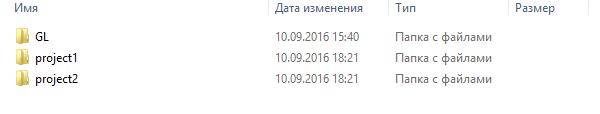
**Цель лабораторной работы:**

1) Нарисовать треугольник в OpenGL приложении используя Vertex Array Object(VAO)

2) Реализовать простейший фрагментный и вершинный шейдер для закраски треугольника в красный цвет.

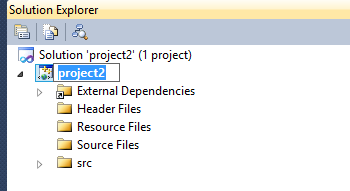
**Порядок исполнения:**

1) Создайте копию папку проектом лабараторной работы 1 - project1 и переименуйте её в project2



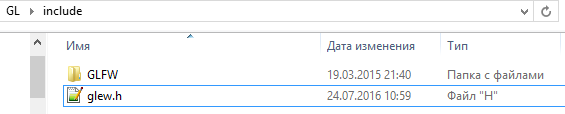
2) В новой папке project2 переименуйте файл project1.sln в project2.sln

3) Переименуйте проект в студии нажав F2 на выделенном проекте:

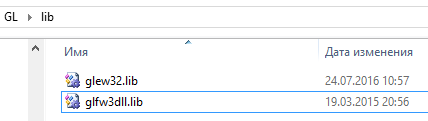


4) Следующим шагом будет подключение библиотеки GLEW которая позволит использовать расширения OpenGL для создания VAO. То есть современного механизма OpenGL для рендеринга геометрии используя вершинные буферы.

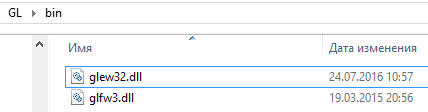
4.1) Из архива " glew-2.0.0-win32.zip" скопируйте файл glew.h, он лежит в папке include\ GL. Этот файл нужно положить в папку GL\include, по аналогии как это выполнялось на 1 лабораторной работе:



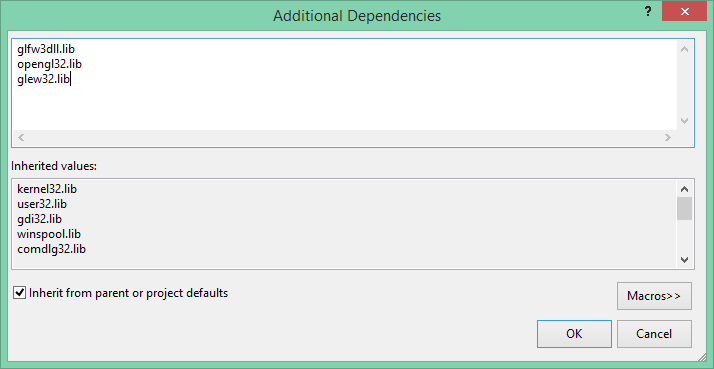
4.2) Из архива " glew-2.0.0-win32.zip" скопируйте файл glew32.lib, он лежит в папке lib\Release\Win32. Этот файл нужно положить в папку GL\lib, по аналогии как это выполнялось в пункте 4.1:



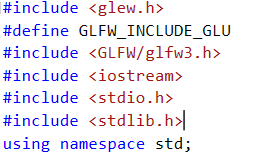
4.3) Из архива " glew-2.0.0-win32.zip" скопируйте файл glew32.dll, он лежит в папке bin\Release\Win32. Этот файл нужно положить в папку GL\bin, по аналогии как это выполнялось в пункте 4.2:

****

4.4) Далее нужно прописать glew32.lib в настройках проекта, по аналогии с тем как это выполнялось для библиотеки glfw3dll.lib в лабораторной работе 1:



4.5) Прописать #include <glew.h> в файле app.h перед включением #include <GLFW/glfw3.h>:



Таким образом к проекту была подключена GLEW - библиотека расширений OpenGL.

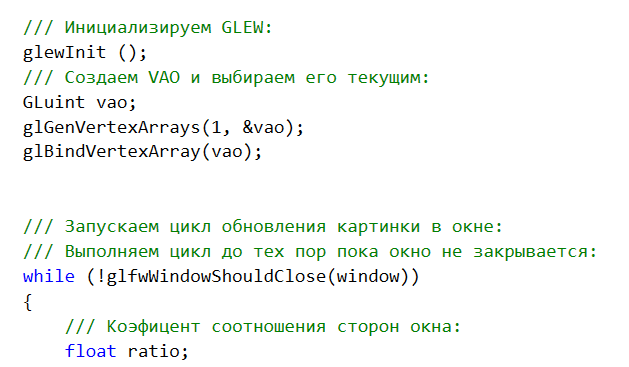
5) Далее выполняем инциализацию GLEW - в файле app.cpp, перед циклом while (!glfwWindowShouldClose(window)) вызываем функцию glewInit ();

6) Сразу после этого выполняем создание VAO- выделяем ID буфера и выполняем привязку этого буфера, переменная в этом случае хранит ссылку на данный буфер OpenGL:

GLuint vao;

glGenVertexArrays(1, &vao);

7) Затем вызываем - glBindVertexArray(vao); Функция glBindVertexArray "выбирает" (делает текущим) VAO с заданным идентификатором. После этого все команды, задающие параметры вершинных буферов, изменяют состояние хранимое в данном VAO. Кроме того именно данный VAO и определяет текущие настройки вершинных буферов. Таким образом у вас должен получится следующий код:



8) Теперь займемся вопросом координат треугольника который мы хотим отрисовать в нашем приложении.

Треугольник можно задать тремя точками. Когда мы говорим про точки в 3д пространстве, мы обычно употребляем термин «вершина». Вершина содержит 3 координаты: X,Y и Z. Вы можете представить для себя эти координаты так:

* X - вправо
* Y - вверх
* Z - за спину

Но есть еще более наглядный способ представить себе эту систему координат: правило правой руки.

* X – большой палец правой руки
* Y – указательный палец правой руки
* Z – средний палец правой руки.

Теперь если вы направите свой большой палец направо, указательный вверх, то средний будет показывать на вас.

Почему Z такое странное? Очень просто: просто за стони лет существования математики и правила правой руки, было создано много методов облегчающих расчеты которые ориентируются на этот неинтуитивный Z. И так, нам нужно три точки в 3д пространстве, чтобы нарисовать треугольник:

*/// Массив из 3 векторов которые будут представлять 3 вершины*  
*static const GLfloat g\_vertex\_buffer\_data[] = {*

*-1.0f, -1.0f, 0.0f,*

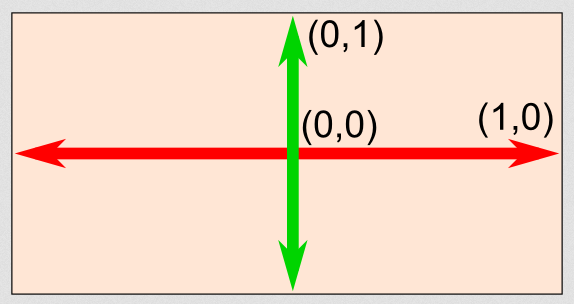
*1.0f, -1.0f, 0.0f,*

*0.0f, 1.0f, 0.0f,*

*};*

Вставьте этот код сразу после создания и выбора VAO.

Первая вершина(-1,-1,0). Это значит, что если мы не перетрансформируем координаты как-нибудь, точка будет на экране в позиции (-1,-1). Что значит -1,-1? Видеокарта трактует экранные координаты так, что центр экрана это 0, а крайние точки: -1 и 1 по X и Y. Вот что у нас получается на широкоформатном мониторе:



Неудобно, да, но поменять это нельзя. Поэтому наш треугольник будет расположен по таким координатам: (-1,-1) это нижний левый угол. (1,-1) – нижний правый. (0,1) – верхняя центральная точка.

9) Следующим шагом мы должны будем передать указанный массив вершин gVertexBufferData

в OpenGL. Для этого необходимо создать вершинный буфер которому мы передадим координаты:

/// Идентификатор вершинного буфера:

GLuint vertexBuffer;

/// Сначала генерируем OpenGL буфер и сохраняем указатель на него в vertexBuffer:

glGenBuffers(1, &vertexBuffer);

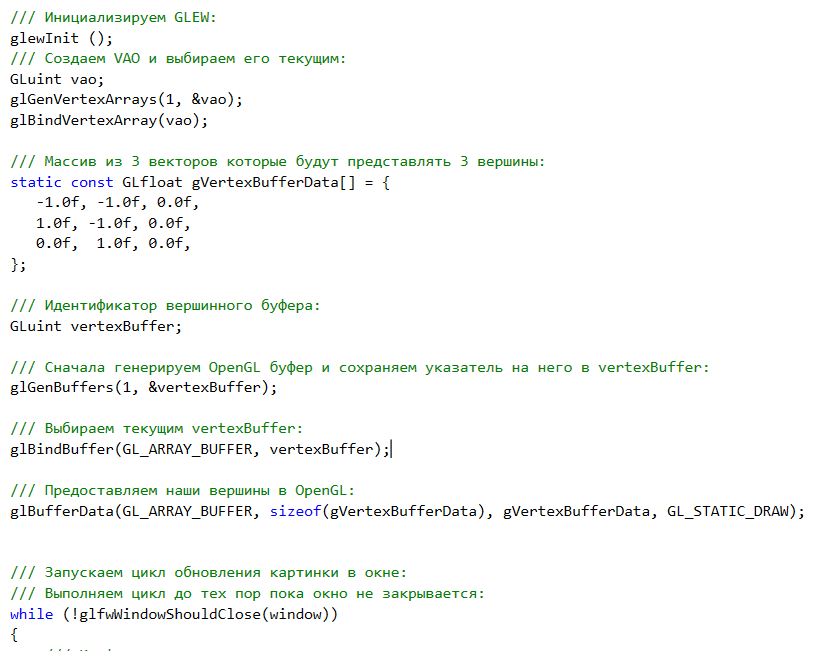
/// Выбираем текущим vertexBuffer:

glBindBuffer(GL\_ARRAY\_BUFFER, vertexBuffer);

/// Предоставляем наши вершины в OpenGL:

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

Таким образом в редакторе студии это выглядит так:



Описанный действия с 5-9 шаг нужно выполнить один раз.

10) На следующем шаге выполняем визуализацию вершинного буфера, для этого сперва закомментируем код из 1 лабораторной работы в цикле while (!glfwWindowShouldClose(window))

визуализации, оставив только последние 2 функции необходимые для нормальной работы приложения:



11) Затем добавим код который нарисует содержимое созданного нами вершинного буфера:

*/// Первый буфер атрибутов: вершины*

*glEnableVertexAttribArray(0);*

*/// Выбираем текущий вершинный буфер:*

*glBindBuffer(GL\_ARRAY\_BUFFER, vertexBuffer);*

*glVertexAttribPointer(*

*0, // Атрибут 0. Сакрального смысла в нуле нет, но число должно совпадать с числом в шейдере*

*3, // количество*

*GL\_FLOAT, // тип*

*GL\_FALSE, // нормализировано ли?*

*0, // шаг*

*(void\*)0 // смещение в буфере*

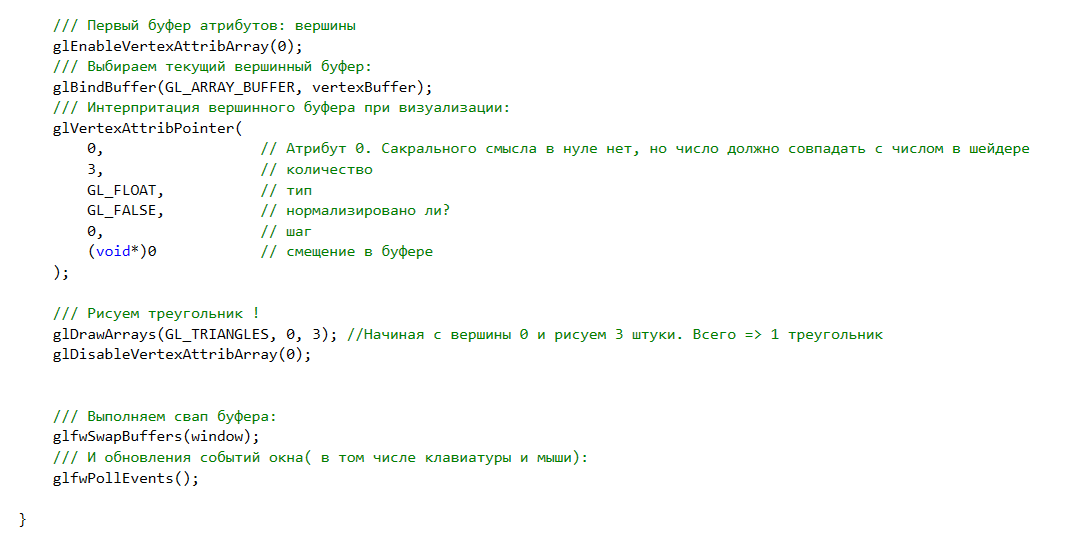
*);*

*/// Рисуем треугольник !*

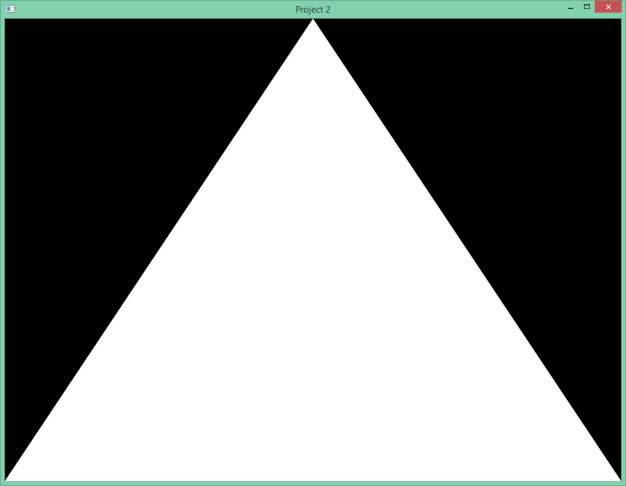
*glDrawArrays(GL\_TRIANGLES, 0, 3); //Начиная с вершины 0 и рисуем 3 штуки. Всего => 1 треугольник*

*glDisableVertexAttribArray(0);*

В редакторе студии это выглядит следующим образом:



Так выглядит запущенная программа с визуализацией треугольника:



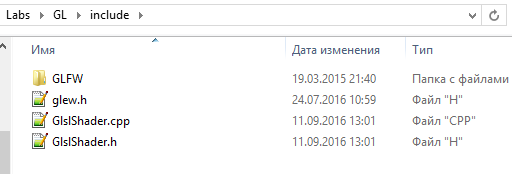
12) Теперь займемся механизмом раскрашивания нарисованного треугольника при помощи шейдера. В самом простом варианте нам понадобится два шейдера:

* Вершинный Шейдер. Это шейдер который будет вызываться для каждой вершины.
* Фрагментный Шейдер. Это шейдер который вызывается при растеризации на каждый пиксель. Если у нас 4х сглаживание (glfwWindowHint(GLFW\_SAMPLES,4);), то фрагментный шейдер вызовется 4 раза на каждый пиксель.

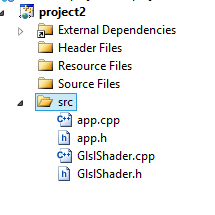
Шейдеры в OpenGL программируются на языке GLSL: GL Shader Language, который есть частью OpenGL. В отличии от С или Java,GLSL компилируется программой при исполнении, что значит, что при каждом старте приложения ваши шейдеры будут перекомпилироваться. Обычно разные шейдеры хранят в разных файлах. Поэтому и у нас будет два файла "simpleShader.frag"

и "simpleShader.vert". Расширение файла не важно, вы можете писать хоть .txt хоть .glsl. Поскольку использование шейдеров предполагается в следующих лабораторных работах, то для этого мы создадим отдельный класс GlslShader который будет заниматься загрузкой и компиляции шейдеров.

13) Для этого создадим в папке GL\include два файла – GlslShader.cpp и GlslShader.h :



Эти 2 файла должны быть также расположены в фильтре src вашего проекта:



14) теперь в пустом файле GlslShader.cpp добавьте в самом начале строчку – #include "GlslShader.h",а в файле пустом файле GlslShader.h добавьте следующее:

#pragma once

#include <glew.h>

#define GLFW\_INCLUDE\_GLU

#include <GLFW/glfw3.h>

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

#include <fstream>

#include <string>

#include <windows.h>

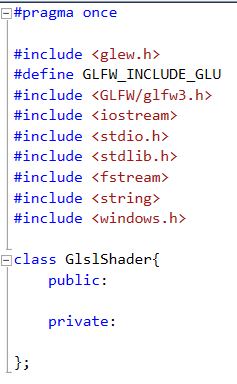
Это необходимые файлы для того чтобы можно было использовать шейдер и загрузить его с учетом пути относительно запускаемого приложения. После этих строчек кода вставьте этот код, который будет описанием класса будущего шейдера:

class GlslShader{

public:

private:

};



Затем в описание этого добавим заголовки функций. А именно в "public:" добавим 2 функции:

/// Вызвав эту функцию мы начинаем применять данный шейдер в отрисовке:

void use();

/// Основная функнция загрузки и компиляции вершинного и фрагментного шейдера и линковки в одну шейдерную программу GL:

void setup(string vertexShaderPath, string fragmentShaderPath);

а в "private:" это:

/// Зарегистрированный ID нашей шейдерной программы в GL, или проще говоря идентификатор:

GLuint programID;

/// Получить путь к расположению файла нашего приложения вместе с именем exe:

string exePathWithName();

/// Тоже самое но уже без имени файла:

string exePath();

Также не забудем добавить

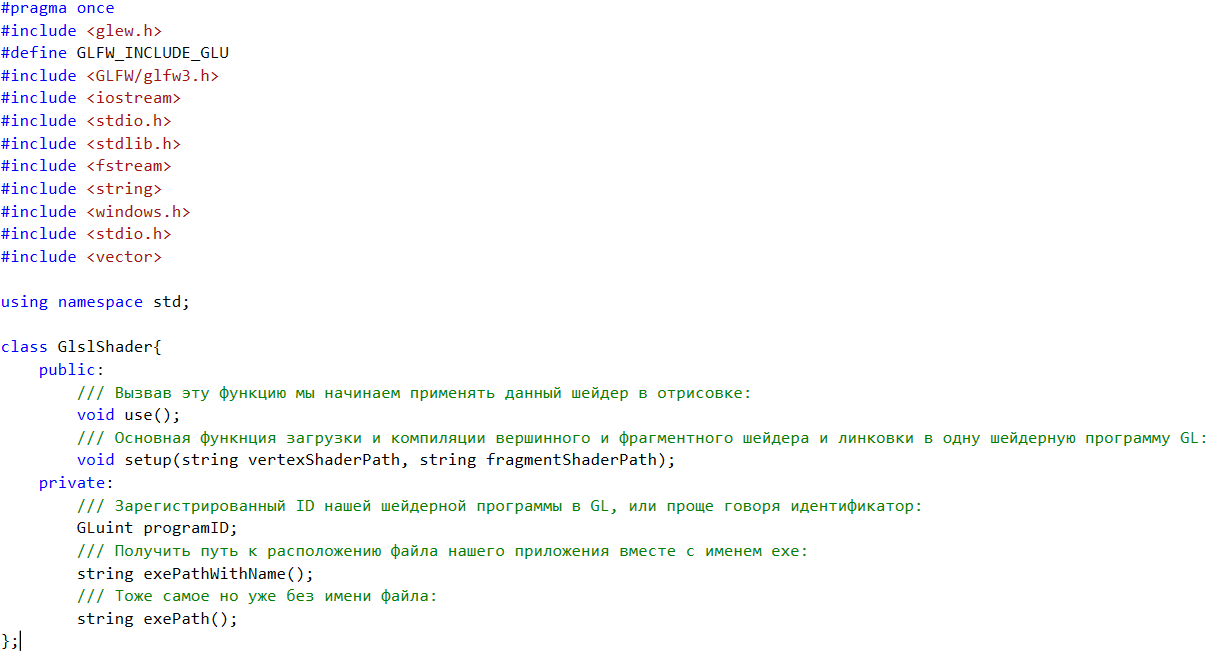
#include <stdio.h>

using namespace std;

#include <vector>

Для доступа к стандартным функциями C++

В итоге в "GlslShader.h" файле мы получим следующее:



15) Теперь в файле "GlslShader.cpp" добавим реализацию описанных функций.

15.1) Сперва добавим вспомогательные функции:

/// Получить путь к exe вместе с именем файла.

string GlslShader::exePathWithName() {

char buffer[MAX\_PATH];

GetModuleFileNameA( NULL, buffer, MAX\_PATH );

return string( buffer );

}

/// Получить путь к exe вместе с без имени файла.

string GlslShader::exePath() {

string buffer = exePathWithName();

string::size\_type pos = buffer.find\_last\_of( "\\/" );

return string( buffer ).substr( 0, pos);

}

15.2) Затем добавим основную функцию данного класса:

/// Основная функнция загрузки и компиляции вершинного и фрагментного шейдера и линковки в одну шейдерную программу GL:

void GlslShader::setup(string vertexShaderPath, string fragmentShaderPath){

/// Создаем шейдеры:

GLuint vertexShaderID = glCreateShader(GL\_VERTEX\_SHADER);

GLuint fragmentShaderID = glCreateShader(GL\_FRAGMENT\_SHADER);

/// Читаем вершинный шейдер из файла:

string vertexShaderCode;

vertexShaderPath = exePath() + "\\" + vertexShaderPath;

ifstream vertexShaderStream(vertexShaderPath.c\_str(), std::ios::in);

if(vertexShaderStream.is\_open()){

std::string Line = "";

while(getline(vertexShaderStream, Line))

vertexShaderCode += "\n" + Line;

vertexShaderStream.close();

}

/// Читаем фрагментный шейдер из файла:

string fragmentShaderCode;

fragmentShaderPath = exePath() + "\\" + fragmentShaderPath;

ifstream fragmentShaderStream(fragmentShaderPath.c\_str(), std::ios::in);

if(fragmentShaderStream.is\_open()){

std::string Line = "";

while(getline(fragmentShaderStream, Line))

fragmentShaderCode += "\n" + Line;

fragmentShaderStream.close();

}

GLint result = GL\_FALSE;

int infoLogLength;

/// Компилируем вершинный шейдер:

cout << "Compiling shader : " << vertexShaderPath << endl;

char const \* vertexSourcePointer = vertexShaderCode.c\_str();

/// Передаем код шейдера в OpenGL:

glShaderSource(vertexShaderID, 1, &vertexSourcePointer , NULL);

/// Собственно компиляция:

glCompileShader(vertexShaderID);

/// Устанавливаем параметры для возможности получения ошибок в текстовой консоли:

glGetShaderiv(vertexShaderID, GL\_COMPILE\_STATUS, &result);

glGetShaderiv(vertexShaderID, GL\_INFO\_LOG\_LENGTH, &infoLogLength);

vector<char> vertexShaderErrorMessage(infoLogLength);

glGetShaderInfoLog(vertexShaderID, infoLogLength, NULL, &vertexShaderErrorMessage[0]);

fprintf(stdout, "%s\n", &vertexShaderErrorMessage[0]);

/// Компилируем фрагментный шейдер:

printf("Compiling shader : %s\n", fragmentShaderPath);

char const \* fragmentSourcePointer = fragmentShaderCode.c\_str();

/// Передаем код шейдера в OpenGL:

glShaderSource(fragmentShaderID, 1, &fragmentSourcePointer , NULL);

glCompileShader(fragmentShaderID);

/// Устанавливаем параметры фрагментного шейдера для возможности получения ошибок в текстовой консоли:

glGetShaderiv(fragmentShaderID, GL\_COMPILE\_STATUS, &result);

glGetShaderiv(fragmentShaderID, GL\_INFO\_LOG\_LENGTH, &infoLogLength);

std::vector<char> fragmentShaderErrorMessage(infoLogLength);

glGetShaderInfoLog(fragmentShaderID, infoLogLength, NULL, &fragmentShaderErrorMessage[0]);

fprintf(stdout, "%s\n", &fragmentShaderErrorMessage[0]);

/// Создаем шейдерную программу из фрагментного и вершинного шейдера:

fprintf(stdout, "Linking program\n");

programID = glCreateProgram();

/// Линкуем шейдеры в программу:

glAttachShader(programID, vertexShaderID);

glAttachShader(programID, fragmentShaderID);

glLinkProgram(programID);

/// Устанавливаем параметры для programID для возможности получения ошибок в текстовой консоли:

glGetProgramiv(programID, GL\_LINK\_STATUS, &result);

glGetProgramiv(programID, GL\_INFO\_LOG\_LENGTH, &infoLogLength);

std::vector<char> programErrorMessage( max(infoLogLength, int(1)) );

glGetProgramInfoLog(programID, infoLogLength, NULL, &programErrorMessage[0]);

fprintf(stdout, "%s\n", &programErrorMessage[0]);

/// Удаляем локальные фрагментные и вершинные шейдеры:

glDeleteShader(vertexShaderID);

glDeleteShader(fragmentShaderID);

}

15.3) Добавим функцию use():

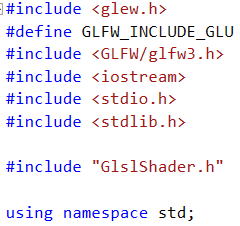
/// Вызвав эту функцию мы начинаем применять данный шейдер в отрисовке:

void GlslShader::use(){

glUseProgram(programID);

}

16) Теперь все готово чтобы можно было начать использовать данный класс в нашем приложении. В конце файле app.h прописываем #include "GlslShader.h" это должно выглядеть так:



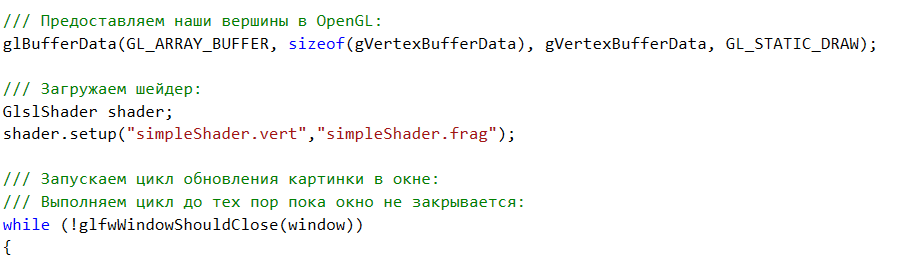
17) Теперь в app.cpp сразу перед циклом обновления while пишем следующие:

/// Загружаем шейдер:

GlslShader shader;

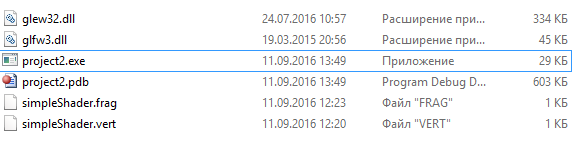
shader.setup("simpleShader.vert","simpleShader.frag");

тем самым мы объявляем переменную-экземпляр класса GlslShader которая и будет шейдерной программой в нашем приложении. В коде это должно выглядеть так:

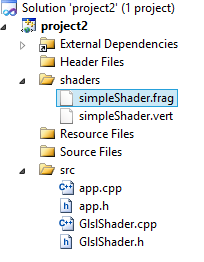


Теперь можно приступить собственно к написанию вершинного и фрагментного шейдера.

Для этого в папке где лежит наш бинарный(exe) файл приложения создадим два пустых текстовых файла – "simpleShader.vert","simpleShader.frag":



18) Теперь создадим фильтр shaders куда добавим наши созданные пустые файлы где потом будут находится фрагментный и вершинный шейдер:



19) Займемся написанием вершинного шейдера simpleShader.vert. Первая строчка будет указанием компилятору, что мы хотим использовать синтаксис OpenGL 3.3:

#version 330 core

Вторая строчка объявляет входные данные:

layout(location = 0) in vec3 vertexPosition\_modelspace;

Тут нужно объяснение поподробнее:

“vec3” это вектор состоящий из трех компонентов в GLSL.

“layout(location=0)” – указание для компилятора, в каком из буферов будет находится атрибут vertexPosition\_modelspace. Каждая вершина может иметь множество атрибутов: Позиция, цвета, текстурные координаты, и еще кучу других вещей. GLSL не знает ничего про цвет, он видит лишь vec3. А наша задача – рассказать, что из себя представляет каждый входящий буфер. Это мы делаем устанавливая layout в то же самое значение, что и glVertexAttribPointer. Мы там поставили «0», но с таким же успехом могли поставить и 12(но это число не может быть больше чем glGetIntegerv(GL\_MAX\_VERTEX\_ATTRIBS, &v)). Самое главное, чтобы эти числа были одинаковы по обе стороны. «vertexPosition\_modelspace» - идентификатор «переменной». Можете назвать как угодно. В нашем случае будет содержать положение вершины которая обрабатывается в данный момент вершинным шейдером. «in» значит, что это входящий параметр. Вскоре у нас будет и «out».Точно так же как и в С++ в GLSL есть функция main:

void main(){

Тут все проще простого. Мы просто устанавливаем положение вершины в значение которое пришло к нам из буфера. Так что если нам пришло (1,1) одна из вершин треугольника будет в верхнем правом углу экрана. В последующих лабораторных мы рассмотрим, что в этом шейдере можно создавать сложные визуальные эффекты основанные на изменении положении вершин, например анимация воды, ткани, травы и всего прочего что состоит из множества вершин которые можно перемещать:

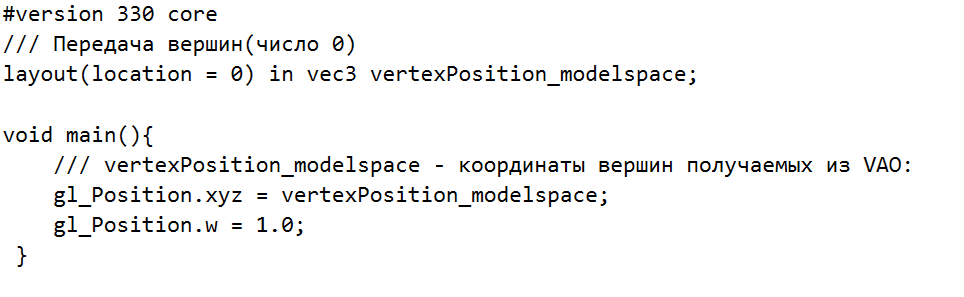
gl\_Position.xyz = vertexPosition\_modelspace;

gl\_Position.w = 1.0;

}

gl\_Position – это одна из встроенных переменных и мы просто обязаны установить какое-то значение в неё.

В конечном счете итоговый код в вершинном шейдере должен выглядеть следующим образом:



20) В нашем первом фрагментном шейдере simpleShader.frag мы выполним очень простую операцию: установим цвет каждого фрагмента в красный цвет(не забудьте, что так как у нас 4х сглаживание, то на каждый пиксель у нас будет 4 фрагмента):

*#version 330 core*

*out vec3 color;*

*void main(){*

*color = vec3(1,0,0);*

*}*

Да, vec3(1,0,0) – это красный цвет. Так происходит потому, что компьютер видит цвет, как совокупность Красного, Зеленого и Голубого. Поэтому (1,0,0) это Полностью Красный, нет ни зеленого ни голубого.

21) Таким образом код пиксельного и вершинного шейдера написан, сами шейдеры загружаются и компилируются, осталось только применить наш шейдер к рисуемому треугольнику. Но сперва давайте зададим отличный от черного фон:

вот этими строчками в цикле while() в файле app.cpp:

/// Выполняем очистки буфера цвета:

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

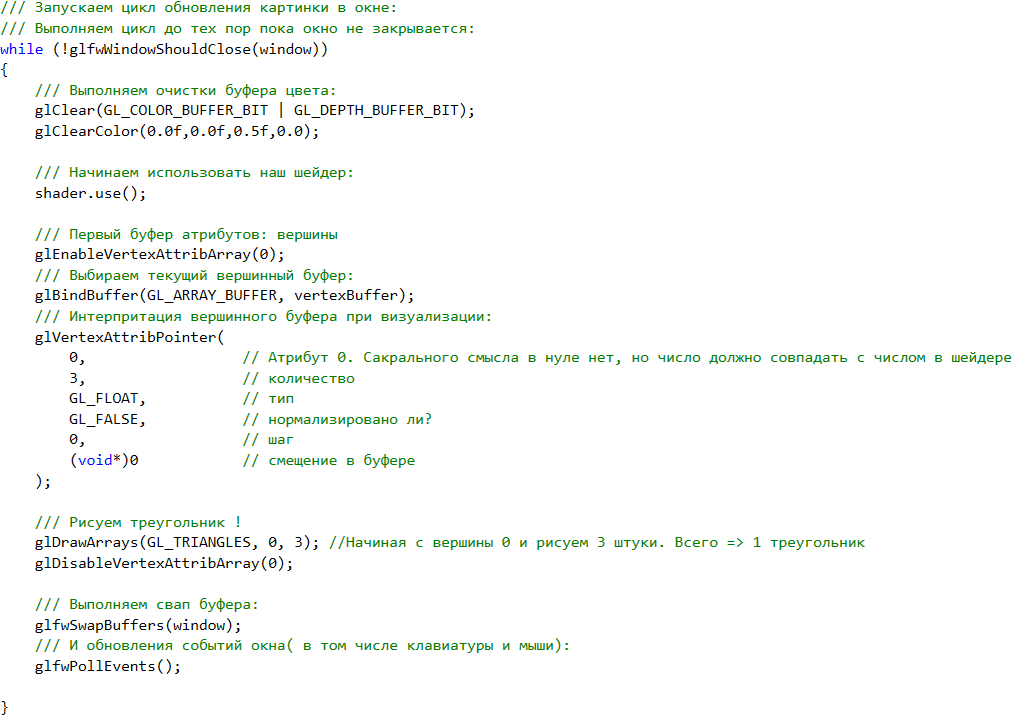
glClearColor(0.0f,0.0f,0.5f,0.0);

22) Остался последний шаг -

/// Начинаем использовать наш шейдер:

shader.use();

В коде это должно выглядеть так:



Если все было сделано верно, то при запуске приложения мы увидим раскрашенный шейдером красный треугольник на синем фоне:

