**Цель лабораторной работы:**

1) Ознакомиться с принципами пространственных трансформаций OpenGL.

2) Разработать простейшие OpenGL приложение осуществляющие поворот, перемещение и масштабирование геометрической модели(в данном случае рассмотрим треугольник). Все реализовать при помощи вершинных шейдеров. Управление через клавиатуру.

**Порядок исполнения:**

Это одна из самых важных лабораторных работ. Она содержит важнейшие принципы осуществления пространственных трансформаций. Внимательно и вдумчиво выполните эту работу. Начнем с теоретического описания.

1. **Гомогенные координаты.**

В предыдущих лабораторных мы предполагали, что вершина расположена по координатам (x, y, z). Теперь давайте добавим еще одну координату – w. Отныне вершины у нас будут по координатам (x, y, z, w) Вскоре вы поймете, что к чему, но пока примите это как данность:

* Если w==1, тогда вектор (x,y,z,1) – это позиция в пространстве
* Если w==0, тогда вектор (x,y,z,0) – это направление.

Запомните это как аксиому без доказательств!!! И что это нам дает? Ну, для вращения ничего. Если вы вращаете точку или направление, то получите один и тот же результат. Но если вы вращаете перемещение(когда вы двигаете точку в определенном направлении), то все кардинально меняется. А что значит «переместить направление»? Ничего особенного.

Гомогенные координаты позволяют нам оперировать единым матаппаратом для обоих случаев.

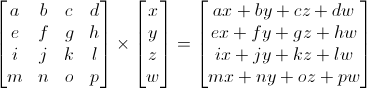
1. **Матрицы Трансформаций.**
   1. **Введение в матрицы**

Если по простому, то матрица, это просто массив чисел с фиксированным количеством строк и столбцов. Например, матрица 2 на 3 будет выглядеть так:

[http://1.bp.blogspot.com/-bAH4TcFXZls/Uc69clacECI/AAAAAAAAAEw/psw-4wB76Zc/s68/lessons3_01.png](http://1.bp.blogspot.com/-bAH4TcFXZls/Uc69clacECI/AAAAAAAAAEw/psw-4wB76Zc/s68/lessons3_01.png)

В 3D графике почти всегда используются матрицы 4х4. Это позволяет трансформировать (x,y,z,w) вершины. Для этого, мы умножаем вектор позиции на матрицу трансформации.

**Матрица\*Вершину  = Трансформированная вершина.** (обратите внимание на то что порядок умножения очень важен!)

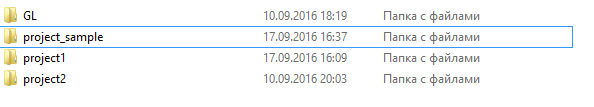
[](http://3.bp.blogspot.com/-iln9uPBlKvM/Uc69cgHhBTI/AAAAAAAAAE0/rE1lx2qrMLU/s367/lessons3_02.gif)

Все не так страшно как выглядит. Укажите пальцем левой руки на a, а пальцем правой руки на x. Это будет ax. Переместите левый палец на следующее число b, а правый палец вниз на следующее число – y. У нас получилось by. Еще раз – cz. И еще раз – dw. Теперь суммируем все получившиеся числа – ax+by+cz+dw. Мы получили наш новый x. Повторите то же самое для каждой строки и вы получите новый вектор (x,y,z,w).

Однако это довольно скучная операция, так что для её выполнения мы будем использовать готовые функции. В С++ мы это будем делать при помощи библиотеки GLM. Для этого сначала мы подключим её к нашему проекту.

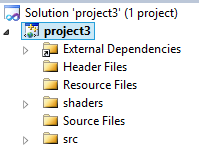
Для этого выполним следующий набор действий:

1) Из папки Lab\_3 скопируйте папку project\_sample и перенесите её в папку где лежит папка GL:

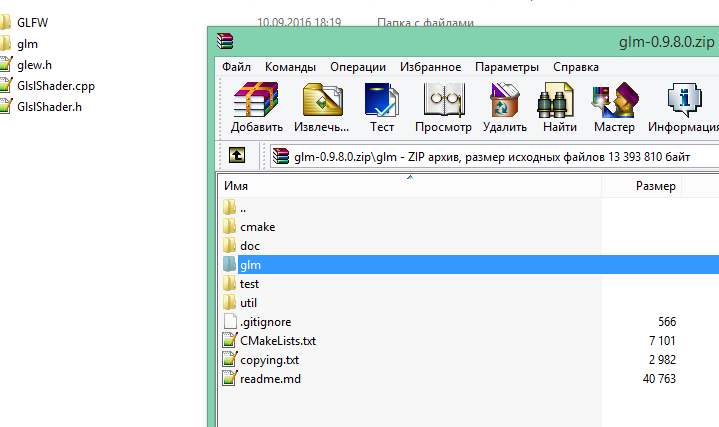


2) Далее переименуйте папку project\_sample в project3. Также внутри папки project3 переименуйте файл project\_sample.sln в project3.sln

3) Откройте в студии файл-решение project3.sln. Название решения также переименуйте в project3:



Теперь можно подключать библиотеку GLM к этому проекту. Эта библиотека подключается очень просто, достаточно всего лишь открыть архив glm-0.9.8.0.zip и перенести оттуда папку glm в папку GL\include:

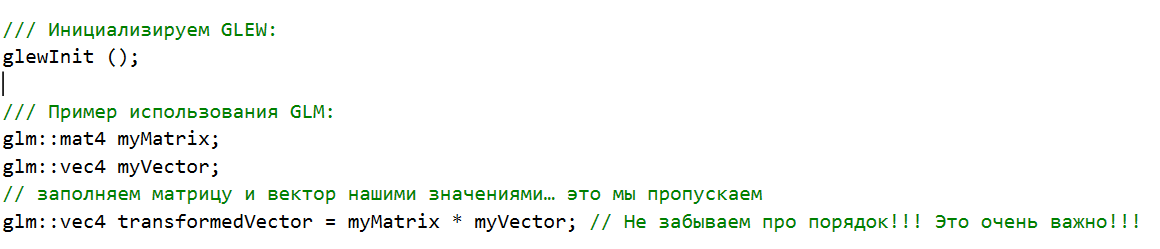


4) В файл app.h пропишем следующие:

#include <glm/glm.hpp>

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

После этого можно сразу же прописать в нашем С++ коде пример умножения матрицы на вектор, используя glm:



Теперь сравним это с тем, как это выглядит в шейдере GLSL:

mat4 myMatrix;

vec4 myVector;

// заполняем матрицу и вектор нашими значениями… это мы пропускаем

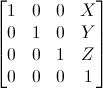
vec4 transformedVector = myMatrix \* myVector;

Синтаксис схож с GLM на 100%. Это одно из самых важных преимуществ библиотеки GLM(сокращение от OpenGL Mathematics).

* 1. **Матрица перемещений.**

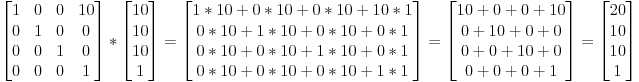
Теперь рассмотри одну из самых простых для понимания матриц – матрицу перемещения.

Вот она:

[](http://4.bp.blogspot.com/-qQpZ5iUhNmk/Uc69cuqZQaI/AAAAAAAAAEs/vTsU57kSGjM/s103/lessons3_03.png)

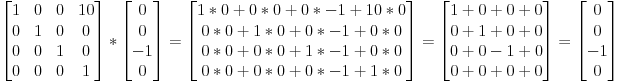
Тут X, Y, Z – это значения, которые мы хотим добавить к нашей позиции вершины.

Итак, если нам нужно переместить вектор (10,10,10,1) на 10 пунктов, по позиции Х, то:

[](http://3.bp.blogspot.com/-IAbUW8Z7GfQ/Uc69dPrNjmI/AAAAAAAAAFc/W1TXRui9oBs/s639/lessons3_04.png)

Как вы, я надеюсь, помните, 1 значит, что вектор представляет собой позицию, а не направление.

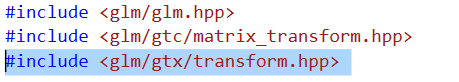
А теперь давайте попробуем таким же образом трансформировать направление (0,0,-1,0):



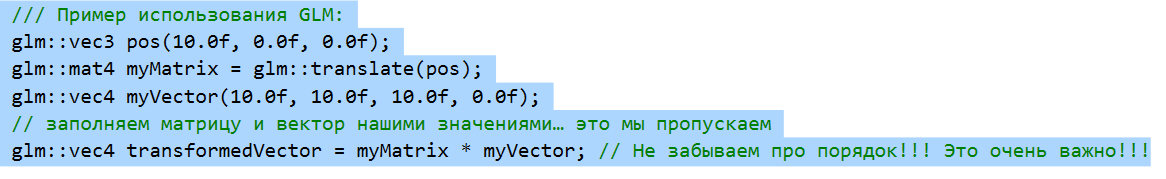
И в итоге у нас получился тот же вектор (0,0,-1,0). Как описывалось выше двигать направление не имеет смысла.

Теперь давайте посмотрим как это выглядит в коде C++ с использованием GLM.

Сперва подключим дополнительный заголовочный файл:



В С++ коде это выглядит так:

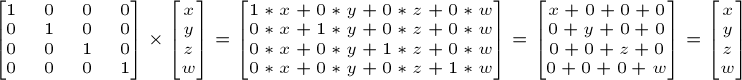


А в GLSL: В GLSL так редко кто делает. Чаще всего с помощью функции glm::translate(). Сначала создают матрицу в С++, а затем отправляют её в GLSL, и уже там делают лишь одно умножение:

vec4 transformedVector = myMatrix \* myVector;

* 1. **Единичная матрица.**

Это специальная матрица. Она не делает ничего. Но я упоминаю её, так как важно знать, что умножение A на единичную матрицу в результате дает А. Можно сказать что это значение матриц opengl по умолчанию. То есть по умолчанию отсутствует какое либо пространственное преобразование:



В С++: это выглядит так:

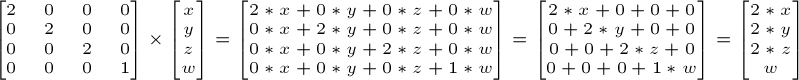
glm::mat4 myIdentityMatrix = glm::mat4(1.0f);

* 1. **Матрица масштабирования.**

Матрица масштабирования так же достаточно проста:

[](http://1.bp.blogspot.com/-p8dVVKIU9Tw/Uc69d6b_-NI/AAAAAAAAAFY/p6BEyfHmss0/s98/lessons3_07.png)

Поэтому если вам хочется увеличить вектор(позицию или направление, не важно) в два раза по всем направлениям:

[](http://1.bp.blogspot.com/-EqFBcst2EX4/Uc69eIxAu7I/AAAAAAAAAFU/0JKyWIKHdCA/s799/lessons3_08.png)

А координата w не поменялась.

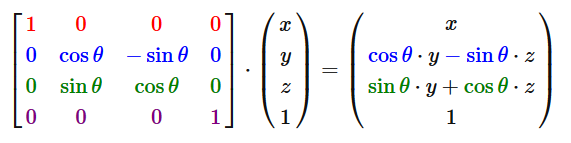
Реализация в C++:

*glm::mat4 myScalingMatrix = glm::scale(2.0f, 2.0f ,2.0f);*

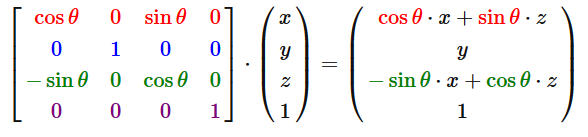
* 1. **Матрица вращения.**

Это матрица уже более сложная чем предыдущие. Поэтому рассмотрим лишь общие частные случаи. Так для каждой оси вращения существует отдельная матрица вращения:

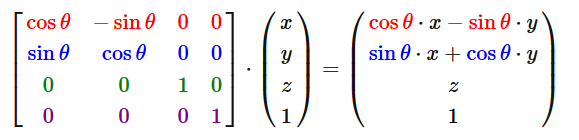
Вращение вокруг оси x:

****

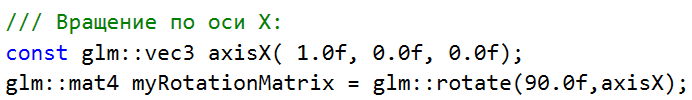
Вращение вокруг оси y:



Вращение вокруг оси z:

****

Реализация в C++:

****

* 1. **Совмещенные трансформации.**

Как вы наверное можете предположить в реальных задачах почти всегда приходится использовать совмещенные трансформации, то есть модель необходимо отмасштабировать, затем нужным образом повернуть и только затем правильно с позиционировать. С точки зрения математики это выглядеть как последовательное умножение соответствующих матриц на оригинальный вектор-позицию. В C++ коде это выглядит также:

*TransformedVector = TranslationMatrix \* RotationMatrix \* ScaleMatrix \* OriginalVector;*

Если мы будем применять трансформации в другом порядке, то не получим такой же результат. Вот попробуйте:

* Сделайте шаг вперед(не свалите компьютер со стола) и повернитесь влево
* Повернитесь влево и сделайте один шаг вперед.

Да, нужно всегда помнить про порядок действий при управлении, например, игровым персонажем. Сначала, если нужно, делаем масштабирование, потом выставьте направление(вращение) а потом перемещайте. Давайте разберем небольшой пример(я убрал вращение для облегчения расчетов).

Не правильный способ:

* Перемещаем 3D модель корабля на (10,0,0). Его центр теперь на 10 по Х от центра.
* Увеличиваем размер нашего корабля в 2 раза. Каждая координата умножается на 2 относительно центра который далеко… И в итоге у нас получается корабль необходимого размера но по позиции 2\*10=20. Что не совсем то, чего мы хотели.

Правильный способ:

* Увеличиваем размер корабля в 2 раза. Теперь у нас есть большой корабль расположенный по центру.
* Перемещаем корабль. Размер корабля не изменился и он расположен в нужном месте.

Умножение матрицы на матрицу выполняется почти так же, как и умножение матрицы на вектор.

В С++:

glm::mat4 myModelMatrix = myTranslationMatrix \* myRotationMatrix \* myScaleMatrix;  
 glm::vec4 myTransformedVector = myModelMatrix \* myOriginalVector;

В GLSL:  
 mat4 transform = mat2 \* mat1;  
 vec4 out\_vec = transform \* in\_vec;

И так, мы рассмотрели основные механизмы пространственных трансформаций для 3D графики в целом и OpenGL в частности.

1. **Матрицы Модели, Вида и Проекции**

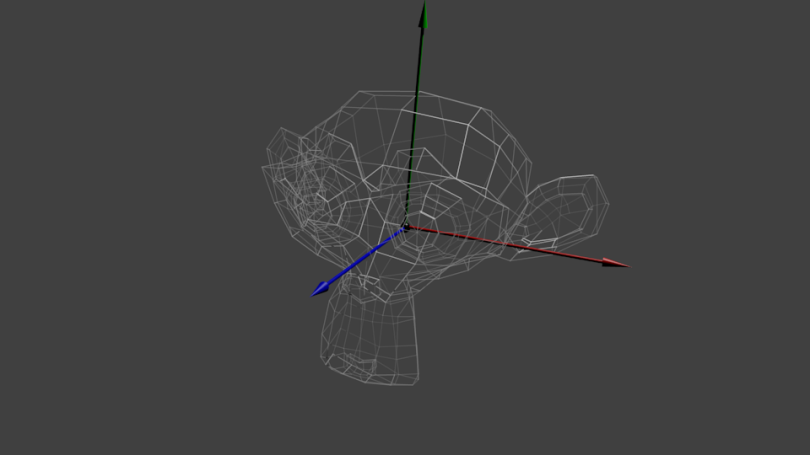
Теперь рассмотрим подробнее основные матрицы OpenGL.

Для иллюстраций предполагаем, что мы уже умеем рисовать в OpenGL известную 3D модель программы Blender – голову обезьяны Сюзанны).

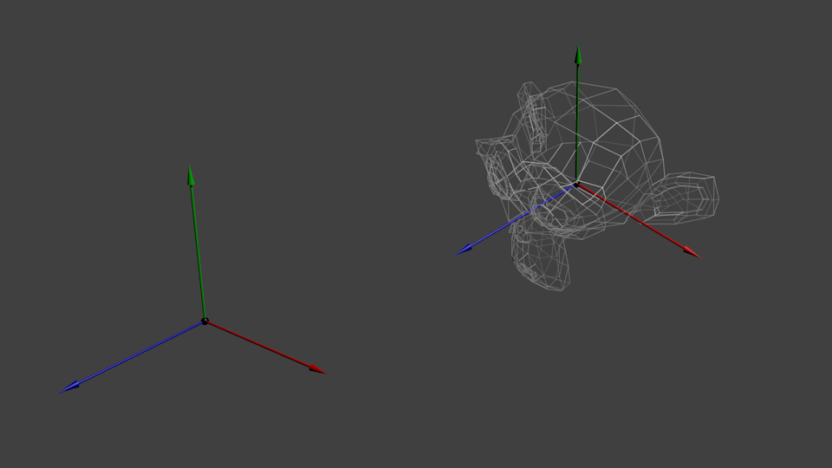
Матрицы Модели, Вида и Проекции очень удобный метод разделения трансформаций. Если сильно хочется, вы можете не использовать их(мы же не использовали их в уроках 1 и 2). Но я настойчиво рекомендую вам пользоваться ими. Просто почти все 3D библиотеки, игры и.т.д используют их для разделения трансформаций.

* 1. **Матрица Модели**

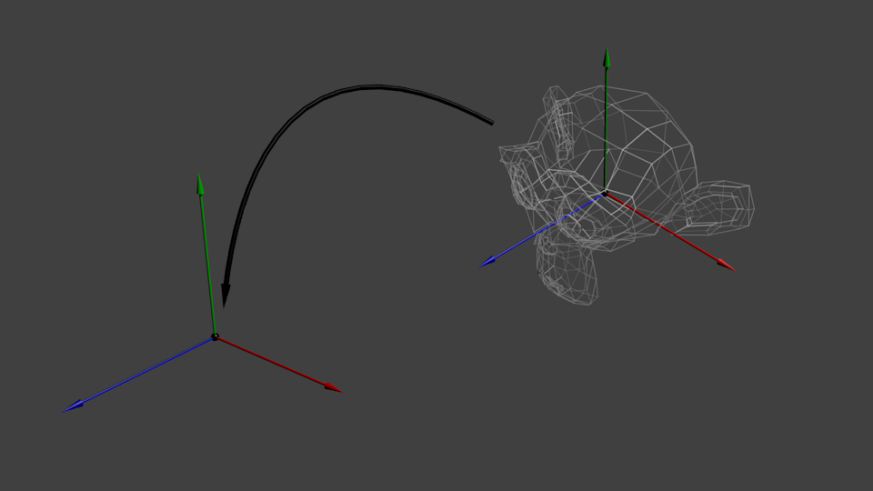
Данная модель, как и треугольник который мы рисовали в 1 и 2 лаб. работах, задана набором вершин. X, Y, Z координаты заданы относительно центра объекта. Так вот, если вершина расположена по координатам (0,0,0), то она находится в центре всего объекта:



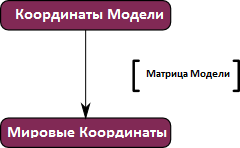
Теперь мы имеем возможность двигать нашу модель. Например, потому, что пользователь управляет ей с помощью клавиатуры и мыши. Это сделать очень просто: масштабирование\*вращение\*перемещение и все. Вы применяете вашу матрицу ко всем вершинам в каждом кадре(в GLSL, а не в C++) и все перемещается. Все что не перемещается – расположено в центре «мира».



Вершины находятся в мировом пространстве Черная стрелка на рисунке показывает, как мы переходим из пространства модели, в мировое пространство(Все вершины были заданы относительно центра модели, а стали заданы относительно центра мира)



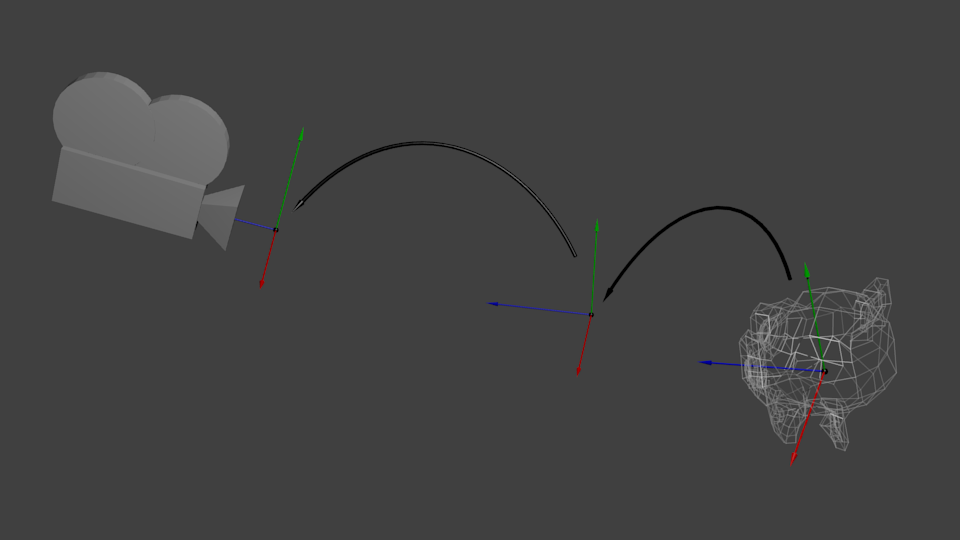
Эту трансформацию можно отобразить следующей диаграммой:

[](http://3.bp.blogspot.com/-Urk6-MNm2zs/Uc69friHsQI/AAAAAAAAAF0/fjFbO2Q3NzU/s240/lessons3_12.png)

* 1. **Матрица Модели**

Предположим что вы стоите с фотоаппаратом и хотите сфотографировать гору. Если вы хотите сфотографировать гору под каким-нибудь углом, то можно передвинуть фотокамеру…или гору. В реальной жизни это невозможно, но очень легко и удобно в компьютерной графике. По умолчанию наша камера находится в центре Мировых Координат. Чтобы двигать наш мир нужно создать новую матрицу. К примеру нам нужно переместить нашу камеру на 3 единицы вправо(+Х). Это то же самое, что переместить весь мир на 3 единицы влево(-Х). И пока вы пытаетесь это осмыслить, давайте попробуем:

*glm::mat4 ViewMatrix = glm::translate(-3.0f, 0.0f ,0.0f);*



Теперь давайте посмотрим на эту интересную функцию:

**glm::mat4 CameraMatrix = glm::LookAt(**

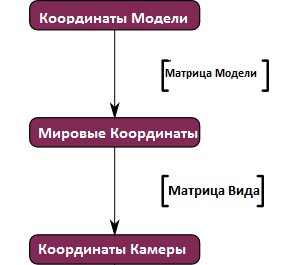
**eye**, /// Позиция камеры в мировых

**center**, координатах, /// точка на которую мы хотим посмотреть в мировых координатах

**upVector** /// Вектор ориентации камеры. скорее всего glm::vec3(0,1,0) будет нормальной ориентацией по умолчанию, а (0,-1,0) будет все показывать вверх ногами, что иногда необходимо.

**);**

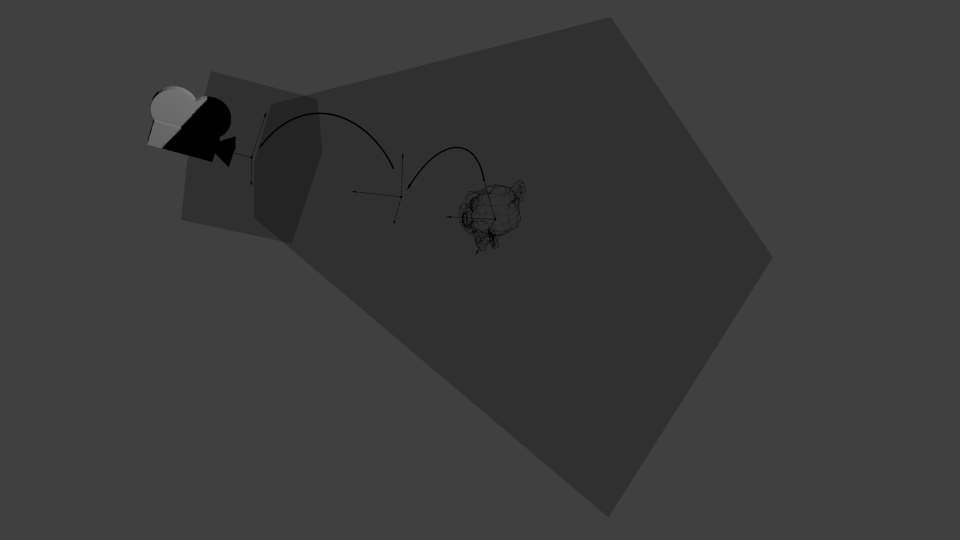
Вот иллюстрация к вышесказанному:



* 1. **Матрица проекции**

Теперь давайте рассмотрим матрицу проекции. Сейчас мы имеем координаты в пространстве камеры. Это значит, что после всех этих трансформаций, вершина которой посчастливилось оказаться в x==0 и y==0 будет отрендерена в центре экрана. Но мы же не можем пользоваться лишь координатами X,Y, чтобы понять куда рисовать вершину: дистанция к камере(Z) должна тоже учитываться! Если у нас есть две вершины, то одна из них будет более ближе к центру экрана чем другая, так как у неё больше координата Z.

Это называется перспективная проекция:



И к большому счастью для нас, матрица 4х4 может представлять собой и перспективные трансформации:

*glm::mat4 projectionMatrix = glm::perspective(*

*FoV, // Горизонтальное Поле Вида в градусах. Или величина приближения. Как будто «линза» на камере. Обычно между 90(суперширокий, как рыбий глаз) и 30(как небольшая подзорная труба)*

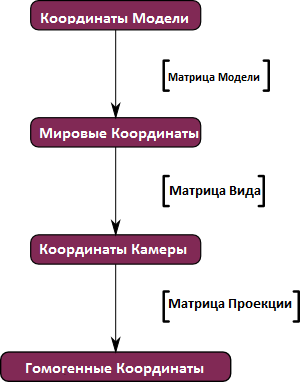
*4.0f / 3.0f, // Соотношение сторон. Зависит от размера вашего окна. Например, 4/3 == 800/600 == 1280/960, знакомо, не правда ли?*

*0.1f, // Ближнее поле отсечения. Его нужно задавать как можно большим, иначе будут проблемы с точностью.*

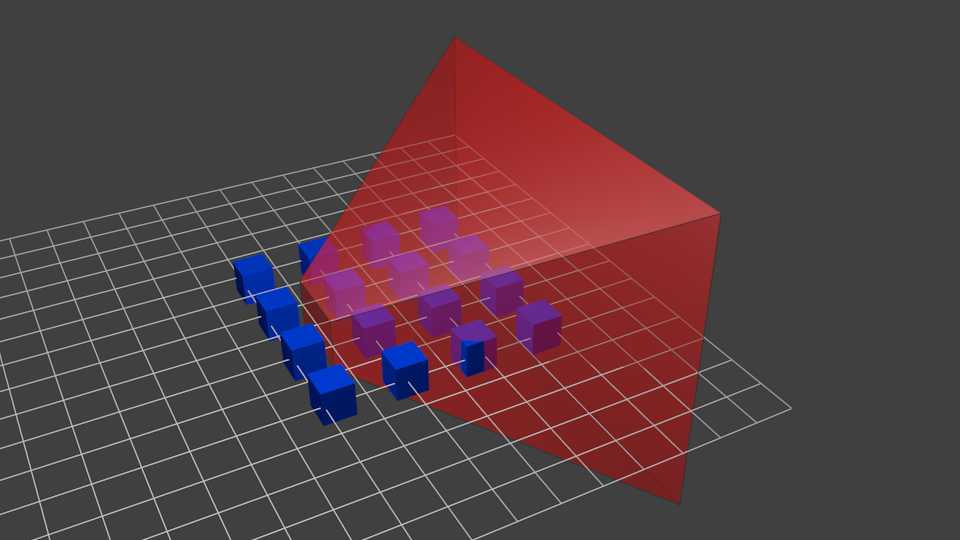
*100.0f // Дальнее поле отсечения. Нужно держать как можно меньшим.*

*);*

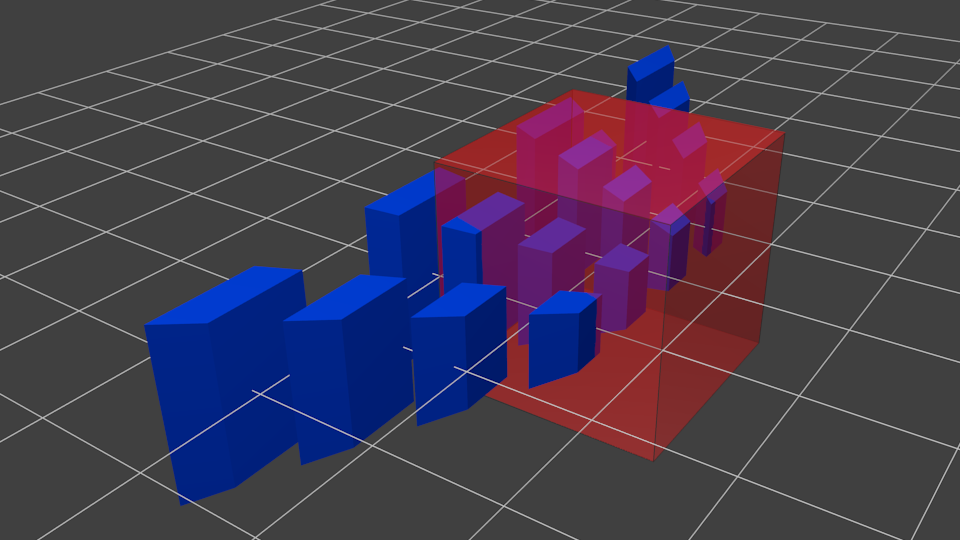
Давайте еще раз проговорим что мы сделали. Мы ушли от пространства камеры(все вершины заданы в координатах относительно камеры) в гомогенное пространство(все вершины в координатах маленького куба(-1,1 по всем осям куба). Все что находится в кубе – находится на экране) И вот так выглядит финальная диаграмма пространственных трансформаций OpenGL:



Вот еще одна картинка чтобы стало яснее, что же происходит при это трансформациях. Перед умножением на проекционную матрицу у нас есть голубые объекты заданные в пространстве камеры и красный объект, который представляет собой поле вида камеры: пространство которое попадает в объектив камеры:

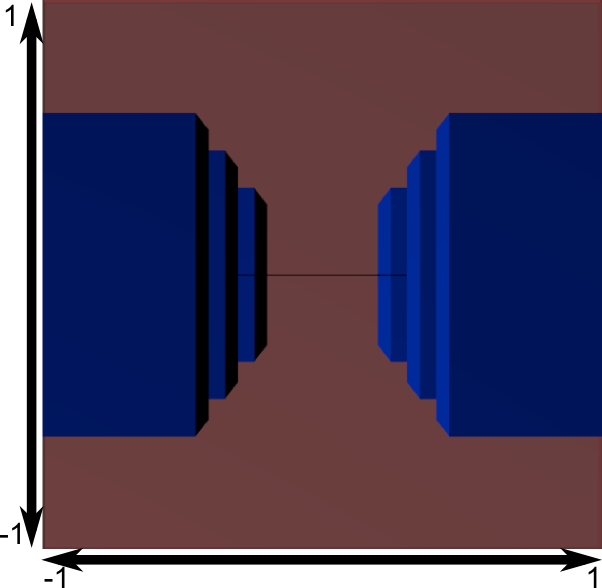


После умножения на проекционную матрицу у нас выходит следующее:

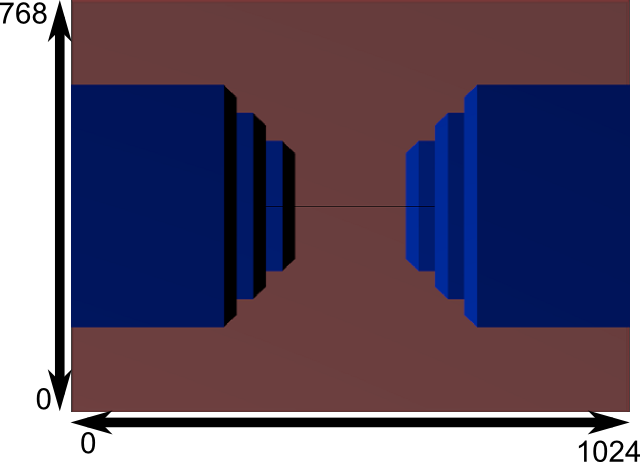


На предыдущей картинке поле вида превратилось в идеальный куб(с координатами вершин от -1 до 1 по всем осям.), а все объекты деформированы в перспективе. Все голубые объекты которые близко к камере – стали большими, а которые дальше – маленькими. Так же как и в реальной жизни.

Вот какой вид у нас открывается из «объектива»:



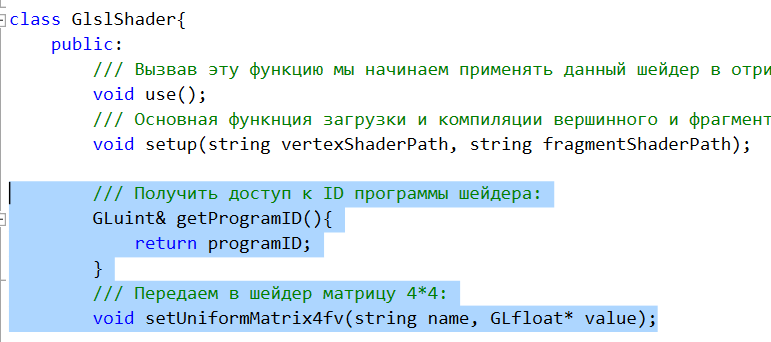
Однако оно квадратное, и нужно применить еще одно математическое преобразование, чтобы подогнать картинку под размеры окна:



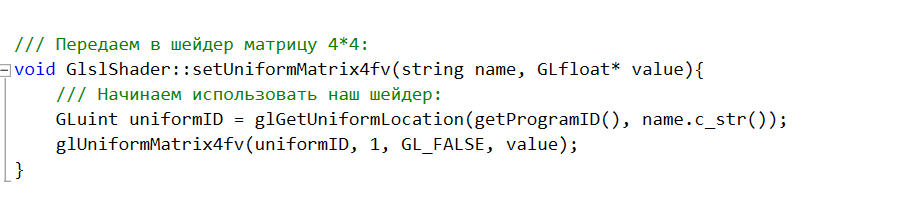
1. **Практическое использование пространственных трансформаций OpenGL.**

Теперь перейдем к практике. Как возможно вы уже поняли project\_sample - базовый проект, который по сути является результатом 2 лабораторной работы где мы рисовали треугольник и раскрашивали его в красный цвет. В это приложение мы добавим возможность масштабирования, поворота и перемещения этого треугольника.

Для этого сначала нам нужно подготовить наш класс шейдера чтобы можно было передать в шейдерную программу нашу вычисленную матрицу. Сначала открываем файл GlslShader.h и туда прописываем 1 функцию, и еще один заголовок функции:

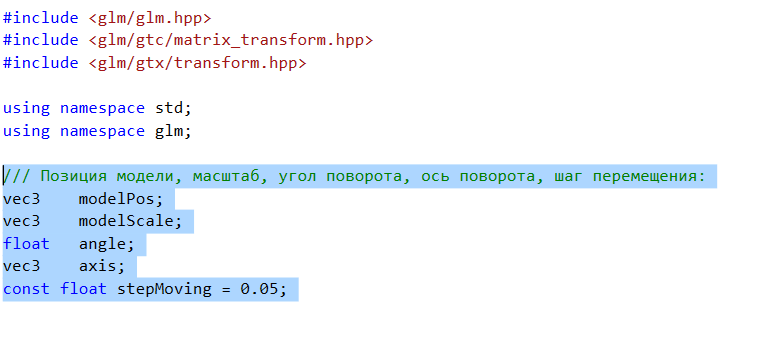


Первая функция по сути просто так назывемый "getter" которая просто дает доступ к private переменной класса. Теперь открываем файл GlslShader.cpp и там прописываем тело функции которая позволит передать матрицу непостредственно в шейдер:

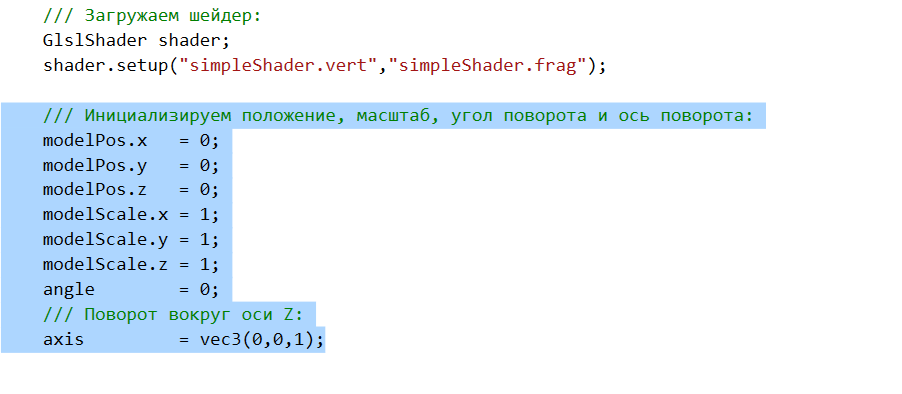


Теперь открываем файл app.h и прописываем using namespace glm; Это нужно чтобы каждый раз не обращаться к этому пространству имен, иными словами писать не glm::vec3, а просто vec3.

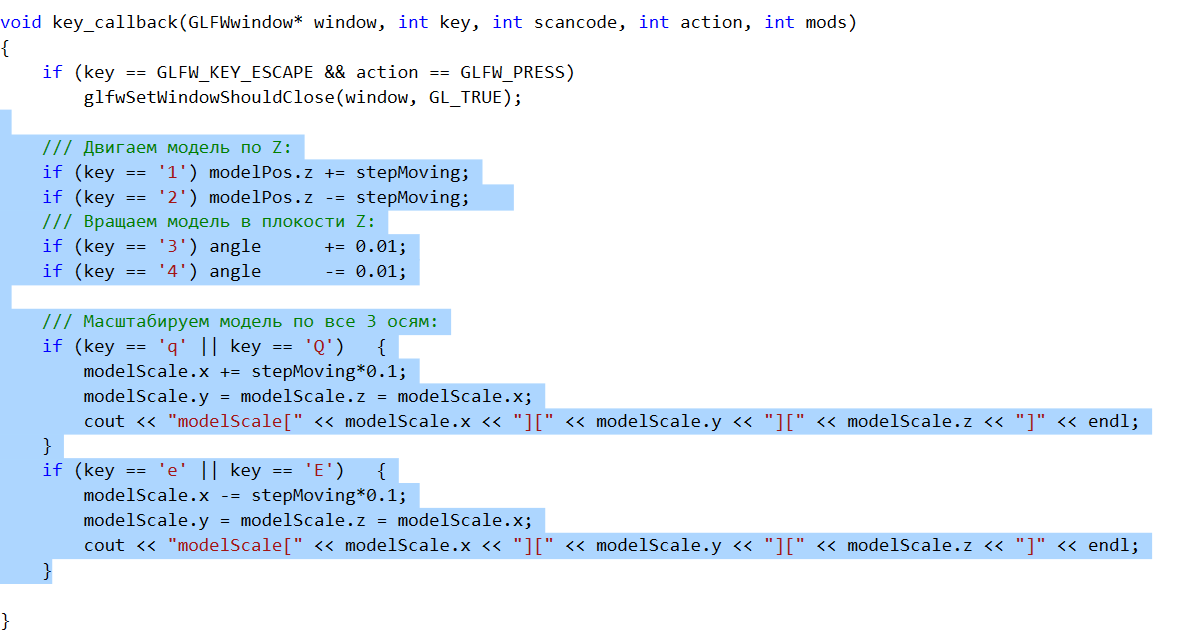
Там же объявим необходимые вспомогательные переменные чтобы мы могли двигать, вертеть и масштабировать наш треугольник:

****

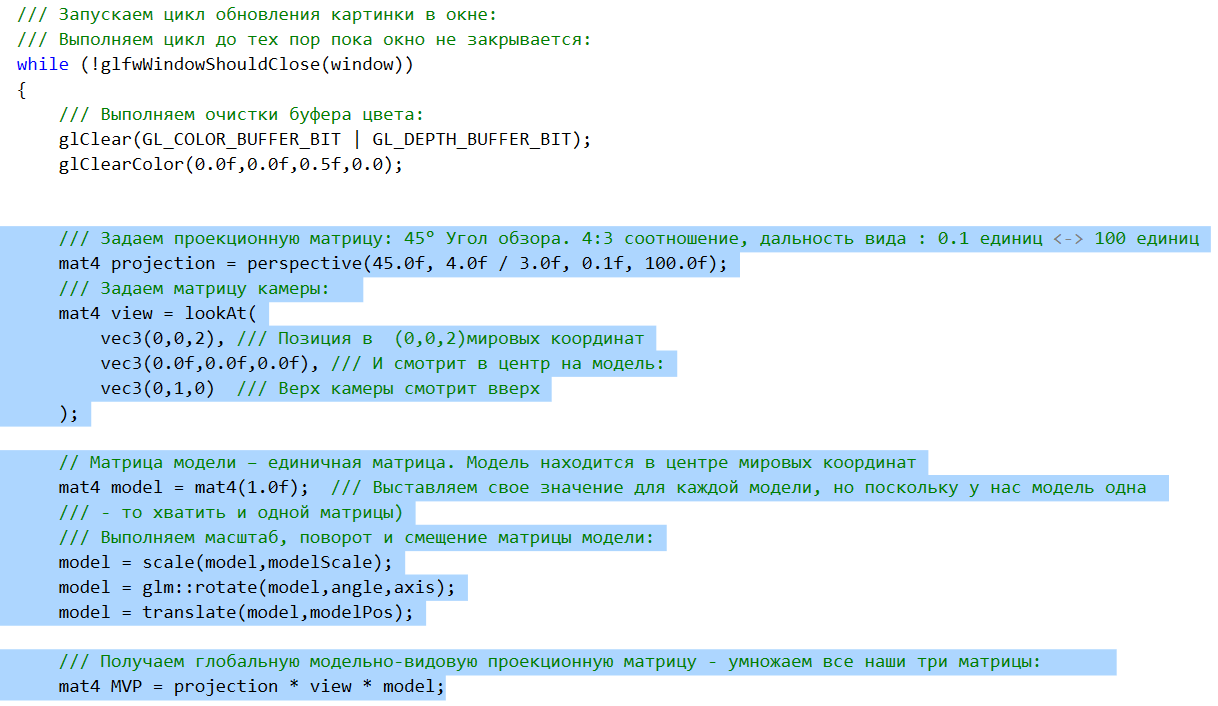
Далее идем в файл app.cpp, функцию main() и там сразу после загрузки шейдера инициализируем переменные:

****

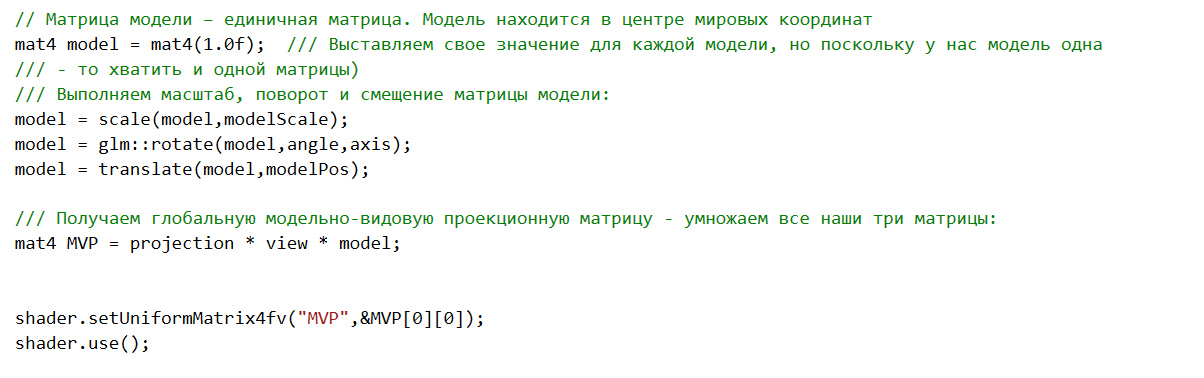
Теперь делаем так чтобы масштаб, угол поворота и позиция модели по z могли меняться с клавиатуры. Для этого идем в функцию key\_callback() и там прописываем код который по нажатиям определенных клавиш будет менять значения вышеописанных переменных:

****

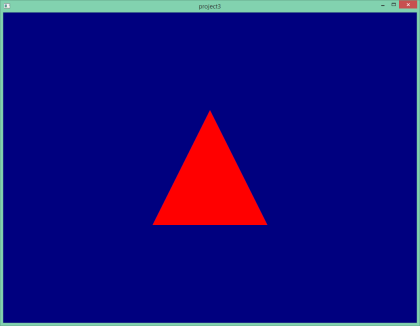
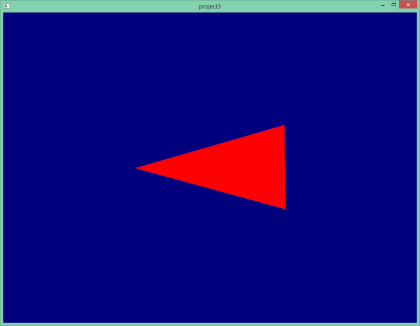
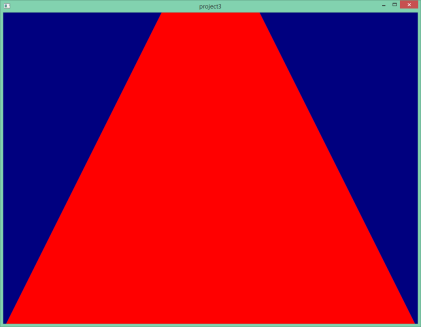
Теперь все готово к тому чтобы вычислить нашу модельно-видовую проекционную матрицу и передать её в shader для визуализации треугольника с учетом пространственных трансформаций. В функции main() в цикле, сразу после очистки буфера цвета мы начинаем формировать последовательно 3 матрицы – проекционную, видовую и модельную. Затем мы перемножаем эти матрицы и получаем в результате глобальную МВП матрицу которую затем передаем в шейдер:



Последний шаг – передаем матрицу в шейдер при помощи нашей только что написанной функции:



После этого приложение можно запускать, мы должны увидеть уже знакомый треугольник, но с возможностью его крутить и масштабировать, обратите внимание на перспективные искажение при повороте треугольника на 2 рисунке:

На этом можно считать лабораторную работу №3 завершенной!