**Цель лабораторной работы:**

1) Загрузить текстуру из BMP файла.

2) Корректно наложить загруженную текстуру на куб используя текстурные координаты UV.

3) Выполнить легкую модернизацию класса камеры для облегчения навигации - добавить движение вверх и вниз.

4) Освоить принципы фильтрации текстур и mipmaping. Загрузить DXT-5 сжатую текстуру с мипмап уровнями из файла формата DDS.

5) Реализуем передачу 2 текстур в шейдер и их произвольное смешивание.

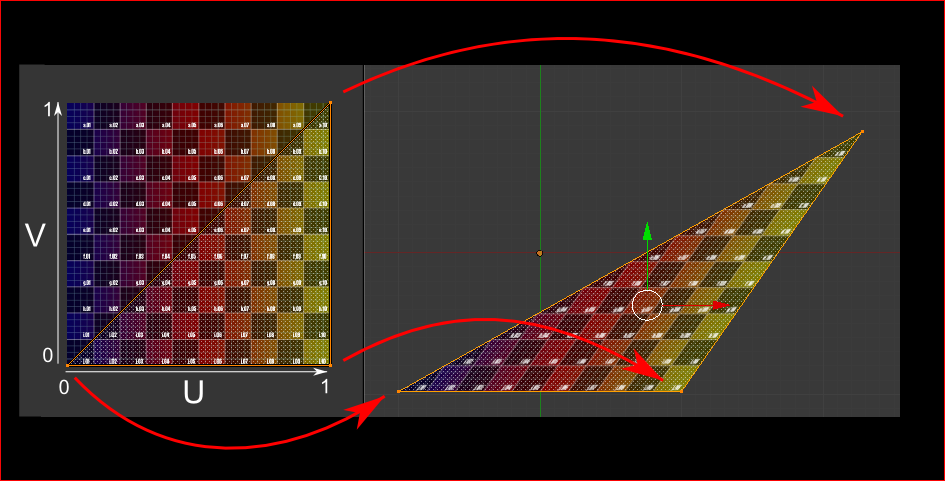
**Порядок исполнения:**

Переместите project5\_start\_sample в папку где лежит GL. По аналогии переименуйте папку этого проекта, файл .sln и название проекта в студии в project5. Сделайте это в полной аналогии как это происходило на прошлых лаб. работах. В файле main.cpp переименуйте функцию так чтобы в заголовке окна приложения был текст " project5".

**UV координаты**

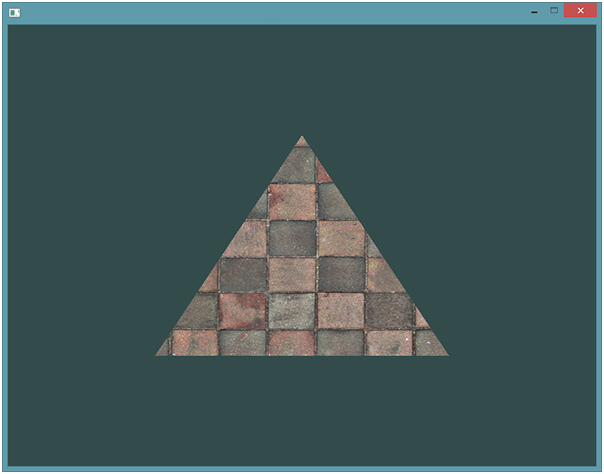
При текстурировании меша, нужно каким-то образом объяснить OpenGL какая часть текстуры будет использоваться для заливки каждого из треугольников. Это делается с помощью UV координат.

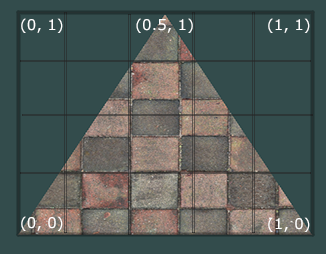
Каждая вершина может иметь не только позицию, а еще цвет и еще, например, два float значения – координаты U и V. Эти координаты используются для доступа к текстуре следующим образом:



Обратите внимание, что текстура растягивается на деформированном треугольнике!

Еще пример:

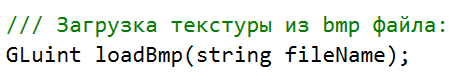




Обратите внимания что координаты UV - нормализованы, то есть лежат в диапазоне [0,1] не смотря на произвольное разрешение картинки-текстуры. Это в принципе все что нам нужно знать на данный момент о текстурных координатах.

Теперь займемся загрузкой BMP файла. Знание устройства bmp формата, это совсем не обязательное знание, так каждая вторая библиотека по работе с графикой умеет загружать bmp. Но так как это очень просто и может помочь нам понять как это работает внутри, мы сделаем этот велосипед. Мы напишем загрузчик BMP файлов с нуля чтобы понять как он работает и забудем про него.

Для начала в App.h, в класса App объявим прототип(заголовок) функции загружающею текстуру:



Там же добавим стандартный для OpenGL идентификатор, в данном случае идентификатор текстуры:

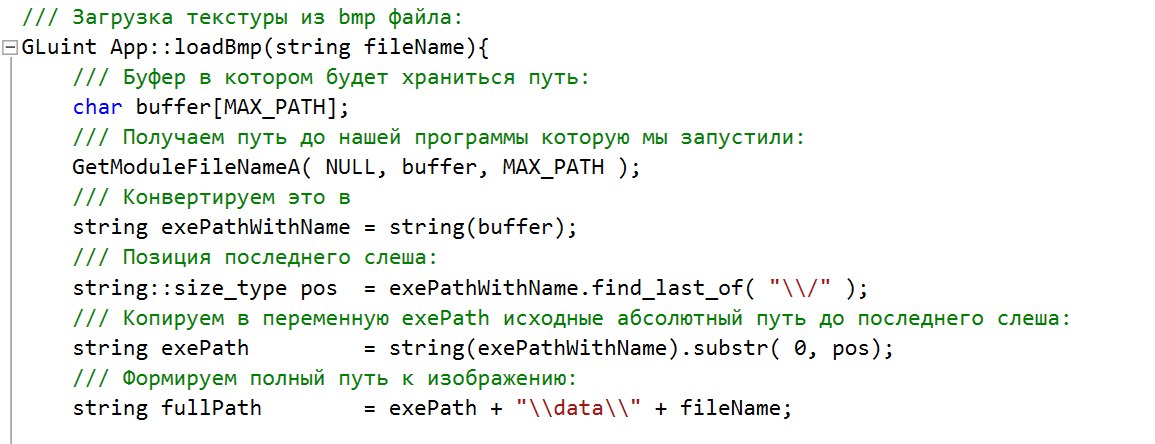


После загрузки картинки из файала и передачи её пикселей в OpenGL именно эта переменная будет ассоциацией с загруженной текстурой.

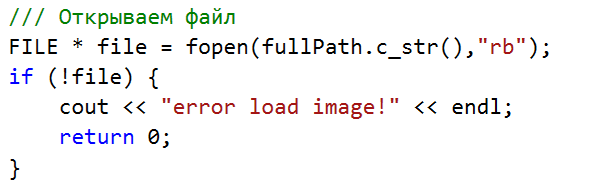
Теперь давайте разберем, как читать BMP файл. Сначала нам нужно прочитать некоторые данные. Нужно установить следующие переменные при чтении BMP файла:

*unsigned char header[54]; /// каждый BMP файл начинается с 54 байтного заголовка*  
*unsigned int dataPos; /// Позиция в файле где сами данные начинаются*  
*unsigned int width, height;*  
*unsigned int imageSize; /// = ширина\*высота\*3*  
*/// Сами RGB данные в массиве произвольной длины из unsigned char*  
*unsigned char \* data;*

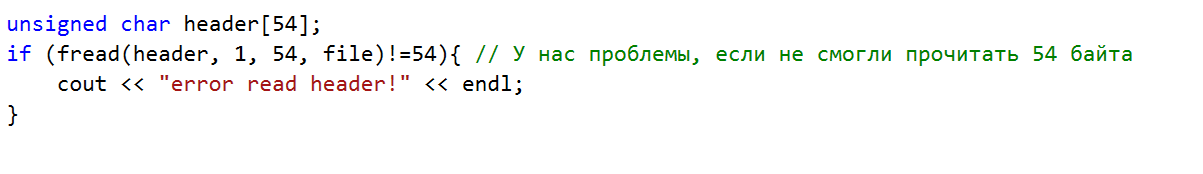
Займемся реализацией функции загрузки в файле app.cpp. Поскольку все изображения, ровно как и шейдеры будут лежать в папке data рядом c exe файлом программы, то для начала нам нужно вычислить корректный абсолютный путь до картинки которую мы будем загружать как текстуру opengl:



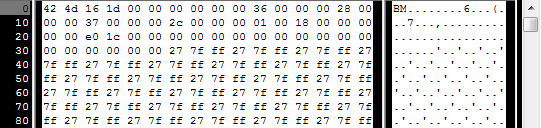
Теперь, имея полный абсолютный путь к нашей картинки fileName можно прочитать её как бинарный файл(rb - режим только чтение для бинарного файла):



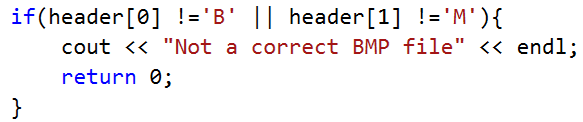
BMP файл начинается с 54 байтного заголовка. Этот заголовок содержит такую информацию как: «Это действительно BMP файл?», размер изображения, количество битов на пиксель, итд. Давайте сначала прочтем заголовок:

Функция fread возвращает размер прочитанных данных.

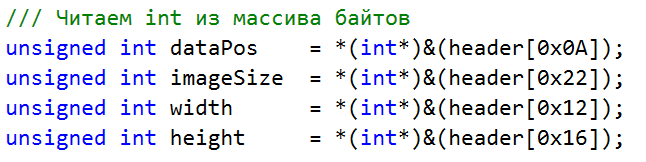
Каждый файл BMP начинается с заголовка BM. Это можно увидеть открыв BMP файл в HEX редакторе:



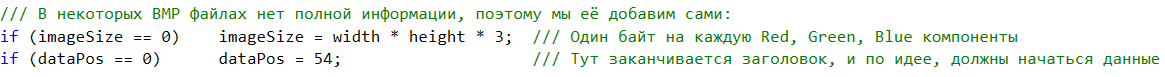
Исходя из этого мы должны проверить первые 2 символа в заголовке:



Когда заголовок у нас прочитан, мы можем получить смещение в файле на данные, и размер картинки. dataPos - позиция откуда у нас начинают идти пиксельные данные в прочитанном файле:



В некоторых BMP файлах нет полной информации, поэтому мы её добавим сами:



Теперь когда у нас есть вся необходимая информация, мы можем выделить память под картинку и прочитать в неё данные:

/// Создаем буфер:

unsigned char \* data = new unsigned char [imageSize];

/// Читаем данные из файла в буфер:

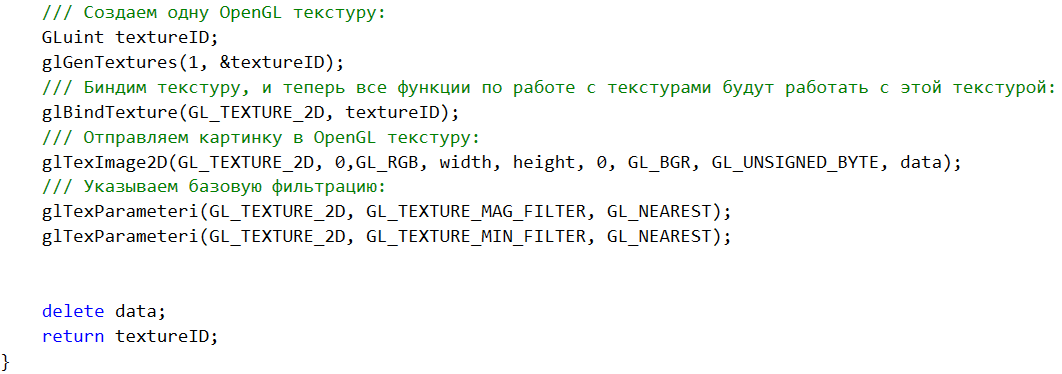
fread(data,1,imageSize,file);

/// Теперь все данные в памяти, и можно закрыть файл:

fclose(file);

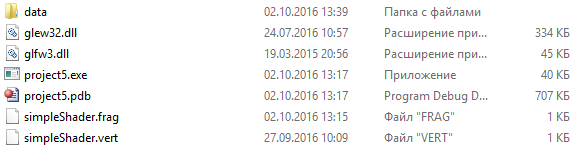
Но это все было не совсем связанное с OpenGL, а теперь будет поинтереснее. Создание текстур, это почти то же самое, что и создание вершинных буферов: создаем текстуру, биндим её, заполняем и конфигурируем.

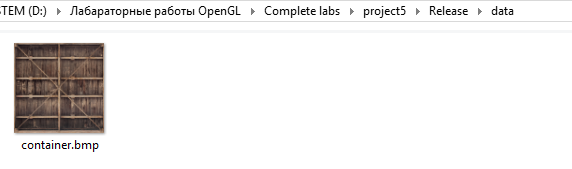
В функции glTexImage2D,GL\_RGB указывает на то, что мы работаем с трехкомпонентным цветом, а GL\_BGR указывает на то, как этот цвет хранится в памяти. По какой-то исторической причине в BMP файлах цвет хранится не как Red-Green-Blue, а как Blue-Green-Red, о чем и нужно уведомить OpenGL:



Таким образом мы полностью сделали реализацию функции загрузки изображения – loadBmp();

Создадим папку data рядом с exe файлом и перенесем туда файл "container.bmp"



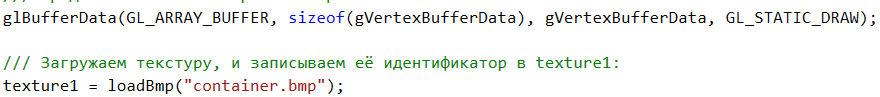


Еще одно важное замечание: Используйте текстуры со значениями ширины и высоты как степень двойки. :

* хорошо : 128\*128\*, 256\*256, 1024\*1024, 2\*2…
* плохо : 127\*128, 3\*5, …
* хорошо, но странно : 128\*256

Это позволит оптимально хранить текстуры в видеопамяти GPU.

В функции setupScene вызовем нашу функцию, вызовем эту её сразу после gBufferData:



Теперь рассмотрим необходимые изменения фрагментного шейдера для наложения текстуры на модель:

#version 330 core

/// Интерполированное значение текстурных координат из вершинного шейдера:

in vec2 UV;

/// Выходной цвет фрагмента-пикселя:

out vec3 color;

/// Текстура полученная из OpenGL:

uniform sampler2D shaderTexture1;

void main(){

/// Результирующий цвет — цвет точки в текстуре по координатам UV

color = texture(shaderTexture1,UV).rgb;

}

Три вещи на которые нужно обратить внимание:

* Фрагментный шейдер нуждается в UV координатах
* Нам нужен так называемый сэмплер(sampler2D), чтобы шейдер знал из какой текстуры извлекать цвет
* Доступ к цвету фрагмента из текстуры происходит с помощью функции texture() которая возвращает нам цвет в формате (R,G,B,A) vec4. Вскоре мы рассмотрим что такое A компонента.

Вершинный шейдер тоже очень простой:

#version 330 core

/// Передача вершин(число 0)

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

/// Передача цветов вершин(число 1)

layout(location = 1) in vec2 vertexUV;

uniform mat4 MVP;

// Исходящие данные: будут интерполированы для каждого фрагмента

out vec2 UV;

void main(){

/// vertexPosition\_modelspace - координаты вершин получаемых из VAO:

/// Координата вершины, c w = 1.0:

vec4 v = vec4(vertexPosition\_modelspace,1);

gl\_Position = MVP \* v;

/// координата UV данной вершины. Тут никаких особых преобразований не нужно делать:

UV = vertexUV;

}

Что такое “layout(location = 1) in vec2 vertexUV”? Если забыли, посмотрите в лабораторной работе 4. Здесь мы будем делать почти то же самое, вот только вместо триплетов цвета, мы передадим в шейдер пары координат UV.

Теперь подготовим нашу 3D модель для текстурирования, сначала слегка поменяем описание вершин куба из прошлой работы, просто заменив старый код на новый:

/// Массив из 3 векторов которые будут представлять 3 вершины:

static const GLfloat gVertexBufferData[] = {

/// 1 Грань:

-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,

/// 2 Грань:

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,

/// 3 Грань:

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,

/// 4 Грань:

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,

/// 5 Грань:

-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,

/// 6 Грань:

1.0f, 1.0f, 1.0f,

1.0f, 1.0f,-1.0f,

-1.0f, 1.0f,-1.0f,

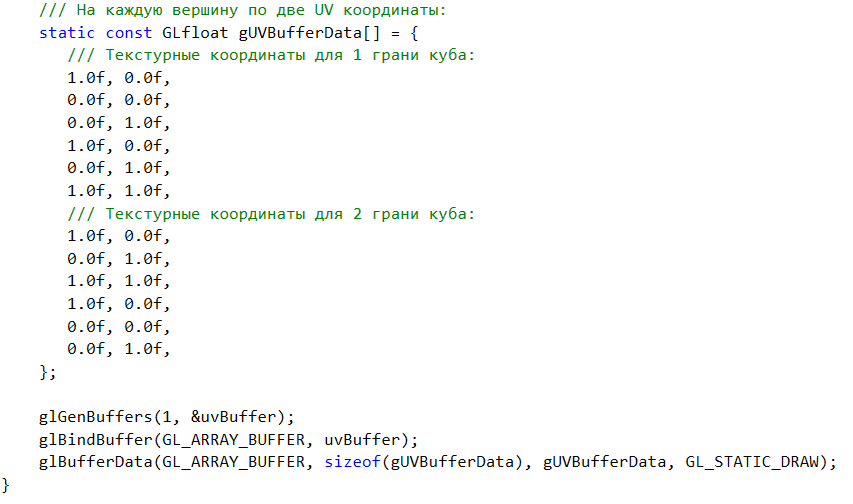
1.0f, 1.0f, 1.0f,

-1.0f, 1.0f,-1.0f,

-1.0f, 1.0f, 1.0f,

};

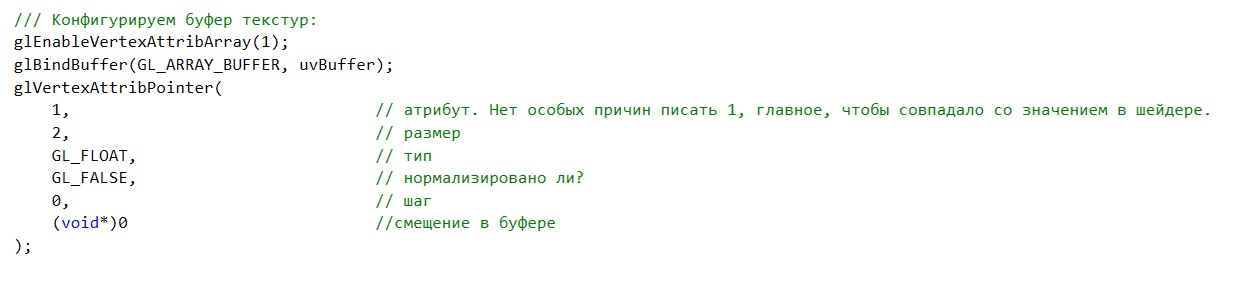
Теперь в функции setupScene() мы вместо буфера цвета создадим буфер текстурных координат и заполним его координатами для 2 граней нашего куба:



2 числа - UV координата для вершины треугольника, 2 треугольника на грань, в каждом 3 вершины.

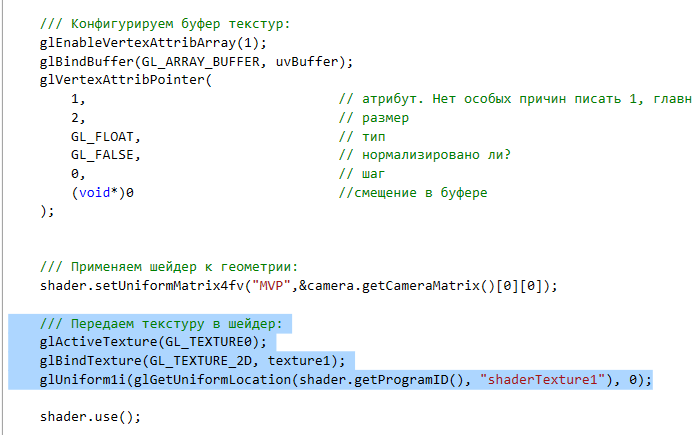
Обратите внимание на сходство работы с буферами и текстурами - по сути одни и те же действия. Создаем ID, далее выбираем его(Bind) и затем передаем данные по идентификатору.

Теперь в функции draw() созданный буфер текстурных координат нужно правильно с конфигурировать, по аналогии как мы это делаем когда передаем координаты вершин и цвета вершин. Делаем это вместо конфигурирования буфера цветов, то есть старый код удаляем и вставляем этот:

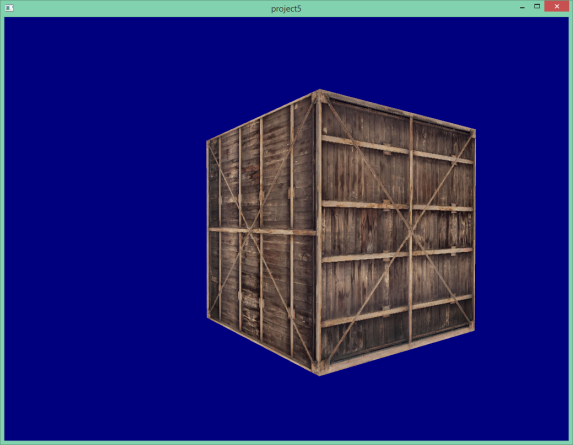
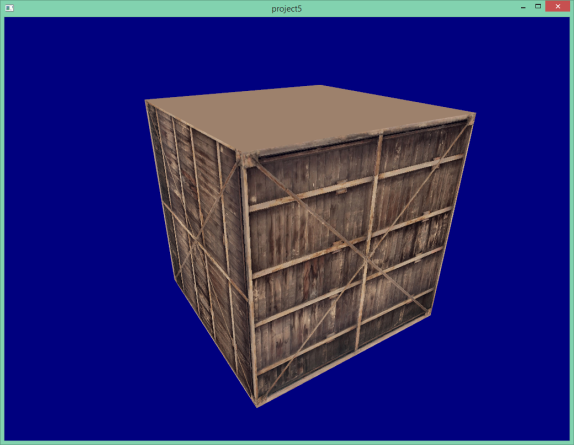
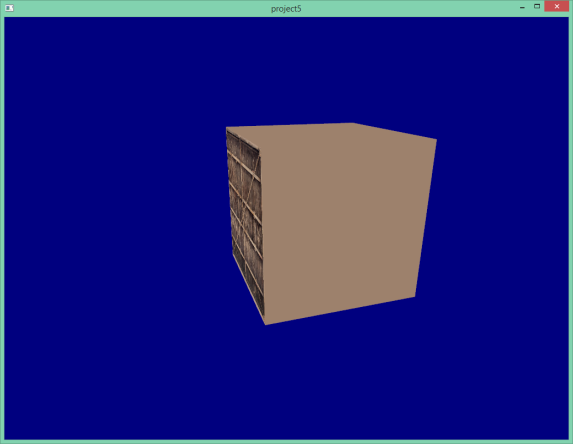
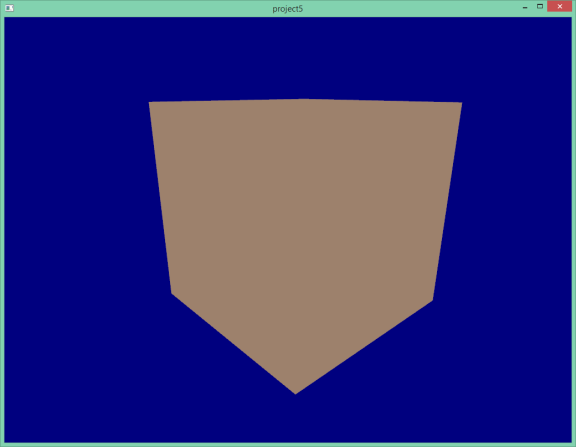


Если внимательно посмотреть то отличия всего 2 - первое это размер, если там было 3(RGB цвета) то здесь их 2. Ну и собственно в glBindBuffer() передается другой буфер - uvBuffer - буфер текстурных координат.

Последним шагом передаем текстуру в шейдер:



Первая функция выбирает активную текстурную единицу для шейдера, вторая функция делает bind загруженной текстуры. 3 функция передает активную текстурную единицу которая теперь хранит нашу текстуру в шейдер в качестве uniform переменной. Теперь запустим приложение и убедимся что текстура накладывается на 2 грани нашего куба:

Обратите внимание на песочный цвет граней для которых не указаны текстурные координаты. По умолчанию шейдер берет цвет с координатами UV(0,0) и применяет его с интерполяцией для всех вершин для которых не указаны текстурные координаты.

Теперь для удобства навигации давайте добавим движение камеры вверх/вниз. Это будет очень удобно при ручном добавлении текстурных координат к 4 оставшимся граням куба. Для этого идите в файл SimpleCamera.cpp, в функцию update(). Там перед расчетом матриц добавьте условия:

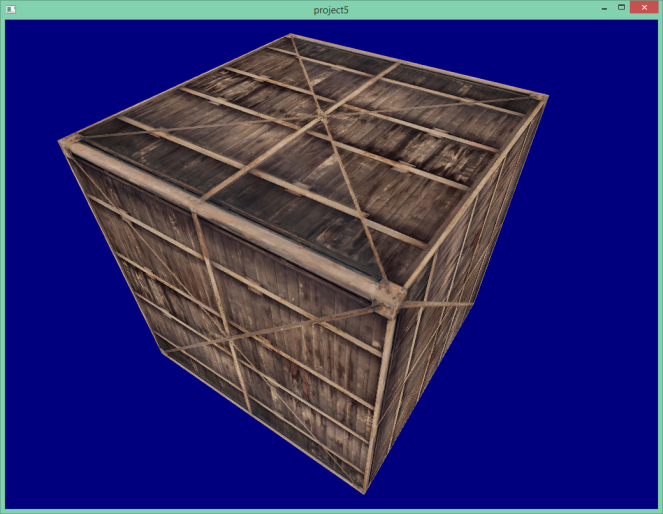
/// Движение вверх и вниз:

if (glfwGetKey(window, GLFW\_KEY\_Q ) == GLFW\_PRESS) {}

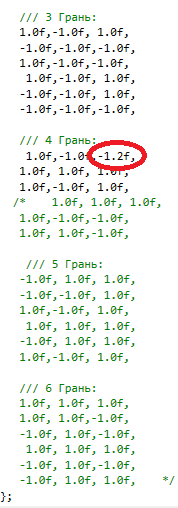
if (glfwGetKey(window, GLFW\_KEY\_E ) == GLFW\_PRESS) {}

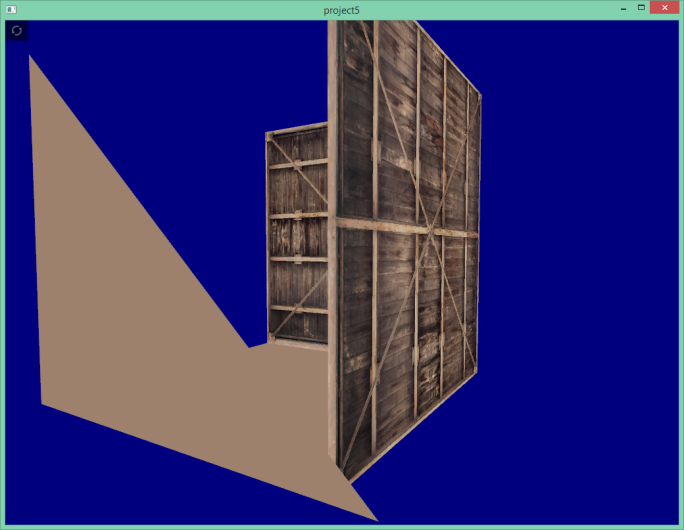
В фигурных скобках вы самостоятельно напишите код который позволит камере двигаться вверх и вниз.

Затем вы самостоятельно добавите текстурные координаты для оставшихся 4 граней, чтобы получился полностью затекстурированный куб имитирующий деревянный ящик:



Дам вводные подсказки. Для корректного наложения текстуры, вам важно понимать какая вершина является первой у треугольника, а какая второй и третьей. Это можно легко понять незначительно изменяя координата вершин куба. Также это будет проще увидеть, если вы будете работать с каждой отдельной гранью или даже треугольником грани, скрывая все лишнее:





Сравнивайте расположение треугольника грани куба и смотрите на первые рисунке в этой лабораторной, где показано как накладывается текстура на треугольник с помощью UV координат.

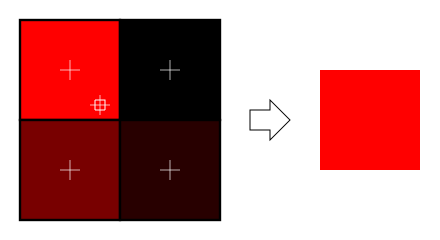
Если справились с выше перечисленными задачами то теперь рассмотрим фильтрацию текстур. Давайте подлетим поближе к одной из граней куба и рассмотрим текстуру внимательнее:



Как видно из предыдущего увеличенного скриншота, качество текстуры не сильно хорошее. Но это потому что в загрузчике текстуры мы написали:

*glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);*  
*glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);*

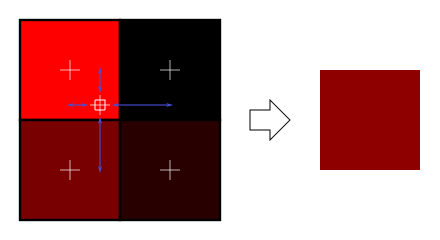
Это значит, что наш фрагментный шейдер в функции texture() просто берет значение текселя по координатам UV и выводит его на экран:



Но у нас есть в запасе несколько механизмов которые помогут улучшить картинку.

**Линейная фильтрация**

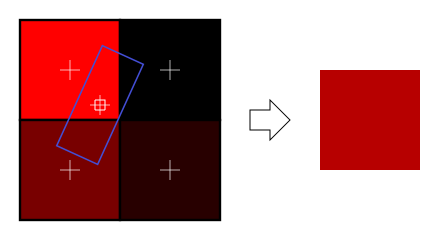
Когда включена линейная фильтрация, функция texture() смотрит не только на один конкретный тексель, но и на несколько вокруг него и смешивает их цвета. Из-за чего у нас пропадает некрасивые четкие зубцы в текстуре:



С такой фильтрацией все становится немножко получше, но есть еще один метод — **Анизотропная фильтрация**, который дает еще более красивую картинку. Но он значительно медленнее.

**Анизотропная фильтрация**

По этому методу мы аппроксимируем картинку которая действительно должна быть видима через фрагментный шейдер. Например, если мы будем смотреть на эту текстуру сбоку и немного повернуто, то анизотропная фильтрация будет вычислять цвет расположенный в голубом прямоугольнике беря фиксированное число выборок(анизотропный уровень) вдоль основного направления:



Теперь давайте добавим линейную фильтрацию текстур, а для большей наглядности сделаем включение/выключение интерполяции в реальном времени с клавиатуры. Для этого добавим в функцию void App::key\_callback следующие строки кода:

if (key == '1') {

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

}

if (key == '2') {

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

}

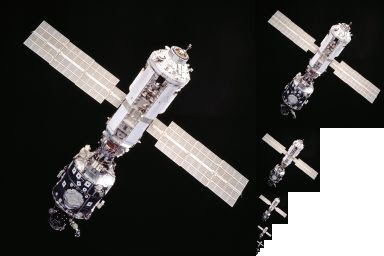
Теперь выполним сборку, и запустим программу:





Результат очевиден.

Теперь рассмотрим такую важную вещь как **Мипмаппинг**. И у линейной и у анизотропной фильтрации есть проблема. Если текстура видна из далека, то смешивания четырех смежных текселей не поможет. Например наша 3д модель может быть расположена так далеко, что она будет занимать всего один пиксель на экране. В таком случае результирующая точка должна быть усредненным цветом всех текселей текстуры. Но видеокарта этого не будет делать из-за того, что это будет слишком долго. Вместо этого придумали другой метод — мипмаппинг:



* Во время загрузки текстуры мы изменяем её размер в два раза, потом результирующую еще в два раза и.т.д, пока у нас не получится текстура 1х1 (все промежуточные сохраняем, конечно же).
* Когда рисуем модель, то видеокарта выбирает наиболее подходящую текстуру среди мипмапов в зависимости от того, каким должен быть тексель по размеру.
* Делать выборку текселя из подходящего мипмапа можно любым способом, хоть nearest хоть linear хоть anisotropic.
* Если хочется еще лучшего качества, можно попробовать делать выборку из двух ближайших мипмапов и смешивать полученные цвета.

К нашему счастью все эти алгоритмы нам делать заново не нужно. OpenGL все умеет делать сам, если его хорошенько воззвать его к нужным функциям.

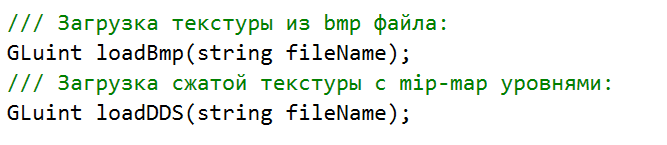
В специальных программах, таких как "**The Compressonator**" создаются текстуры содержащие внутри уменьшенные экземпляры самих себя(мип уровни). Более того, такие такие текстуры сжимаются специальные алгоритмами. Эти сжатые текстуры прямо совместимы с GPU. Теперь когда вы будете вызывать функцию texture() из GLSL, видеокарта автоматически на лету будет распаковывать данные. На первый взгляд это может показаться медленным, но на самом деле это даёт следующие преимущества:

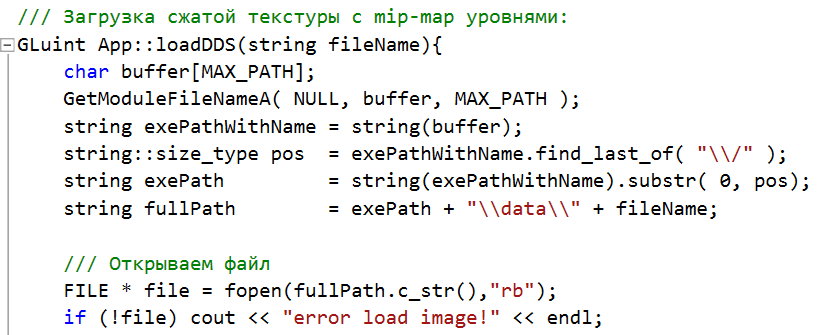
* Значительно меньшее потребление памяти(как обычной RAM, так и памяти видео карты)
* Намного меньше затраты на передачу данных из оперативной памяти в GPU.
* Распаковка сжатых данных совсем не замедляет обработку так как это происходит аппаратно в GPU. То есть графические процессоры обладают аппаратными возможностями декодирования таких сжатых текстур.

Сейчас мы реализуем загрузку сжатой DXT-5 текстуры содержащие mip уровни. Загруженную текстуру мы также будем накладывать на куб.

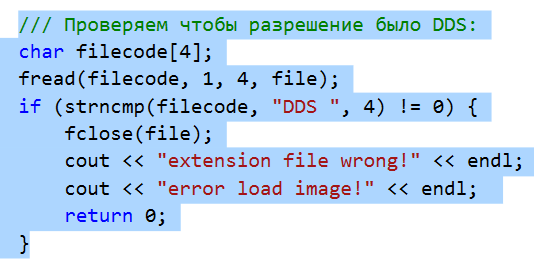
**Как использовать сжатые текстуры.**

Давайте сначала посмотрим как загружать сжатое изображение. Это очень похоже на загрузку BMP, вот только заголовок организован слегка иначе. И так по аналогии с функцией loadBmp() добавим заголовок и реализацию новой функции loadDDS():

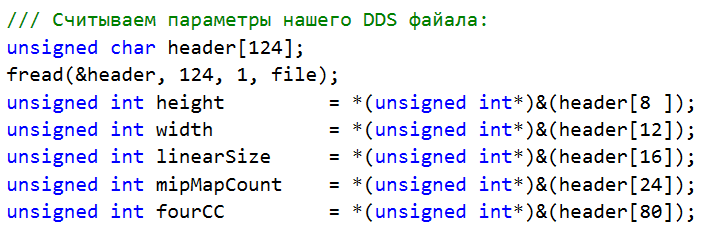




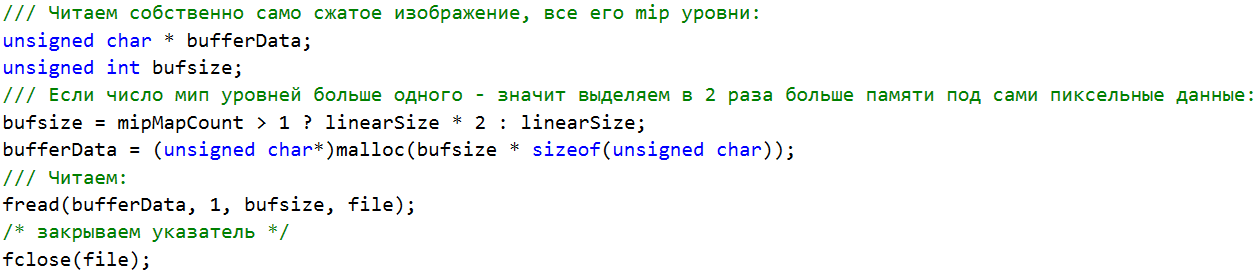
Как видите начало функции абсолютно идентично функции loadBmp(), а вот дальше уже пойдут отличия:



Затем считываем заголовок и получаем необходимые параметры:



Параметры: ширина/высота картинки, её размер в байтах, число мип уровней, и какой алгоритм сжатия использовался. Теперь читаем пиксели изображения:



Добавим необходимые константы в файл app.h, Прямо перед объявлением класса:

#define FOURCC\_DXT1 0x31545844 // Equivalent to "DXT1" in ASCII

#define FOURCC\_DXT3 0x33545844 // Equivalent to "DXT3" in ASCII

#define FOURCC\_DXT5 0x35545844 // Equivalent to "DXT5" in ASCII

Далее определяем тип сжатия полученных данных:

/// Определяем используемый формат сжания доступный для OpenGL:

unsigned int components = (fourCC == FOURCC\_DXT1) ? 3 : 4;

/// Формат сжатия OpenGL:

unsigned int format;

/// Выбираем доступный формат OpenGL:

switch(fourCC){

case FOURCC\_DXT1:

format = GL\_COMPRESSED\_RGBA\_S3TC\_DXT1\_EXT;

break;

case FOURCC\_DXT3:

format = GL\_COMPRESSED\_RGBA\_S3TC\_DXT3\_EXT;

break;

case FOURCC\_DXT5:

format = GL\_COMPRESSED\_RGBA\_S3TC\_DXT5\_EXT;

break;

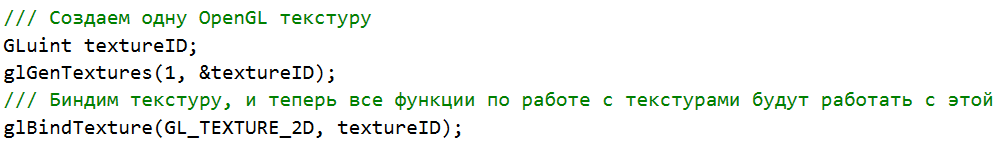
default:

free(bufferData);

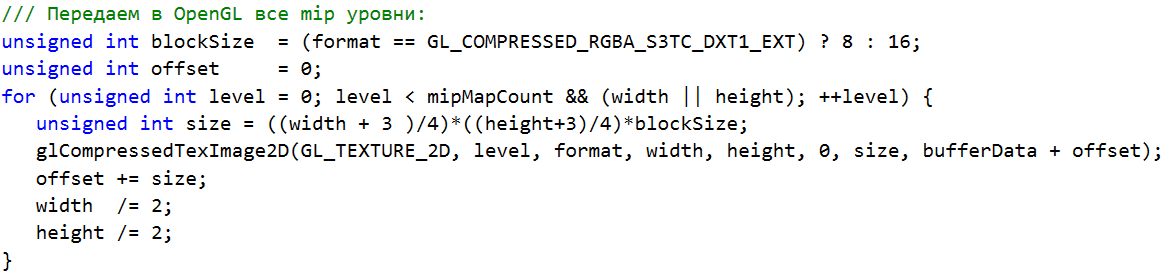
return 0;

}

Создание текстуры такое же как и всегда:

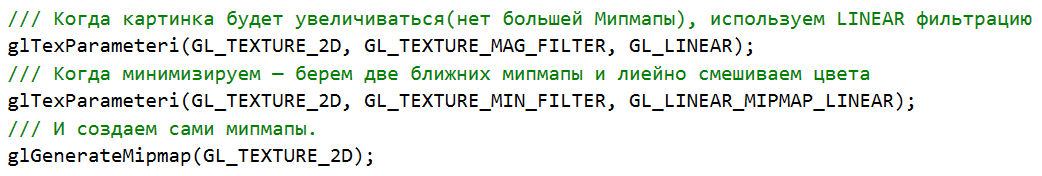


А вот теперь рассмотрим самое интересное:

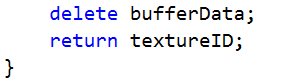


Как видите, мы запускаем цикл, который итерационно добавляет мип уровни в OpenGL, передавая соответствующий фрагмент пикселей в OpenGL. То есть, здесь мы за каждую итерацию передаем по одному мип уровню - одному изображению в GL. Начинаем с самого большого мип уровня - оригинального изображения с исходной шириной и высотой. Затем в следующею итерацию мы берем следующий мип уровень, который меньше в 2 раза чем предыдущий. И так до тех пор пока не загрузим все мип уровни(mipMapCount) и соответственно ширина и высота не станут равны нулю (именно по этому width и height переменные типа int).

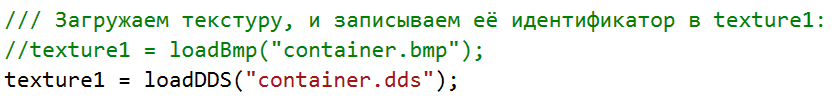
Осталось задать интерполяции и собственно сгенерировать из переданных данных MipMap текстуру которую мы и будем накладывать на наш куб:



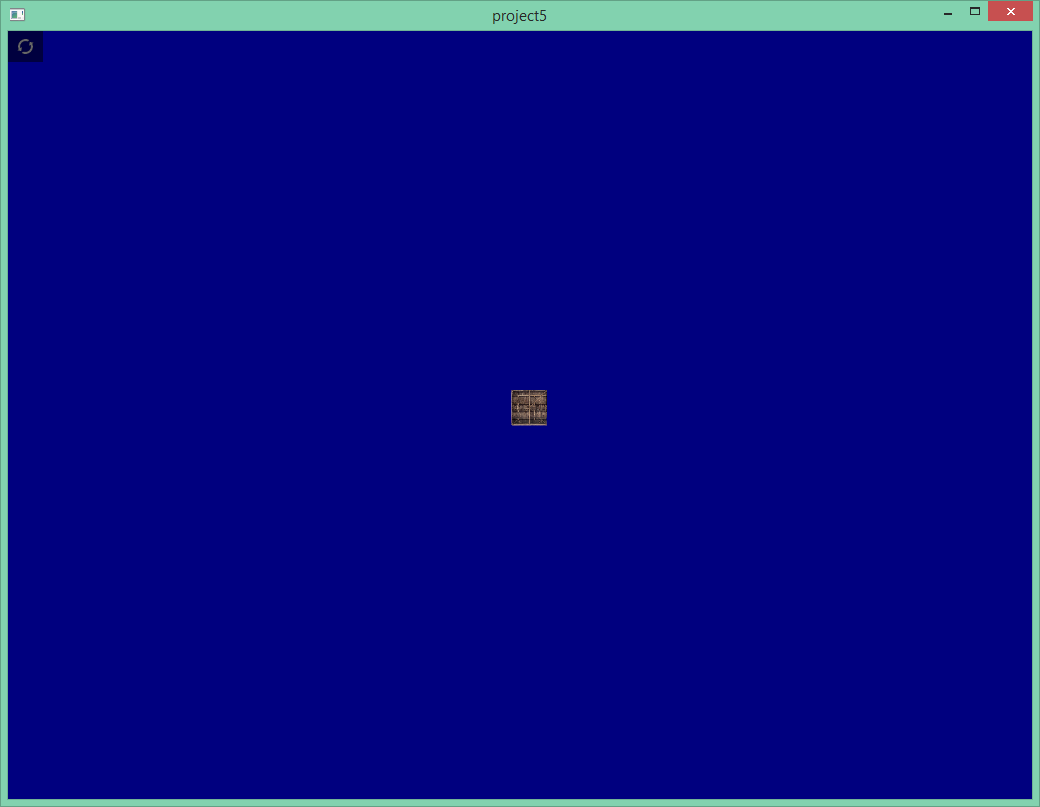
Ну и в самом конце также как в loadBmp вернуть texureID и удалить временный буфер с данными из памяти:



Теперь перенесем в папку data из папки Lab\_5 файл "container.DDS" этот файл полная копия файла Bmp, с тем отличием что он сжат DXT-5 и содержит мип уровни вплоть до картинки 1\*1 пиксель. Затем изменим в setupScene() строчку загрузки текстуры:



Теперь приложение будет загружать сжатутю текстуру с мипмаппингом. Давайте сравним изменения в качестве. Вот как выглядит обычная текстура без сжатия и мапмап уровней:



А так выглядит вариант с мипмап уровнями и сжатием:



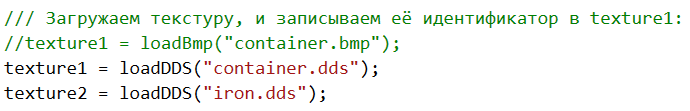
Как видите на втором изображении нет выраженных пиксельных артефактов, поскольку подбирается оптимальная по размеру текстура из мип-мап уровней. В статичной картинке это не так заметно как в движении, в реальном времени. Удостоверьтесь в этом сами.

Теперь мы перейдем к последней части нашей лабораторной работы, где мы рассмотрим возможность одновременного наложения на объект нескольких текстур с определенным алгоритмом смешивания на уровне шейдера. также рассмотрим передачу дополнительных параметров в шейдер.

Скопируйте из папки Lab\_5 в папку data файл "iron.DDS" это текстура для нашего железного ящика.

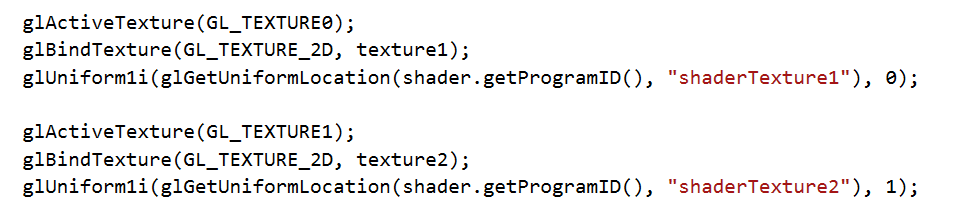
В файле App.h, в класс App добавьте еще один идентификатор: GLuint texture2; и еще переменную float mixFactor; При помощи неё мы будем смешивать 2 текстуры.

В функции setupScene() добавьте загрузку еще одной текстуры:



И приравняем mixFactor к 0 сразу после загрузки текстур.

Затем в функции draw добавьте передачу еще одной текстуры в шейдер:

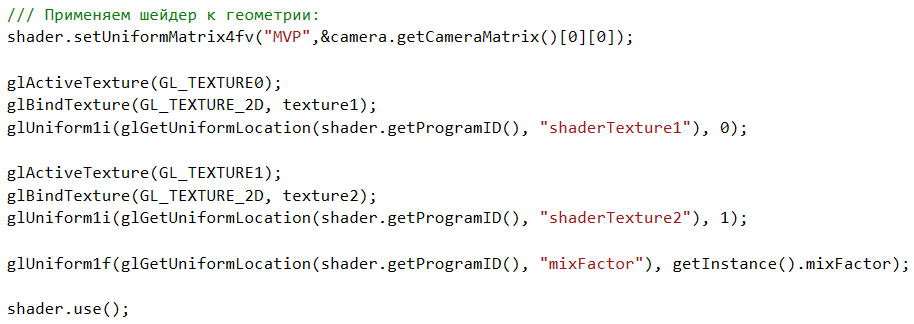


Обратите внимание на последний аргумент glUniform1i он равен 1 во втором случае, и также на аргумент glActiveTexture - GL\_TEXTURE1. Тоже единица. Это говорит о том что мы передаем в шейдер активную текстуру номер 1. За этой активной текстурой закреплена как раз вторая текстура которую мы будем накладывать на куб вместе с первой.

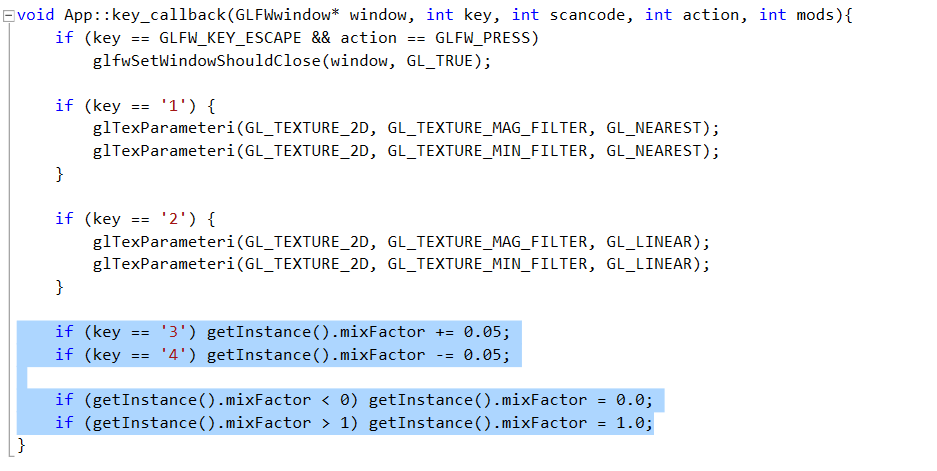
Передадим mixFactor в шейдер:

glUniform1f(glGetUniformLocation(shader.getProgramID(), "mixFactor"), getInstance().mixFactor);

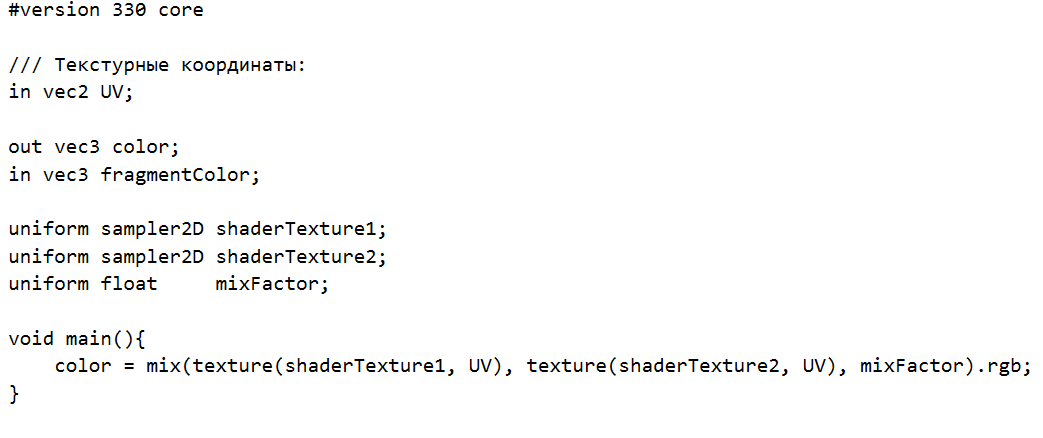
Должно получится так:



Теперь в обработчике клавиатуры пропишем взаимодействие с переменной mixFactor:



Затем модернизируем фрагментный шейдер:

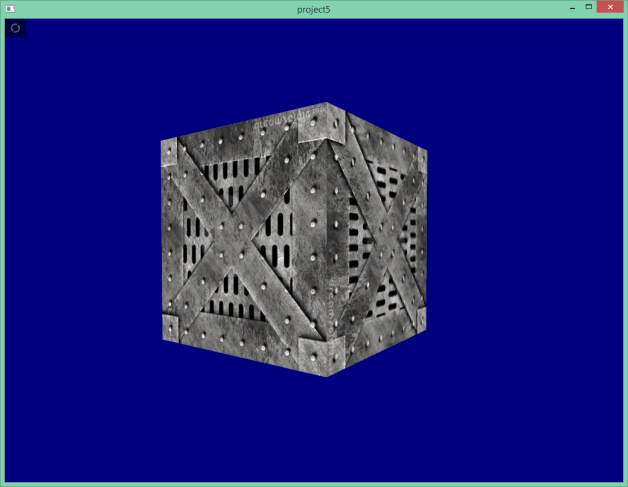
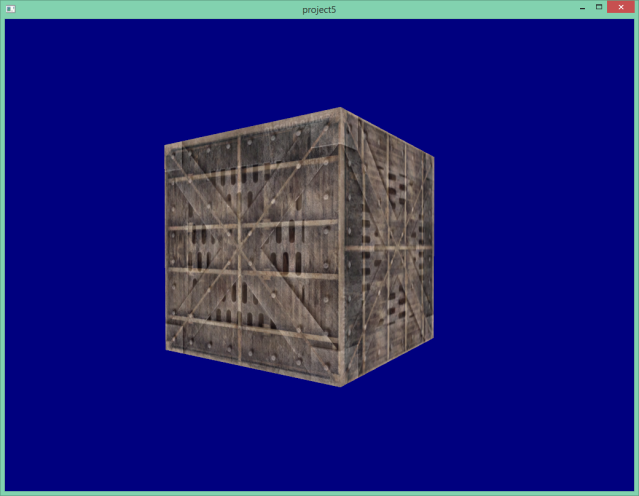


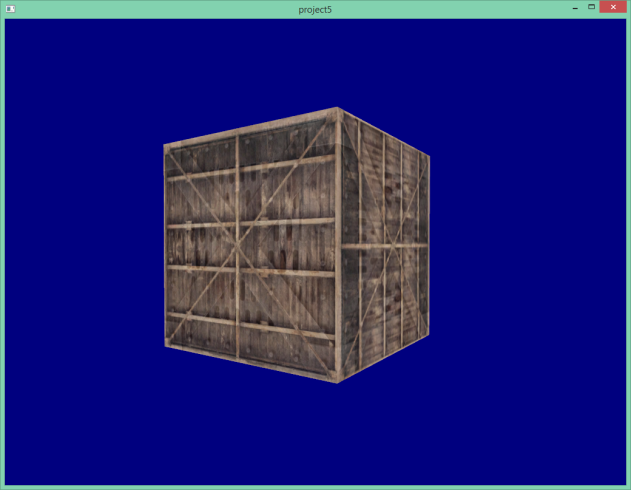
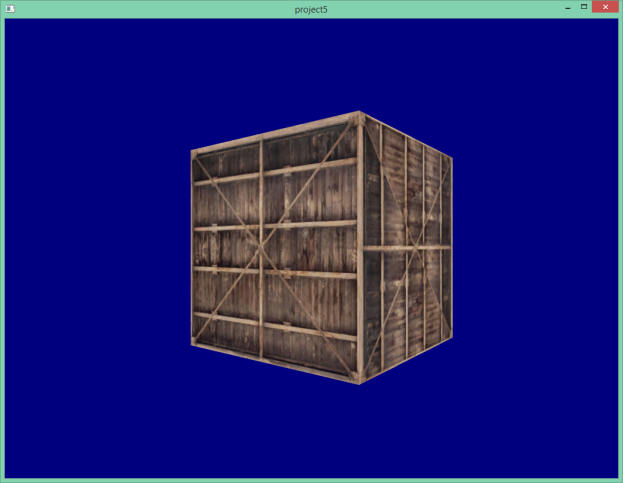
Как видите изменения минимальные, мы добавили 2 текстуру и дополнительную переменную uniform - mixFactor.

Самое интересное в функции main(). Здесь мы получаем два пикселя(2 цвета по сути) с первой и второй текстуры соответственно.

Далее мы выполняем линейное, пропорциональное смешивание этих двух цветов. Смешивание происходит в пропорции mixFactor. То есть, если mixFactor=0.2 значит у нас будет 20 процентов от одного цвета и 80 процентов второго цвета. Здесь, для этого мы используем готовую шейдерную функцию mix() которая делает это используя графический процессор видеокарты. Попробуйте написать свою реализацию смешивания двух изображений, аналогичной функции mix. Попробуйте сделать смешивания двух изображений используя сумму, разность и умножение 2 пикселей.

Что получается в итоге при разных значениях смешивания:

На этом лабораторная 5 завершена.