**Цель лабораторной работы:**

1) Выполнить рефакторинг приложения полученного в 3 лабораторной работе для более удобной работы в последствии, улучшении читабельности кода и в целом лучшего понимания процессов OpenGL. Основа рефакторинга - создание класса "app" - нового каркаса приложения который мы будем использовать в дальнейшем.

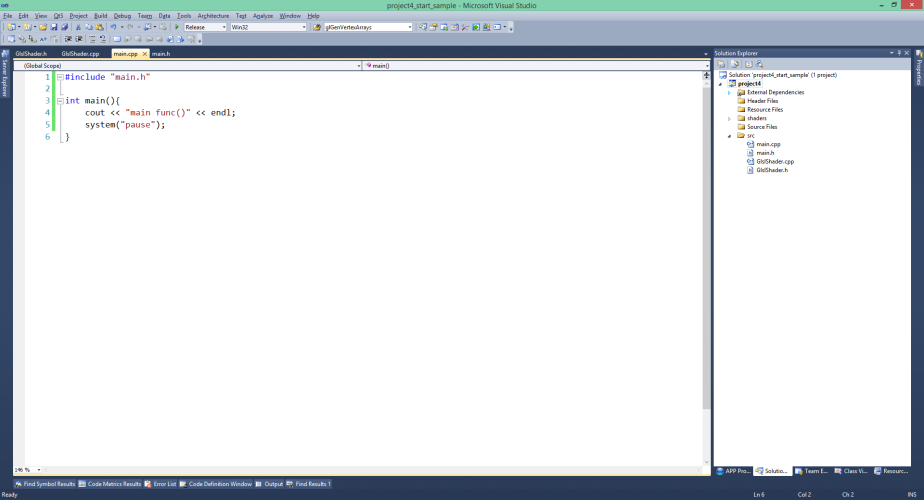
2) Разработать виртуальную камеру, которая позволит перемещаться в пространстве. Управление камерой будет выполнятся привычным образом - движение клавиши WASD, движение мышки - поворот камеры.

3) Вместо треугольника в пространстве создадим немного более сложную фигуру - разноцветный куб. Вокруг которого и будем летать на камере.

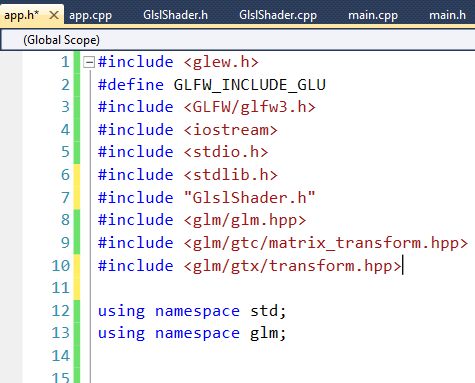
**Порядок исполнения:**

1. **Рефакторинг**

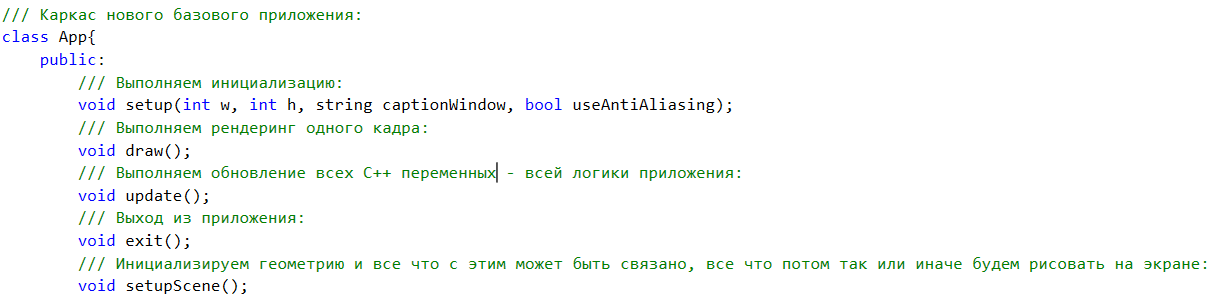
Начнем с рефакторинга. Переместите project4\_start\_sample в папку где лежит GL. По аналогии переименуйте папку этого проекта, файл .sln и название проекта в студии в project4. Сделайте это в полной аналогии как это происходило на прошлых лаб. работах. Откройте этот проект в студии и обратите внимание что в этом проекте нету файлов app(cpp и h). Есть только файлы main. Основная суть рефакторинга будет заключатся в том что мы создадим класс app - класс который и будет нашим приложением. В файле main.cpp мы будем создавать только экземпляр нашего приложение и далее будем его запускать. То есть мы отделим всю логику приложения от головной функции main() и сформируем отдельный класс.



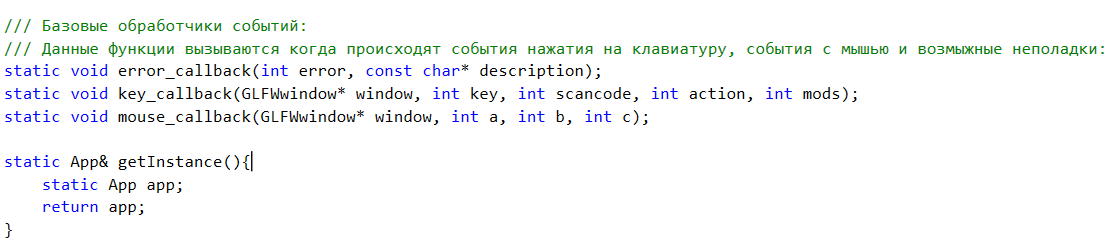
И так начните с того что создайте пару файлов app.cpp и app.h. Эти файлы должны также как и файл main лежат в папке src. Затем переместите все содержимое файла main.h в файл app.h:



А теперь давайте опишем каркас будущего приложение в файле app.h. Сперва создадим заголовок класса App и его основные функции:

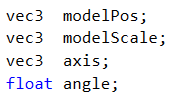


Затем добавим обработчики события и получение статичной переменной нашего приложения:

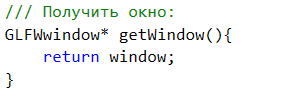


Обратите внимание что первые 3 функции по сути такие же функции что мы использовали ранее, только с приставкой *static*, это значит что эти функции могут вызываться отдельно без создания экземпляра класса, так как будто они были глобальные, то есть также как они использовались ранее - без привязки к какому либо классу. Особенностью функции статик является и то что они не могут менять локальные переменные класса, именно поэтому нам потребуется получит синглтон нашего приложения - это даст возможность связать внутренние переменные с этими статичными функциями. Эти сложности обусловлены особенностями работы функций обработчиков событий библиотеки GLFW которую мы используем.

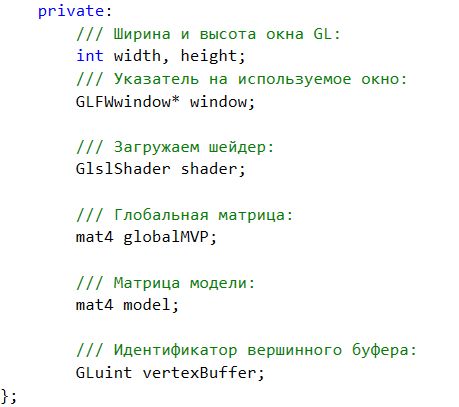
Теперь добавим переменные которые мы использовали в 3 работе для перемещения и вращения треугольника:



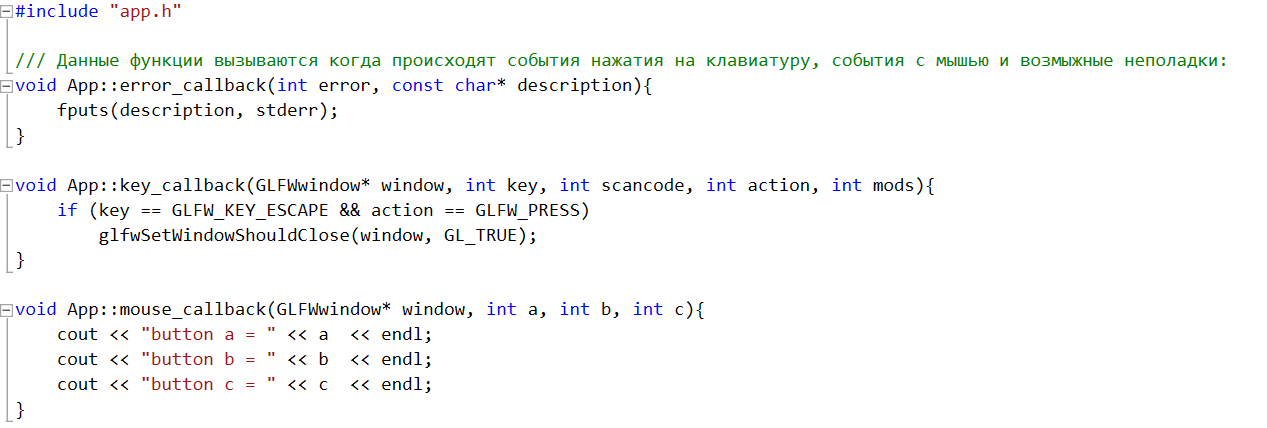
Заметьте, что эти переменные локальные, то есть они являются членами класса App, который мы сейчас разрабатываем. Также давайте добавим функцию которую будет возвращать указатель на используемое нами GLFW окно:



На этом публичные, доступные извне переменные и функции класса App заканчиваются. Теперь опишем приватные переменные, доступные только внутри нашего класс App:

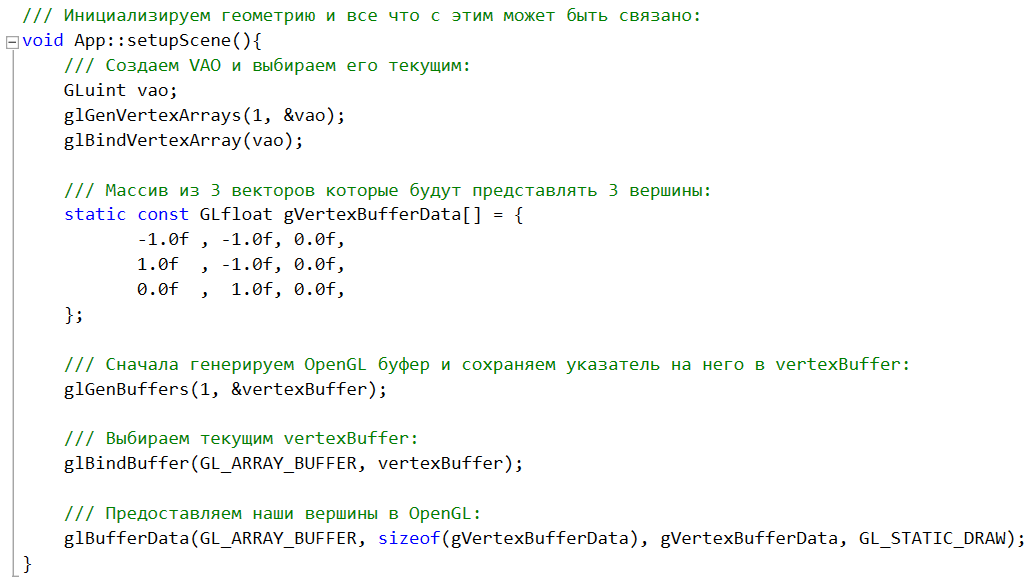


Займемся теперь файлом app.cpp. А именно опишем реализации(implementation) функций который мы описали только что в файле заголовке(.h). Начнем с обработчиков событий:



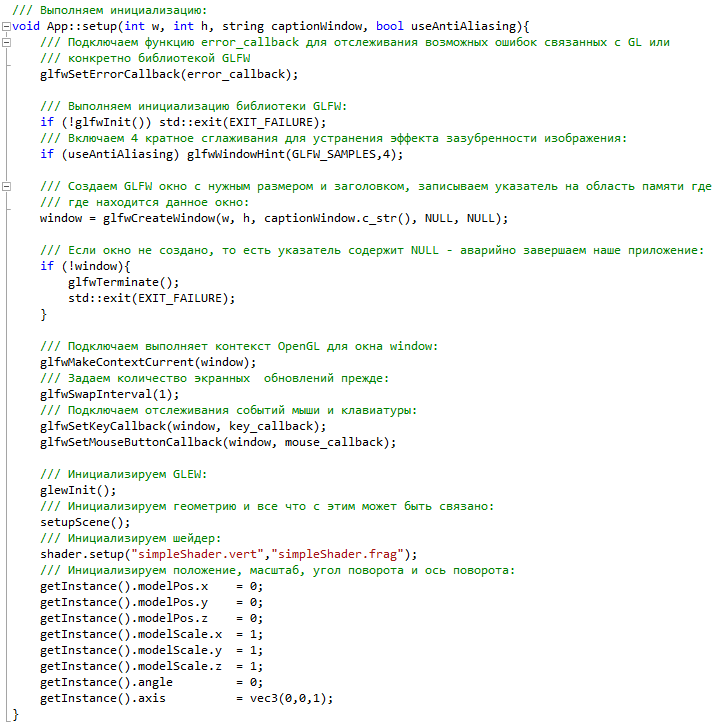
Заметьте что здесь мы уже не пишем static – указывать что функция будет статичной нужно только при объявлении функции.

Теперь добавим функцию которая создает простейшею геометрию в нашем приложении:



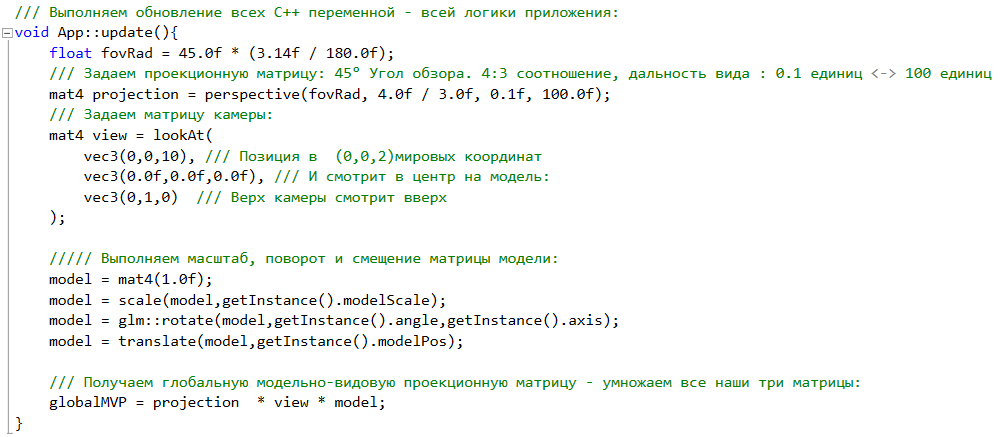
Заметьте что код содержимого этой функции тот же самый что был в прошлой работе, по этому можете смело, но внимательно копировать куски кода из прошлой работы.

Добавим функцию инициализации нашего приложения:

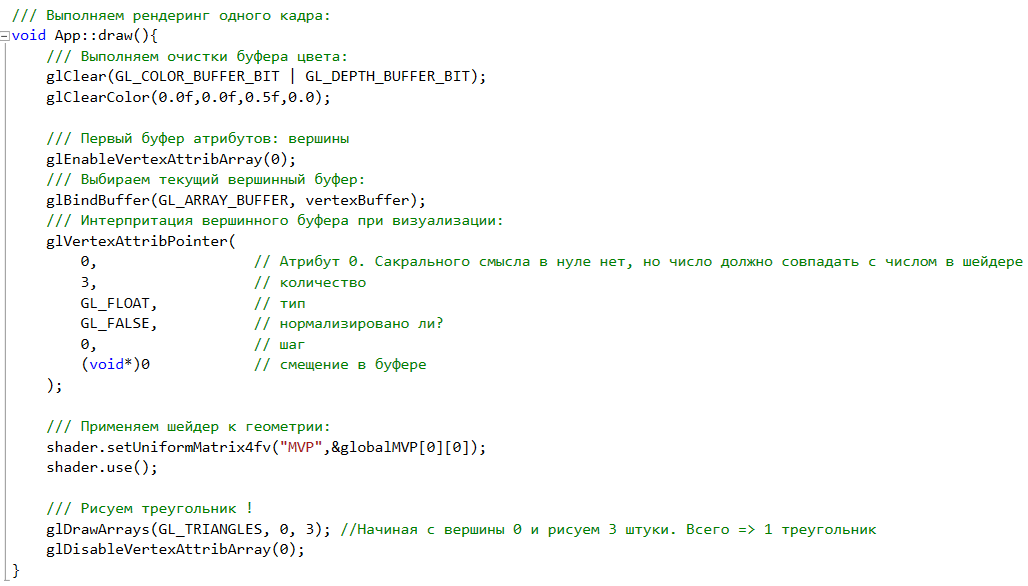


Опять же видно что большая часть кода этой функции использовалась в прошлой работе. Исключение составляет инициализация положения, масштаба и угла используя синглтон - это наш "мост" который будет связывать работу обычных функций класса App и статичных функций-обработчиков событий клавиатуры и мыши.

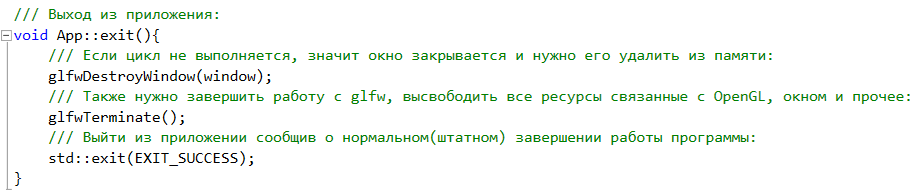
Теперь опишем основную функцию обновления нашего приложения, это функция будет вызваться циклически, перед вызовом функции draw. В эту функцию мы перенесем код который будет вычислять нашу матрицу globalMVP в соответствии с изменением позиции, угла и масштаба:



Далее добавим функцию draw():



И наконец последняя функция exit():



Таким образом мы описали класс нашего приложения. В дальнейшем мы будет работать с этим классом, расширяя его возможности.

Теперь создадим экземпляр класса нашего приложения и запустим его в работу. Для идем в класс main.cpp и добавляем в основную функцию следующие строчки кода:

Сразу в самом начале функции:

App myApp;

myApp.setup(1024,768,"project4",true);

Затем добавляем основной цикл:

/// Запускаем главный цикл обновления:

/// Выполняем цикл до тех пор пока окно не закрывается:

while (!glfwWindowShouldClose(myApp.getWindow())){

/// Выполняем обновление нашего приложения:

myApp.update();

/// Рисуем кадр:

myApp.draw();

/// Выполняем свап буфера:

glfwSwapBuffers(myApp.getWindow());

/// И обновления событий окна( в том числе клавиатуры и мыши):

glfwPollEvents();

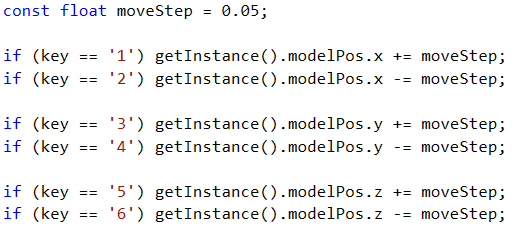
}

В самом конце функции, после выше указанных строчек добавим завершение работы приложения:

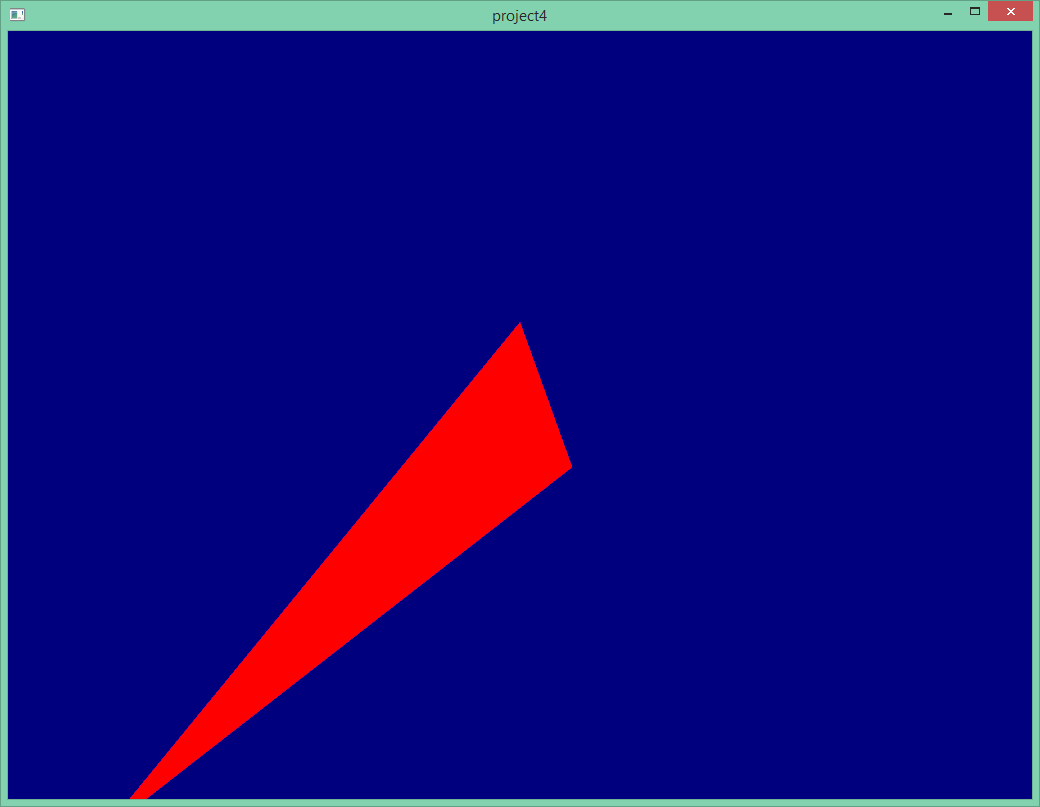
myApp.exit();

То есть как только цикл перестает работать мы запускаем процедуру завершения работы нашего приложения.

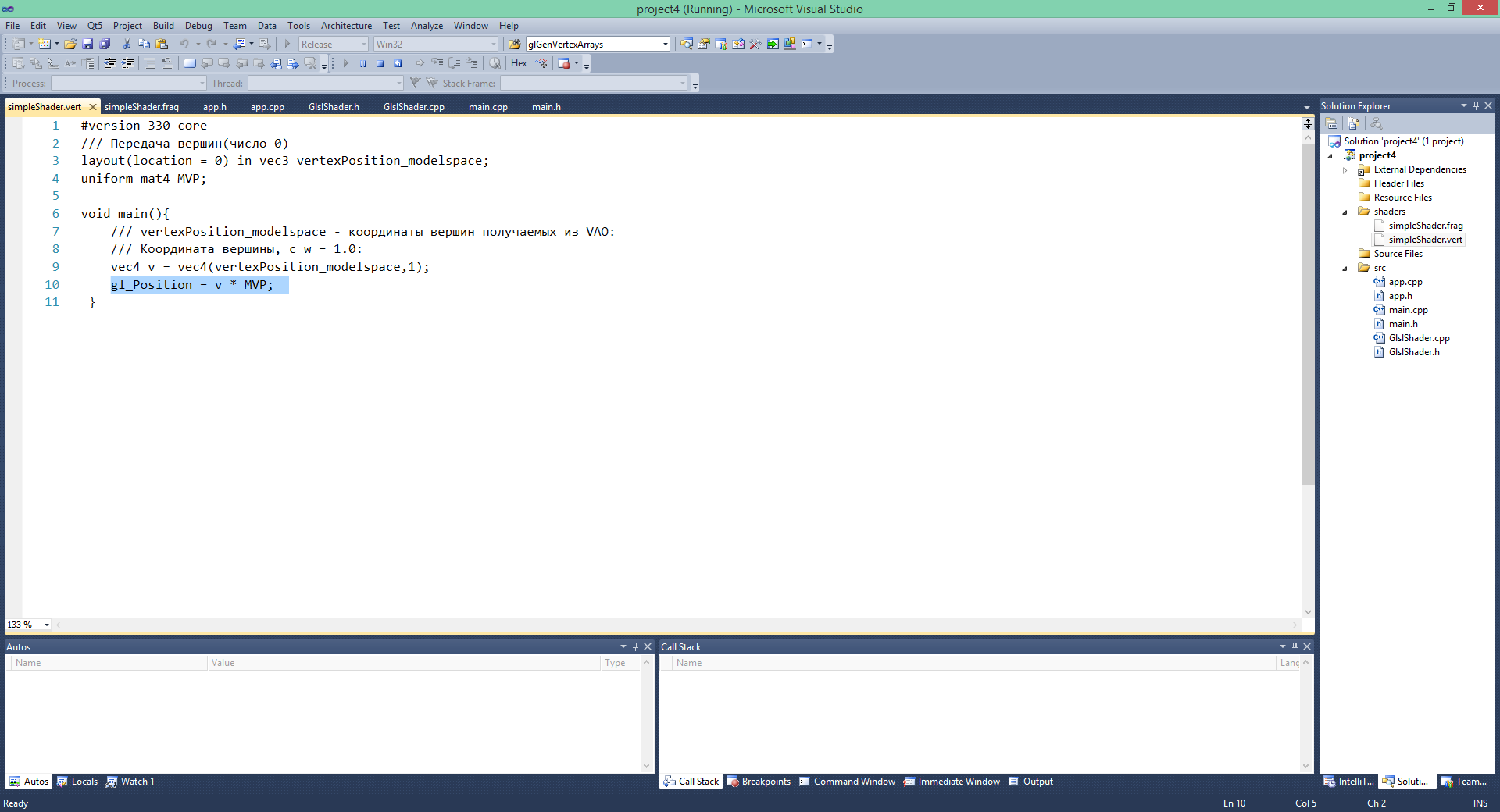
Теперь добавим в обработчик события клавиатуры код который позволит двигать модель по всем 3 осям:



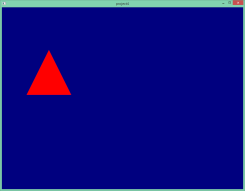
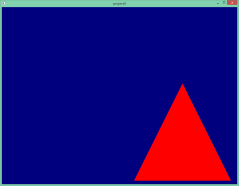
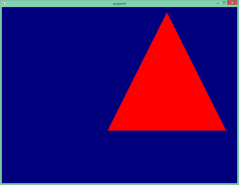
Отлично, теперь компилируем и запускаем приложение, давайте несколько раз нажмем на клавишу "1" и вызовем смещение модели по Х:



И что же мы видем, треугольник вместо перемещения по Х вытянулся, самые внимательные возможно заметили этот эффект еще на прошлой работе. Ключ этой проблемы нарушенный порядок умножения вершины и матрицы. Открываем файл "simpleShader.vert" и смотрим внимательно на код:



Теперь меняем порядок произведения на такой: gl\_Position = MVP \* v; Вот теперь позиционирование модели корректное:

1. **Рисуем куб вместо треугольника.**

В этой части лабораторной работы мы будем учиться следующим вещам:

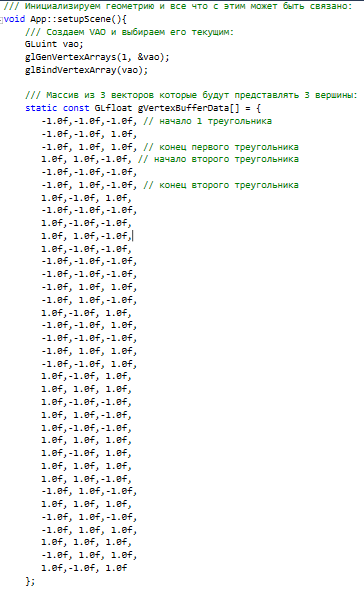
* Нарисуем объемный куб вместо уже порядком надоевшего плоского треугольника.
* Добавим разные цветы на грани куба
* Узнаем что такое Z-буфер

Для начала давайте добавим вершины которые образуют наш куб. У куба есть 6 квадратных граней. Но, так как OpenGL ничего не знает о гранях и умеет работать лишь с треугольниками нужно разбить каждую квадратную грань на 2 треугольника и у нас получится 12 треугольников. Запишем координаты вершин точно так же, как и для одного треугольника:

// Наши вершины. Три дробных числа подряд дают нам одну координату 3д вершины.3 вершины подряд дают нам один треугольник.  
// У куба 6 граней с 2 треугольниками на каждой грани, что дает на 6\*2=12 треугольников, или 12\*3=36 вершин  
static const GLfloat gVertexBufferData[] = {  
 -1.0f,-1.0f,-1.0f, // начало 1 треугольника

-1.0f,-1.0f, 1.0f,  
 -1.0f, 1.0f, 1.0f, // конец первого треугольника  
 1.0f, 1.0f,-1.0f, // начало второго треугольника  
 -1.0f,-1.0f,-1.0f,  
 -1.0f, 1.0f,-1.0f, // конец второго треугольника  
 1.0f,-1.0f, 1.0f,  
 -1.0f,-1.0f,-1.0f,  
 1.0f,-1.0f,-1.0f,  
 1.0f, 1.0f,-1.0f,  
 1.0f,-1.0f,-1.0f,  
 -1.0f,-1.0f,-1.0f,  
 -1.0f,-1.0f,-1.0f,  
 -1.0f, 1.0f, 1.0f,  
 -1.0f, 1.0f,-1.0f,  
 1.0f,-1.0f, 1.0f,  
 -1.0f,-1.0f, 1.0f,  
 -1.0f,-1.0f,-1.0f,  
 -1.0f, 1.0f, 1.0f,  
 -1.0f,-1.0f, 1.0f,  
 1.0f,-1.0f, 1.0f,  
 1.0f, 1.0f, 1.0f,  
 1.0f,-1.0f,-1.0f,  
 1.0f, 1.0f,-1.0f,  
 1.0f,-1.0f,-1.0f,  
 1.0f, 1.0f, 1.0f,  
 1.0f,-1.0f, 1.0f,  
 1.0f, 1.0f, 1.0f,  
 1.0f, 1.0f,-1.0f,  
 -1.0f, 1.0f,-1.0f,  
 1.0f, 1.0f, 1.0f,  
 -1.0f, 1.0f,-1.0f,  
 -1.0f, 1.0f, 1.0f,  
 1.0f, 1.0f, 1.0f,  
 -1.0f, 1.0f, 1.0f,  
 1.0f,-1.0f, 1.0f  
};

Так это выглядит в студии:



Теперь идем в функцию draw() и меняем там всего одну функцию:

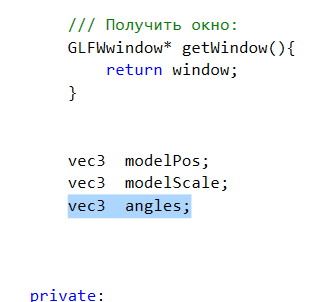
glDrawArrays(GL\_TRIANGLES, 0, 12\*3);

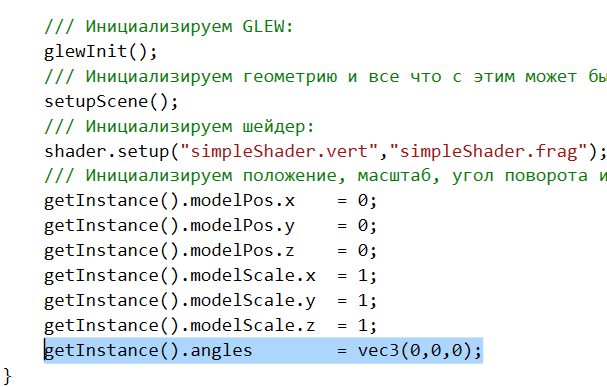
Так как сейчас рисуем уже не 1 треугольник с 3 вершинами, а 12 треугольников с 3 вершинами в каждой.

***Обращу ваше внимание на следующие моменты:***

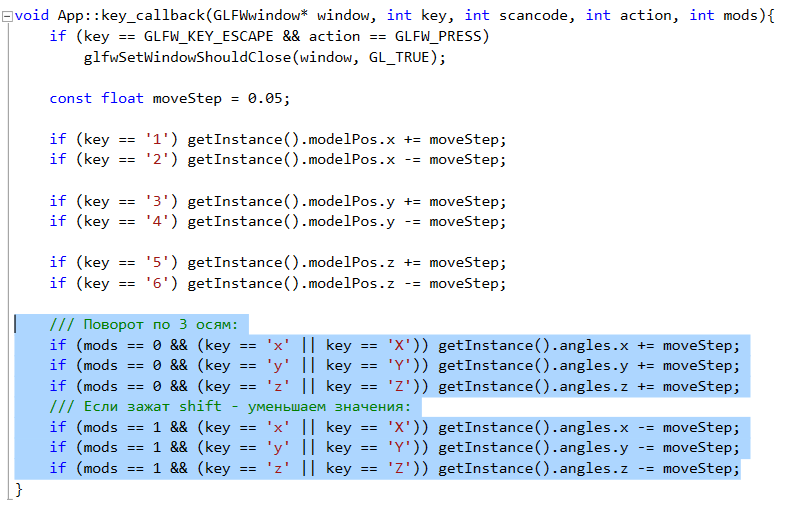
* Наша модель жестко задана в программе. То есть если мы хотим её поменять, то нам нужно изменить исходный код и перекомпилировать. В следующих работах мы научимся читать модель динамически из файла.
* Каждая вершина тут записана по меньшей мере 3 раза(можете сами поискать строчку «“-1.0f,-1.0f,-1.0f» в предыдущем куске кода). Это большое расточительство. В следующих работах мы научимся экономить память.

Теперь давайте добавим возможность поворота нашего куба по всем осям. Вместо одной переменной angle сделаем мы сделаем 3 компонентный вектор angles где будут хранится углы по всем 3 осям:

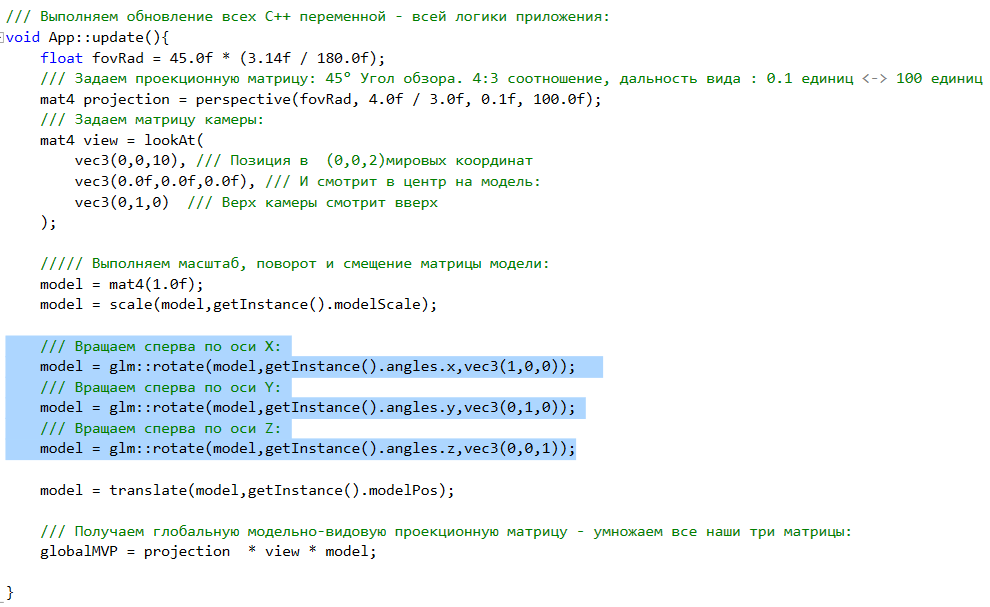




Далее идем в функцию обработчик клавиатуры и добавляем изменения углов по соответствующим осям:



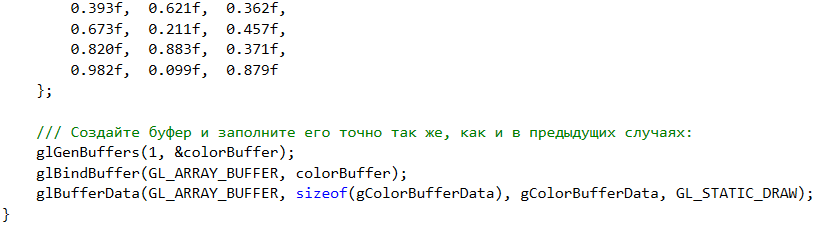
Теперь идем в функцию update() и меняем механизм вращения:



Теперь мы можем не только двигать куб но и вращать его по всем 3 осям. Далее добавим цвет для каждой грани. Цвет – с точки зрения видеокарты, точно то же самое, что и позиция: просто данные. В терминах OpenGL – это «атрибуты». Мы пользовались этими атрибутами вызывая функцию glEnableVertexAttribArray() и glVertexAttribPointer(). Теперь можно попробовать добавить еще один атрибут. Код будет почти такой же. Сначала нужно объявить наши цвета: один RGB триплет на каждую вершину. Я использовал случайные значения, поэтому результат может быть не такой красивый, как хотелось бы, но вы можете попробовать сделать красивее:

/*// Один цвет на каждую вершину:*   
*static const GLfloat gСolorBufferData[] = {*  
 *0.583f, 0.771f, 0.014f,*  
 *0.609f, 0.115f, 0.436f,*  
 *0.327f, 0.483f, 0.844f,*  
 *0.822f, 0.569f, 0.201f,*  
 *0.435f, 0.602f, 0.223f,*  
 *0.310f, 0.747f, 0.185f,*  
 *0.597f, 0.770f, 0.761f,*  
 *0.559f, 0.436f, 0.730f,*  
 *0.359f, 0.583f, 0.152f,*  
 *0.483f, 0.596f, 0.789f,*  
 *0.559f, 0.861f, 0.639f,*  
 *0.195f, 0.548f, 0.859f,*  
 *0.014f, 0.184f, 0.576f,*  
 *0.771f, 0.328f, 0.970f,*  
 *0.406f, 0.615f, 0.116f,*  
 *0.676f, 0.977f, 0.133f,*  
 *0.971f, 0.572f, 0.833f,*  
 *0.140f, 0.616f, 0.489f,*  
 *0.997f, 0.513f, 0.064f,*  
 *0.945f, 0.719f, 0.592f,*  
 *0.543f, 0.021f, 0.978f,*  
 *0.279f, 0.317f, 0.505f,*  
 *0.167f, 0.620f, 0.077f,*  
 *0.347f, 0.857f, 0.137f,*  
 *0.055f, 0.953f, 0.042f,*  
 *0.714f, 0.505f, 0.345f,*  
 *0.783f, 0.290f, 0.734f,*  
 *0.722f, 0.645f, 0.174f,*  
 *0.302f, 0.455f, 0.848f,*  
 *0.225f, 0.587f, 0.040f,*  
 *0.517f, 0.713f, 0.338f,*  
 *0.053f, 0.959f, 0.120f,*  
 *0.393f, 0.621f, 0.362f,*  
 *0.673f, 0.211f, 0.457f,*  
 *0.820f, 0.883f, 0.371f,*  
 *0.982f, 0.099f, 0.879f*  
*};*

Идем в функцию setupScene() и добавим этот код в самом конце функции. Далее создаем буфер по аналогии с тем, как мы этот делали для вершин:



После этого идем в функцию draw() и там добавим конфигурирование буфера цвета, по аналогии как это делается для вершин:

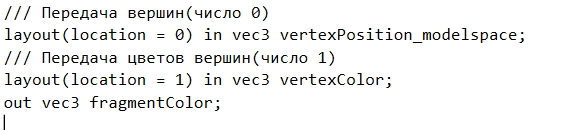


Теперь необходимо доработать вершинный и фрагментный шейдер, таким образом чтобы он смог назначить цвет каждой вершине нашего куба, промежуточные значения(между вершинами) будут автоматически интерполироваться. И так начнем с вершинного шейдера. Добавим туда код который позволит получит атрибут цвета конкретной вершины:

layout(location = 1) in vec3 vertexColor;

Обратите внимание 1 - это номер атрибута который вы указываете при конфигурировании буфера функцией glVertexAttribPointer().

Далее указываем out vec3 fragmentColor; - это выходной интерполированный цвет который мы передаем во фрагментный шейдер из вершинного:



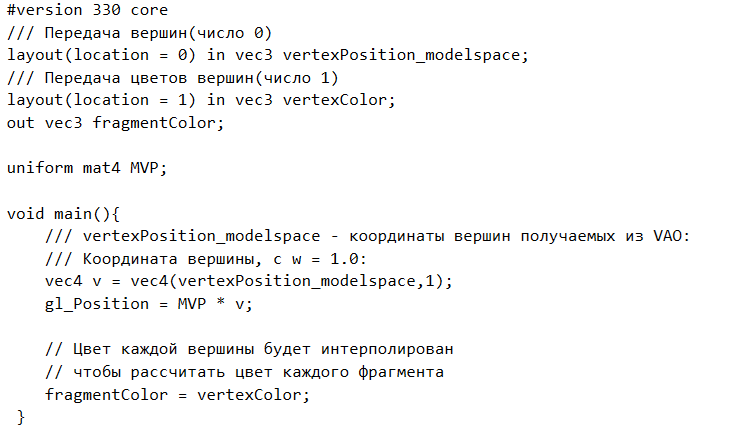
Далее в самом конце функции main() в вершинном шейдере укажем значение выходного цвета для передачи в фрагментный шейдер:

/// Цвет каждой вершины будет интерполирован

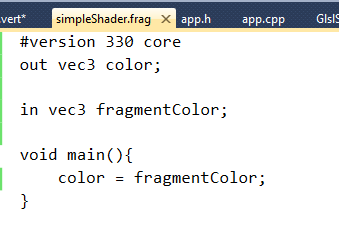
/// чтобы рассчитать цвет каждого фрагмента

fragmentColor = vertexColor;

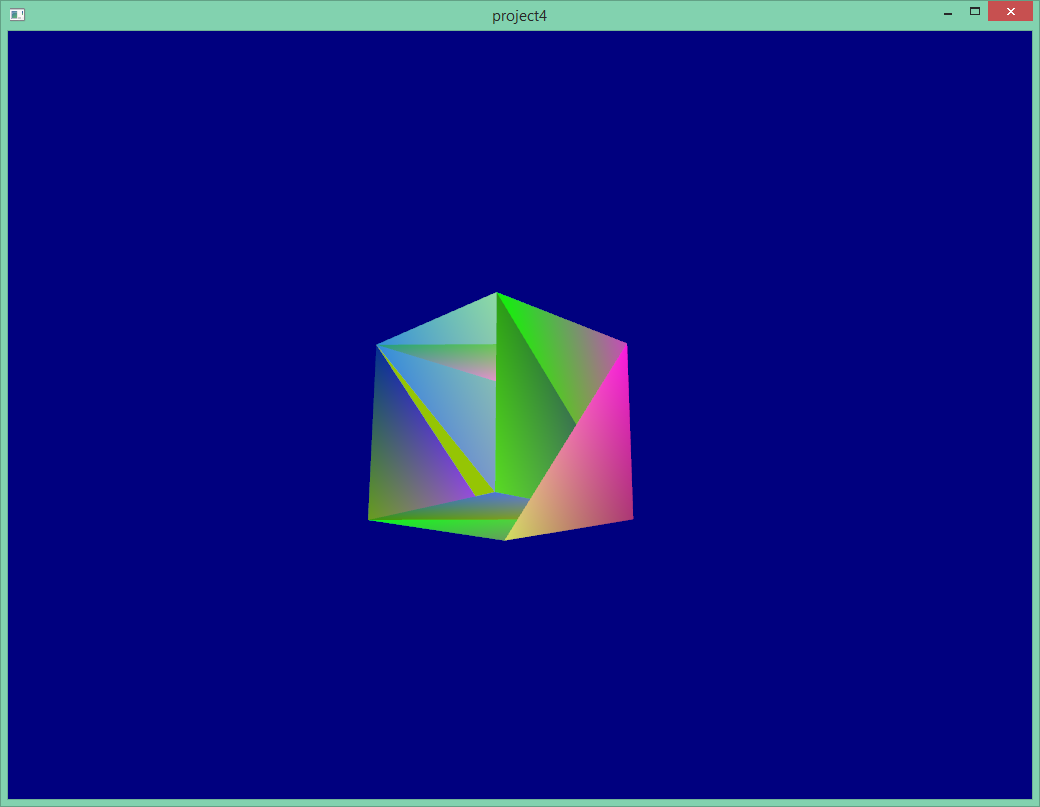
Полный код вершинного шейдера выглядит так:



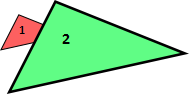
Теперь займемся фрагментным шейдером, его изменения минимальны, вместо назначения красного цвета мы укажем значения переменной fragmentColor, которая является для него входным значением:



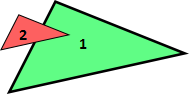
Теперь запустим программу и немного по вращаем наш кубик. И Увидим что визуализируется что-то не корректное грани куба как будто просвечивают друг через друга:



Чтобы понять, почему так получилось, давайте разберем, что получается, когда мы рисуем «дальние» и «ближние» треугольники:

[](http://2.bp.blogspot.com/-M3yLOCW4tJ8/Uc76Sj0WeKI/AAAAAAAAAHg/hk7z-6-9JTg/s189/lessons4_02.png)

В принципе так и надо. А теперь нарисуем «дальний» треугольник последним:

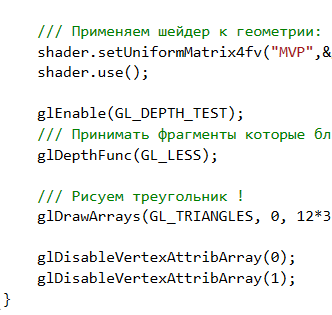
[](http://4.bp.blogspot.com/-k7pBOue30XQ/Uc76ShwTnmI/AAAAAAAAAHs/oppwc0ReWZo/s189/lessons4_03.png)

В итоге получается, что «дальний» треугольник перекрывает «ближний», хотя, по идее, такого быть не должно. Это же и получилось с нашим кубом: Некоторые грани, которые должны были бы быть сзади, повылазили вперед. Нам поможет Z-буфер!

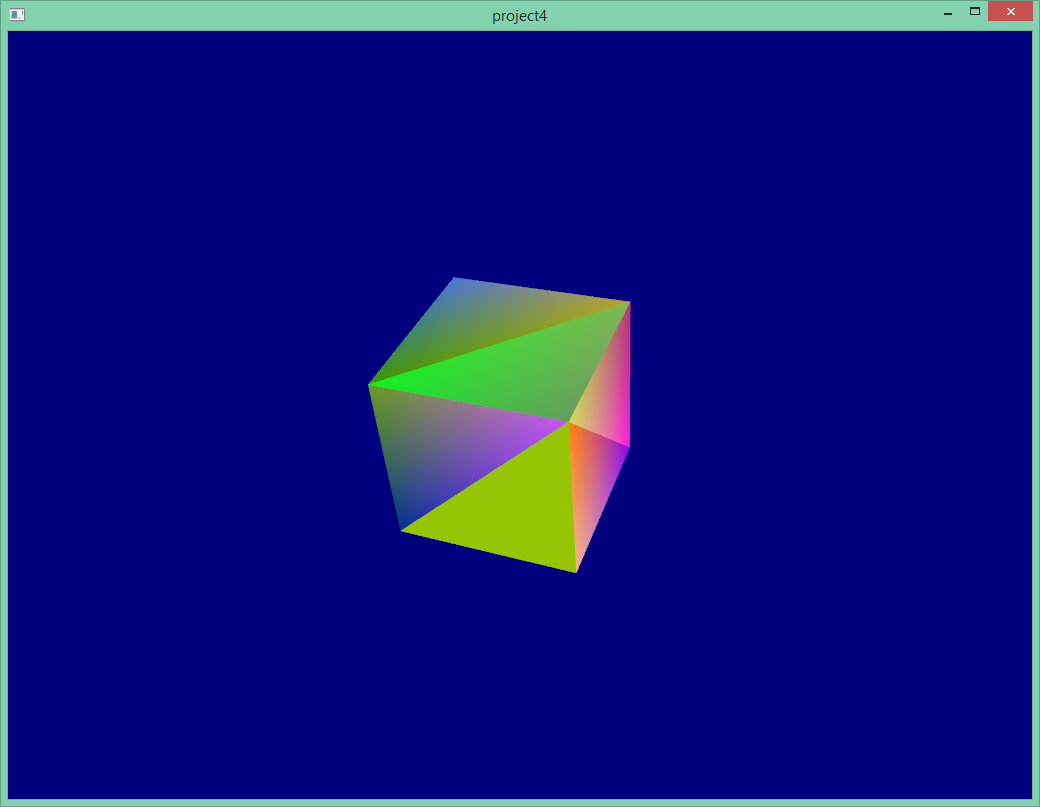
Чтобы избавить нас от этой проблемы, нужно просто сохранять «глубину»(Z компоненту) для каждого фрагмента в отдельном буфере, и каждый раз когда нам хочется нарисовать фрагмент, мы должны проверять, можем ли мы его записать(если новый фрагмент ближе к нам чем предыдущий записанный). Это можно было бы сделать и самому, но гораздо проще попросить это сделать за нас видеокарту:

*glEnable(GL\_DEPTH\_TEST);*  
*/// Принимать фрагменты которые ближе к камере*  
*glDepthFunc(GL\_LESS);*

Добавляем это в функции draw().Также обратите внимание на добавленную строчку отключение 2 вершинного атрибура:



Этого достаточно чтобы уже корректно нарисовать наш куб:

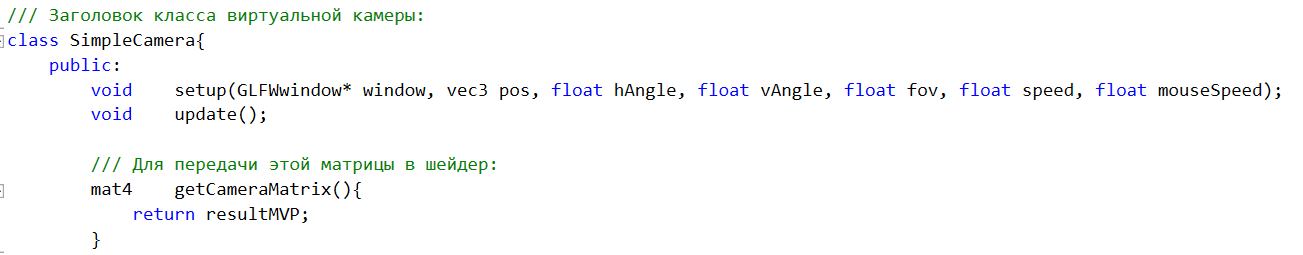


1. **Виртуальная камера для навигации пространства:**

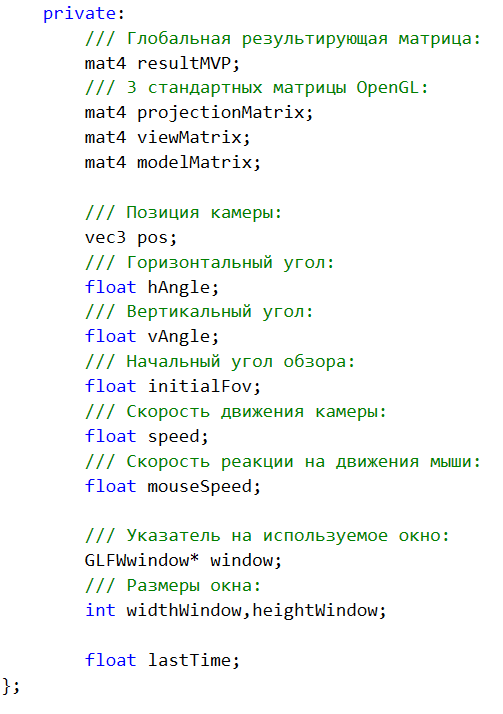
Уже по традиции создадим 2 пустых файла SimpleCamera(cpp,h). Затем в файле SimpleCamera.h добавим уже знакомые по app.h include файлы:



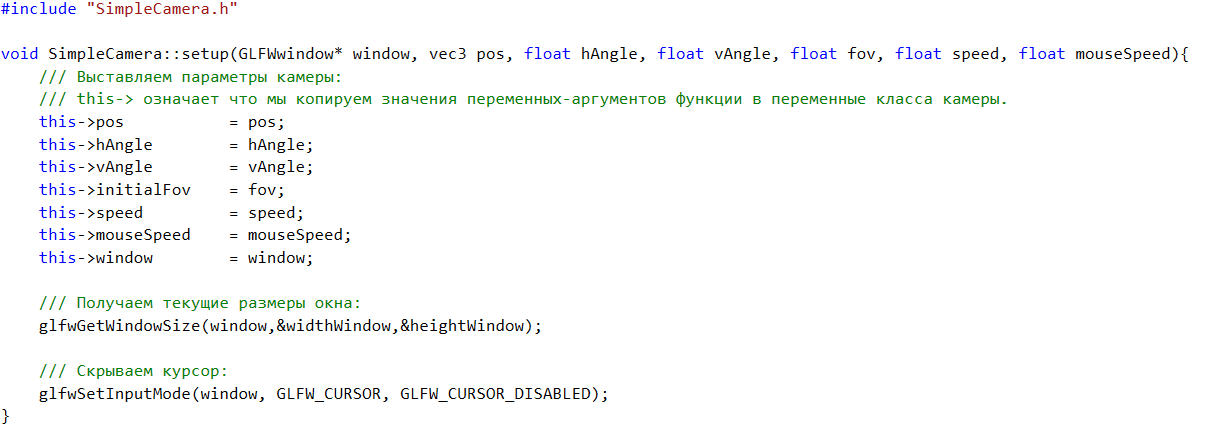
Далее по аналогии создадим описание класса камеры, сначала public часть:



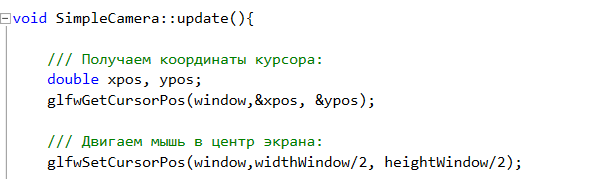
Затем, основная приватная часть класса:



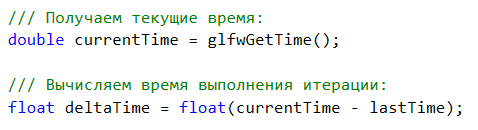
Теперь идем в файл SimpleCamera.cpp и делаем реализацию функции setup камеры:



Здесь все просто - мы копируем переменные аргументы функции в переменные класса камеры, затем определяем размер окна и скрываем курсор мышки. Весь механизм работы камеры сосредоточен в функции update() камеры. Рассмотрим её подробнее:



Тут также все очевидно мы получаем координаты мышки, затем возвращаем курсор мышки в центр окна. Нам необходимо передвигать курсор назад в центр экрана, иначе он вскоре выйдет за пределы окна, или упрется в край монитора и больше не будет двигаться. Далее мы получаем время в секундах с момента запуска программы, и вычисляем разницу во времени (deltaTime) между текущим вызовом функции update и последним вызовом этой функции:



Мы ввели переменную deltaTime из-за того, что вы вряд ли бы хотели смещать камеру на 1 каждый кадр, так как:

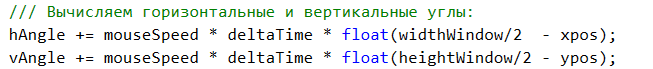
* Если у вас быстрый компьютер, и сцена рендерится, например, со скоростью в 100fps, то камера будет перемещаться со скоростью 100 единиц в секунду.
* Если у вас медленный компьютер, и сцена рендерится, например, со скоростью в 20fps, то камера будет перемещаться со скоростью 20 единиц в секунду.

Чтобы избежать этих проблем с разной скоростью перемещения на разных компьютерах обычно пользуются таким методом как deltaTime — или время которое прошло с последнего кадра.

* Если у вас быстрый компьютер, и у вас 100 fps, то вы двигаетесь со скоростью 1/100\*скорость за один кадр. Итого 1\*скорость в секунду.
* Если у вас медленны компьютер, и у вас 20 fps, то вы двигаетесь со скоростью 1/20\*скорость за один кадр. Итого 1\*скорость в секунду.

В самом конце функции update переменная lastTime = currentTime;

Теперь вычислим горизонтальные и вертикальные углы:



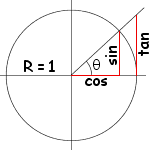
Краткое описание того, что делает этот код:

* widthWindow/2 - положение по Х. Иными словами чем дальше мы от центра окна, тем на больший угол нужно повернуться.
* mouseSpeed - просто множитель который определяет скорость вращения.

Теперь мы можем посчитать вектор, который будет представлять направление в Мировом Пространстве в которое смотрит камера.



Это стандартный способ перехода от сферических координат в декартовы. Вот краткое описание этого перехода:



Формула в коде выше — это то же самое, но в 3D пространстве. Теперь нам нужно правильно посчитать “up” вектор. Заметьте, «вверх», это совсем не обязательно в сторону +Y. Если вы опустите голову вниз, то ваша макушка будет направлена горизонтально. Единственная для нас константа за которую можно привязаться — вектор который выходит справа от камеры всегда горизонтальный.

Это можно проверить, если вы выпрямите свою правую руку вбок и начнете крутиться вокруг своей оси, или смотреть вверх и вниз, рука будет горизонтальной.

Давайте объявим вектор «вправо»: его координата Y будет 0, так как он горизонтальный, а координаты X и Z будут вычисляться так:

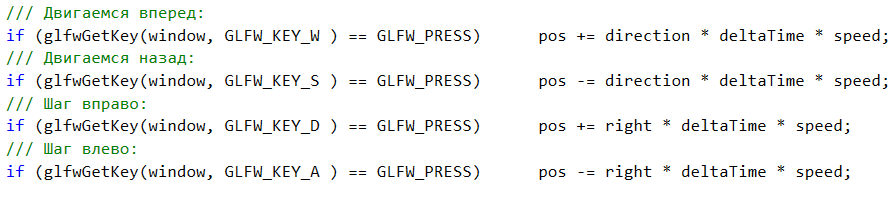


Теперь у нас есть вектор «вправо» и вектор «направление». Вектор «вверх» это вектор который перпендикулярный этим двум. Для нахождения перпендикулярных векторов издавна есть прекрасный метод — векторное произведение(cross product):

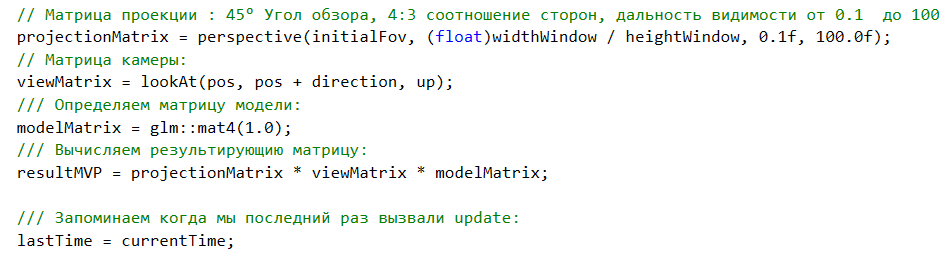


Чтобы запомнить как работает векторное произведение, просто вспомните правило правой руки. Первый вектор, это большой палец, второй вектор — указательный, а средний палец — это и есть векторное произведение. Все очень просто и удобно.

Теперь вычислим координаты позиции:



Теперь можно вычислить матрицы:



Основная матрица которая меняется в данном случае viewMatrix. Функция lookAt, напомню, использует следующие аргументы:

lookAt(  
 position,  // Положение камеры  
 position+direction, // А вот сюда мы смотрим  
 up

);

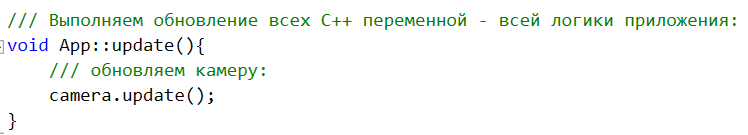
Обратите внимание что это почти тот же код что мы использовали в прошлой работе. То есть по сути все что мы сделали: определили направление взгляда, вектор "вверх" и позицию камеры, ну и конечно завернули все это в отдельный класс, чтобы с этим в последствии было удобнее работать. Теперь в коде приложения добавим экземпляр камеры и удалим старый код:

Добавим в конце App:setup() инициализацию камеры:

/// Задаем параметры камеры:

camera.setup(getWindow(),vec3(0,0,5),3.14,0.0f,45.f,3,0.01);

Затем обновим update() приложения:



В void App::draw() поменяем одну строчку:

shader.setUniformMatrix4fv("MVP",&camera.getCameraMatrix()[0][0]);

То есть используем уже матрицу класса камеры.

На этом выполнение лабораторной работы №4 считаем законченной.