

)glsl"

;

Пиксельный шейдер

Как и в вершинном шейдере в первой строке указывается версия языка GLSL. Во второй строке определяется выходная переменная fragColor, которая имеет тип данных vec4. Выходная переменная в пиксельном шейдере может быть только одна. Таким образом на выходе могут быть только пиксели (поэтому шейдер называется пиксельным).

В теле функции main выходной переменной fragColor присваивается цвет в формате RGBA (1, 0, 0, 1). Значение цвета в GLSL нормировано от 0 до 1.

На этом пока закончим описание исходного кода шейдеров.

Теперь рассмотрим функцию для создания объекта, хранящего байт-код шейдеров:

static GLuint CreateProgram() {

GLuint vertex\_shader = glCreateShader(GL\_VERTEX\_SHADER),

pixel\_shader = glCreateShader(GL\_FRAGMENT\_SHADER);

//Конвертируем исходный код из std::string в const char \*

const char\* vertex\_shader\_src = vShader.c\_str();

const char\* fragment\_shader\_src = fShader.c\_str();

glShaderSource(vertex\_shader, 1, &vertex\_shader\_src, 0);

glCompileShader(vertex\_shader);

Check(vertex\_shader); //Проверка компиляции

glShaderSource(pixel\_shader, 1, &fragment\_shader\_src, 0);

glCompileShader(pixel\_shader);

Check(pixel\_shader); //Проверка компиляции

GLuint program = glCreateProgram();

glAttachShader(program, vertex\_shader);

glAttachShader(program, pixel\_shader);

glLinkProgram(program);

Check(program, true); //Проверка компоновки

glDeleteShader(vertex\_shader);

glDeleteShader(pixel\_shader);

return program;

}

Функция создания программы из шейдеров

Шейдеры представляют из себя отдельные модули. Поэтому они поочередно компилируются, после чего компонуются в одну программу.

В первую очередь создаются объекты шейдеров с помощью функции glCreateShader (glCreateShader возвращает идентификатор шейдера). После чего исходный код шейдеров, который мы рассмотрели ранее, передается в соответствующие объекты с помощью функции glShaderSource, а также компиляция с помощью функции glCompileShader. Результат компиляции проверяется с помощью пользовательской функции Check (см. исходный код примера).

После компиляции двух шейдеров создается объект program, который будет содержать скомпонованную программу из этих шейдеров.

Следующим этапом рассмотрим процесс создания вершинного буфера (Vertex buffer Object) и вершинного массива (Vertex Array Object).

...

//Загрузка реализаций функций OpenGL из драйверов

gladLoadGLLoader((GLADloadproc)glfwGetProcAddress);

//Массив вершин

const float vertices[] = {

0.0f, 0.5f,

0.5f, -0.5f,

-0.5f, -0.5f

};

GLuint vao; //Массив вершин (Vertex Array)

glGenVertexArrays(1, &vao); //Сгенерировать идентификатор

glBindVertexArray(vao); //Выбрать массив вершин

GLuint vbo; //Буфер вершин (Vertex Buffer)

glGenBuffers(1, &vbo); //Сгенерировать идентификатор

glBindBuffer(GL\_ARRAY\_BUFFER, vbo); //Выбрать буфер

//Заполнить буфер данными

glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);

//Определить интерпретацию считывания из буфера для идентификатора 0 в шейдере glVertexAttribPointer(

0, //Идентификатор в вершинном шейдере

2, //Количество значений передаваемых по идентификатору

GL\_FLOAT, //Тип данных

GL\_FALSE, //Нужна ли нормализация (от 0 до 1)

sizeof(GL\_FLOAT) \* 2, //Шаг считывания из буфера

(void\*)0 //Сдвиг в буфере

);

//Разрешить передачу данных по нулевому идентификатору

glEnableVertexAttribArray(0);

GLuint program = CreateProgram();

...

Создание вершинного массива и вершинного буфера

Вершинный буфер (Vertex Buffer Object) – буфер, содержащий вертексы фигуры.

Вершинный массив (Vertex Array Object) – несмотря на свое название не означает массив в прямом смысле. Вершинный массив определяет каким образом разбить массив вершин для считывания в шейдере.

Рассмотрим блок кода, приведенный выше.

OpenGL для доступа к объектам использует беззнаковые целочисленные значения – идентификаторы.

Функции glGenVertexArrays() и glGenBuffers() генерирует идентификатор для вершинных массивов и буферов соответственно. Функции glBindVertexBuffer() и glBindBuffer() выбирают активный вершинный массив и буфер. glBufferData() записывает массив данных vertices в буфер vbo. Функция glVertexAttribPointer разбивает буфер вершин для считывания в шейдере. glEnableVertexAttribArray() разрешает передачу данных по идентификатору в шейдере. Функция glBufferData вызывается при выбранном активном буфере с помощью функции glBindBuffer, а функции glVertexAttribPointer и glEnableVertexAttribArray вызываются при выбранном активном массиве вершин с помощью функции glBindVertexArray. Чтобы изменить объект, созданный в OpenGL, необходимо чтобы его идентификатор находился в состоянии OpenGL.

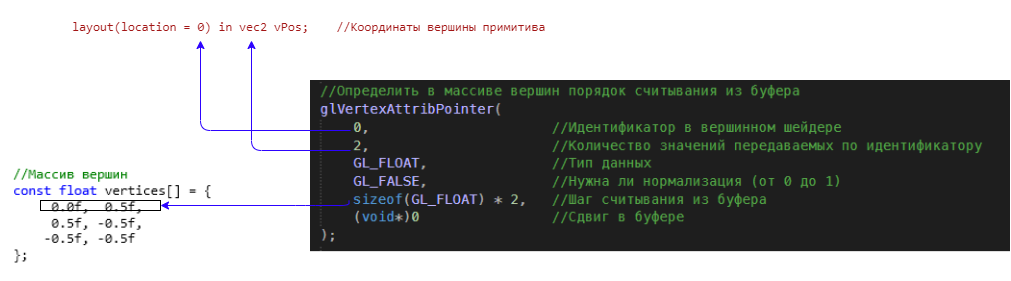


Рис. 4. Иллюстрация к применению функции glVertexAttribPointer

Если мы вернемся к ранее рассмотренному вершинному шейдеру, то сможем с помощью рис. 4 понять смысл вызова функции glVertexAttribPointer.

Перейдем к главному циклу программы.

...

//Пока окно не должно быть закрытым

while (!glfwWindowShouldClose(window))

{

glClear(GL\_COLOR\_BUFFER\_BIT); //Очистить экран

glClearColor(0.3, 0.3, 0.3, 1.0f); //Цвет очистки экрана

glUseProgram(program); //Выбираем активную программу из шейдеров

glBindVertexArray(vao); //Выбираем активный массив вершин

glDrawArrays(GL\_TRIANGLES, 0, 3); //Отрисовать три вершины

...

Функция создания графического конвейера

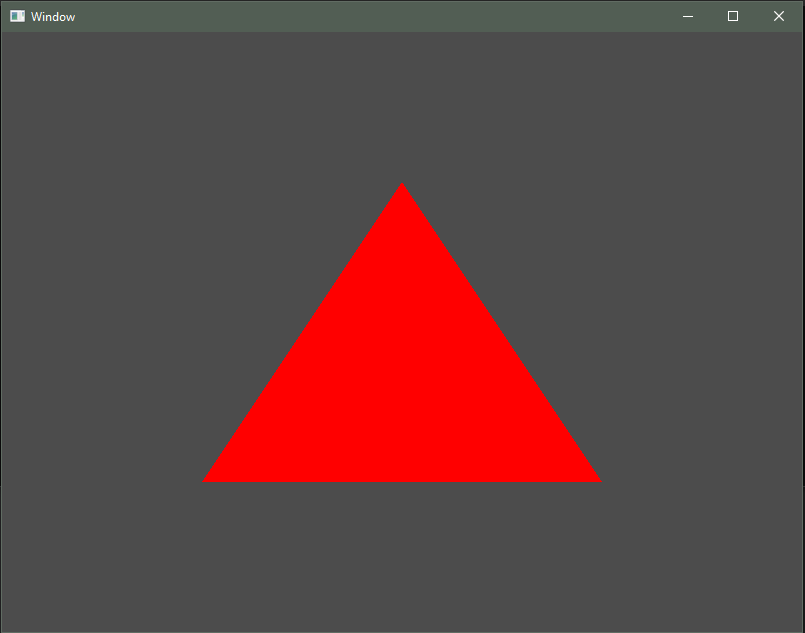
Функция glUseProgram выбирает активную программу (набор шейдеров), которую мы создали ранее. Функция glDrawArrays запускает графический конвейер (визуализирует треугольник). В результате получим:

Рис. 5. Визуализация треугольника

На данном этапе исходный код будет иметь следующий вид:

#include <glad/glad.h>

#include <GLFW/glfw3.h> //Библиотека GLFW

#include <string>

#include <iostream>

//Исходный код вершинного шейдера

std::string vShader = R"glsl(

#version 440 core //Версия GLSL

#line 10

layout(location = 0) in vec2 vPos; //Координаты вершины примитива

void main() {

gl\_Position = vec4(vPos.x, vPos.y, 0.0f, 1.0f);

}

)glsl"

;

//Исходный код пиксельного шейдера

std::string fShader = R"glsl(

#version 440 core //Версия GLSL

#line 22

out vec4 fragColor; //Выходной цвет пикселя

void main() {

fragColor = vec4(1.0f, 0.0f, 0.0f, 1.0f); //Изменение значения выходного цвета пикселя

}

)glsl"

;

static void Check(GLuint obj, bool isProgram = false) {

int status;

char msg[512];

if(!isProgram)

glGetShaderiv(obj, GL\_COMPILE\_STATUS, &status);

else

glGetProgramiv(obj, GL\_LINK\_STATUS, &status);

if (!status) {

if (!isProgram)

glGetShaderInfoLog(obj, 512, 0, msg);

else

glGetProgramInfoLog(obj, 512, 0, msg);

std::cout << msg << std::endl;

}

}

static GLuint CreateProgram() {

GLuint vertex\_shader = glCreateShader(GL\_VERTEX\_SHADER),

pixel\_shader = glCreateShader(GL\_FRAGMENT\_SHADER);

//Конвертируем исходный код из std::string в const char \*

const char\* vertex\_shader\_src = vShader.c\_str();

const char\* fragment\_shader\_src = fShader.c\_str();

glShaderSource(vertex\_shader, 1, &vertex\_shader\_src, 0);

glCompileShader(vertex\_shader);

Check(vertex\_shader); //Проверка компиляции

glShaderSource(pixel\_shader, 1, &fragment\_shader\_src, 0);

glCompileShader(pixel\_shader);

Check(pixel\_shader); //Проверка компиляции

GLuint program = glCreateProgram();

glAttachShader(program, vertex\_shader);

glAttachShader(program, pixel\_shader);

glLinkProgram(program);

Check(program, true); //Проверка компоновки

glDeleteShader(vertex\_shader);

glDeleteShader(pixel\_shader);

return program;

}

GLFWwindow\* window;

int main() {

glfwInit();//Инициализация GLFW

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);//Retained mode

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3); //Младшая версия GL

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3); //Старшая версия GL

//Параметры: ширина, высота, название окна

window = glfwCreateWindow(800, 600, "Window", nullptr, nullptr); //Создание окна

//Позволяет вызывать OpenGL функции для созданного окна

glfwMakeContextCurrent(window);

//Загрузка реализаций функций OpenGL из драйверов

gladLoadGLLoader((GLADloadproc)glfwGetProcAddress);

//Массив вершин

const float vertices[] = {

0.0f, 0.5f,

0.5f, -0.5f,

-0.5f, -0.5f

};

GLuint vao; //Массив вершин (Vertex Array)

glGenVertexArrays(1, &vao); //Сгенерировать идентификатор

glBindVertexArray(vao); //Выбрать массив вершин

GLuint vbo; //Буфер вершин (Vertex Buffer)

glGenBuffers(1, &vbo); //Сгенерировать идентификатор

glBindBuffer(GL\_ARRAY\_BUFFER, vbo); //Выбрать буфер

//Заполнить буфер данными

glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);

//Определить интерпретацию считывания из буфера для идентификатора 0 в шейдере

glVertexAttribPointer(

0, //Идентификатор в вершинном шейдере

2, //Количество значений передаваемых по идентификатору

GL\_FLOAT, //Тип данных

GL\_FALSE, //Нужна ли нормализация (от 0 до 1)

sizeof(GL\_FLOAT) \* 2, //Шаг считывания из буфера

(void\*)0 //Сдвиг в буфере

);

//Разрешить передачу данных по нулевому идентификатору

glEnableVertexAttribArray(0);

GLuint program = CreateProgram();

//Пока окно не должно быть закрытым

while (!glfwWindowShouldClose(window))

{

glClear(GL\_COLOR\_BUFFER\_BIT); //Очистить экран

glClearColor(0.3, 0.3, 0.3, 1.0f); //Цвет очистки экрана

glUseProgram(program); //Выбираем активную программу

glBindVertexArray(vao); //Выбираем активный массив вершин

glDrawArrays(GL\_TRIANGLES, 0, 3); //Отрисовать три вершины

//Чередовать буферы с изображением

glfwSwapBuffers(window);

//Обработать события ввода

glfwPollEvents();

}

//Освободить ресурсы

glfwDestroyWindow(window);

glfwTerminate();

return 0;

}

Дополним вершинный шейдер и пиксельный шейдеры для передачи цвета с помощью вершинного буфера:

#version 440 core //Версия GLSL

#line 10

layout(location = 0) in vec2 vPos; //Координаты вершины примитива

layout(location = 1) in vec3 vColor; //Цвет вершины примитива

//Выходные данные вершинного шейдера

out VS\_OUT{

vec3 outColor;

}vs\_out;

void main() {

vs\_out.outColor = vColor;

gl\_Position = vec4(vPos.x, vPos.y, 0.0f, 1.0f);

}

Вершинный шейдер

#version 440 core //Версия GLSL

#line 29

//Входные данные пиксельного шейдера

in VS\_OUT{

vec3 outColor;

}fs\_in;

out vec4 fragColor; //Выходной цвет пикселя

void main() {

fragColor = vec4(fs\_in.outColor.r, fs\_in.outColor.g, fs\_in.outColor.b, 1.0f); //Изменение значения выходного цвета пикселя

}

Пиксельный шейдер

В дополненном вершинном шейдере появилась новая входная переменная: vColor. Эта переменная хранит цвет вершины, однако чтобы применить ее, необходимо передать переменную в пиксельный шейдер. С этой целью используется структура-интерфейс VS\_OUT, которая содержит поле outColor. Структура VS\_OUT объявляется выходной с помощью ключевого слова out.

В обновленном пиксельном шейдере происходит прием переменной из вершинного шейдера. Структура VS\_OUT становится входной, поэтому используется ключевое слово in. Далее полученное значение цвета применяется к выходной переменной fragColor.

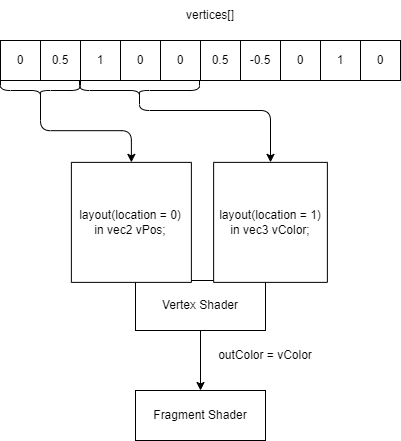


Рис. 6. Иллюстрация работы шейдеров

На рис. 6 показана наглядная иллюстрация к ранее описанным изменениям в шейдерах.

Теперь необходимо ввести изменения в код создания вершинного буфера и вершинного массива:

...

//Массив вершин

const float vertices[] = {

//координаты //цвет

0.0f, 0.5f, 1, 0, 0,

0.5f, -0.5f, 0, 1, 0,

-0.5f, -0.5f, 0, 0, 1

};

GLuint vao; //Массив вершин (Vertex Array)

glGenVertexArrays(1, &vao); // Сгенерировать идентификатор

glBindVertexArray(vao); //Выбрать массив вершин

GLuint vbo; //Буфер вершин (Vertex Buffer)

glGenBuffers(1, &vbo); // Сгенерировать идентификатор

glBindBuffer(GL\_ARRAY\_BUFFER, vbo); //Выбрать буфер

//Заполнить буфер данными

glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);

//Определить интерпретацию считывания из буфера для идентификатора 0 в шейдере glVertexAttribPointer(0, 2, GL\_FLOAT, GL\_FALSE, sizeof(GL\_FLOAT) \* 5, (void\*)0);

//Определить интерпретацию считывания из буфера для идентификатора 1 в шейдере

glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, sizeof(GL\_FLOAT) \* 5, (void\*)(sizeof(float) \* 2));

//Разрешить передачу данных по нулевому идентификатору

glEnableVertexAttribArray(0);

glEnableVertexAttribArray(1);

...

Создание вершинного массива и вершинного буфера

Для каждой вершины (в массиве vertices) добавились еще по три значения – значения каналов цвета вершин. В следствие изменений необходимо доопределить интерпретацию считывания из буфера. Теперь шаг считывания будет равен 5 \* размер float в байтах. По первому идентификатору теперь идет передача значения цвета вершины. Чтобы получить три значения цвета необходимо сделать сдвиг на 2 значения с плавающей точкой в буфере. Рис. 7 демонстрирует описанный процесс.

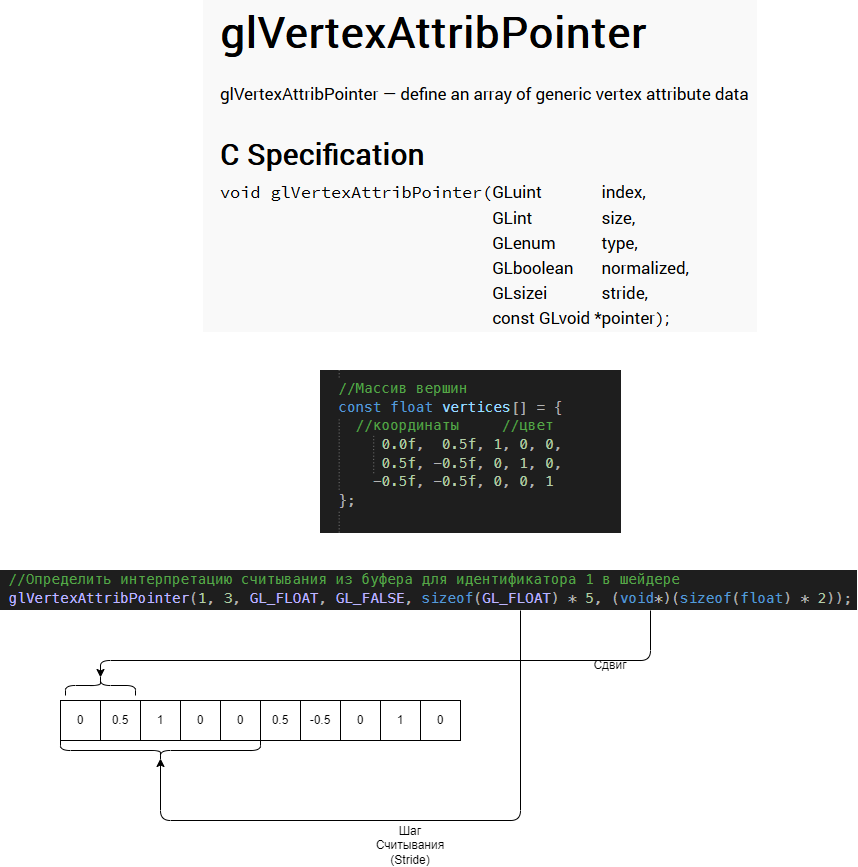


Рис. 7. Демонстрация использования функции glVertexAttribPointer

В результате проделанных операций имеем следующий результат:

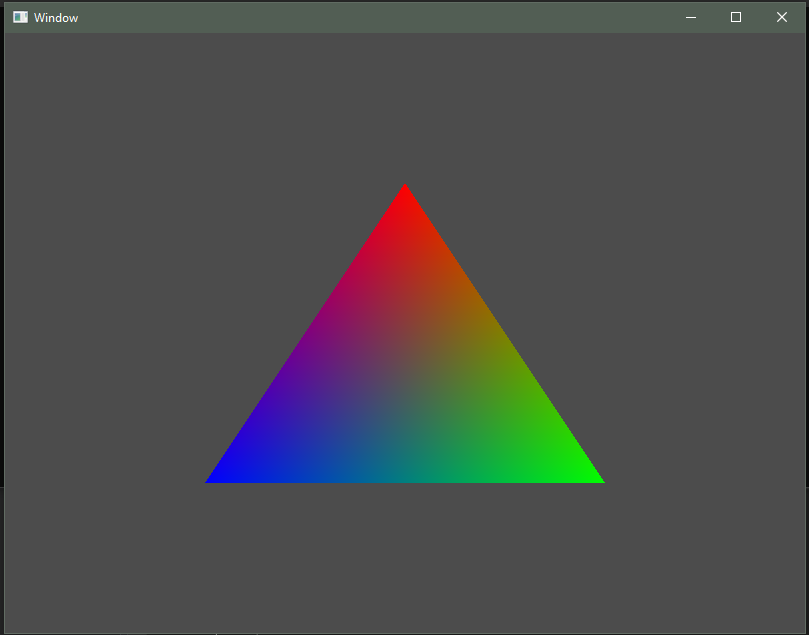


Рис. 8. Визуализация треугольника

Проделав минимальные изменения в коде можем визуализировать прямоугольник:

...

//Массив вершин

const float vertices[] = {

//координаты //цвет

0.5f, 0.5f, 1, 0, 0, //------------

0.5f, -0.5f, 0, 1, 0, //Трегольник 1

-0.5f, -0.5f, 0, 0, 1, //------------

-0.5f, -0.5f, 1, 0, 0, //------------

-0.5f, 0.5f, 0, 1, 0, //Треугольник 2

0.5f, 0.5f, 0, 0, 1, //------------

}; ...

//Пока окно не закрыто

while (!glfwWindowShouldClose(window))

{

...

glDrawArrays(GL\_TRIANGLES, 0, 6); //Отрисовать шесть вершин

...

}

...

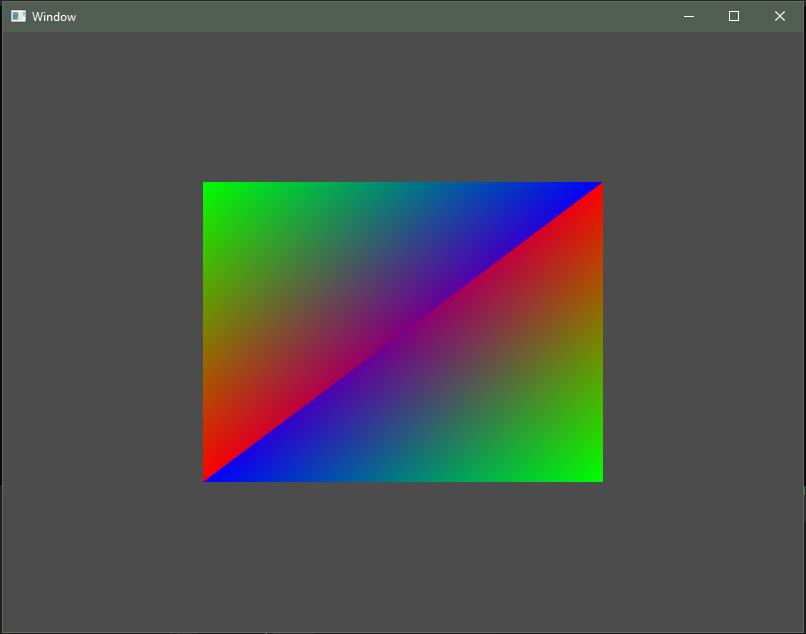
Модификация массива vertices

Рис. 9. Визуализация прямоугольника

Каждые три вершины формируют треугольник. Вершины в OpenGL соединяются линиями по часовой стрелке (или против). Вершины на рис. 10 соединяются в следующем порядке: 0, 1, 2, 2, 3, 0.

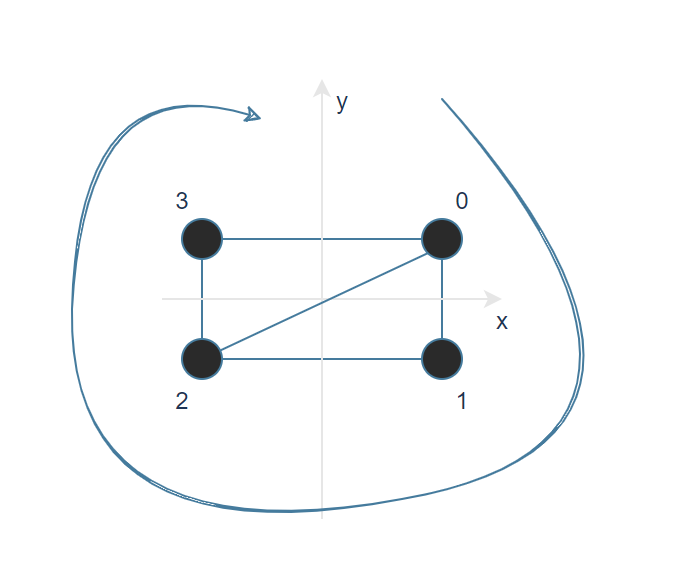


Рис. 10. Порядок соединения вершин прямоугольника

Добавим в вершинный шейдер uniform переменную model и умножим координаты вершины на эту матрицу. Uniform переменная – переменная, которую можно выставить с помощью программного интерфейса.

...

layout(location = 1) in vec3 vColor; //Цвет вершины примитива

uniform mat4 model; //Матрица преобразований

...

void main() {

...

gl\_Position = model \* vec4(vPos.x, vPos.y, 0.0f, 1.0f);

}

...

*Вершинный шейдер*

Чтобы передать матрицу из С++ необходимо воспользоваться интерфейсом OpenGL. В теоретической справке было упомянуто как реализуются геометрические преобразования. Чтобы производить векторные и матричные операции воспользуемся библиотекой GLM (OpenGL Mathematics).

*Передача матрицы преобразований в шейдер*

...

GLuint program = CreateProgram();

//Матрица переноса

glm::mat4 translationMat = glm::translate(glm::mat4(1.0f), glm::vec3(0.2, 0.2, 0));

//Матрица масштабирования

glm::mat4 scaleMat = glm::scale(glm::mat4(1.0f), glm::vec3(0.5, 0.5, 0));

//Матрица вращения

glm::mat4 rotateMat = glm::rotate(glm::radians(45.f), glm::vec3(0, 0, 1));

//Матрица преобразований

glm::mat4 modelMat = translationMat \* rotateMat \* scaleMat;

//Пока окно закрыто

while (!glfwWindowShouldClose(window))

{

//Передача матрицы преобразований в шейдер

glUseProgram(program);

glUniformMatrix4fv(

glGetUniformLocation(program, "model"),

1,

GL\_FALSE,

glm::value\_ptr(modelMat)

);

...

В первую очередь сформируем матрицы переноса, вращения, масштабирования. Как в функцию glm::translate так и в функцию glm::scale первым параметром передается диагональная матрица со значениями равными единице по диагонали. Во втором параметре функций glm::translate и glm::scale передаются вектора, содержащие значения переноса/масштабирования. Функция glm::rotate формирует матрицу вращений на основе угла вращения и оси, вокруг которой происходит вращение.

Функция glUniformMatrix4fv позволяет передать матрицу в шейдер. Первым аргументом является идентификатор переменной-состояния. Идентификатор можно получить с помощью функции glGetUniformLocation, указав в втором аргументе название переменной-состояния.

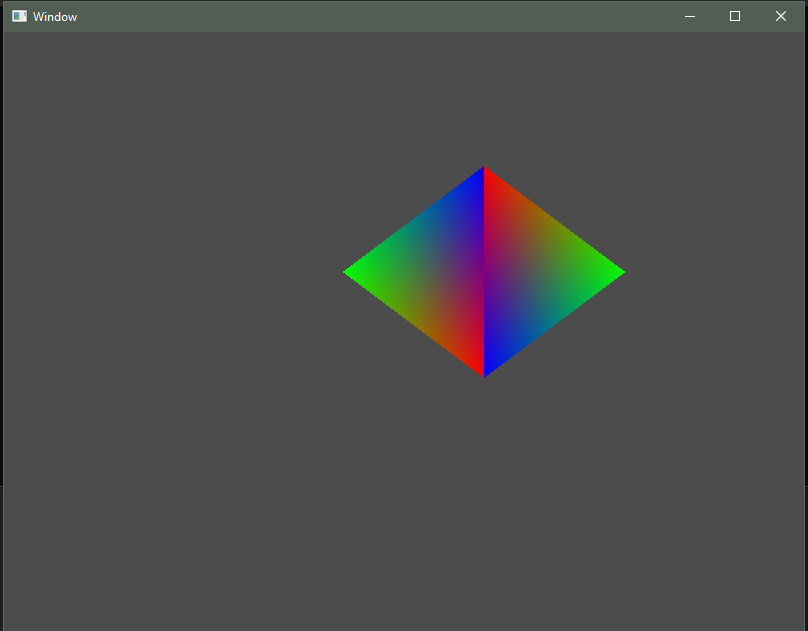


Рис. 11. Применение геометрических преобразований к прямоугольнику

*Подготовка к работе*

1. Загрузить и скомпилировать исходный код программы примера.

*Выполнение работы*

|  |  |
| --- | --- |
| Вариант | Задание |
| 1 | Модифицировать подпрограмму загрузки шейдера добавив возможность считывания исходного кода шейдера из файла (использовать std::fstream). |
| 2 | Разработать класс для хранения и управления графическим объектом. |
| 3 | Реализовать анимацию изменения цвета фона сцены с помощью std::chrono. Использовать переменную DeltaTime. |
| 4 | Реализовать анимацию вращения двухмерного примитива с помощью std::chrono. Использовать переменную DeltaTime. |
| 5 | Реализовать перемещение объекта с помощью клавиатуры. Добавить ускорение при нажатии определенной клавиши. |
| 6 | Разработать подпрограмму для создания вершинного массива (Vertex Array Object), добавив в качестве входных данных массив из параметров вершин. |
| 7 | Вывести на экран два примитива. |
| 8 | Вывести многоугольник на экран. |
| 9 | Реализовать подпрограмму загрузки параметров вершин из файла. |
| 10 | Модифицировать пиксельный шейдер использовав состояние color для задания цвета примитива. Для изменения значения состояния использовать функции glGetUniformLocation и glUniform3f. |