



# Начало работы с OpenGL

<b>1</b>	<b>Первое приложение с OpenGL .....</b>	<b>2</b>
1.1	Предварительная подготовка	
1.2	Добавление элементов рисунка	
1.3	Отрисовка трехмерной сцены	
1.4	Назначение горячих клавиш	
1.5	Добавление файлового диалога	
1.6	Манипуляции с мышью	
1.7	Задание для самостоятельной работы	
1.8	Контрольные вопросы	
<b>2</b>	<b>Освещение .....</b>	<b>64</b>
2.1	Предмет разработки	
2.2	Шейдеры	
2.3	Изменение формата представления объектов сцены	
2.4	Источник света	
2.5	Модель освещения	
2.6	Передвижение источника света	
2.7	Задание для самостоятельной работы	
2.8	Контрольные вопросы	

# 1. Первое приложение с OpenGL

В этом уроке мы создадим каркас приложения, использующего средства OpenGL. Для этого проведем установку некоторых дополнительных средств.

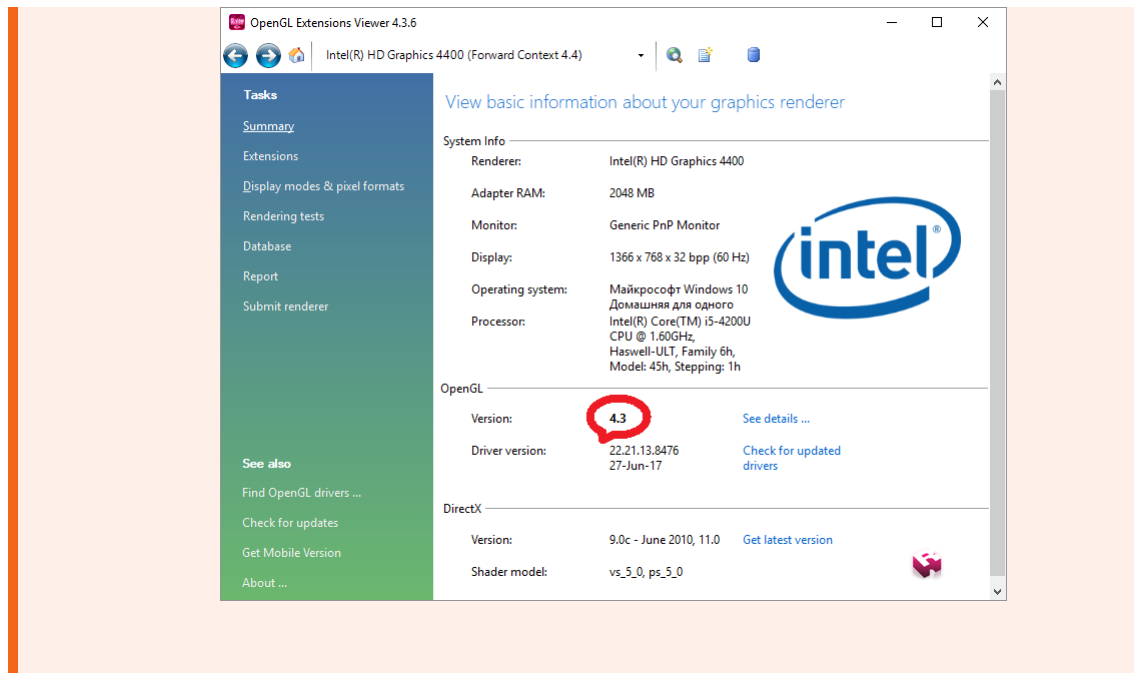
Следует отметить, что серия уроков, посвященных OpenGL создана на основе материала, изложенного в публикации “Learn OpenGL” по адресу <https://learnopengl.com/> (на английском языке). Переводы отдельных уроков этого ресурса можно найти на сайте <https://habrahabr.ru> с помощью поиска по ключу «learnopengl». Рекомендуется обращаться к этому on-line учебнику для более подробных разъяснений.

## 1.1 Предварительная подготовка

В наших приложениях мы будем использовать OpenGL версии 3.3. Для корректной работы проекта необходимо, чтобы драйверы видеокарты поддерживали эту версию. Поэтому, рекомендуем обновить драйверы до последней версии



Узнать текущую версию OpenGL можно, например, с помощью утилиты OpenGL Extension Viewer, которую можно получить по ссылке [http://download.cnet.com/OpenGL-Extensions-Viewer/3000-18487\\_4-34442.html](http://download.cnet.com/OpenGL-Extensions-Viewer/3000-18487_4-34442.html). Следующий рисунок показывает пример запуска этого приложения. На рисунке помечена версия OpenGL. Необходимый номер версии — не ниже 3.3.



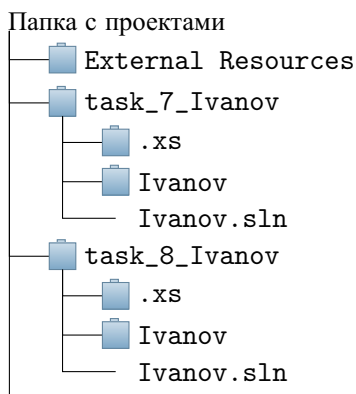
Так как Вы выполняете учебные задания, которые будут отправлены на проверку и будут запускаться проверяющей стороной в системе, отличной от Вашей, очень важно, чтобы Вы не отступали от инструкций этого раздела.

В ходе этого урока Вы установите несколько библиотек. Для того, чтобы повысить Вашу мобильность при создании проекта, библиотеки будем устанавливать в портативной форме: создадим набор папок с файлами библиотек, не проводя их установку в системе, а добавляя эти библиотеки в каждый наш проект, их использующий. Это позволит копировать проект вместе с дополнительными файлами с одного компьютера на другой, не проводя дополнительной установки.

Общий вес такого набора файлов составит несколько дополнительных мегабайт. Поэтому было бы затратно дублировать этот набор с каждым проектом и, тем более, загружать его как часть решения. Вместо этого условимся об определенном расположении папки с библиотеками относительно наших проектов.

У Вас имеется папка, в которой создаются Ваши проекты Visual Studio. Это может что-то, вроде `"C:\tmp\Ivanov"`, а может быть папка по умолчанию, как, например, `"C:\Users\Ivanov\Documents\Visual Studio 2017\Projects"` (фамилия Ivanov здесь используется для примера, как и раньше).

Будем предполагать, что папки проектов для седьмого и последующих заданий будут находиться в одной папке. В ней же создадим папку `"External Resources"`.



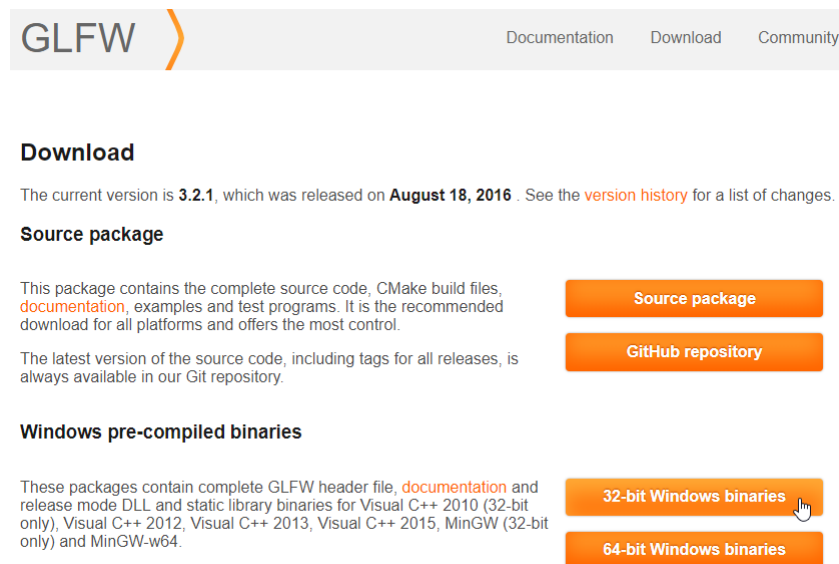
Все необходимые библиотеки будем добавлять в папку **"External Resources"**.



Здесь и в дальнейшем при установке дополнительных компонент не ошибайтесь в написании наименований папок. В противном случае Ваш проект может не компилироваться у проверяющей стороны, а следовательно решение задания может быть не засчитано.

### 1.1.1 Библиотека GLFW

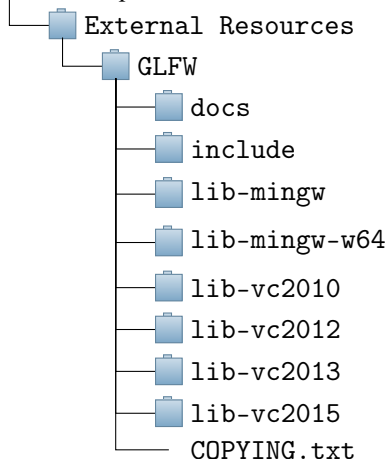
Средства OpenGL позволяют обрабатывать графическую информацию для получения изображения и его вывода. Однако создание окна (окон) для рисования и организация пользовательского интерфейса с таким окном — задачи, специфичные для каждой операционной системы. Поэтому OpenGL целенаправленно пытается абстрагироваться от этих операций. Библиотека GLFW (наряду с другими библиотеками, такими как GLUT, SDL, SFML) предоставляет такие средства. Для того, чтобы её скачать и установить, зайдите на официальный сайт <http://www.glfw.org/download.html> и в подразделе “Windows pre-compiled binaries” нажмите кнопку “32-bit Windows binaries” (см. рисунок)



В результате будет скачан файл с актуальной версией GLFW. На момент написания методических инструкций имя файла `glfw-3.2.1.bin.WIN32.zip`.

Внутри папки `External Resources` создайте папку с именем `GLFW` в которую распакуйте содержимое архива. Должна получиться такая структура папок

Папка с проектами



Набор папок из архива соответствует версии GLFW, актуальной на момент составления методических инструкций, и может отличаться от версии на момент Вашей установки.



За русскоязычной инструкцией по компиляции библиотеки для систем, отличных от Windows, можно обратиться по адресу <https://habrahabr.ru/post/311198/>, пункт *Сборка GLFW*.

### 1.1.2 Библиотека GLAD

Поскольку OpenGL является стандартом/спецификацией, разработчик драйвера может по-своему реализовывать эту спецификацию для конкретной видеокарты. Поскольку существует множество версий драйверов OpenGL, расположение большинства его функций неизвестно во время компиляции и должно быть определено во время выполнения. При создании приложения с использованием OpenGL разработчик должен получить информацию о местонахождении функций, которые ему нужны, и сохранить её для последующего использования. Получение этих мест зависит от операционной системы и в Windows выглядит примерно так:

```
// объявляется тип конкретной функции OpenGL
typedef void (*GL_GENBUFFERS) (GLsizei, GLuint*);
// ищется функция и присваивается переменной определенного ранее типа
GL_GENBUFFERS glGenBuffers = (GL_GENBUFFERS)wglGetProcAddress("glGenBuffers");
// теперь функция используется, как обычная функция
unsigned int buffer;
glGenBuffers(1, &buffer);
```

Как вы видите, код выглядит сложным. Для каждой функции такой процесс является достаточно громоздким. Но его можно автоматизировать, используя одну из предназначенных для этого библиотек, например, GLAD или GLEW.

Мы будем использовать библиотеку GLAD. Чтобы установить GLAD, нужно в web-форме на официальном сайте сделать запрос — для какой версии OpenGL нам нужна библиотека. Откройте web-форму по адресу <http://glad.dav1d.de/>.

Glad  
Multi-Language GL/GLSL/OpenGL/GLX/WGL Loader-Generator based on the official specs.

Language: C/C++ ← 1

Specification: OpenGL ← 2

API: gl: Version 3.3 ← 3

gles1: None

gles2: None

gles3: None

Profile: Core ← 4

Extensions:

Search:

GL\_3DFX\_multisample  
GL\_3DFX\_tbuffer  
GL\_3DFX\_texture\_compression\_FXT1  
GL\_AMD\_blend\_minmax\_factor  
GL\_AMD\_conservative\_depth  
GL\_AMD\_debug\_output  
GL\_AMD\_depth\_clamp\_separate

Options:

☒ Generate a loader 5

☐ Omit KHR

☐ Local Files

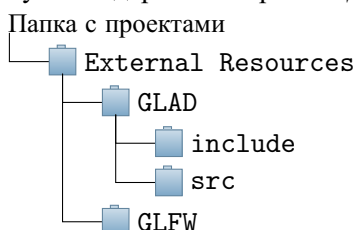
6 ↓

GENERATE

Заполним форму. В поле *Language* выберите язык C++ (пункт 1 на рисунке выше); в поле *Specification* выберите *OpenGL* (пункт 2); в разделе *API* в пункте *gl* выберите версию OpenGL — *Version 3.3* (пункт 3); в поле *Profile* выберите профиль библиотеки GLAD, не включающий функции более ранних версий OpenGL — *Core* (пункт 4). Убедитесь, что отмечен галочкой пункт *Generate a loader* (пункт 5) и нажмите кнопку *GENERATE*.

Откроется страница, с которой Вы сможете загрузить файл *glad.zip* (кликните на его имени для загрузки).

Внутри папки *External Resources* создайте папку с именем *GLAD* в которую распакуйте содержимое архива. Должна получиться такая структура папок

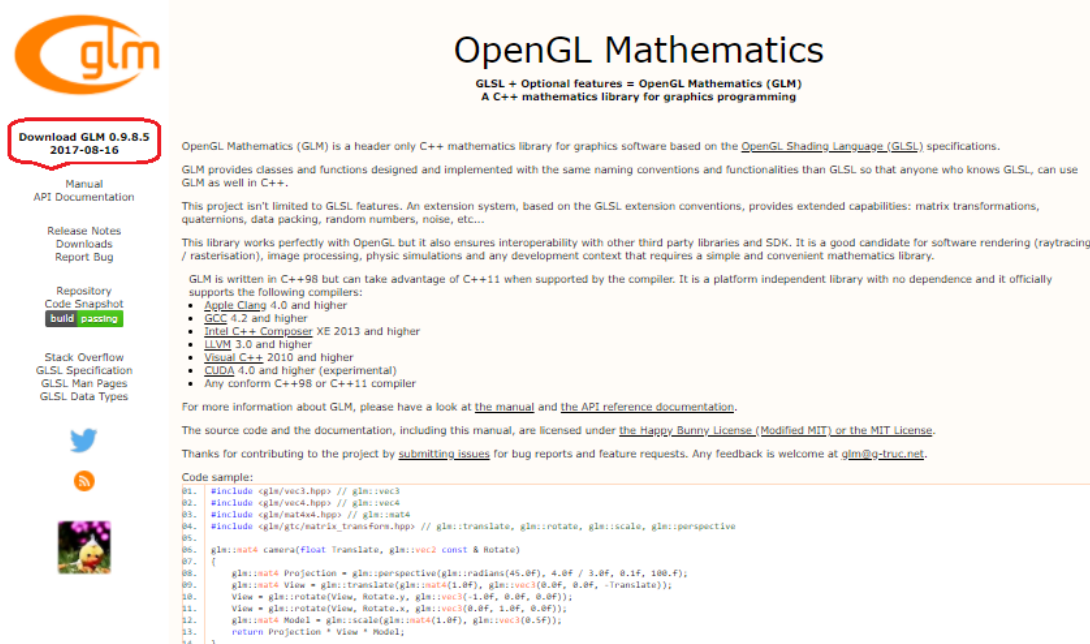


Набор папок из архива соответствует версии GLAD, актуальной на момент составления методических инструкций, и может отличаться от версии на момент Вашей установки.

### 1.1.3 Библиотека GLM

До сих пор Вы использовали собственные реализации матричных операций и функций для получения матриц геометрических преобразований. В дальнейшем, вместо этих реализаций мы будем использовать функции библиотеки (набора библиотек) GLM.

Для установки GLM перейдите на официальный сайт <https://glm.g-truc.net>. В левом верхнем углу страницы Вы найдете ссылку для скачивания актуальной версии библиотеки (см. рисунок).



**Download GLM 0.9.8.5**  
2017-08-16

Manual  
API Documentation  
Release Notes  
Downloads  
Report Bug  
Repository  
Code Snapshot  
build: passing  
Stack Overflow  
GLSL Specification  
GLSL Man Pages  
GLSL Data Types

OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the [OpenGL Shading Language \(GLSL\)](#) specifications. GLM provides classes and functions designed and implemented with the same naming conventions and functionalities than GLSL so that anyone who knows GLSL, can use GLM as well in C++.

This project isn't limited to GLSL features. An extension system, based on the GLSL extension conventions, provides extended capabilities: matrix transformations, quaternions, data packing, random numbers, noise, etc...

This library works perfectly with OpenGL but it also ensures interoperability with other third party libraries and SDK. It is a good candidate for software rendering (raytracing / rasterisation), image processing, physic simulations and any development context that requires a simple and convenient mathematics library.

GLM is written in C++98 but can take advantage of C++11 when supported by the compiler. It is a platform independent library with no dependence and it officially supports the following compilers:

- [Apple Clang](#) 4.0 and higher
- [GCC](#) 4.2 and higher
- [Intel C++ Composer](#) XE 2013 and higher
- [LLVM](#) 3.0 and higher
- [Visual C++](#) 2010 and higher
- [CUDA](#) 4.0 and higher (experimental)
- Any conform C++98 or C++11 compiler

For more information about GLM, please have a look at [the manual](#) and [the API reference documentation](#).

The source code and the documentation, including this manual, are licensed under [the Happy Bunny License \(Modified MIT\)](#) or the [MIT License](#).

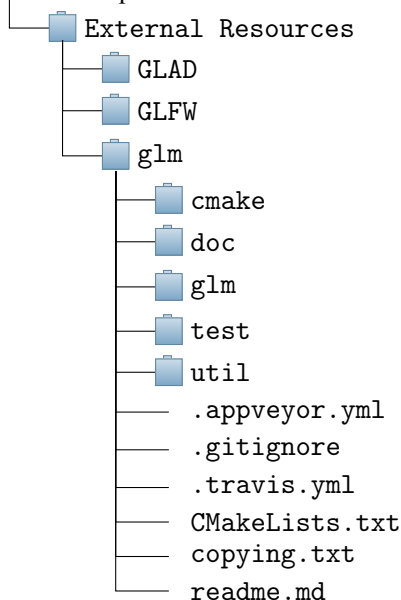
Thanks for contributing to the project by [submitting issues](#) for bug reports and feature requests. Any feedback is welcome at [glm@g-truc.net](mailto:glm@g-truc.net).

Code sample:

```
01. #include <glm/vect3.hpp> // glm::vec3
02. #include <glm/vect4.hpp> // glm::vec4
03. #include <glm/mat4x4.hpp> // glm::mat4
04. #include <glm/gtc/matrix_transform.hpp> // glm::translate, glm::rotate, glm::scale, glm::perspective
05.
06. glm::mat4 camera(float Translate, glm::vec2 const & Rotate)
07. {
08.     glm::mat4 Projection = glm::perspective(glm::radians(45.0f), 4.0f / 3.0f, 0.1f, 100.f);
09.     glm::mat4 View = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, -1.0f), -Translate);
10.     View = glm::rotate(View, Rotate.y, glm::vec3(1.0f, 0.0f, 0.0f));
11.     View = glm::rotate(View, Rotate.x, glm::vec3(0.0f, 1.0f, 0.0f));
12.     glm::mat4 Model = glm::scale(glm::mat4(1.0f), glm::vec3(0.5f));
13.     return Projection * View * Model;
14. }
```

Нажмите на ссылку. Будет скачан файл `glm-0.9.8.5.zip` (имя файла на момент создания методических рекомендаций). В архиве находится папка `glm`. Вытащите папку `glm` из архива со всем её содержимым и поместите внутрь папки `External Resources`. Должна получиться такая структура папок

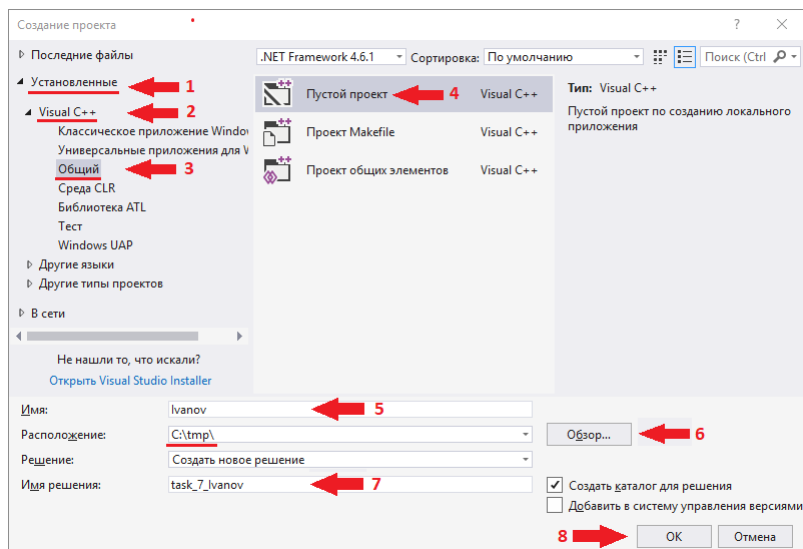
Папка с проектами



Приведенный состав папки `glm` является актуальным на момент составления методических инструкций, и может отличаться от версии на момент Вашей установки.

### 1.1.4 Установки нового проекта

Приложения, которые мы будем создавать — консольные приложения с использованием дополнительного окна для вывода графической информации. Для создания проекта такого приложения следует выбрать пункт меню *Файл* → *Создать* → *Проект*. Появится окно создания проекта.

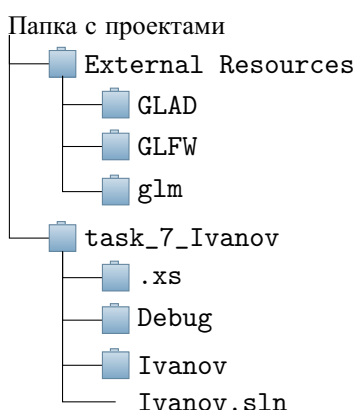


В этом окне нужно выбрать *Установленные* → *Visual C++* → *Общий* → *Пустой проект* (пункты 1–4 на рисунке).

В нижней части окна задайте имя создаваемого проекта — **Ваша фамилия латинскими буквами** (пункт 5). Укажите месторасположение будущего проекта (поле *Расположение*) — установите папку, внутри которой располагается папка **"External Resources"** (пункт 6). Укажите имя папки для проекта: для решения 7-го задания это `task_7_Ivanov` (вместо слова `Ivanov` должна быть Ваша фамилия) (пункт 7).

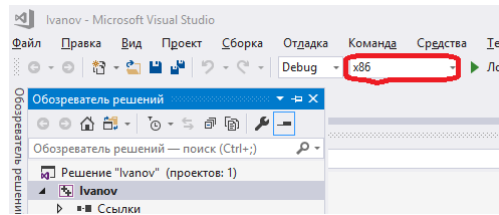
Для завершения создания проекта нажмите *OK* (пункт 8).

Будет создан пустой проект. Структура папок должна оказаться следующей.



Прежде чем менять установки проекта, в окне Visual Studio открытого проекта следует установить платформу приложения — *x86* (см. рисунок).

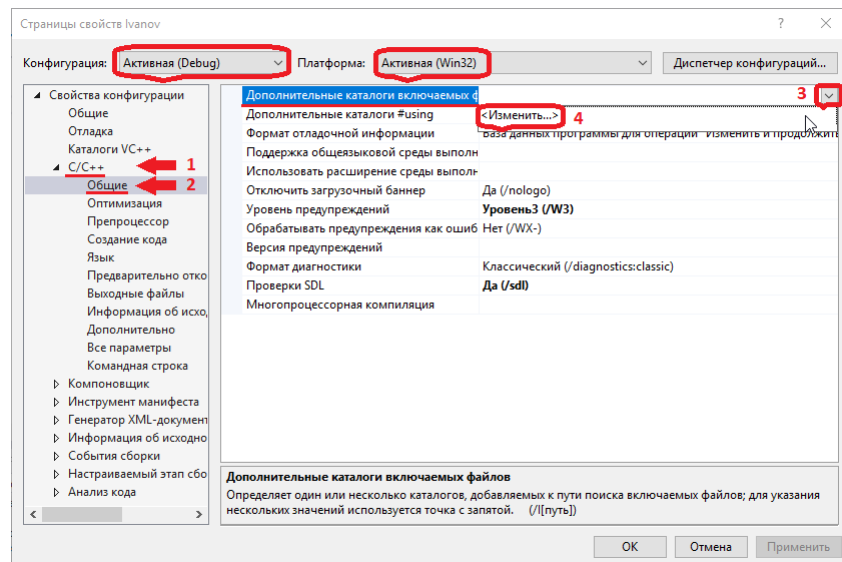




Добавим в проект файл исходного кода, который назовем `Main.cpp`, в котором добавим пока пустую функцию `main`.

```
int main () {  
    return 0;  
}
```

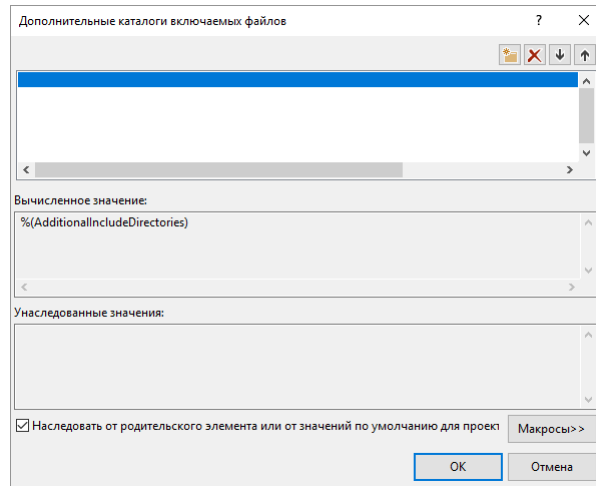
Теперь откроем окно свойств проекта.



Сначала убедитесь, что установки, которые Вы проводите, — для платформы *Win32* (см. рисунок, опция вверху окна), а опция *Конфигурация* — та же, что является активной (рядом с названием платформы на предыдущем рисунке, обычно *Debug*) или *Все конфигурации*.

Слева в окне выберите *Свойства конфигурации* → *C/C++* → *Общие* (пункты 1–2 на рисунке).

Нажмите на флажок в строке *Дополнительные каталоги включаемых файлов* и из выпадающего списка выберите *<Изменить>* (пункты 3 и 4). Появится окно



Верхнее поле этого окна предназначено для редактирования списка подключаемых папок. Для того, чтобы добавить очередную папку, кликните на пустой строке и Вы перейдете к её редактированию. Введите значение

```
$(SolutionDir)/../External Resources/glm
```

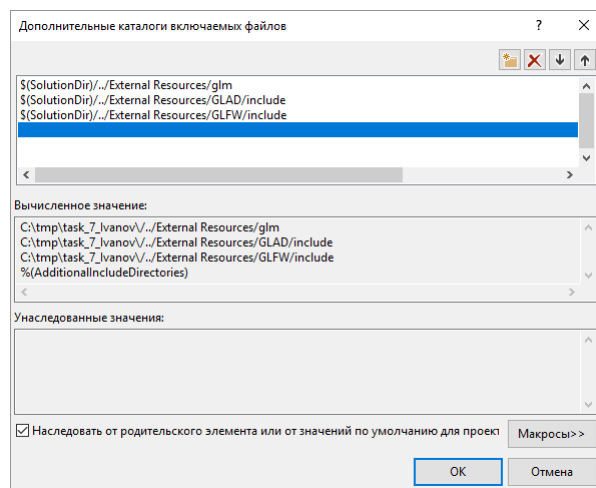
Часть введенной строки `$(SolutionDir)` является условным обозначением для папки вашего проекта (в моем случае это `C:\tmp\task_7_Ivanov`); часть `/..` говорит, что относительно предыдущей папки мы поднимаемся по структуре на уровень выше (т.е. к `C:\tmp`) и дальнейший путь будет продолжаться от полученной позиции (т.е. путь к папке `glm` — `C:\tmp\External Resources\glm`).

После добавления пути к папке `glm` аналогичным образом укажем путь к папкам `GLAD\include` и `GLFW\include`:

```
$(SolutionDir)/../External Resources/GLAD/include
```

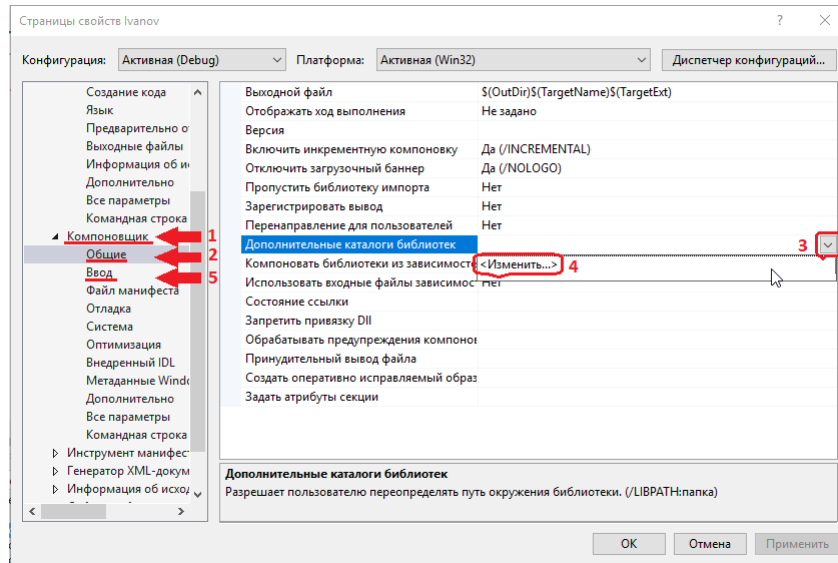
```
$(SolutionDir)/../External Resources/GLFW/include
```

Получим окно в следующем виде.



Для завершения ввода и возврата в окно свойств проекта нажмите *OK*.

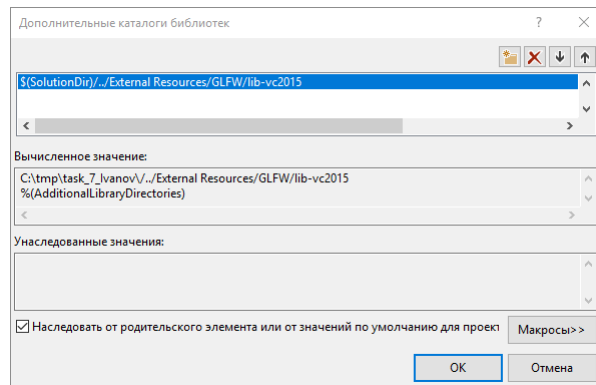
Слева в окне выберите *Компоновщик* → *Общие* (пункты 1–2 на рисунке).



Нажмите на флажок в строке *Дополнительные каталоги библиотек* и из выпадающего списка выберите *<Изменить>* (пункты 3 и 4). Появится такое же окно, как и в предыдущем случае. Здесь нужно прописать путь

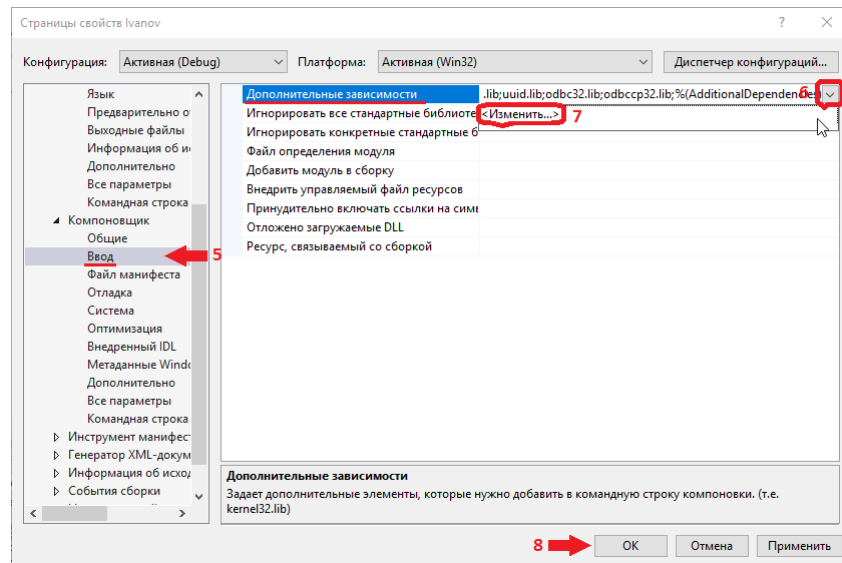
`$(SolutionDir)/../External Resources/GLFW/lib-vc2015`

Получим окно в следующем виде.



Нажмите *OK* для возврата в окно свойств проекта.

Здесь же, в разделе *Компоновщик* выберите пункт *Ввод* (пункт 5 на рисунке).



Нажмите на флажок в строке *Дополнительные зависимости* и из выпадающего списка выберите *<Изменить>* (пункты 6 и 7). Опять появится похожее окно, но в нем нам нужно будет ввести имена подключаемых библиотек (по одному имени в каждой строке).

opengl32.lib  
glfw3.lib

!!! Мы не проводили какой-то специальной установки `opengl32.lib`, так как эта установка проведена в рамках установки Microsoft SDK, что делается автоматически при установке Visual Studio.

!!! Мы не указываем путь и не прописываем здесь файлы библиотек GLM и GLAD так как первая представляет из себя только набор файлов заголовков, а вторую мы в дальнейшем подключим непосредственно включением файла программного кода в проект.

Нажмите *OK* для возврата в окно свойств проекта. Здесь нажмите *OK* для того, чтобы сохранить установки.

Выберите пункт меню *Проект → Добавить существующий элемент*. В появившемся окне выберите файл `External Resources/GLAD/src/glad.c`.

Вернемся к редактированию файла `main.cpp`. Подключим необходимые заголовочные файлы

```
#include <glad\glad.h>
#include <GLFW\glfw3.h>

#include <glm\glm.hpp>
#include <glm\gtc\transform.hpp>
#include <glm\gtc\type_ptr.hpp>
```

!!! Важно, чтобы файл `glad.h` был подключен раньше `glfw3.h`.

Если все свойства проекта прописаны правильно, то компиляция проекта должна выполняться без ошибок.

### 1.1.5 Первое приложение

Пока наше приложение ничего не делает. Определим в нем окно для рисования.

Для этого добавим в функцию `main` следующие строки.

Сначала проведем инициализацию GLFW, вызвав `glfwInit`.

```
glfwInit(); // Инициализация GLFW
```

Перед выходом из программы (перед командой `return 0;`) завершим работу GLFW.

```
glfwTerminate(); // завершить работу GLFW
```

Теперь проведем начальные установки параметров GLFW с помощью процедуры `glfwWindowHint`. У этой процедуры два аргумента: первый ссылается на параметр, а второй — значение, которое мы собираемся установить этому параметру.

Укажем версию OpenGL, которую мы будем использовать — 3.3. Такая установка будет проведена двумя вызовами `glfwWindowHint`: первый вызов устанавливает номер версии до десятичной точки, а второй — после.

```
// Задается минимальная требуемая версия OpenGL.  
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3); // Номер до десятичной точки  
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3); // Номер после десятичной точки
```

Кроме того, укажем, что мы не будем использовать средства предыдущих версий OpenGL, а используем только средства версии 3.3 (3.3 core)

```
// Используем только средства указанной версии без совместимости с более ранними  
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

Теперь создадим окно для вывода графики с помощью `glfwCreateWindow`.

```
// Создаем окно  
GLFWwindow* window = glfwCreateWindow(800, 600, "Task 7. Ivanov", NULL, NULL);
```

Первые два параметра функции — разрешение окна. Третий параметр — заголовок окна (в этом и последующих проектах заголовок окна должен соответствовать заданию). Последние два параметра в наших решениях оставим равными `NULL`.

После вызова `glfwCreateWindow` проверим, создано ли окно успешно. Если не создано, завершим работу программы.

```
if (window == NULL){ // если ссылка на окно не создана  
    std::cout << "Вызов glfwCreateWindow закончился неудачей." << std::endl;  
    glfwTerminate(); // завершить работу GLFW  
    return -1; // завершить программу  
}  
glfwMakeContextCurrent(window); // делаем окно window активным (текущим)
```

Так как в коде используем `cout <<`, в начале `Main.cpp` подключим `iostream`.

```
#include <iostream>
```

Вернемся к коду функции `main`.

Если окно создано (мы не вышли из программы), то делаем окно активным: дальнейшие вызовы процедур OpenGL будут относиться к этому окну.

```
glfwMakeContextCurrent(window); // делаем окно window активным (текущим)
```



Все процедуры, которые мы использовали до сих пор, очевидно относятся к библиотеке GLFW: имена всех процедур этой библиотеки начинаются с префикса `glfw` (константы начинаются с префикса `GLFW`). Процедуры OpenGL (обеспечиваемые использованием GLAD) имеют префикс `gl`, а собственные процедуры GLAD — префикс `glad`. Процедуры библиотеки GLM имеют префикс `glm`.

Проведем инициализацию GLAD. Для этого функции `gladLoadGLLoader` передаем функцию `glfwGetProcAddress` для загрузки указателей функций OpenGL. Если инициализация пройдет неудачно, завершим программу.

```
// Инициализация GLAD
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
    std::cout << "Не удалось загрузить GLAD" << std::endl;
    glfwTerminate(); // завершить работу GLFW
    return -1;       // завершить программу
}
```

Теперь нужно сообщить OpenGL диапазон координат в окне на экране. Мы можем установить эти значения через процедуру `glViewport`. У этой процедуры четыре параметра: первые два — координаты, соответствующие левому нижнему углу окна; вторые два — ширина и высота окна в пикселах.

```
// сообщаем диапазон координат в окне
// (0, 0) - координаты левого нижнего угла, 800x600 - размеры окна в пикселах
glViewport(0, 0, 800, 600);
```

Приложение уже можно запустить: при запуске мелькнет два окна: одно — окно консоли приложения (в которое, в том числе, организуется вывод командами `cout`, при необходимости), а второе — окно созданное GLFW.

Для того, чтобы все-таки увидеть созданное окно организуем цикл.

```
while(!glfwWindowShouldClose(window)) { // пока окно window не должно закрыться
    glfwSwapBuffers(window); // поменять местами буферы изображения
    glfwPollEvents(); // проверить, произошли ли какие-то события
}
```

Здесь функция `glfwWindowShouldClose` проверяет, получил ли GLFW инструкцию к закрытию заданного окна.

При получении изображения в окне включена двойная буферизация — все команды, касающиеся отрисовки, исполняются не в окне, а в «запасном» буфере. Смена изображения в окне происходит только тогда, когда второй буфер становится активным (а буфер окна, который содержит предыдущее изображение, становится «запасным»). Но смена содержимого буферов здесь происходит не автоматически, как в предыдущих наших проектах, а проводится с помощью процедуры `glfwSwapBuffers`.

Процедура `glfwPollEvents` проверяет, были ли вызваны какие либо события (вроде ввода с клавиатуры или перемещения мыши) и обеспечивает вызов процедур — обработчиков событий.

То есть, общий смысл нашего цикла — вывести то, что есть на данный момент, после чего проверить, произошли ли какие-то события, вызывать их обработчики и вернуться к началу цикла.

Получим, на текущем этапе, следующий код `Main.cpp`.

```

1  #include <iostream>
2
3  #include <glad\glad.h>
4  #include <GLFW\glfw3.h>
5
6  #include <glm\glm.hpp>
7  #include <glm\gtw\transform.hpp>
8  #include <glm\gtc\type_ptr.hpp>
9
10 int main () {
11     glfwInit(); // Инициализация GLFW
12     // Проведение начальных установок GLFW
13     // Дается минимальная требуемая версия OpenGL.
14     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3); // Номер до десятичной точки
15     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3); // Номер после десятичной точки
16     // Используем только средства указанной версии без совместимости с более ранними
17     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
18
19     // Создаем окно
20     GLFWwindow* window = glfwCreateWindow(800, 600, "Task 7. Ivanov", NULL, NULL);
21     if (window == NULL) { // если ссылка на окно не создана
22         std::cout << "Вызов glfwCreateWindow закончился неудачей." << std::endl;
23         glfwTerminate(); // завершить работу GLFW
24         return -1;      // завершить программу
25     }
26     glfwMakeContextCurrent(window); // делаем окно window активным (текущим)
27
28     // Инициализация GLAD
29     if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
30         std::cout << "Не удалось загрузить GLAD" << std::endl;
31         glfwTerminate(); // завершить работу GLFW
32         return -1;      // завершить программу
33     }
34
35     // сообщаем диапазон координат в окне
36     // (0, 0) - координаты левого нижнего угла, 800x600 - размеры окна в пикселах
37     glViewport(0, 0, 800, 600);
38
39     while (!glfwWindowShouldClose(window)) { // пока окно window не должно закрыться
40         glfwSwapBuffers(window); // поменять местами буферы изображения
41         glfwPollEvents(); // проверить, произошли ли какие-то события
42     }
43
44     glfwTerminate(); // завершить работу GLFW
45     return 0;
46 }
```

Теперь запуск проекта должен привести к появлению двух черных окон. Когда мы закрываем окно *Task 7. Ivanov*, вызов `glfwWindowShouldClose(window)` возвращает истину, что приводит к завершению цикла и программы.

Обработчики событий (callbacks, в терминологии GLFW, — функции обратного вызова) — функции C++, имеющие определенную сигнатуру, назначенные на эту роль с помощью специальных процедур GLFW.

Добавим сначала в наше приложение обработчик события *Resize* (здесь его наименование *FramebufferSizeCallback*). Добавим перед функцией `main`

```

// обработчик события Resize
void framebuffer_size_callback(GLFWwindow* window, int width, int height) {
```

```
}
```

У процедуры три аргумента: первый — окно GLFW, второй и третий — размеры окна после растяжения.

В тело этой процедуры поместим вызов `glViewport` с обновленными размерами окна.

```
glViewport(0, 0, width, height);
```

В общем виде процедура `framebuffer_size_callback` примет следующий вид.

```
1 // обработчик события Resize
2 void framebuffer_size_callback(GLFWwindow* window, int width, int height) {
3     glViewport(0, 0, width, height);
4 }
```

Теперь, сразу после объявления окна `window` активным (после вызова `glfwMakeContextCurrent`), назначим описанную процедуру обработчиком события для этого окна с помощью `glfwSetFramebufferSizeCallback`.

```
// Назначение обработчика события Resize
glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
```

Похожим образом назначим обработчик нажатия клавиш. Перед функцией `main` опишем процедуру

```
// Обработчик нажатия клавиш
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode) {
}
```

Первый аргумент, как и в предыдущем случае, окно, для которого определяется обработчик события; второй — идентификатор нажатой клавиши — определяется константой библиотеки GLFW; третий — код нажатой клавиши; четвертый — действие произошедшее с клавишей; последний параметр — информация о нажатом модификаторе (**Shift**, **Ctrl** и т. п.).

Добавим в тело процедуры обработку нажатия клавиши **Escape**. Установим реакцию на **Escape** — выход из цикла. Для этого, с помощью процедуры `glfwSetWindowShouldClose` назначим результат функции `glfwWindowShouldClose` — `GL_TRUE`.

```
if (action != GLFW_RELEASE) { // если клавиша нажата
    switch (key) { // анализируем обрабатываемую клавишу
        case GLFW_KEY_ESCAPE: // если клавиша - Escape
            // устанавливаем, что окно window должно быть закрыто
            glfwSetWindowShouldClose(window, GL_TRUE);
            break;
        default:
            break;
    }
}
```



Обратите внимание, что в Windows Forms были отдельные обработчики событий для нажатия клавиш (*KeyDown*) и для отжатия клавиш (*KeyUp*) и некоторые другие. Здесь эти события обрабатываются единой процедурой, но с дополнительным аргументом `action`.



Получим обработчик нажатия клавиш в общем виде.

```
1 // Обработчик нажатия клавиш
2 void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode) {
3     if (action != GLFW_RELEASE) { // если клавиша нажата
4         switch (key) { // анализируем обрабатываемую клавишу
5             case GLFW_KEY_ESCAPE: // если клавиша - Escape
6                 // устанавливаем, что окно window должно быть закрыто
7                 glfwSetWindowShouldClose(window, GL_TRUE);
8                 break;
9             default:
10                break;
11         }
12     }
13 }
```

Назначим эту процедуру обработчиком события с помощью `glfwSetKeyCallback`. Добавим вызов этой процедуры в функцию `main` после назначения обработчика события `Resize`.

```
// Назначение обработчика нажатия клавиш
glfwSetKeyCallback(window, key_callback);
```

Если теперь запустить приложение, то можно выйти из программы, нажав **Escape**.

Наконец, добавим в цикл заливку окна нашим собственным цветом. Для этого сначала назначим цвет, которым зальем окно: сразу после входа в цикл добавим

```
glClearColor(0.2f, 0.3f, 0.3f, 1.0f); // назначаем цвет заливки
```

После этого вызовем процедуру очищения (заливки) окна заданным цветом

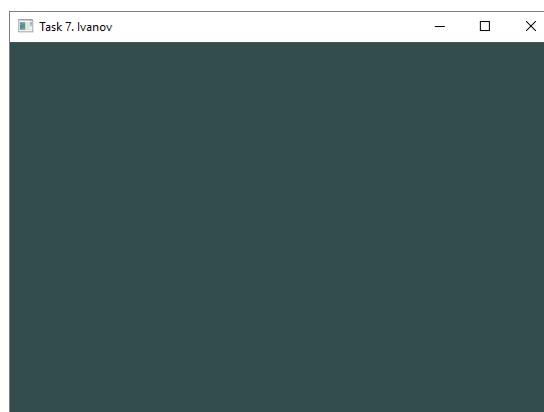
```
glClear(GL_COLOR_BUFFER_BIT); // очищаем буфер заданным цветом
```

Здесь аргумент `GL_COLOR_BUFFER_BIT` означает, что очистке подлежит так называемый буфер цвета — область памяти, в которой хранятся данные о цвете каждого пиксела.



Обратите внимание, что цвет задается через 4 компоненты: первые три — компоненты RGB, заданные вещественными величинами от 0 до 1 (тогда как раньше у нас эти компоненты задавались целыми числами от 0 до 255). Последняя компонента — так называемая *alpha* — дополнительная величина от 0 до 1, отвечающая обычно за степень «прозрачности».

Запустив приложение обнаружим, что окно покрашено темнозеленосиним цветом.



На текущий момент мы имеем следующий вид содержимого файла `Main.cpp`.

```

1  #include <iostream>
2
3  #include <glad\glad.h>
4  #include <GLFW\glfw3.h>
5
6  #include <glm\glm.hpp>
7  #include <glm\gtw\transform.hpp>
8  #include <glm\gtc\type_ptr.hpp>
9
10 // обработчик события Resize
11 void framebuffer_size_callback(GLFWwindow* window, int width, int height) {
12     glViewport(0, 0, width, height);
13 }
14
15 // Обработчик нажатия клавиш
16 void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode) {
17     if (action != GLFW_RELEASE) { // если клавиша нажата
18         switch (key) { // анализируем обрабатываемую клавишу
19             case GLFW_KEY_ESCAPE: // если клавиша - Escape
20                 // устанавливаем, что окно window должно быть закрыто
21                 glfwSetWindowShouldClose(window, GL_TRUE);
22                 break;
23             default:
24                 break;
25         }
26     }
27 }
28
29 int main () {
30     glfwInit(); // Инициализация GLFW
31     // Проведение начальных установок GLFW
32     // Задается минимальная требуемая версия OpenGL.
33     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3); // Номер до десятичной точки
34     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3); // Номер после десятичной точки
35     // Используем только средства указанной версии без совместимости с более ранними
36     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
37
38     // Создаем окно
39     GLFWwindow* window = glfwCreateWindow(800, 600, "Task 7. Ivanov", NULL, NULL);
40     if (window == NULL) { // если ссылка на окно не создана
41         std::cout << "Вызов glfwCreateWindow закончился неудачей." << std::endl;
42         glfwTerminate(); // завершить работу GLFW
43         return -1; // завершить программу
44     }
45     glfwMakeContextCurrent(window); // делаем окно window активным (текущим)
46     // Назначение обработчика события Resize
47     glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
48     // Назначение обработчика нажатия клавиш
49     glfwSetKeyCallback(window, key_callback);
50
51     // Инициализация GLAD
52     if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
53         std::cout << "Не удалось загрузить GLAD" << std::endl;
54         glfwTerminate(); // завершить работу GLFW
55         return -1; // завершить программу
56     }
57
58     // сообщаем диапазон координат в окне
59     // (0, 0) - координаты левого нижнего угла, 800x600 - размеры окна в пикселах
60     glViewport(0, 0, 800, 600);
61
62     while (!glfwWindowShouldClose(window)) { // пока окно window не должно закрыться
63         glClearColor(0.2f, 0.3f, 0.3f, 1.0f); // назначаем цвет заливки
64         glClear(GL_COLOR_BUFFER_BIT); // очищаем буфер заданным цветом
65         glfwSwapBuffers(window); // поменять местами буферы изображения
66         glfwPollEvents(); // проверить, произошли ли какие-то события
67     }
68 }

```

```

69  glfwTerminate(); // завершить работу GLFW
70  return 0;
71  }

```

## 1.2 Добавление элементов рисунка

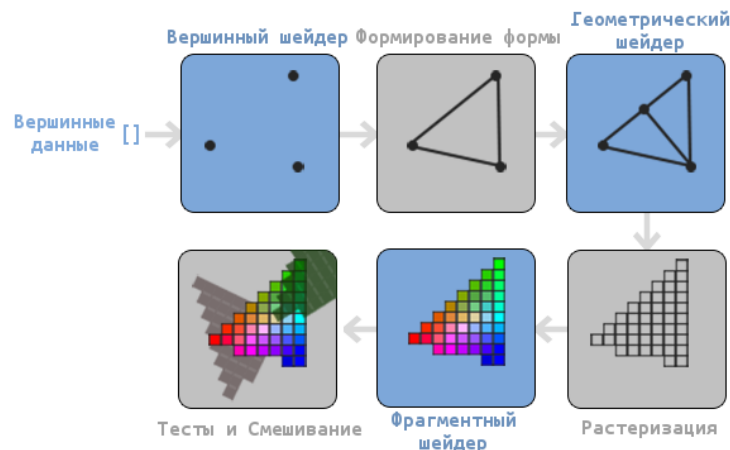
### 1.2.1 Графический конвейер

Координатное пространство, в котором задается изображение в OpenGL — трехмерное, но при этом окно на экране — это двумерный массив пикселей. Большая часть работы OpenGL — преобразование 3D-координат в двумерное пространство для отрисовки на экране. Процесс преобразования управляется графическим конвейером OpenGL. Графический конвейер можно разделить на 2 большие части: первая часть преобразовывает 3D-координаты в 2D-координаты, а вторая часть преобразовывает 2D-координаты в цветные пиксели.

Графический конвейер можно разделить на несколько этапов, где на вход каждого этапа поступает результат работы предыдущего. Все этапы эти крайне специализированы и могут с легкостью исполняться параллельно. По причине их параллельной природы большинство современных GPU имеют множество процессоров для быстрой обработки данных графического конвейера с помощью запуска большого количества маленьких программ на каждом этапе конвейера. Эти маленькие программы называются шейдерами.

Некоторые из шейдеров могут быть написаны нами для замены стандартных. Это дает нам гораздо больше возможностей тонкой настройки специфичных мест конвейера, и именно из-за того, что они работают на GPU, позволяет нам сохранить процессорное время. Шейдеры пишутся на языке GLSL (OpenGL Shading Language), синтаксис которого схож с синтаксисом языка C.

На изображении ниже можно увидеть примерное представление всех этапов графического конвейера. Синим выделены этапы для которых можно специфицировать собственные шейдеры.



Как Вы видите, графический конвейер содержит большое количество секций, где каждая занимается своей частью обработки вершинных данных в полностью отрисованный пиксел. Кратко опишем каждую секцию конвейера, чтобы дать представление о том как он работает.

На вход конвейера передается массив элементов — набор вершин (вершина — vertex). Каждая вершина задается своими трехмерными координатами и, возможно, дополнительными атрибутами (например цвет). Во время отрисовки системе следует указать, что составить из переданного набора данных: хотим ли мы отрисовать набор точек, набор треугольников или просто одну ломаную линию (такие фигуры, называются примитивами).

Первый этап конвейера — вершинный шейдер (Vertex Shader), который принимает на вход одну вершину. Основная задача вершинного шейдера — преобразование одних 3D-координат в другие. Возможность изменения этого шейдера позволяет выполнять преобразования в зависимости от параметров вершин. Предполагается, что на выходе вершинного шейдера координаты вершины заданы в пространстве отсечения.

Сборка примитивов — этап, принимающий на вход все вершины от вершинного шейдера, формирующие примитив и собирает из них сам примитив.

Результат этапа сборки примитивов передается геометрическому шейдеру (Geometry Shader). Он же в свою очередь на вход принимает набор вершин, формирующих примитивы, из которого может сформировать новый набор возможно иных примитивов.

Результат работы геометрического шейдера передается на этап растеризации, где результирующие примитивы будут соотноситься с пикселями на экране, формируя «фрагмент» для фрагментного шейдера (Fragment Shader). Фрагмент в OpenGL — все данные, которые нужны OpenGL для того, чтобы отрисовать пиксел. Перед выполнением фрагментного шейдера, выполняется отсечение примитива (по умолчанию, относительно куба от -1 до 1 по каждой из осей), при котором отбрасываются все его части, которые находятся вне поля зрения, повышая таким образом производительность.

Основная цель фрагментного шейдера — вычисление конечного цвета пиксела. Зачастую фрагментный шейдер содержит всю информацию о 3D сцене, которую можно использовать для модификации финального цвета (типа освещения, теней, цвета источника света и т.д.).

После того, как определение всех соответствующих цветовых значений будет закончено, результат пройдет еще один этап, который называется альфа тестирование и смешивание. Этот этап проверяет соответствующее значение глубины (мы вернемся к этому позже) фрагмента и использует его для проверки местоположения фрагмента относительно других объектов: спереди или сзади. Этот этап также проверяет значения прозрачности и смешивает цвета, если это необходимо. Таким образом, результирующий цвет пиксела может отличаться от цвета, вычисленного фрагментным шейдером.

Обычно геометрический шейдер оставляется стандартным. Но на видеокартах не существует стандартного вершинного и фрагментного шейдера, в следствие чего в современном OpenGL Вы вынуждены задавать их самостоятельно.

### 1.2.2 Шейдерная программа

Шейдер — программа, исполняемая на графическом процессоре (GPU). Графическое приложение, использующее шейдеры, будучи однажды скомпилированным для какой либо платформы, должно запускаться и работать на разных компьютерах на этой платформе. Но единство операционной системы не гарантирует единства графических процессоров. GPU могут иметь различные архитектуры, но шейдер должен на них исполняться одинаково успешно. Поэтому, компиляция шейдера производится не во время сборки всего приложения, а в ходе выполнения приложения — для той архитектуры GPU, которая является актуальной при запуске.

Шейдеры загружаются в систему в составе шейдерной программы. Шейдерная программа — комбинация шейдеров используемая для некоторого набора данных. В эту комбинацию обязательно входит один вершинный шейдер, один фрагментный шейдер. Возможно присутствие по одному экземпляру шейдера каждого другого вида.

Для того, чтобы составить шейдерную программу, нужно определить текст каждого шейдера, скомпилировать его, провести сборку скомпилированных шейдеров в единое целое. Рассмотрим пример сборки шейдерной программы из простеньких шейдеров.

### Вершинный шейдер

Сначала мы должны написать сам шейдер на специальном языке GLSL, а затем собрать его, чтобы приложение могло с ним работать. Вот код простейшего шейдера:

```
#version 330 core
layout (location = 0) in vec3 position;
void main() {
    gl_Position = vec4(position.x, position.y, position.z, 1.0);
}
```

Каждый шейдер начинается с комментария, в котором указывается версия OpenGL. В нашем случае это `#version 330 core` (для версий, начиная с 3.3 соответствие комментария версии прямое: 330 для OpenGL версии 3.3, 420 — для 4.2. Для более ранних версий следует обращаться к спецификациям). Кроме того, мы явно указали, что используем core profile.

В первой строке после комментария указан входной параметр шейдера — `position`, типа `vec3` (координаты трехмерной точки). О том, что это входной параметр, свидетельствует ключевое слово `in`. Кроме того, явно указана позиция входного параметра через `layout (location = 0)` (что это значит, будет сказано позднее).

Результат вычислений вершинного шейдера должен быть присвоен предопределенной переменной `gl_Position`, имеющей тип `vec4` — однородные координаты точки в пространстве отсечения. В нашем случае, в процедуре `main` переменной `gl_Position` просто присваиваются однородные координаты входного параметра `position`.

Чтобы описать этот шейдер в нашем приложении, зададим его текст в `Main.cpp` в функции `main` в виде массива символов

```
//=====
//                               ВЕРШИННЫЙ ШЕЙДЕР
//=====
const char *vertexShaderSource =
    "#version 330 core\n"
    "layout (location = 0) in vec3 position;\n"
    "void main() {\n"
    "    gl_Position = vec4(position.x, position.y, position.z, 1.0);\n"
    "}\n0";
//=====
```

Добавим это описание перед циклом отрисовки.

Теперь создадим шейдерный объект. Доступ к созданным объектам OpenGL осуществляется через целочисленный идентификатор: мы будем хранить его в переменной типа `GLuint`. Создать шейдерный объект можем с помощью `glCreateShader`:

```
GLuint vertexShader; // шейдерный объект - вершинный шейдер
vertexShader = glCreateShader(GL_VERTEX_SHADER); // создаем объект
```

Во время создания шейдера мы должны указать его тип: поскольку нам нужен вершинный шейдер, указываем `GL_VERTEX_SHADER`.

Теперь привязываем исходный код шейдера (из переменной `vertexShaderSource`) к объекту шейдера.

```
// привязываем исходный код к шейдерному объекту
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
```

Функция `glShaderSource` в качестве первого аргумента принимает шейдер, который необходимо скомпилировать. Второй аргумент описывает количество символьных массивов, в нашем случае — лишь один. Третий параметр — это сам исходный код шейдера, а четвертый параметр мы оставим в `NULL`.

Проведем компиляцию шейдера.

```
glCompileShader(vertexShader); // компилируем шейдер
```

Теперь добавим проверку того, что компиляция прошла успешно. Для этого, сначала определим параметр, в который запросим статус компиляции нашего шейдера.

```
GLint success; // результат компиляции
// запрашиваем статус компиляции шейдера в переменную success
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
```

Процедура `glGetShaderiv` присваивает своему третьему аргументу параметр заданного шейдера, определенный вторым аргументом.

Теперь проверим значение `success`: если значение `GL_FALSE`, то произошла ошибка компиляции и нужно выдать сообщение об ошибке.

```
if (!success) { // если компиляция прошла с ошибкой
    GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog); // запрашиваем сообщение
    // выводим сообщение об ошибке на экран
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
}
```

### Фрагментный шейдер

Приведем код простейшего фрагментного шейдера.

```
#version 330 core
out vec4 color;
void main() {
    color = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

Задача фрагментного шейдера — вычислить цвет каждого пиксела. Цвет вычисляется в виде четырехмерного вектора вещественных чисел, где числа имеют те же значения, которые мы использовали при задании цвета заливки окна.

Наш шейдер выдает один и тот же цвет для каждого фрагмента, для которого он запускается.

Начинается описание шейдера с объявления версии. Ключевое слово `out` в следующей строке говорит о том, что объявляется выходная переменная с именем `color` и типом `vec4`. В теле процедуры `main` лишь один оператор присваивания: переменной `color` присваивается значение некоторого конкретного цвета.

Как и в случае вершинного шейдера, опишем этот шейдер в нашей программе в виде массива символов.

```
//=====
//                                ФРАГМЕНТНЫЙ ШЕЙДЕР
//=====
const char *fragmentShaderSource =
    "#version 330 core\n"
```

```
"out vec4 color;\n"
"void main() {\n"
"    color = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
"}\n\n0";
//=====
```

Дальнейший процесс создания шейдерного объекта и компиляции шейдера почти дословно повторяет подобный процесс для вершинного шейдера.

```
GLuint fragmentShader; // шейдерный объект - фрагментный шейдер
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER); // создаем объект
// привязываем исходный код к шейдерному объекту
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader); // компилируем шейдер
// запрашиваем статус компиляции шейдера в описанную ранее переменную success
glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
if (!success) { // если компиляция прошла с ошибкой
    GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
    glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog); // запрашиваем сообщение
    // выводим сообщение об ошибке на экран
    std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog << std::endl;
}
```

### Сборка шейдерной программы

При соединении шейдеров в программу переменные, объявленные выходными в одном шейдере (если такое объявление имело место), сопоставляются с входными переменными другого шейдера. Если входные и выходные значения не совпадают, сборка программы закончится неудачей и может быть получен протокол о соответствующих ошибках.

Процесс создания и компоновки программы несколько похож на создание и компиляцию шейдерного объекта. Сначала объявим переменную — идентификатор программы.

```
// Шейдерная программа
GLuint shaderProgram; // идентификатор шейдерной программы
```

Затем создаем программный объект.

```
shaderProgram = glCreateProgram(); // создаем программный объект
```

Теперь присоединим наши скомпилированные шейдеры к программе, в том порядке, в котором они должны работать: сначала вершинный шейдер, а затем фрагментный. Присоединение производим с помощью процедуры `glAttachShader`.

```
glAttachShader(shaderProgram, vertexShader); // присоединяем вершинный шейдер
glAttachShader(shaderProgram, fragmentShader); // фрагментный шейдер
```

Когда присоединены все необходимые шейдеры, можно произвести компоновку программы.

```
glLinkProgram(shaderProgram); // компоновка программы
```

Теперь проверим успешность компоновки. Здесь, вместо `glGetShaderiv` используем `glGetProgramiv`, а вместо `glGetShaderInfoLog` — `glGetProgramInfoLog`.



```
// запрашиваем статус компоновки шейдерной программы в переменную success
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
if (!success) { // если компоновка прошла с ошибкой
    GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
    glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog); // запрашиваем сообщение
    // выводим сообщение об ошибке на экран
    std::cout << "ERROR::SHADER::PROGRAM::LINK_FAILED\n" << infoLog << std::endl;
}
```

У нас теперь есть единая шейдерная программа (которую идентифицирует переменная `shaderProgram`), объединяющая в себе оба шейдера. Сами шейдерные объекты теперь нам стали не нужны и их можно удалить.

```
// удаление шейдерных объектов
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

### 1.2.3 Передача данных

Для простоты представим координаты трех вершин заданными уже в пространстве отсечения. Определим их в массиве элементов типа `GLfloat`

```
//=====
//   НАБОР ИСХОДНЫХ ДАННЫХ ДЛЯ ОТРИСОВКИ
//=====
GLfloat vertices[] = {
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};
```

Несмотря на то, что у нас есть эти данные, OpenGL не может использовать их, пока они находятся вне памяти на видеокарте. Для того, чтобы обеспечить доступ OpenGL к этим данным, необходимо их скопировать на видеокарту и сказать системе что они из себя представляют и как с ними следует обращаться.

Вершинные данные организуются с помощью, так называемых, объектов вершинного массива (Vertex Array Object). Создадим такой объект с помощью `glGenVertexArrays`.

```
GLuint vertexArray; // объект вершинного массива
// создаем вершинный массив, идентификатор которого присваиваем vertexArray
glGenVertexArrays(1, &vertexArray);
```

Для накопления информации в вершинном массиве, его следует сделать активным.

```
glBindVertexArray(vertexArray); // делаем активным вершинный массив
```

Теперь, последующие команды, относящиеся к наполнению/изменению/чтению памяти GPU будут ассоциироваться с этим объектом.

Следующим шагом является выделение некоторой памяти, которую может видеть OpenGL, и заполнение этой памяти нашими данными. Это делается с помощью так называемого буферного объекта.

Буферный объект представляет собой линейный массив памяти, управляемый и распределенный OpenGL по воле пользователя. Содержимое этой памяти контролируется



пользователем, но пользователь имеет только косвенный контроль над ней. Можно представить себе буферный объект как массив памяти GPU. Графический процессор может быстро считывать эту память, поэтому хранение данных в ней имеет преимущества производительности.

Мы можем создать такой объект с помощью функции `glGenBuffers`:

```
GLuint vertexBuffer; // идентификатор буферного объекта
// создаем буферный объект, идентификатор которого присваиваем vertexBuffer
glGenBuffers(1, &vertexBuffer);
```

В OpenGL есть большое количество различных типов буферных объектов. Так как буферный объект определяет просто некоторую область памяти, то её содержимое может рассматриваться по-разному в различном контексте. Прежде чем что-то проделывать с буферным объектом, он должен быть привязан к контексту. В силу специфики OpenGL общение с буферным объектом происходит не напрямую, а через контекст. Так, в нашем примере, чтобы переслать данные из массива `vertices` в буферный объект `vertexBuffer` нужно сначала привязать буферный объект к контексту.

```
// привязка vertexBuffer к GL_ARRAY_BUFFER
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
```

Контекст `GL_ARRAY_BUFFER` означает, что наш буферный объект рассматривается как массив вершинных данных. После этой привязки все операции с `GL_ARRAY_BUFFER` будут относиться к `vertexBuffer`.

Теперь мы можем вызвать `glBufferData` для копирования вершинных данных в этот буфер.

```
// в буфер, привязанный к GL_ARRAY_BUFFER копируем содержимое vertices
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

Процедура `glBufferData` выполняет две операции. Первая — выделяется память для буфера, связанного с первым аргументом `GL_ARRAY_BUFFER`. Второй параметр процедуры — размер в байтах необходимой памяти: в нашем случае он вычисляется с помощью `sizeof(vertices)`. Таким образом, мы выделяем достаточно памяти GPU для хранения данных наших вершин.

Вторая операция — копирование данных из массива, заданного третьим аргументом, в объект буфера. Третий аргумент процедуры — начальный адрес области памяти, из которой следует копировать данные. В нашем случае `vertices` — массив, и обращение к имени приводит к получению адреса первой ячейки массива.

Четвертый аргумент определяет, каким образом видеокарте предстоит обращаться с переданными ей данными. В частности, значение `GL_STATIC_DRAW` указывает, что данные либо никогда не будут изменяться, либо будут изменяться очень редко.

Теперь нужно сказать системе как она будет интерпретировать эти данные при передаче на вход вершинному шейдеру. Для каждой вершины в буфере может находиться различная информация, описывающая эту вершину. Предполагается, что эта информация представлена в виде набора атрибутов этой вершины, где каждый атрибут — отдельный параметр шейдера.

В нашем вершинном шейдере один входной параметр `position`, для которого указано расположение `location = 0`. Опишем, как информация для этого расположения будет считываться из вершинного буфера. Это можно организовать с помощью процедуры `glVertexAttribPointer`.

```
// описание расположения параметра вершинного шейдера в вершинном буфере  
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
```

Здесь первый аргумент соответствует значению `location` вершинного шейдера (номеру атрибута). Второй и третий аргумент — количество и тип элементов буфера на один параметр шейдера. Так как тип `position` — `vec3`, указываем, что на один параметр шейдера отводится три вещественных числа. Четвертый аргумент оставляем равным `GL_FALSE` (он указывает, нужно ли проводить дополнительную нормализацию каждого элемента буфера). Пятый аргумент задает шаг в байтах для перехода к этому же атрибуту следующей вершины. Когда значения параметра шейдера располагаются в буфере непосредственно друг за другом (как в нашем случае), можно указать шаг равный 0: в этом случае OpenGL вычисляет шаг автоматически (на основании значений второго и третьего аргумента), но здесь мы указали шаг в явном виде. Последний параметр имеет тип `GLvoid*` и поэтому требует такое странное приведение типов. Этот параметр задает первоначальное смещение для чтения данных относительно начала буфера. В нашем случае параметр шейдера будет считываться с самого начала буфера, смещение отсутствует, поэтому здесь передаем значение 0.

Теперь нужно включить атрибут с помощью `glEnableVertexAttribArray`.

```
glEnableVertexAttribArray(0); // включение параметра 0 для шейдера
```

После того, как данные скопированы, мы можем разорвать связь между `GL_ARRAY_BUFFER` и буферным объектом `vertexBuffer`.

```
glBindBuffer(GL_ARRAY_BUFFER, 0); // отвязка буферного объекта
```

Это не является строго необходимой операцией, так как любая последующая привязка к `GL_ARRAY_BUFFER` просто отвяжет то, что было привязано до этого. Но обычно рекомендуется отвязывать объекты, которые Вы связываете.

Данные вершинного массива `vertexArray` сформированы. Выключим его активность (до момента, когда потребуется отрисовка) с помощью той же процедуры `glBindVertexArray`, которой передадим нулевой аргумент.

```
glBindVertexArray(0); // отключение вершинного массива
```

Формирование данных в памяти GPU закончено.

### 1.2.4 Отрисовка примитивов

Теперь обратимся к циклу отрисовки. Следующие строки добавим после процедуры очистки формы.

Сначала укажем системе, какую шейдерную программу мы хотим использовать.

```
glUseProgram(shaderProgram); // шейдерную программу shaderProgram делаем активной
```

Теперь включаем (делаем активным) вершинный массив, из которого в шейдерную программу будут поступать исходные данные.

```
glBindVertexArray(vertexArray); // делаем активным вершинный массив
```

Теперь проведем отрисовку примитивов, соответствующих данным из вершинного массива. Так как в этом массиве описан, всего лишь, набор точек, то интерпретировать

содержимое массива можно по-разному. Например, можно считать, что последовательность точек определяет ломаную линию (как в задании 6), можно предположить, что каждая пара точек в последовательности определяет отрезок (как в задании 2), а можно решить, что каждая тройка точек определяет закрашенный треугольник и т. д.

Отрисовка примитивов проводится с помощью процедуры `glDrawArrays`. Представим, что наши три вершины задают один треугольник. Тогда для его отрисовки добавим вызов

```
glDrawArrays(GL_TRIANGLES, 0, 3); // отрисовка одного треугольника
```

Первый аргумент задает тип примитивов для отрисовки: `GL_TRIANGLES` говорит нам, что мы рисуем треугольники. Второй аргумент указывает, с какого элемента в вершинном массиве нужно начать считывать точки (вершины), а третий аргумент — количество точек, которое нужно прочесть.

После того, как отрисовка примитивов закончена, снова сделаем неактивным вершинный массив.

```
glBindVertexArray(0); // отключаем вершинный массив
```

Если все введено корректно, то файл `Main.cpp` примет вид

```
1  #include <iostream>
2
3  #include <glad\glad.h>
4  #include <GLFW\glfw3.h>
5
6  #include <glm\glm.hpp>
7  #include <glm\gtw\transform.hpp>
8  #include <glm\gtc\type_ptr.hpp>
9
10 // обработчик события Resize
11 void framebuffer_size_callback(GLFWwindow* window, int width, int height) {
12     glViewport(0, 0, width, height);
13 }
14
15 // Обработчик нажатия клавиш
16 void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode) {
17     if (action != GLFW_RELEASE) { // если клавиша нажата
18         switch (key) { // анализируем обрабатываемую клавишу
19             case GLFW_KEY_ESCAPE: // если клавиша - Escape
20                 // устанавливаем, что окно window должно быть закрыто
21                 glfwSetWindowShouldClose(window, GL_TRUE);
22                 break;
23             default:
24                 break;
25         }
26     }
27 }
28
29 int main () {
30     glfwInit(); // Инициализация GLFW
31     // Проведение начальных установок GLFW
32     // Задается минимальная требуемая версия OpenGL.
33     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3); // Номер до десятичной точки
34     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3); // Номер после десятичной точки
35     // Используем только средства указанной версии без совместимости с более ранними
36     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
37
38     // Создаем окно
39     GLFWwindow* window = glfwCreateWindow(800, 600, "Task 7. Ivanov", NULL, NULL);
40     if (window == NULL) { // если ссылка на окно не создана
41         std::cout << "Вызов glfwCreateWindow закончился неудачей." << std::endl;
42         glfwTerminate(); // завершить работу GLFW
43         return -1; // завершить программу
44     }
45     glfwMakeContextCurrent(window); // делаем окно window активным (текущим)
```

```

46 // Назначение обработчика события Resize
47 glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
48 // Назначение обработчика нажатия клавиш
49 glfwSetKeyCallback(window, key_callback);
50
51 // Инициализация GLAD
52 if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
53     std::cout << "Не удалось загрузить GLAD" << std::endl;
54     glfwTerminate(); // завершить работу GLFW
55     return -1;      // завершить программу
56 }
57
58 // сообщаем диапазон координат в окне
59 // (0, 0) - координаты левого нижнего угла, 800x600 - размеры окна в пикселах
60 glViewport(0, 0, 800, 600);
61
62 //=====
63 //                ВЕРШИНЫЙ ШЕЙДЕР
64 //=====
65 const char *vertexShaderSource =
66     "#version 330 core\n"
67     "layout (location = 0) in vec3 position;\n"
68     "void main() {\n"
69     "    gl_Position = vec4(position.x, position.y, position.z, 1.0);\n"
70     "}\n0";
71 //=====
72
73 GLuint vertexShader; // шейдерный объект - вершинный шейдер
74 vertexShader = glCreateShader(GL_VERTEX_SHADER); // создаем объект
75 // привязываем исходный код к шейдерному объекту
76 glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
77 glCompileShader(vertexShader); // компилируем шейдер
78 GLint success; // результат компиляции
79 // запрашиваем статус компиляции шейдера в переменную success
80 glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
81 if (!success) { // если компиляция прошла с ошибкой
82     GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
83     glGetShaderInfoLog(vertexShader, 512, NULL, infoLog); // запрашиваем сообщение
84     // выводим сообщение об ошибке на экран
85     std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
86 }
87
88 //=====
89 //                ФРАГМЕНТНЫЙ ШЕЙДЕР
90 //=====
91 const char *fragmentShaderSource =
92     "#version 330 core\n"
93     "out vec4 color;\n"
94     "void main() {\n"
95     "    color = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
96     "}\n0";
97 //=====
98
99 GLuint fragmentShader; // шейдерный объект - фрагментный шейдер
100 fragmentShader = glCreateShader(GL_FRAGMENT_SHADER); // создаем объект
101 // привязываем исходный код к шейдерному объекту
102 glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
103 glCompileShader(fragmentShader); // компилируем шейдер
104 // запрашиваем статус компиляции шейдера в описанную ранее переменную success
105 glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
106 if (!success) { // если компиляция прошла с ошибкой
107     GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
108     glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog); // запрашиваем сообщение
109     // выводим сообщение об ошибке на экран
110     std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog << std::endl;
111 }
112
113 // Шейдерная программа
114 GLuint shaderProgram; // идентификатор шейдерной программы
115 shaderProgram = glCreateProgram(); // создаем программный объект
116 glAttachShader(shaderProgram, vertexShader); // присоединяем вершинный шейдер
117 glAttachShader(shaderProgram, fragmentShader); // и фрагментный шейдер
118 glLinkProgram(shaderProgram); // компоновка программы
119 // запрашиваем статус компоновки шейдерной программы в переменную success

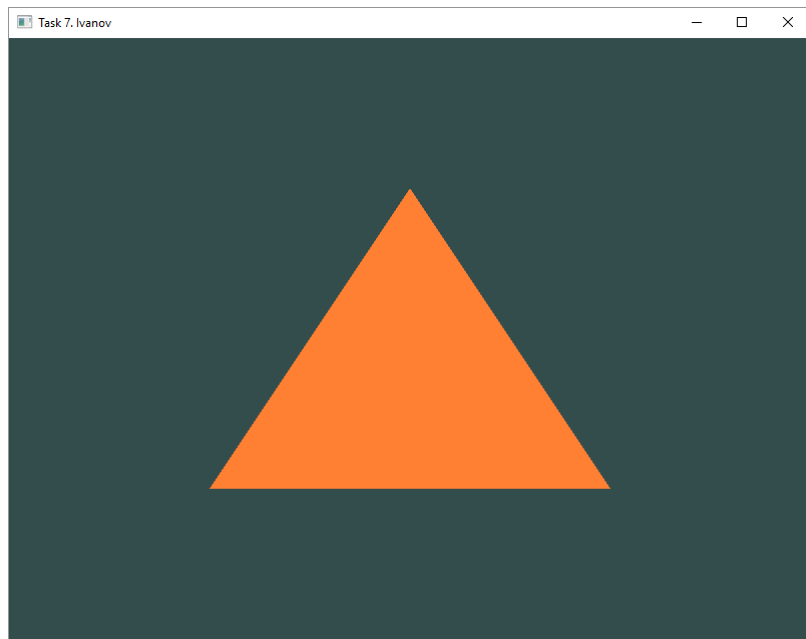
```

```

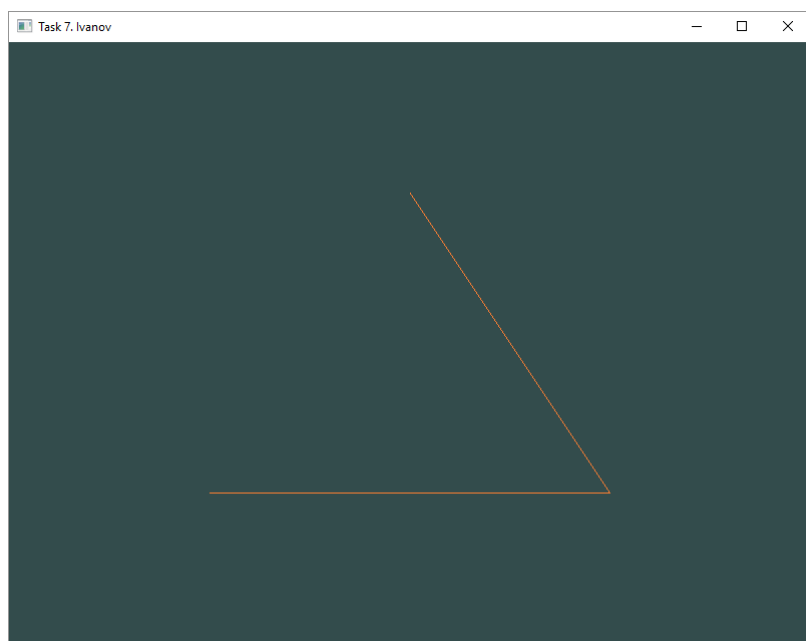
120 glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
121 if (!success) { // если компоновка прошла с ошибкой
122     GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
123     glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog); // запрашиваем сообщение
124     // выводим сообщение об ошибке на экран
125     std::cout << "ERROR::SHADER::PROGRAM::LINK_FAILED\n" << infoLog << std::endl;
126 }
127
128 // удаление шейдерных объектов
129 glDeleteShader(vertexShader);
130 glDeleteShader(fragmentShader);
131
132 //=====
133 // НАБОР ИСХОДНЫХ ДАННЫХ ДЛЯ ОТРИСОВКИ
134 //=====
135 GLfloat vertices[] = {
136     -0.5f, -0.5f, 0.0f,
137     0.5f, -0.5f, 0.0f,
138     0.0f, 0.5f, 0.0f
139 };
140
141 GLuint vertexArray; // объект вершинного массива
142 // создаем вершинный массив, идентификатор которого присваиваем vertexArray
143 glGenVertexArrays(1, &vertexArray);
144 glBindVertexArray(vertexArray); // делаем активным вершинный массив
145
146 GLuint vertexBuffer; // идентификатор буферного объекта
147 // создаем буферный объект, идентификатор которого присваиваем vertexBuffer
148 glGenBuffers(1, &vertexBuffer);
149 // привязка vertexBuffer к GL_ARRAY_BUFFER
150 glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
151 // в буфер, привязанный к GL_ARRAY_BUFFER копируем содержимое vertices
152 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
153 // описание расположения параметра вершинного шейдера в вершинном буфере
154 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
155 glEnableVertexAttribArray(0); // включение параметра 0 для шейдера
156 glBindBuffer(GL_ARRAY_BUFFER, 0); // отвязка буферного объекта
157
158 glBindVertexArray(0); // отключение вершинного массива
159
160 while (!glfwWindowShouldClose(window)) { // пока окно window не должно закрыться
161     glClearColor(0.2f, 0.3f, 0.3f, 1.0f); // назначаем цвет заливки
162     glClear(GL_COLOR_BUFFER_BIT); // очищаем буфер заданным цветом
163
164     glUseProgram(shaderProgram); // шейдерную программу shaderProgram делаем активной
165     glBindVertexArray(vertexArray); // делаем активным вершинный массив
166     glDrawArrays(GL_TRIANGLES, 0, 3); // отрисовка одного треугольника
167     glBindVertexArray(0); // отключаем вершинный массив
168
169     glfwSwapBuffers(window); // поменять местами буферы изображения
170     glfwPollEvents(); // проверить, произошли ли какие-то события
171 }
172
173 glfwTerminate(); // завершить работу GLFW
174 return 0;
175 }

```

Наша программа приняла работоспособную форму. Её запуск приведет к отрисовке треугольника.



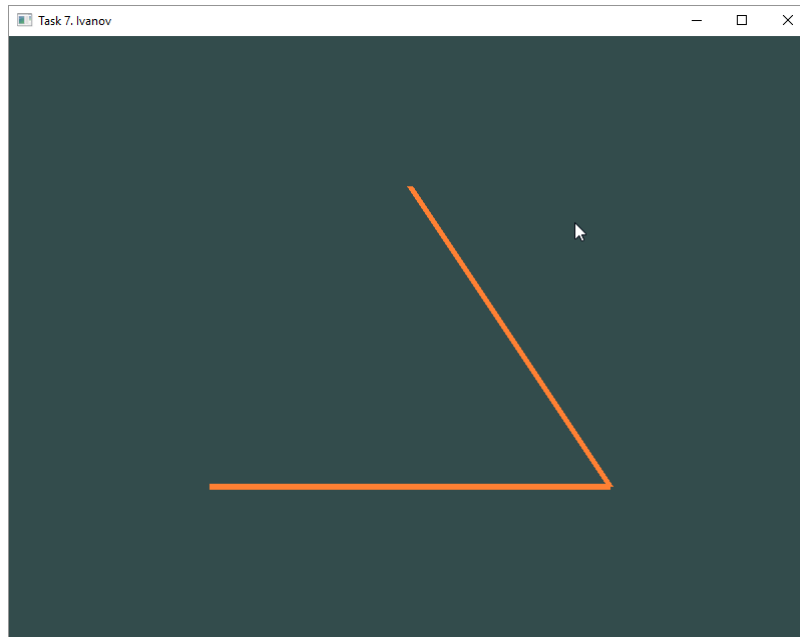
Если в вызове процедуры `glDrawArrays` заменить `GL_TRIANGLES` на `GL_LINE_STRIP`, то получим ломаную, определенную нашей последовательностью точек.



Можно изменить толщину отрисовываемой линии, если перед вызовом `glDrawArrays` поставить вызов `glLineWidth`.

```
glLineWidth(6); // устанавливаем толщину линии - 6
```

Получим изображение



В рамках этого урока мы, как и раньше, будем общаться с отрисовкой наборов ломаных.

### 1.3 Отрисовка трехмерной сцены

Будем считать, что трехмерная сцена задается текстовым файлом в том же формате, что в задании 6. Мы написали достаточно объемный программный код для чтения информации из такого файла и представления этой информации в графической форме. В этом разделе мы адаптируем уже написанный программный код в рамках приложения для отрисовки с использованием OpenGL.

Сначала обсудим основные изменения в проекте по отношению к нашей реализации.

#### 1.3.1 Адаптация готовых фрагментов программного кода

Прежде всего, как уже говорили, мы не будем использовать собственные реализации матричных операций. Вместо этого мы используем матричные определения и операции, предоставляемые библиотекой GLM. Мы использовали в описаниях `Matrix.h` и `Transform.h` по большей части имена типов, наименования и сигнатуры операций такие же, какие предоставляет библиотека GLM. Поэтому переход от наших определений к новым не должен вызвать затруднений.

Если подключен модуль `glm\glm.hpp` библиотеки GLM то нам доступны векторные и матричные типы с теми же именами, что мы использовали ранее, но определенные в пространстве имен `glm`. Поэтому, обращение к этим именам будем производить с префиксом `glm::`. То же касается большинства функций и процедур, определенных на векторах/матрицах.

Стоит обратить внимание, что для векторов в GLM имеются конструкторы от одного числового аргумента, которые создают вектор, все компоненты которого равны этому аргументу. Например

```
glm::vec4 v = glm::vec4(1.f); // v = (1.f, 1.f, 1.f, 1.f)
```

В GLM присутствует большинство тригонометрических функций, поэтому при необходимости их использования библиотеку `math.h` подключать не нужно. Также есть полезная для нас функция `radians`, переводящая заданный в градусах угол в радианы.

Функция `glm::normalize`, в отличие от реализованной нами функции `normalize`, возвращает единичный вектор, сонаправленный с аргументом (аналог нашей функции `norm`).

Что же касается перехода из однородных координат в евклидовы, то он не оформлен в виде отдельной процедуры. Но на самом деле он нам и не нужен. Действительно, при переходе к однородным координатам мы всегда приписываем единицу в качестве дополнительной компоненты. При использовании стандартных преобразований последняя компонента однородных координат точки всегда остается равной единице. Поэтому, переход от таких однородных координат к евклидовым будет состоять просто в том, чтобы отбросить единицу (так как деление на единицу оставшихся координат ничего не меняет). Только в двух случаях в наших предыдущих решениях получалась четвертая (дополнительная) координата неравная единице. Первый раз — в функции нормализации вектора (функции `norm`), реализация которой в GLM уже есть. Второй раз — при переходе к двумерным координатам после применения перспективного преобразования. Но этот переход в OpenGL происходит без нашего участия: вершинный шейдер должен выдавать однородные координаты точки, а переход к евклидовым координатам проводится в графическом конвейере автоматически. Таким образом, надобность в функции перехода из однородных координат у нас отпала (хотя такую функцию можно было бы легко реализовать).

В случае подключения модуля `glm\gtc\transform.hpp` мы будем иметь в запасе функции, возвращающие матрицы трехмерных преобразований вращения, масштабирования и переноса в почти привычном для нас виде. Только в функциях `glm::scale` и `glm::translate` не три вещественных аргумента, а один — типа `glm::vec3`. Кроме этого, стоит отметить, что функция `glm::rotate` не определена для случая, когда первый аргумент типа `double`. Чтобы не возникало ошибок компиляции, первый аргумент должен быть приведен к типу `float`.

Функция `glm::ortho` в качестве двух последних аргументов принимает не значения координаты  $z$ , а расстояния до плоскости окна наблюдения и плоскости горизонта (как в функциях `frustum` и `perspective`).

Стоит учесть ещё один момент: при считывании компонент вектора из файла не следует считывать значения прямо в вектор, как например

```
glm::vec3 v;  
in >> v.x >> v.y >> v.z;
```

Компиляция такого фрагмента пройдет успешно, но система подсказок Visual Studio Intellisense скорее всего перестанет корректно функционировать для последующего кода. Лучше использовать вспомогательные скалярные переменные, для дальнейшего формирования из них нужного вектора. Например, для предыдущего фрагмента,

```
float x, y, z;  
in >> x >> y >> z;  
glm::vec3 v = vec3(x, y, z);
```

Адаптация кода, не относящегося к использованию операций с матрицами, по большей части сводится к тому, что мы не подключили пространство имен `std`, из-за чего появится необходимость в добавлении префиксов `std::` у некоторых имен типов и процедур.



Рекомендуем, при выполнении этого урока, не подключать пространства имен `glm` и `std` (не использовать `using namespace`), чтобы получить ясное представление о



принадлежности тех или иных имен к определенным библиотекам. По большей части, Intellisense отметит Вам те фрагменты кода, в которые необходимо внести исправления.

### 1.3.2 Структуры данных для информации из файла

Организуем структуры данных, в которые будем производить считывание информации из файла, и из которых будем загружать информацию в вершинные массивы. За основу возьмем структуру, описанную в файле `Figure.h`, полученном в ходе выполнения 6-го урока. Скопируйте этот файл внутрь папки с проектом (в моем случае — папка `Ivanov`) и добавьте файл в проект (пункт меню *Проект* → *Добавить существующий элемент* и выбор файла `Figure.h` из папки `task_7_Ivanov\Ivanov`).

Сейчас файл имеет следующий вид.

```

1  #pragma once
2  #include "Matrix.h"
3  #include <vector>
4
5  class path {
6  public:
7      std::vector<vec3> vertices; // последовательность точек
8      vec3 color; // цвет, разбитый на составляющие RGB
9      float thickness; // толщина линии
10     path(std::vector<vec3> verts, vec3 col, float thickn) {
11         vertices = verts;
12         color = col;
13         thickness = thickn;
14     }
15 };
16
17 class model {
18 public:
19     std::vector<path> figure; // составляющие рисунка
20     mat4 modelM; // модельная матрица
21     model(std::vector<path> fig, mat4 mat) {
22         figure = fig;
23         modelM = mat;
24     }
25 };

```

Адаптируем это описание под наш проект.  
Заменим подключение файла `Matrix.h` на

```
#include <glm\glm.hpp>
```

Перед именами типов `vec3` и `mat4` добавим префикс `glm::`.

Каждый объект типа `path` описывает отдельную ломаную, которую при использовании OpenGL целесообразно выводить отдельным вызовом процедуры `glDrawArrays`. Поступим относительно примитивным способом. Для каждой такой ломаной сформируем вершинный массив, который сделаем составной частью объекта. Тогда в дальнейшем, при отрисовке ломаной, нам понадобится лишь сделать этот вершинный массив активным и вызвать для него `glDrawArrays`. То есть, мы сделаем объекты OpenGL (идентификаторы объектов) составляющими объекта класса `path`.

Подключим библиотеку GLAD, для чего в самом начале файла добавим строку

```
#include <glad\glad.h>
```

Добавим в класс `path` описание поля для вершинного массива

```
GLuint vertexArray; // вершинный массив (объект OpenGL)
```

Кроме того, добавим приватное поле для вершинного буфера.

```
private:
GLuint vertexBuffer; // вершинный буфер (объект OpenGL)
```

Оформим формирование вершинного массива в отдельном приватном методе `setupPath`

```
void setupPath() {
}
```

В рамках этого метода мы повторим действия, которые мы проводили при создании вершинного массива в разделе 1.2.4. Сначала создадим вершинный массив и вершинный буфер (переменные для них уже имеются).

```
glGenVertexArrays(1, &vertexArray); // создаем вершинный массив
glGenBuffers(1, &vertexBuffer); // создаем вершинный буфер
```

Потом сделаем вершинный массив активным.

```
glBindVertexArray(vertexArray); // делаем вершинный массив активным
```

Буферный объект связываем с контекстом.

```
// связываем vertexBuffer с GL_ARRAY_BUFFER
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
```

Копируем список вершин в буфер.

```
// копируем содержимое vertices в вершинный буфер vertexBuffer
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec3), &vertices[0],
             GL_STATIC_DRAW);
```



В отличие от предыдущего случая, `vertices` является не массивом, а вектором. С этим связано использование конструкции `&vertices[0]` (адрес первого элемента вектора) на том месте, где мы использовали раньше просто имя массива.

Описываем расположение описаний точек в вершинном буфере

```
// описание расположения параметра вершинного шейдера в вершинном буфере
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(glm::vec3), (GLvoid*)0);
```

Включаем параметр для шейдера.

```
glEnableVertexAttribArray(0); // включение параметра 0 для шейдера
```

Отвязываем буферный объект и делаем вершинный массив неактивным

```
glBindBuffer(GL_ARRAY_BUFFER, 0); // отвязка буферного объекта
glBindVertexArray(0); // отключение вершинного массива
```

Получили метод `setupPath` в следующем виде.

```
1 void setupPath() {
2     // создаем вершинный массив и вершинный буфер
3     glGenVertexArrays(1, &vertexArray); // создаем вершинный массив
4     glGenBuffers(1, &vertexBuffer); // создаем вершинный буфер
5     glBindVertexArray(vertexArray); // делаем вершинный массив активным
6     // связываем vertexBuffer с GL_ARRAY_BUFFER
7     glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
8     // копируем содержимое vertices в вершинный буфер vertexBuffer
9     glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec3), &vertices[0], GL_STATIC_DRAW);
10    // описание расположения параметра вершинного шейдера в вершинном буфере
11    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(glm::vec3), (GLvoid*)0);
12    glEnableVertexAttribArray(0); // включение параметра 0 для шейдера
13    glBindBuffer(GL_ARRAY_BUFFER, 0); // отвязка буферного объекта
14    glBindVertexArray(0); // отключение вершинного массива
15 }
```

Добавим вызов метода `setupPath` последним действием в конструктор `path`. Конструктор примет следующий вид.

```
1 path(std::vector<glm::vec3> verts, glm::vec3 col, float thickn) {
2     vertices = verts;
3     color = col;
4     thickness = thickn;
5     setupPath();
6 }
```

Весь файл `Figure.h` примет следующий вид.

```
1 #pragma once
2 #include <glad\glad.h>
3 #include <glm\glm.hpp>
4
5 #include <vector>
6
7 class path {
8 public:
9     std::vector<glm::vec3> vertices; // последовательность точек
10    glm::vec3 color; // цвет, разбитый на составляющие RGB
11    float thickness; // толщина линии
12    GLuint vertexArray; // вершинный массив (объект OpenGL)
13    path(std::vector<glm::vec3> verts, glm::vec3 col, float thickn) {
14        vertices = verts;
15        color = col;
16        thickness = thickn;
17        setupPath();
18    }
19
20 private:
21    GLuint vertexBuffer; // вершинный буфер (объект OpenGL)
22    void setupPath() {
23        // создаем вершинный массив и вершинный буфер
24        glGenVertexArrays(1, &vertexArray); // создаем вершинный массив
25        glGenBuffers(1, &vertexBuffer); // создаем вершинный буфер
26        glBindVertexArray(vertexArray); // делаем вершинный массив активным
27        // связываем vertexBuffer с GL_ARRAY_BUFFER
28        glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
29        // копируем содержимое vertices в вершинный буфер vertexBuffer
30        glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec3), &vertices[0], GL_STATIC_DRAW);
31        // описание расположения параметра вершинного шейдера в вершинном буфере
32        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(glm::vec3), (GLvoid*)0);
33        glEnableVertexAttribArray(0); // включение параметра 0 для шейдера
34        glBindBuffer(GL_ARRAY_BUFFER, 0); // отвязка буферного объекта
35        glBindVertexArray(0); // отключение вершинного массива
36    }
37 };
38
```

```

39 class model {
40 public:
41     std::vector<path> figure; // составляющие рисунка
42     glm::mat4 modelM; // модельная матрица
43     model(std::vector<path> fig, glm::mat4 mat) {
44         figure = fig;
45         modelM = mat;
46     }
47 };

```

Чтобы проверить, что компиляция файла проходит без ошибок, в конце блока подключения заголовочных файлов в `Main.cpp` добавим следующую строку.

```
#include "Figure.h"
```

Так как мы пока не используем описанные в файле `Figure.h` структуры, наш проект будет работать без изменений.

Все дальнейшие изменения кода будут касаться файла `Main.cpp`.

### 1.3.3 Чтение из файла

Организуем чтение входного файла по той же схеме, по которой мы это делали в обработчике нажатия кнопки на форме в шестом задании.

Нам потребуются средства библиотек `fstream`, `sstream` и `string`. Подключим их.

```

#include <fstream>
#include <sstream>
#include <string>
#include <vector>

```

После блока команд `#include` добавим описание глобальных переменных — аналогов переменных файла `MyForm.h` в 6-м задании. Скопируем его из файла `MyForm.h`. Добавим префиксы `std::` и `glm::` там где необходимо. Кроме того, имена `near` и `far` вступают в конфликт с библиотечными константами Visual Studio, поэтому переименуем их в `near_view`, `far_view`. Получим следующий блок описаний

```

1 //=====
2 //                                ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
3 //=====
4 std::vector<model> models;
5 glm::mat4 T; // матрица, в которой накапливаются все преобразования
6 glm::vec3 S, P, u; // координаты точки наблюдения
7                 // точки, в которую направлен вектор наблюдения
8                 // вектора направления вверх
9 float dist; // вспомогательная переменная - расстояние между S и P
10 float fovy, aspect; // угол обзора и соотношение сторон окна наблюдения
11 float fovy_work, aspect_work; // рабочие переменные для fovy и aspect
12 float near_view, far_view; // расстояния до окна наблюдения и до горизонта
13 float n, f; // рабочие переменные для near_view и far_view
14 float l, r, t, b; // рабочие вспомогательные переменные
15                 // для значений координат левой, правой,
16                 // нижней и верхней координаты в СКН
17 enum projType { Ortho, Frustum, Perspective } pType; // тип трехмерной проекции
18 //=====

```

После описания глобальных переменных вставим копию процедуры `initWorkPars` (опять из файла `MyForm.h`) для вычисления рабочих параметров. Адаптируем ее под наш проект: изменим тип результата на `void`, заменим `near` и `far` на `near_view` и `far_view`, добавим префикс `glm::` там, где необходимо. Получим процедуру в следующем виде.

```

1 void initWorkPars() { // инициализация рабочих параметров камеры
2     n = near_view;
3     f = far_view;
4     fovy_work = fovy;
5     aspect_work = aspect;
6     float Vy = 2 * near_view * glm::tan(fovy / 2);
7     float Vx = aspect * Vy;
8     l = -Vx / 2;
9     r = l;
10    b = -Vy / 2;
11    t = b;
12    dist = glm::length(P - S);
13    T = glm::lookAt(S, P, u);
14 }

```

Теперь добавим процедуру чтения из файла, которая, как и раньше, будет заполнять глобальные переменные. Назовем ее `readFromFile` и будем передавать ей имя файла для чтения.

```

void readFromFile(const char* fileName) { // чтение сцены из файла fileName
}

```

Так как в этой процедуре имя файла предполагается известным, то телом этой процедуры будет аналог фрагмента метода `btnOpen_Click` из `MyForm.h`, начиная с объявления потока для чтения из файла, т. е. фрагмент, начинающийся с

```

// объявление и открытие файла
ifstream in;
in.open(fileName);

```

и заканчивающийся

```

initWorkPars();
Refresh();
}

```

Скопируем этот фрагмент.

Первое изменение, которое следует внести в полученный код, чтобы не было каких-то дальнейших недоразумений, — заменить блок, относящийся к чтению команды `camera`

```

1 if (cmd == "camera") { // положение камеры
2     s >> S.x >> S.y >> S.z; // координаты точки наблюдения
3     s >> P.x >> P.y >> P.z; // точка, в которую направлен вектор наблюдения
4     s >> u.x >> u.y >> u.z; // вектор направления вверх
5 }

```

на аналог, в котором прямое чтение компонент векторов заменено на чтение в промежуточные скалярные переменные:

```

if (cmd == "camera") { // положение камеры
    float x, y, z; // промежуточные переменные для чтения из файла, чтобы не упал
    ↪ Intellisense
    s >> x >> y >> z; // координаты точки наблюдения
    S = glm::vec3(x, y, z);
    s >> x >> y >> z; // точка, в которую направлен вектор наблюдения
    P = glm::vec3(x, y, z);
    s >> x >> y >> z; // вектор направления вверх
    u = glm::vec3(x, y, z);
}

```

Теперь можно приступить к внесению остальных исправлений.

Всю последовательность действий оставляем без изменений, только адаптируем код, как было сказано ранее.

До блока, относящегося к команде `screen` адаптация будет заключаться в добавлении префикса `std::` для имен библиотек `vector`, `string`, `fstream`, `sstream` и префикса `glm::` для имен из библиотеки GLM.

В блоке команды `screen`

```

1 else if (cmd == "screen") { // положение окна наблюдения
2     s >> fovy_work >> aspect >> near >> far; // параметры команды
3     fovy = fovy_work / 180.f * Math::PI; // перевод угла из градусов в радианы
4 }

```

мы сталкиваемся с вычислением угла в радианах при заданном значении в градусах. Заменим соответствующее выражение в этом блоке на вызов функции `glm::radians`. Получим

```

else if (cmd == "screen") { // положение окна наблюдения
    s >> fovy_work >> aspect >> near_view >> far_view; // параметры команды
    fovy = glm::radians(fovy_work); // перевод угла из градусов в радианы
}

```

В блоке, относящемся к команде `model` встречаются матричные операции в строке

```
initM = scale(S, S, S) * translate(-mVcx, -mVcy, -mVcz);
```

Как оговаривалось в разделе 1.3.1, функциям `scale` и `translate` следует передавать векторы, координатами которых являются коэффициенты преобразований. То есть указанную строку заменим на следующую.

```
initM = glm::scale(glm::vec3(S)) * glm::translate(glm::vec3(-mVcx, -mVcy, -mVcz));
```

Аналогичные преобразования следует провести в блоках для команд `translate` и `scale`.

В блоке для команды `rotate` снова вызовем функцию для перевода градусов в радианы.

Вызов `Refresh()` в конце процедуры остается лишним. Избавимся от него.

Наконец, вспомним, что компоненты RGB для цвета в OpenGL задаются вещественными значениями от 0.f до 1.f. В нашем входном файле эти компоненты задаются значениями от 0 до 255. Изменим код нашей процедуры, чтобы в объекте класса `path` сохранялись значения в формате OpenGL.

Объект класса `path` создается в последней строке блока, соответствующего команде `path` (после адаптации под наш проект):

```
figure.push_back(path(vertices, glm::vec3(r, g, b), thickness));
```

Чтобы передать конструктору `path` «нормализованный» вектор цвета, разделим каждую компоненту второго аргумента конструктора на `255.f`, т. е.

```
figure.push_back(path(vertices, glm::vec3(r, g, b) / 255.f, thickness));
```

Процедура чтения из файла готова. Её общий вид должен быть следующий.

```
1 void readFromFile(const char* fileName) { // чтение сцены из файла fileName
2     // объявление и открытие файла
3     std::ifstream in;
4     in.open(fileName);
5     if (in.is_open()) {
6         // файл успешно открыт
7         models.clear(); // очищаем имеющийся список рисунков
8         // временные переменные для чтения из файла
9         glm::mat4 M = glm::mat4(1.f); // матрица для получения модельной матрицы
10        glm::mat4 initM; // матрица для начального преобразования каждого рисунка
11        std::vector<glm::mat4> transforms; // стек матриц преобразований
12        std::vector<path> figure; // список ломаных очередного рисунка
13        float thickness = 2; // толщина со значением по умолчанию 2
14        float r, g, b; // составляющие цвета
15        r = g = b = 0; // значение составляющих цвета по умолчанию (черный)
16        std::string cmd; // строка для считывания имени команды
17        // непосредственно работа с файлом
18        std::string str; // строка, в которую считываем строки файла
19        std::getline(in, str); // считываем из входного файла первую строку
20        while (in) { // если очередная строка считана успешно
21            // обрабатываем строку
22            if ((str.find_first_not_of(" \t\r\n") != std::string::npos) && (str[0] != '#')) {
23                // прочитанная строка не пуста и не комментарий
24                std::stringstream s(str); // строковый поток из строки str
25                s >> cmd;
26                if (cmd == "camera") { // положение камеры
27                    float x, y, z;
28                    s >> x >> y >> z; // координаты точки наблюдения
29                    S = glm::vec3(x, y, z);
30                    s >> x >> y >> z; // точка, в которую направлен вектор наблюдения
31                    P = glm::vec3(x, y, z);
32                    s >> x >> y >> z; // вектор направления вверх
33                    u = glm::vec3(x, y, z);
34                }
35                else if (cmd == "screen") { // положение окна наблюдения
36                    s >> fovy_work >> aspect >> near_view >> far_view; // параметры команды
37                    fovy = glm::radians(fovy_work); // перевод угла из градусов в радианы
38                }
39                else if (cmd == "color") { // цвет линии
40                    s >> r >> g >> b; // считываем три составляющие цвета
41                }
42                else if (cmd == "thickness") { // толщина линии
43                    s >> thickness; // считываем значение толщины
44                }
45                else if (cmd == "path") { // набор точек
46                    std::vector<glm::vec3> vertices; // список точек ломаной
47                    int N; // количество точек
48                    s >> N;
49                    std::string str1; // дополнительная строка для чтения из файла
50                    while (N > 0) { // пока не все точки считали
51                        std::getline(in, str1); // считываем в str1 из входного файла очередную строку
52                        // так как файл корректный, то на конец файла проверять не нужно
53                        if ((str1.find_first_not_of(" \t\r\n") != std::string::npos) && (str1[0] != '#')) {
54                            // прочитанная строка не пуста и не комментарий
55                            // значит в ней пара координат
56                            float x, y, z; // переменные для считывания
57                            std::stringstream s1(str1); // еще один строковый поток из строки str1
58                            s1 >> x >> y >> z;
59                            vertices.push_back(glm::vec3(x, y, z)); // добавляем точку в список
60                            N--; // уменьшаем счетчик после успешного считывания точки
61                        }
51                    }
62                }
63            }
64        }
65    }
66 }
```

```

62     }
63     // все точки считаны, генерируем ломаную (path) и кладем ее в список figure
64     figure.push_back(path(vertices, glm::vec3(r, g, b) / 255.f, thickness));
65 }
66 else if (cmd == "model") { // начало описания нового рисунка
67     float mVcx, mVcy, mVcz, mVx, mVy, mVz; // параметры команды model
68     s >> mVcx >> mVcy >> mVcz >> mVx >> mVy >> mVz; // считываем значения переменных
69     float S = mVx / mVy < 1 ? 2.f / mVy : 2.f / mVx;
70     // сдвиг точки привязки из начала координат в нужную позицию
71     // после которого проводим масштабирование
72     initM = glm::scale(glm::vec3(S)) * glm::translate(glm::vec3(-mVcx, -mVcy, -mVcz));
73     figure.clear();
74 }
75 else if (cmd == "figure") { // формирование новой модели
76     models.push_back(model(figure, M * initM));
77 }
78 else if (cmd == "translate") { // перенос
79     float Tx, Ty, Tz; // параметры преобразования переноса
80     s >> Tx >> Ty >> Tz; // считываем параметры
81     M = glm::translate(glm::vec3(Tx, Ty, Tz)) * M; // добавляем перенос к общему преобразованию
82 }
83 else if (cmd == "scale") { // масштабирование
84     float S; // параметр масштабирования
85     s >> S; // считываем параметр
86     M = glm::scale(glm::vec3(S)) * M; // добавляем масштабирование к общему преобразованию
87 }
88 else if (cmd == "rotate") { // поворот
89     float theta; // угол поворота в градусах
90     float nx, ny, nz; // координаты направляющего вектора оси вращения
91     s >> theta >> nx >> ny >> nz; // считываем параметры
92     // добавляем вращение к общему преобразованию
93     M = glm::rotate(glm::radians(theta), glm::vec3(nx, ny, nz)) * M;
94 }
95 else if (cmd == "pushTransform") { // сохранение матрицы в стек
96     transforms.push_back(M); // сохраняем матрицу в стек
97 }
98 else if (cmd == "popTransform") { // откат к матрице из стека
99     M = transforms.back(); // получаем верхний элемент стека
100    transforms.pop_back(); // выкидываем матрицу из стека
101 }
102 }
103 // считываем очередную строку
104 std::getline(in, str);
105 }
106 initWorkPars();
107 }
108 }

```

Можно провести компиляцию программы, чтобы посмотреть, что мы не допустили ошибок при её составлении. Но добавленный функционал мы в программе пока не используем, поэтому проект будет работать без изменений.

### 1.3.4 Отрисовка сцены

Изменим содержимое цикла отрисовки в процедуре `main` с тем, чтобы в нем выводились ломаные, определенные входным файлом, информация о которых сохраняется в списке `models`.

Закомментируйте фрагмент кода в цикле, начинающийся и заканчивающийся командами `glBindVertexArray`:

```

1 glBindVertexArray(vertexArray); // делаем активным вершинный массив
2 glLineWidth(6); // устанавливаем толщину линии - 6
3 glDrawArrays(GL_LINE_STRIP, 0, 3);
4 glBindVertexArray(0); // отключаем вершинный массив

```

Вместо этого блока скопируем сюда фрагмент, ответственный за отрисовку сцены, из обработчика события *Paint*:



```

1 mat4 proj; // матрица перехода в пространство отсечения
2 switch (pType) {
3     case Ortho: // прямоугольная проекция
4         proj = ortho(l, r, b, t, -n, -f);
5         break;
6     case Frustum: // перспективная проекция с Frustum
7         proj = frustum(l, r, b, t, n, f);
8         break;
9     case Perspective: // перспективная проекция с Perspective
10        proj = perspective(fovy_work, aspect_work, n, f);
11        break;
12    }
13    // матрица кадрирования
14    mat3 cdr = cadrRL(vec2(-1.f, -1.f), vec2(2.f, 2.f), vec2(Wcx, Wcy), vec2(Wx, Wy));
15    mat4 C = proj * T; // матрица перехода от мировых координат в пространство отсечения
16    for (int k = 0; k < models.size(); k++) { // цикл по моделям
17        vector<path> figure = models[k].figure; // список ломаных очередной модели
18        mat4 TM = C * models[k].modelM; // матрица общего преобразования модели
19        for (int i = 0; i < figure.size(); i++) {
20            path lines = figure[i]; // lines - очередная ломаная линия
21            Pen^ pen = gcnew Pen(Color::FromArgb(lines.color.x, lines.color.y, lines.color.z));
22            pen->Width = lines.thickness;
23            // начальная точка первого отрезка в трехмерных евклидовых координатах
24            vec3 start_3D = normalize(TM * vec4(lines.vertices[0], 1.0));
25            // начальная точка первого отрезка в координатах экрана
26            vec2 start = normalize(cdr * vec3(vec2(start_3D), 1.f));
27            for (int j = 1; j < lines.vertices.size(); j++) { // цикл по конечным точкам (от единицы)
28                // конечная точка отрезка в трехмерных евклидовых координатах
29                vec3 end_3D = normalize(TM * vec4(lines.vertices[j], 1.0));
30                // конечная точка отрезка в координатах экрана
31                vec2 end = normalize(cdr * vec3(vec2(end_3D), 1.f));
32                vec2 tmpEnd = end; // продублировали координаты точки для будущего использования
33                if (clip(start, end, minX, minY, maxX, maxY)) { // если отрезок видим
34                    // после отсечения, start и end - концы видимой части отрезка
35                    g->DrawLine(pen, start.x, start.y, end.x, end.y); // отрисовка видимых частей
36                }
37                start = tmpEnd; // конечная точка неотсеченного отрезка становится начальной точкой
38                // следующего
39            }
40        }
41    }
42 }

```

Адаптируем этот фрагмент для нашей отрисовки.

В операторе `switch` кроме добавления префиксов, стоит обратить внимание на вызов функции `ortho`. Как было сказано в разделе 1.3.1, два последних аргумента этой функции нужно передавать в той же форме (с тем же знаком), что и функциям `perspective` и `frustum`. Поэтому, уберем смену знака аргументов `n` и `f`:

```
proj = glm::ortho(l, r, b, t, n, f);
```

Преобразование кадрирования OpenGL сделает самостоятельно, поэтому матрица `cdr` становится ненужной.

В цикле по `i` проводится отрисовка каждой ломаной. Заменяем всё содержимое этого цикла на наш новый алгоритм отрисовки. В начале цикла сделаем активным вершинный массив для `i`-й ломаной

```
glBindVertexArray(figure[i].vertexArray); // делаем активным вершинный массив i-й ломаной
```

Установим толщину ломаной.

```
glLineWidth(figure[i].thickness); // устанавливаем толщину линии
```

Чертим ломаную. Количество точек ломаной извлечем из списка `vertices` объекта `path`.

```
glDrawArrays(GL_LINE_STRIP, 0, figure[i].vertices.size()); // отрисовка ломаной
```

и делаем вершинный массив неактивным.

```
glBindVertexArray(0); // отключаем вершинный массив
```

После изменения обций вид нашего фрагмента отрисовки будет следующим.

```
1 glm::mat4 proj; // матрица перехода в пространство отсечения
2 switch (pType) {
3     case Ortho: // прямоугольная проекция
4         proj = glm::ortho(l, r, b, t, n, f);
5         break;
6     case Frustum: // перспективная проекция с Frustum
7         proj = glm::frustum(l, r, b, t, n, f);
8         break;
9     case Perspective: // перспективная проекция с Perspective
10        proj = glm::perspective(fovy_work, aspect_work, n, f);
11        break;
12 }
13 glm::mat4 C = proj * T; // матрица перехода от мировых координат в пространство отсечения
14 for (int k = 0; k < models.size(); k++) { // цикл по моделям
15     std::vector<path> figure = models[k].figure; // список ломаных очередной модели
16     glm::mat4 TM = C * models[k].modelM; // матрица общего преобразования модели
17     for (int i = 0; i < figure.size(); i++) {
18         glBindVertexArray(figure[i].vertexArray); // делаем активным вершинный массив i-й ломаной
19         glLineWidth(figure[i].thickness); // устанавливаем толщину линии
20         glDrawArrays(GL_LINE_STRIP, 0, figure[i].vertices.size()); // отрисовка ломаной
21         glBindVertexArray(0); // отключаем вершинный массив
22     }
23 }
```

Обратите внимание, что пока мы игнорируем цвет ломаной: с цветом мы разберемся позже. Кроме того, хотя матрица `TM` вычисляется, но её значение нигде не используется. То есть, все преобразования сцены (включая модельное преобразование) игнорируются — ломаные пока выводятся в своих локальных координатах.

Мы уже можем протестировать получившийся код. Но сначала прокомментируем фрагмент процедуры `main`, от начала описания набора исходных данных для отрисовки до цикла `while` — этот фрагмент стал ненужным. Вместо него добавим вызов процедуры чтения файла.

```
readFromFile("triangle.txt");
```

Файл `Main.cpp` на текущем этапе примет вид (исключены закомментированные строки кода).

```
1 #include <iostream>
2 #include <fstream>
3 #include <sstream>
4 #include <string>
5 #include <vector>
6
7 #include <glad\glad.h>
8 #include <GLFW\glfw3.h>
9
10 #include <glm\glm.hpp>
11 #include <glm\gtw\transform.hpp>
12 #include <glm\gtc\type_ptr.hpp>
13
```

```

14 #include "Figure.h"
15
16 //=====
17 //                                ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
18 //=====
19 std::vector<model> models;
20 glm::mat4 T; // матрица, в которой накапливаются все преобразования
21 glm::vec3 S, P, u; // координаты точки наблюдения
22 // точки, в которую направлен вектор наблюдения
23 // вектора направления вверх
24 float dist; // вспомогательная переменная - расстояние между S и P
25 float fovy, aspect; // угол обзора и соотношение сторон окна наблюдения
26 float fovy_work, aspect_work; // рабочие переменные для fovy и aspect
27 float near_view, far_view; // расстояния до окна наблюдения и до горизонта
28 float n, f; // рабочие переменные для near_view и far_view
29 float l, r, t, b; // рабочие вспомогательные переменные
30 // для значений координат левой, правой,
31 // нижней и верхней координаты в СКН
32 enum projType { Ortho, Frustum, Perspective } pType; // тип трехмерной проекции
33 //=====
34
35 void initWorkPars() { // инициализация рабочих параметров камеры
36     n = near_view;
37     f = far_view;
38     fovy_work = fovy;
39     aspect_work = aspect;
40     float Vy = 2 * near_view * glm::tan(fovy / 2);
41     float Vx = aspect * Vy;
42     l = -Vx / 2;
43     r = l;
44     b = -Vy / 2;
45     t = b;
46     dist = glm::length(P - S);
47     T = glm::lookAt(S, P, u);
48 }
49
50 void readFromFile(const char* fileName) { // чтение сцены из файла fileName
51     // объявление и открытие файла
52     std::ifstream in;
53     in.open(fileName);
54     if (in.is_open()) {
55         // файл успешно открыт
56         models.clear(); // очищаем имеющийся список рисунков
57         // временные переменные для чтения из файла
58         glm::mat4 M = glm::mat4(1.f); // матрица для получения модельной матрицы
59         glm::mat4 initM; // матрица для начального преобразования каждого рисунка
60         std::vector<glm::mat4> transforms; // стек матриц преобразований
61         std::vector<path> figure; // список ломаных очередного рисунка
62         float thickness = 2; // толщина со значением по умолчанию 2
63         float r, g, b; // составляющие цвета
64         r = g = b = 0; // значение составляющих цвета по умолчанию (черный)
65         std::string cmd; // строка для считывания имени команды
66         // непосредственно работа с файлом
67         std::string str; // строка, в которую считываем строки файла
68         std::getline(in, str); // считываем из входного файла первую строку
69         while (in) { // если очередная строка считана успешно
70             // обрабатываем строку
71             if ((str.find_first_not_of(" \t\r\n") != std::string::npos) && (str[0] != '#')) {
72                 // прочитанная строка не пуста и не комментарий
73                 std::stringstream s(str); // строковый поток из строки str
74                 s >> cmd;
75                 if (cmd == "camera") { // положение камеры
76                     float x, y, z;
77                     s >> x >> y >> z; // координаты точки наблюдения
78                     S = glm::vec3(x, y, z);
79                     s >> x >> y >> z; // точка, в которую направлен вектор наблюдения
80                     P = glm::vec3(x, y, z);
81                     s >> x >> y >> z; // вектор направления вверх
82                     u = glm::vec3(x, y, z);
83                 }
84                 else if (cmd == "screen") { // положение окна наблюдения
85                     s >> fovy_work >> aspect >> near_view >> far_view; // параметры команды
86                     fovy = glm::radians(fovy_work); // перевод угла из градусов в радианы
87                 }

```

```

88     else if (cmd == "color") { // цвет линии
89         s >> r >> g >> b; // считываем три составляющие цвета
90     }
91     else if (cmd == "thickness") { // толщина линии
92         s >> thickness; // считываем значение толщины
93     }
94     else if (cmd == "path") { // набор точек
95         std::vector<glm::vec3> vertices; // список точек ломаной
96         int N; // количество точек
97         s >> N;
98         std::string str1; // дополнительная строка для чтения из файла
99         while (N > 0) { // пока не все точки считали
100             std::getline(in, str1); // считываем в str1 из входного файла очередную строку
101             // так как файл корректный, то на конец файла проверять не нужно
102             if ((str1.find_first_not_of(" \t\r\n") != std::string::npos) && (str1[0] != '#')) {
103                 // прочитанная строка не пуста и не комментарий
104                 // значит в ней пара координат
105                 float x, y, z; // переменные для считывания
106                 std::stringstream s1(str1); // еще один строковый поток из строки str1
107                 s1 >> x >> y >> z;
108                 vertices.push_back(glm::vec3(x, y, z)); // добавляем точку в список
109                 N--; // уменьшаем счетчик после успешного считывания точки
110             }
111         }
112         // все точки считаны, генерируем ломаную (path) и кладем ее в список figure
113         figure.push_back(path(vertices, glm::vec3(r, g, b) / 255.f, thickness));
114     }
115     else if (cmd == "model") { // начало описания нового рисунка
116         float mVcx, mVcy, mVcz, mVx, mVy, mVz; // параметры команды model
117         s >> mVcx >> mVcy >> mVcz >> mVx >> mVy >> mVz; // считываем значения переменных
118         float S = mVx / mVy < 1 ? 2.f / mVy : 2.f / mVx;
119         // сдвиг точки привязки из начала координат в нужную позицию
120         // после которого проводим масштабирование
121         initM = glm::scale(glm::vec3(S)) * glm::translate(glm::vec3(-mVcx, -mVcy, -mVcz));
122         figure.clear();
123     }
124     else if (cmd == "figure") { // формирование новой модели
125         models.push_back(model(figure, M * initM));
126     }
127     else if (cmd == "translate") { // перенос
128         float Tx, Ty, Tz; // параметры преобразования переноса
129         s >> Tx >> Ty >> Tz; // считываем параметры
130         M = glm::translate(glm::vec3(Tx, Ty, Tz)) * M; // добавляем перенос к общему преобразованию
131     }
132     else if (cmd == "scale") { // масштабирование
133         float S; // параметр масштабирования
134         s >> S; // считываем параметр
135         M = glm::scale(glm::vec3(S)) * M; // добавляем масштабирование к общему преобразованию
136     }
137     else if (cmd == "rotate") { // поворот
138         float theta; // угол поворота в градусах
139         float nx, ny, nz; // координаты направляющего вектора оси вращения
140         s >> theta >> nx >> ny >> nz; // считываем параметры
141         // добавляем вращение к общему преобразованию
142         M = glm::rotate(glm::radians(theta), glm::vec3(nx, ny, nz)) * M;
143     }
144     else if (cmd == "pushTransform") { // сохранение матрицы в стек
145         transforms.push_back(M); // сохраняем матрицу в стек
146     }
147     else if (cmd == "popTransform") { // откат к матрице из стека
148         M = transforms.back(); // получаем верхний элемент стека
149         transforms.pop_back(); // выкидываем матрицу из стека
150     }
151 }
152 // считываем очередную строку
153 std::getline(in, str);
154 }
155 initWorkPars();
156 }
157 }
158
159 // обработчик события Resize
160 void framebuffer_size_callback(GLFWwindow* window, int width, int height) {
161     glViewport(0, 0, width, height);

```

```

162 }
163
164 // Обработчик нажатия клавиш
165 void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode) {
166     if (action != GLFW_RELEASE) { // если клавиша нажата
167         switch (key) { // анализируем обрабатываемую клавишу
168             case GLFW_KEY_ESCAPE: // если клавиша - Escape
169                 // устанавливаем, что окно window должно быть закрыто
170                 glfwSetWindowShouldClose(window, GL_TRUE);
171                 break;
172             default:
173                 break;
174         }
175     }
176 }
177
178 int main () {
179     glfwInit(); // Инициализация GLFW
180     // Проведение начальных установок GLFW
181     // Задаются минимальная требуемая версия OpenGL.
182     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3); // Номер до десятичной точки
183     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3); // Номер после десятичной точки
184     // Используем только средства указанной версии без совместимости с более ранними
185     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
186
187     // Создаем окно
188     GLFWwindow* window = glfwCreateWindow(800, 600, "Task 7. Ivanov", NULL, NULL);
189     if (window == NULL) { // если ссылка на окно не создана
190         std::cout << "Вызов glfwCreateWindow закончился неудачей." << std::endl;
191         glfwTerminate(); // завершить работу GLFW
192         return -1; // завершить программу
193     }
194     glfwMakeContextCurrent(window); // делаем окно window активным (текущим)
195     // Назначение обработчика события Resize
196     glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
197     // Назначение обработчика нажатия клавиш
198     glfwSetKeyCallback(window, key_callback);
199
200     // Инициализация GLAD
201     if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
202         std::cout << "Не удалось загрузить GLAD" << std::endl;
203         glfwTerminate(); // завершить работу GLFW
204         return -1; // завершить программу
205     }
206
207     // сообщаем диапазон координат в окне
208     // (0, 0) - координаты левого нижнего угла, 800x600 - размеры окна в пикселах
209     glViewport(0, 0, 800, 600);
210
211     //=====
212     // ВЕРшинный шейдер
213     //=====
214     const char *vertexShaderSource =
215         "#version 330 core\n"
216         "layout (location = 0) in vec3 position;\n"
217         "void main() {\n"
218         "    gl_Position = vec4(position.x, position.y, position.z, 1.0);\n"
219         "}\n0";
220     //=====
221
222     GLuint vertexShader; // шейдерный объект - вершинный шейдер
223     vertexShader = glCreateShader(GL_VERTEX_SHADER); // создаем объект
224     // привязываем исходный код к шейдерному объекту
225     glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
226     glCompileShader(vertexShader); // компилируем шейдер
227     GLint success; // результат компиляции
228     // запрашиваем статус компиляции шейдера в переменную success
229     glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
230     if (!success) { // если компиляция прошла с ошибкой
231         GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
232         glGetShaderInfoLog(vertexShader, 512, NULL, infoLog); // запрашиваем сообщение
233         // выводим сообщение об ошибке на экран
234         std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
235     }

```

```

236
237 //=====
238 //          ФРАГМЕНТНЫЙ ШЕЙДЕР
239 //=====
240 const char *fragmentShaderSource =
241     "#version 330 core\n"
242     "out vec4 color;\n"
243     "void main() {\n"
244     "    color = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
245     "}\n\n0";
246 //=====
247
248 GLuint fragmentShader; // шейдерный объект - фрагментный шейдер
249 fragmentShader = glCreateShader(GL_FRAGMENT_SHADER); // создаем объект
250 // привязываем исходный код к шейдерному объекту
251 glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
252 glCompileShader(fragmentShader); // компилируем шейдер
253 // запрашиваем статус компиляции шейдера в описанную ранее переменную success
254 glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
255 if (!success) { // если компиляция прошла с ошибкой
256     GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
257     glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog); // запрашиваем сообщение
258     // выводим сообщение об ошибке на экран
259     std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog << std::endl;
260 }
261
262 // Шейдерная программа
263 GLuint shaderProgram; // идентификатор шейдерной программы
264 shaderProgram = glCreateProgram(); // создаем программный объект
265 glAttachShader(shaderProgram, vertexShader); // присоединяем вершинный шейдер
266 glAttachShader(shaderProgram, fragmentShader); // и фрагментный шейдер
267 glLinkProgram(shaderProgram); // компоновка программы
268 // запрашиваем статус компоновки шейдерной программы в переменную success
269 glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
270 if (!success) { // если компоновка прошла с ошибкой
271     GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
272     glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog); // запрашиваем сообщение
273     // выводим сообщение об ошибке на экран
274     std::cout << "ERROR::SHADER::PROGRAM::LINK_FAILED\n" << infoLog << std::endl;
275 }
276
277 // удаление шейдерных объектов
278 glDeleteShader(vertexShader);
279 glDeleteShader(fragmentShader);
280
281 readFromFile("triangle.txt");
282
283 while (!glfwWindowShouldClose(window)) { // пока окно window не должно закрыться
284     glClearColor(0.2f, 0.3f, 0.3f, 1.0f); // назначаем цвет заливки
285     glClear(GL_COLOR_BUFFER_BIT); // очищаем буфер заданным цветом
286
287     glUseProgram(shaderProgram); // шейдерную программу shaderProgram делаем активной
288
289     glm::mat4 proj; // матрица перехода в пространство отсечения
290     switch (pType) {
291     case Ortho: // прямоугольная проекция
292         proj = glm::ortho(l, r, b, t, n, f);
293         break;
294     case Frustum: // перспективная проекция с Frustum
295         proj = glm::frustum(l, r, b, t, n, f);
296         break;
297     case Perspective: // перспективная проекция с Perspective
298         proj = glm::perspective(fovy_work, aspect_work, n, f);
299         break;
300     }
301     glm::mat4 C = proj * T; // матрица перехода от мировых координат в пространство отсечения
302     for (int k = 0; k < models.size(); k++) { // цикл по моделям
303         std::vector<path> figure = models[k].figure; // список ломаных очередной модели
304         glm::mat4 TM = C * models[k].modelM; // матрица общего преобразования модели
305         for (int i = 0; i < figure.size(); i++) {
306             glBindVertexArray(figure[i].vertexArray); // делаем активным вершинный массив i-й ломаной
307             glLineWidth(figure[i].thickness); // устанавливаем толщину линии
308             glDrawArrays(GL_LINE_STRIP, 0, figure[i].vertices.size()); // отрисовка ломаной
309             glBindVertexArray(0); // отключаем вершинный массив

```

```

310     }
311 }
312
313     glfwSwapBuffers(window); // поменять местами буферы изображения
314     glfwPollEvents(); // проверить, произошли ли какие-то события
315 }
316
317     glfwTerminate(); // завершить работу GLFW
318     return 0;
319 }

```

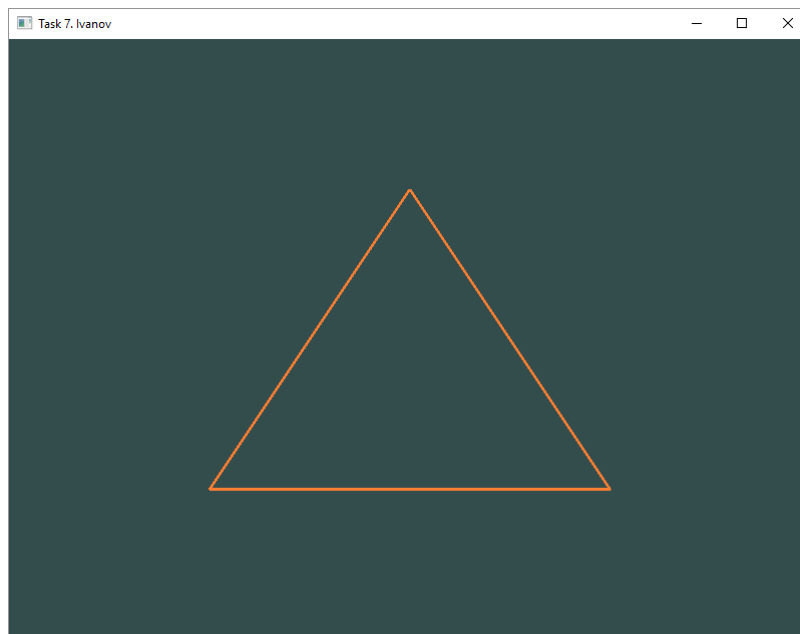
Файл `triangle.txt` Вы можете найти в приложении к заданию 7 или набрать самостоятельно. Его содержимое — описание одного треугольника с теми же координатами вершин, что и во фрагменте, который мы закоментировали.

```

camera 0 0 10 0 0 0 0 1 0
screen 90 1 3 200
model 0 0 0 2 2 2
color 155 200 0
thickness 3
path 4
-0.5 -0.5 0.0
0.5 -0.5 0.0
0.0 0.5 0.0
-0.5 -0.5 0.0
figure

```

Сохраните файл `triangle.txt` в ту же папку, в которой находится `Main.cpp` и запустите проект. В окне должно появиться изображение треугольника.



### 1.3.5 Передача дополнительных параметров в шейдерную программу

То изображение которое мы в настоящий момент строим — неполноценное. Матричные преобразования игнорируются: мы не используем матрицу  $T_M$ , в которой накоплены все преобразования — от модельного до перехода в пространство отсечения. Кроме этого, цвет линий мы не устанавливали — он извлекается из шейдера.

Шейдерам мы передаем только лишь вершинные данные. Конечно, можно было бы с каждой вершиной включить в вершинные данные её цвет. Но это было бы слишком накладно по использованию памяти.

Также является нецелесообразным пересчет всех координат в соответствии со значениями матрицы ТМ перед отправкой вершинных данных в вершинный массив, так как в этом случае, во-первых, мы оставляем на CPU нагрузку по перемножению матриц с координатными векторами точек, во-вторых, после каждого преобразования (например, после нажатия клавиши) нам бы пришлось копировать обновленную информацию о вершинах в GPU, а такое копирование — относительно длительный процесс.

Выходом из такой ситуации является использование, так называемых, uniform-переменных — глобальных переменных шейдеров. Uniform-переменные определяются для шейдерной программы и к ним возможен доступ из любого шейдера. Значение uniform-переменной загружается в GPU с помощью семейства процедур `glUniform`.

Так, например, отрезки наших ломаных имеют цвет, и имеет смысл этот цвет передавать фрагментному шейдеру для установки цвета соответствующих пикселей. Сделаем это с помощью определения uniform-переменной, представляющей трехмерный вектор. Внесем изменения в фрагментный шейдер. В естественном виде (представленный не в виде строки, а в виде текста) фрагментный шейдер сейчас имеет вид.

```
1 #version 330 core
2 out vec4 color;
3 void main() {
4     color = vec4(1.0f, 0.5f, 0.2f, 1.0f);
5 }
```

Перед функцией `main` шейдера объявим uniform-переменную `pathColor` типа `vec3`.

```
uniform vec3 pathColor
```

Теперь в процедуре `main` сделаем так, чтобы на выходе она выдавала тот цвет, который найдет в переменной `pathColor`. Для этого придется выполнить преобразование трехмерного вектора к четырехмерному с добавлением четвертой координаты, равной единице.

```
color = vec4(pathColor, 1.0f);
```

Наш шейдер примет вид.

```
1 #version 330 core
2 out vec4 color;
3 uniform vec3 pathColor
4 void main() {
5     color = vec4(pathColor, 1.0f);
6 }
```

В теле функции `main` заменим определение текста фрагментного шейдера на обновленное.

```
//=====
//          ФРАГМЕНТНЫЙ ШЕЙДЕР
//=====
const char *fragmentShaderSource =
    "#version 330 core\n"
```



```

"out vec4 color;\n"
"uniform vec3 pathColor;\n"
"void main() {\n"
"    color = vec4(pathColor, 1.0f);\n"
"}\n\n0";
//=====

```

Имя переменной `pathColor` относится к программе на GLSL, а не к нашей программе на C++. Пересылка значения такой переменной осуществляется по её местоположению в шейдерной программе. Расположение `uniform`-переменной в шейдерной программе можно узнать после компоновки программы: его вернет вызов функции `glGetUniformLocation`.

В строки программы после удаления шейдерных объектов (строки 277-279 приведенного выше кода) добавим запрос расположения переменной `pathColor` в программе `shaderProgram`.

```

// запрашиваем у программы shaderProgram расположение переменной pathColor
GLint pathColorLocation = glGetUniformLocation(shaderProgram, "pathColor");

```

Теперь вернемся в цикл отрисовки. Нам нужно менять значение цвета для каждой ломаной линии. Внутри цикла `for` по `i` (сразу после начала цикла) с помощью `glUniform` загрузим информацию о цвете линии в память GPU по адресу, находящемуся теперь в переменной `pathColorLocation`.

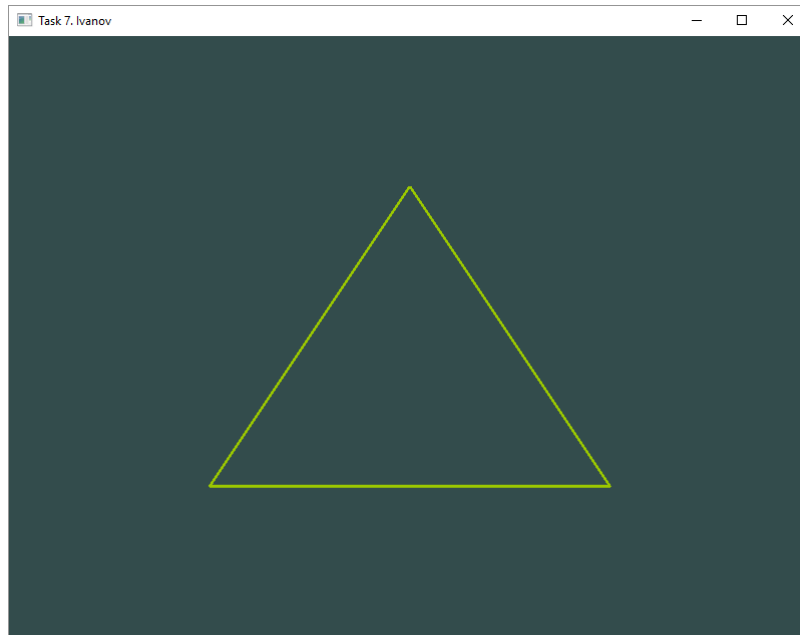
```

// пересылка цвета линии в переменную pathColor шейдерной программы
glUniform3fv(pathColorLocation, 1, glm::value_ptr(fgure[i].color));

```

Здесь мы используем процедуру `glUniform3fv`. Суффикс `3fv` имени процедуры означает, что мы пересылаем трехмерный вектор с вещественными значениями координат. Первый аргумент указывает место, в которое нужно переслать значение. Второй аргумент задает количество векторов, которые мы пересылаем (эта же процедура используется для пересылки массивов векторов). Третий аргумент задает значение для пересылки. Но тип `glm::vec3` по структуре не эквивалентен типу `vec3` языка GLSL, поэтому здесь используется дополнительное преобразование, заданное функцией `glm::value_ptr`.

Если запустить программу, то треугольник в окне поменяет свой цвет.



Похожим образом поступим с исполнением матричных преобразований. Чтобы к каждой точке вершинного массива было применено преобразование, заданное матрицей  $TM$ , передадим эту матрицу в вершинный шейдер.

Сейчас вершинный шейдер имеет следующий вид.

```
1 #version 330 core
2 layout (location = 0) in vec3 position;
3 void main() {
4     gl_Position = vec4(position.x, position.y, position.z, 1.0);
5 }
```

Объявим перед функцией `main` шейдера uniform-переменную `clipView` типа `mat4`.

```
uniform mat4 clipView;
```

Изменим процесс получения значения `gl_Position`: домножим слева полученный вектор `position`, переведенный в однородные координаты, на матрицу `clipView`.

```
gl_Position = clipView * vec4(position, 1.0);
```

Получим шейдер в общем виде.

```
1 #version 330 core
2 layout (location = 0) in vec3 position;
3 uniform mat4 clipView;
4 void main() {
5     gl_Position = clipView * vec4(position, 1.0);
6 }
```

В теле функции `main` заменим определение текста вершинного шейдера на обновленное.

```
//=====
//                                ВЕРШИННЫЙ ШЕЙДЕР
```

```
//=====
const char *vertexShaderSource =
    "#version 330 core\n"
    "layout (location = 0) in vec3 position;\n"
    "uniform mat4 clipView;\n"
    "void main() {\n"
    "    gl_Position = clipView * vec4(position, 1.0);\n"
    "}\n0";
//=====
```

После запроса расположения uniform-переменной `pathColor` определим расположение переменной `clipView`

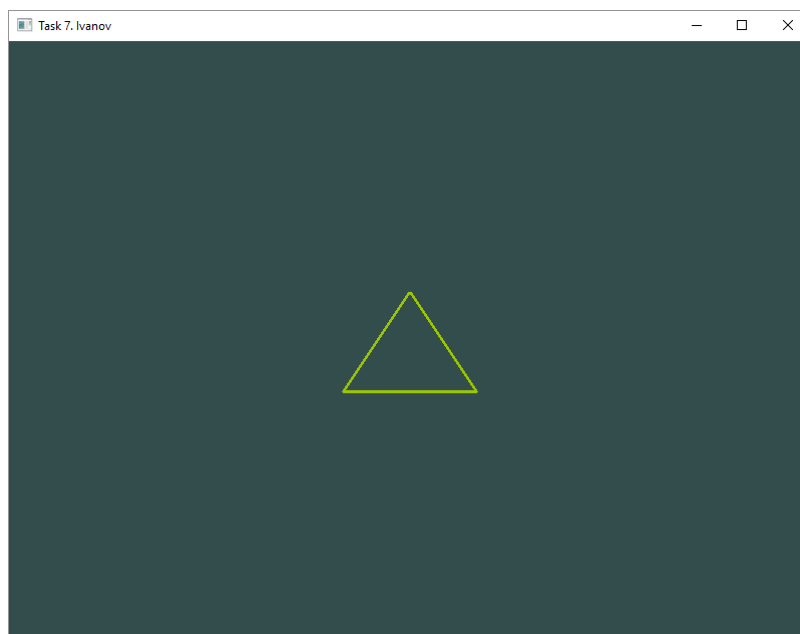
```
// запрашиваем у программы shaderProgram расположение переменной clipView
GLint clipViewLocation = glGetUniformLocation(shaderProgram, "clipView");
```

Теперь, в цикле отрисовки, сразу после вычисления значения матрицы `TM` перешлем ее значение на месторасположение переменной `clipView`.

```
// пересылка матрицы TM в переменную clipView шейдерной программы
glUniformMatrix4fv(clipViewLocation, 1, GL_FALSE, glm::value_ptr(TM));
```

Здесь суффиксом имени процедуры выступает `Matrix4fv`, что соответствует пересылке квадратной матрицы четвертого порядка с вещественными значениями. Третий аргумент вызова процедуры указывает, нужно ли транспонировать матрицу после пересылки. Остальные аргументы (первый, второй и последний) имеют тот же смысл, что и у процедуры `glUniform3fv`.

Если запустить программу, то треугольник на экране изменит свои размеры.



Общий вид процедуры `main` нашего проекта (без закомментированных частей) окажется следующим.

```

1  int main () {
2      glfwInit(); // Инициализация GLFW
3      // Проведение начальных установок GLFW
4      // Задаются минимальная требуемая версия OpenGL.
5      glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3); // Номер до десятичной точки
6      glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3); // Номер после десятичной точки
7      // Используем только средства указанной версии без совместимости с более ранними
8      glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
9
10     // Создаем окно
11     GLFWwindow* window = glfwCreateWindow(800, 600, "Task 7. Ivanov", NULL, NULL);
12     if (window == NULL) { // если ссылка на окно не создана
13         std::cout << "Вызов glfwCreateWindow закончился неудачей." << std::endl;
14         glfwTerminate(); // завершить работу GLFW
15         return -1;      // завершить программу
16     }
17     glfwMakeContextCurrent(window); // делаем окно window активным (текущим)
18     // Назначение обработчика события Resize
19     glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
20     // Назначение обработчика нажатия клавиш
21     glfwSetKeyCallback(window, key_callback);
22
23     // Инициализация GLAD
24     if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
25         std::cout << "Не удалось загрузить GLAD" << std::endl;
26         glfwTerminate(); // завершить работу GLFW
27         return -1;      // завершить программу
28     }
29
30     // сообщаем диапазон координат в окне
31     // (0, 0) - координаты левого нижнего угла, 800x600 - размеры окна в пикселах
32     glViewport(0, 0, 800, 600);
33
34     //=====
35     //          ВЕРШИННЫЙ ШЕЙДЕР
36     //=====
37     const char *vertexShaderSource =
38         "#version 330 core\n"
39         "layout (location = 0) in vec3 position;\n"
40         "uniform mat4 clipView;\n"
41         "void main() {\n"
42         "    gl_Position = clipView * vec4(position, 1.0);\n"
43         "}\n0";
44     //=====
45
46     GLuint vertexShader; // шейдерный объект - вершинный шейдер
47     vertexShader = glCreateShader(GL_VERTEX_SHADER); // создаем объект
48     // привязываем исходный код к шейдерному объекту
49     glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
50     glCompileShader(vertexShader); // компилируем шейдер
51     GLint success; // результат компиляции
52     // запрашиваем статус компиляции шейдера в переменную success
53     glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
54     if (!success) { // если компиляция прошла с ошибкой
55         GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
56         glGetShaderInfoLog(vertexShader, 512, NULL, infoLog); // запрашиваем сообщение
57         // выводим сообщение об ошибке на экран
58         std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
59     }
60
61     //=====
62     //          ФРАГМЕНТНЫЙ ШЕЙДЕР
63     //=====
64     const char *fragmentShaderSource =
65         "#version 330 core\n"
66         "out vec4 color;\n"
67         "uniform vec3 pathColor;\n"
68         "void main() {\n"
69         "    color = vec4(pathColor, 1.0f);\n"
70         "    //\" color = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
71         "}\n0";
72     //=====
73
74     GLuint fragmentShader; // шейдерный объект - фрагментный шейдер

```

```

75 fragmentShader = glCreateShader(GL_FRAGMENT_SHADER); // создаем объект
76 // привязываем исходный код к шейдерному объекту
77 glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
78 glCompileShader(fragmentShader); // компилируем шейдер
79 // запрашиваем статус компиляции шейдера в описанную ранее переменную success
80 glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
81 if (!success) { // если компиляция прошла с ошибкой
82     GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
83     glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog); // запрашиваем сообщение
84     // выводим сообщение об ошибке на экран
85     std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog << std::endl;
86 }
87
88 // Шейдерная программа
89 GLuint shaderProgram; // идентификатор шейдерной программы
90 shaderProgram = glCreateProgram(); // создаем программный объект
91 glAttachShader(shaderProgram, vertexShader); // присоединяем вершинный шейдер
92 glAttachShader(shaderProgram, fragmentShader); // и фрагментный шейдер
93 glLinkProgram(shaderProgram); // компоновка программы
94 // запрашиваем статус компоновки шейдерной программы в переменную success
95 glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
96 if (!success) { // если компоновка прошла с ошибкой
97     GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
98     glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog); // запрашиваем сообщение
99     // выводим сообщение об ошибке на экран
100     std::cout << "ERROR::SHADER::PROGRAM::LINK_FAILED\n" << infoLog << std::endl;
101 }
102
103 // удаление шейдерных объектов
104 glDeleteShader(vertexShader);
105 glDeleteShader(fragmentShader);
106
107 // запрашиваем у программы shaderProgram расположение переменной pathColor
108 GLint pathColorLocation = glGetUniformLocation(shaderProgram, "pathColor");
109 // запрашиваем у программы shaderProgram расположение переменной clipView
110 GLint clipViewLocation = glGetUniformLocation(shaderProgram, "clipView");
111
112 readFromFile("triangle.txt");
113
114 while (!glfwWindowShouldClose(window)) { // пока окно window не должно закрыться
115     glClearColor(0.2f, 0.3f, 0.3f, 1.0f); // назначаем цвет заливки
116     glClear(GL_COLOR_BUFFER_BIT); // очищаем буфер заданным цветом
117
118     glUseProgram(shaderProgram); // шейдерную программу shaderProgram делаем активной
119
120     glm::mat4 proj; // матрица перехода в пространство отсечения
121     switch (pType) {
122     case Ortho: // прямоугольная проекция
123         proj = glm::ortho(l, r, b, t, n, f);
124         break;
125     case Frustum: // перспективная проекция с Frustum
126         proj = glm::frustum(l, r, b, t, n, f);
127         break;
128     case Perspective: // перспективная проекция с Perspective
129         proj = glm::perspective(fovy_work, aspect_work, n, f);
130         break;
131     }
132     glm::mat4 C = proj * T; // матрица перехода от мировых координат в пространство отсечения
133     for (int k = 0; k < models.size(); k++) { // цикл по моделям
134         std::vector<path> figure = models[k].figure; // список ломаных очередной модели
135         glm::mat4 TM = C * models[k].modelM; // матрица общего преобразования модели
136         // пересылка матрицы TM в переменную clipView шейдерной программы
137         glUniformMatrix4fv(clipViewLocation, 1, GL_FALSE, glm::value_ptr(TM));
138         for (int i = 0; i < figure.size(); i++) {
139             // пересылка цвета линии в переменную pathColor шейдерной программы
140             glUniform3fv(pathColorLocation, 1, glm::value_ptr(figure[i].color));
141             glBindVertexArray(figure[i].vertexArray); // делаем активным вершинный массив i-й ломаной
142             glLineWidth(figure[i].thickness); // устанавливаем толщину линии
143             glDrawArrays(GL_LINE_STRIP, 0, figure[i].vertices.size()); // отрисовка ломаной
144             glBindVertexArray(0); // отключаем вершинный массив
145         }
146     }
147
148     glfwSwapBuffers(window); // поменять местами буферы изображения

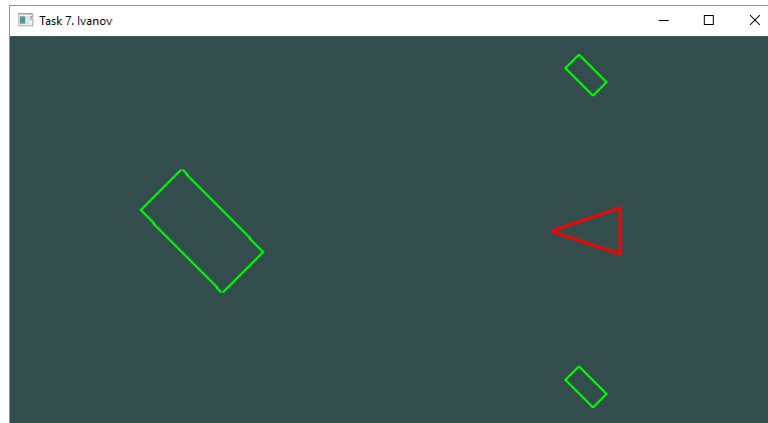
```

```

149     glfwPollEvents(); // проверить, произошли ли какие-то события
150 }
151
152 glfwTerminate(); // завершить работу GLFW
153 return 0;
154 }

```

Если в качестве входного файла взять файл `Geometric.txt`, то после его загрузки получим следующее изображение (окно растянуто в соответствии с пропорциями изображения).



## 1.4 Назначение горячих клавиш

Теперь, когда матричные преобразования учитываются, мы можем дополнить приложение, сделать так, чтобы изображение в окне реагировало на нажатие клавиш.

Установим ту же раскладку реакций на нажатие клавиш, что мы назначали ранее в проекте шестого задания. Но, в том проекте реакция на нажатие клавиши **Escape** — вернуться к исходным параметрам изображения. Сейчас же реакция на эту клавишу — выход из приложения. Но закрыть приложение мы можем стандартными средствами: нажатием **Alt-F4** или нажатием на кнопку с крестиком в правом верхнем углу окна приложения. Поэтому, вместо уже назначенной реакции установим ту, что имела место в шестом задании.

Опять позаимствуем программный код из предыдущего проекта: возьмем фрагмент обработчика события `KeyDown` — все варианты `case` оператора `switch`. Здесь приведен фрагмент, который был реализован в ходе изложения шестого урока.

```

1 case Keys::Escape:
2     initWorkPars();
3     break;
4 case Keys::W:
5     T = lookAt(vec3(0, 0, -1), vec3(0, 0, -2), vec3(0, 1, 0)) * T;
6     break;
7 case Keys::S:
8     T = lookAt(vec3(0, 0, 1), vec3(0, 0, 0), vec3(0, 1, 0)) * T;
9     break;
10 case Keys::A:
11     T = lookAt(vec3(-1, 0, 0), vec3(-1, 0, -1), vec3(0, 1, 0)) * T;
12     break;
13 case Keys::R: {
14     vec3 u_new = mat3(rotate(0.1, vec3(0, 0, 1))) * vec3(0, 1, 0);
15     T = lookAt(vec3(0, 0, 0), vec3(0, 0, -1), u_new) * T;
16     break;
17 }
18 case Keys::T: {
19     if (Control::ModifierKeys == Keys::Shift) {

```

```

20 // матрица вращения относительно точки P
21 mat4 M = rotateP(0.1, vec3(1, 0, 0), vec3(0, 0, -dist));
22 vec3 u_new = mat3(M) * vec3(0, 1, 0); // вращение направления вверх
23 vec3 S_new = normalize(M * vec4(0, 0, 0, 1)); // вращение начала координат
24 // переход к СКН в которой начало координат в новой точке, а направление
25 // наблюдения - в точку P
26 T = lookAt(S_new, vec3(0, 0, -dist), u_new) * T;
27 }
28 else {
29   mat4 M = rotate(0.1, vec3(1, 0, 0)); // матрица вращения относительно 0x
30   vec3 u_new = mat3(M) * vec3(0, 1, 0); // вращение направления вверх
31   // вращение точки, в которую смотрит наблюдатель
32   vec3 P_new = normalize(M * vec4(0, 0, -1, 1));
33   T = lookAt(vec3(0, 0, 0), P_new, u_new) * T;
34 }
35 break;
36 }
37 case Keys::I:
38   if (Control::ModifierKeys == Keys::Shift) {
39     t -= 1;
40   }
41   else {
42     t += 1;
43   }
44   break;
45 case Keys::J:
46   if (Control::ModifierKeys == Keys::Shift) {
47     l += 1;
48   }
49   else {
50     l -= 1;
51   }
52   break;
53 case Keys::D1:
54   pType = Ortho;
55   break;
56 case Keys::D3:
57   pType = Perspective;
58   break;

```

Адаптация этого кода заключается прежде всего в замене констант, соответствующих клавишам: вместо `Keys::Escape` — `GLFW_KEY_ESCAPE`, вместо `Keys::W` — `GLFW_KEY_W`, вместо `Keys::D1` — `GLFW_KEY_1` и т.д. Вместо сравнения

`Control::ModifierKeys == Keys::Shift`

следует установить сравнение `mode == GLFW_MOD_SHIFT`. Дальнейшие изменения касаются, в основном, замены вызовов наших функций на соответствующие функции библиотеки GLM, за исключением следующего момента.

В библиотеке GLM отсутствует аналог нашей функции `rotateP`. Поэтому позаимствуем реализацию этой функции из файла `Transform.h` и поместим перед процедурой `key_callback`. Скорректируем в ней типы данных и имена функций. Получим следующую реализацию функции.

```

// матрица вращения относительно оси, заданной вектором n,
// проходящей через точку P
glm::mat4 rotateP(float theta, glm::vec3 n, glm::vec3 P) {
  return glm::translate(P) * glm::rotate(theta, n) * glm::translate(-P);
}

```



Не забудьте о двух моментах:

1. функция `glm::rotate` в нашем случае должна принимать первый аргумент типа `float`, т.е. вместо `0.1` следует писать `0.1f`;
2. вместо функции `normalize` (переход из однородных координат), следует использовать `glm::vec3` (отбрасывание четвертой координаты).

После изменений процедура `key_callback` примет вид.

```

1 // Обработчик нажатия клавиш
2 void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode) {
3     if (action != GLFW_RELEASE) { // если клавиша нажата
4         switch (key) { // анализируем обрабатываемую клавишу
5             case GLFW_KEY_ESCAPE: // если клавиша - Escape
6                 initWorkPars();
7                 break;
8             case GLFW_KEY_W:
9                 T = glm::lookAt(glm::vec3(0, 0, -1), glm::vec3(0, 0, -2), glm::vec3(0, 1, 0)) * T;
10                break;
11             case GLFW_KEY_S:
12                 T = glm::lookAt(glm::vec3(0, 0, 1), glm::vec3(0, 0, 0), glm::vec3(0, 1, 0)) * T;
13                break;
14             case GLFW_KEY_A:
15                 T = glm::lookAt(glm::vec3(-1, 0, 0), glm::vec3(-1, 0, -1), glm::vec3(0, 1, 0)) * T;
16                break;
17             case GLFW_KEY_R: {
18                 glm::vec3 u_new = glm::mat3(glm::rotate(0.1f, glm::vec3(0, 0, 1))) * glm::vec3(0, 1, 0);
19                 T = glm::lookAt(glm::vec3(0, 0, 0), glm::vec3(0, 0, -1), u_new) * T;
20                } break;
21             }
22             case GLFW_KEY_T: {
23                 if (mode == GLFW_MOD_SHIFT) {
24                     // матрица вращения относительно точки P
25                     glm::mat4 M = rotateP(0.1, glm::vec3(1, 0, 0), glm::vec3(0, 0, -dist));
26                     glm::vec3 u_new = glm::mat3(M) * glm::vec3(0, 1, 0); // вращение направления вверх
27                     glm::vec3 S_new = glm::vec3(M * glm::vec4(0, 0, 0, 1)); // вращение начала координат
28                     // переход к СКН в которой начало координат в новой точке, а направление
29                     // наблюдения - в точку P
30                     T = glm::lookAt(S_new, glm::vec3(0, 0, -dist), u_new) * T;
31                 }
32                 else {
33                     glm::mat4 M = glm::rotate(0.1f, glm::vec3(1, 0, 0)); // матрица вращения относительно 0x
34                     glm::vec3 u_new = glm::mat3(M) * glm::vec3(0, 1, 0); // вращение направления вверх
35                     // вращение точки, в которую смотрит наблюдатель
36                     glm::vec3 P_new = glm::vec3(M * glm::vec4(0, 0, -1, 1));
37                     T = glm::lookAt(glm::vec3(0, 0, 0), P_new, u_new) * T;
38                 }
39                 break;
40             }
41             case GLFW_KEY_I:
42                 if (mode == GLFW_MOD_SHIFT) {
43                     t -= 1;
44                 }
45                 else {
46                     t += 1;
47                 }
48                 break;
49             case GLFW_KEY_J:
50                 if (mode == GLFW_MOD_SHIFT) {
51                     l += 1;
52                 }
53                 else {
54                     l -= 1;
55                 }
56                 break;
57             case GLFW_KEY_1:
58                 pType = Ortho;

```



```

59     break;
60     case GLFW_KEY_3:
61         pType = Perspective;
62         break;
63     default:
64         break;
65 }
66 }
67 }

```

## 1.5 Добавление файлового диалога

В этом разделе добавим диалог открытия файла в качестве реакции на нажатие клавиши **F3**. Для этого, перед подключением файла `glad.h` подключим заголовочный файл `Windows.h`.

```
#include <Windows.h>
```



Если `Windows.h` подключить позже `glad.h`, то система не найдет необходимых нам элементов. Вместо `Windows.h` можно подключить `Commdlg.h`, но такое подключение должно быть после `glad.h`: если подключить `Commdlg.h` перед `glad.h`, то возникнет конфликт.

Теперь добавим соответствующую ветвь в операторе `switch`.

```

case GLFW_KEY_F3: {
}

```

В блок из фигурных скобок добавим описание и вызов файлового диалога.

```

OPENFILENAME openFileDialog; // диалог открытия файла
char fileName[260]; // буфер для имени файла
// Инициализация файлового диалога
ZeroMemory(&openFileDialog, sizeof(openFileDialog));
openFileDialog.lStructSize = sizeof(openFileDialog);
openFileDialog.hwndOwner = NULL;
openFileDialog.lpstrFile = fileName;
openFileDialog.lpstrFile[0] = '\0';
openFileDialog.nMaxFile = sizeof(fileName);
openFileDialog.lpstrFilter = "Text files (*.txt)\0*.txt\0All files 2007\0*.*\0";
openFileDialog.nFilterIndex = 1;
openFileDialog.lpstrFileTitle = NULL;
openFileDialog.nMaxFileTitle = 0;
openFileDialog.lpstrInitialDir = NULL;
openFileDialog.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST | OFN_HIDEREADONLY;
// Вывод файлового диалога и получение результата
if (GetOpenFileName(&openFileDialog)) {
    // если файл выбран успешно
}
break;

```

Здесь `fileName` — символьный массив, в котором, в случае выбора файла и нажатия кнопки *ОК*, будет сохранено имя файла для чтения. Поэтому в тело условного оператора добавим вызов нашей процедуры чтения файла.

```
readFromFile(fileName);
```



В задачи урока не входит изучение типа данных `OPENFILENAME` или каких-то других средств WinAPI. Поэтому, приведенный код рассмотрим как данность. Но настоятельно рекомендуем самостоятельно изучить упомянутые средства и подробно разобраться в приведенном фрагменте кода.

## 1.6 Манипуляции с мышью

Добавим в наше приложение реакции на операции с мышью. Будем отслеживать движения курсора в окне и в зависимости от его сдвига осуществлять поворот направления наблюдения.

Для этого мы определим процедуру, которую назначим обработчиком движения мыши.

Перед процедурой (это важно) `key_callback` добавим процедуру `cursorPos_callback`.

```
// обработчик положения курсора
void cursorPos_callback(GLFWwindow* window, double xpos, double ypos) {
}
```

Как мы видим, этот обработчик событий получает координаты текущего положения курсора (аргументы типа `double`).

Создадим две вспомогательные переменные, в которых будем сохранять значения координат курсора при предыдущем вызове. В блоке описания глобальных переменных добавим

```
double lastX, lastY; // последняя позиция курсора
```

Вернемся теперь к определению процедуры `cursorPos_callback`. Зная предыдущие и текущие значения координат курсора, мы знаем вектор сдвига курсора в окне:  $(xpos - lastX, ypos - lastY)$ . Это будет означать, что мы хотим повернуть направление наблюдения в эту же сторону. Такой поворот будет поворотом вокруг оси, перпендикулярной вектору сдвига курсора и лежащей в координатной плоскости  $z = 0$  системы координат наблюдателя — оси, заданной вектором  $(lastY - ypos, lastX - xpos, 0)$ .

Добавим соответствующий код в процедуру.

```
// вычисляем вектор, задающий ось вращения
glm::vec3 n = glm::vec3(lastY - ypos, lastX - xpos, 0);
```

Угол вращения возьмем пропорциональным длине сдвига. Так как вектор `n` той же длины, то мы используем длину этого вектора. Коэффициент, задающий пропорцию возьмем достаточно малым числом: `0.002f`. Можем определить матрицу вращения.

```
// создаем матрицу вращения
glm::mat4 M = glm::rotate(glm::length(n) * 0.002f, n);
```

Осталось осуществить вращение точки, на которую смотрит наблюдатель, и перейти к новой системе координат наблюдателя, в которой изменяется направление наблюдения.

```
// вращаем точку (0, 0, -1), на которую смотрит наблюдатель
glm::vec3 P = M * glm::vec4(0, 0, -1, 1);
// добавляем к преобразованиям переход к новой системе координат наблюдателя
T = glm::lookAt(glm::vec3(0), P, u) * T;
```

Наконец, сохраним текущие координаты курсора в глобальных переменных, для последующего использования.

```
lastX = xpos;
lastY = ypos;
```

Получится процедура следующего вида.

```
1 // обработчик положения курсора
2 void cursorPos_callback(GLFWwindow* window, double xpos, double ypos) {
3     // вычисляем вектор, задающий ось вращения
4     glm::vec3 n = glm::vec3(lastY - ypos, lastX - xpos, 0);
5     // создаем матрицу вращения
6     glm::mat4 M = glm::rotate(glm::length(n) * 0.002f, n);
7     // вращаем точку (0, 0, -1), на которую смотрит наблюдатель
8     glm::vec3 P = M * glm::vec4(0, 0, -1, 1);
9     // добавляем к преобразованиям переход к новой системе координат наблюдателя
10    T = glm::lookAt(glm::vec3(0), P, glm::vec3(0, 1, 0)) * T;
11    lastX = xpos;
12    lastY = ypos;
13 }
```

Теперь назначим эту процедуру на роль обработчика событий. В процедуре `main`, после вызова процедуры `glfwSetKeyCallback` добавим

```
// назначение обработчика положения курсора
glfwSetCursorPosCallback(window, cursorPos_callback);
```

Приложение можно запустить. Но оно теперь функционирует не так, как хотелось бы. Во-первых, изображение «улетает» при первом появлении курсора в окне. Во-вторых, сложно управлять положением изображения в окне, вследствие того, что когда курсор выдвигается за пределы окна, положение изображения перестает меняться, а при возврате курсора в окно изображение «прыгает». Исправим эту ситуацию.

Укажем системе, чтобы она вычисляла позицию курсора даже в том случае, когда он находится вне окна. Это возможно когда курсор выключен. Добавим после вызова `glfwSetCursorPosCallback` установку такой опции для курсора.

```
// отключение курсора
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

Здесь значение `GLFW_CURSOR_DISABLED` для параметра `GLFW_CURSOR` как раз означает, что мы собираемся отключить курсор.

После запуска обнаружим, что при первом передвижении мыши изображение все равно «улетает», но теперь стало проще управлять его положением. Но теперь добавилось такое неудобство, что курсор, пропав, уже не появляется и закрыть мышью окно приложения не получится (поможет только **Alt-F4**).

Добавим реакцию на нажатие клавиши **F5**, на которую назначим выключение и включение курсора. Вместе с курсором будем включать и выключать обработчик его положения. В операторе `switch` процедуры `key_callback` добавим соответствующую ветвь `case`.

```
case GLFW_KEY_F5:
    break;
```

В теле этой ветви добавим условную конструкцию, в которой сравнивается значение параметра `GLFW_CURSOR` (мы можем его извлечь с помощью `glfwGetInputMode`) со значением `GLFW_CURSOR_DISABLED`.

```
if (glfwGetInputMode(window, GLFW_CURSOR) == GLFW_CURSOR_DISABLED) {
} else {
}
```

В первой ветви `if` (равенство имеет место) включим курсор и отключим обработчик его положения.

```
// включается курсор
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
// отключается обработчик положения курсора
glfwSetCursorPosCallback(window, NULL);
```

Передача `NULL` в качестве второго параметра `glfwSetCursorPosCallback` означает, что мы отключаем обработчик события.

Во второй ветви `if` отключим курсор и включим обработчик положения курсора.

```
// отключается курсор
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
// включается обработчик положения курсора
glfwSetCursorPosCallback(window, cursorPos_callback);
```

Теперь при запущенном приложении курсор включается и выключается при нажатии **F5**. Но мы не избавились от «прыжков» изображения в окне при выключении курсора. Это можно почти исправить, если мы будем отслеживать и запоминать координаты курсора в окне, при включенном курсоре. Для этого, сразу после процедуры `cursorPos_callback` определим еще один обработчик положения курсора `cursorPosSave_callback`, в котором будем просто сохранять позицию курсора в глобальных переменных.

```
// обработчик позиции курсора при включенном курсоре
void cursorPosSave_callback(GLFWwindow* window, double xpos, double ypos) {
    lastX = xpos;
    lastY = ypos;
}
```

Теперь, в описании реакции на клавишу **F5** заменим отключение обработчика положения курсора на назначение процедуры `cursorPosSave_callback` таким обработчиком. Соответствующий блок `case` в процедуре `key_callback` примет вид.

```

1 case GLFW_KEY_F5:
2     if (glfwGetInputMode(window, GLFW_CURSOR) == GLFW_CURSOR_DISABLED) {
3         glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
4         glfwSetCursorPosCallback(window, cursorPosSave_callback);
5     }
6     else {
7         glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
8         glfwSetCursorPosCallback(window, cursorPos_callback);
9     }
10    break;

```

Чтобы изображение не «улетало» при запуске приложения, назначим процедуру `cursorPosSave_callback` вместо `cursorPos_callback` обработчиком положения курсора в процедуре `main` и уберем из функции `main` отключение курсора.

Эффект «улетания» останется только для случая, когда курсор отключается при его нахождении вне окна. Но это уже не такое большое неудобство, что имело место ранее.

Обновленный файл `car.txt` приложен к заданию 7. При загрузке этого файла и отображении в перспективной проекции Вы можете протестировать блуждание по трехмерному пространству.

## 1.7 Задание для самостоятельной работы

### Задание 7

1. Создайте проект, включающий в себя все, что было описано выше в качестве примера. Проект должен называться Вашей фамилией, записанной латинскими буквами. Основное окно приложения должно называться *Task 7. Ivanov*, где фамилия *Ivanov* должна быть заменена на Вашу, записанную латинскими буквами.
2. Добавьте в приложение весь функционал горячих клавиш, имевший место в задании 6:
  - **2** — включение перспективной проекции с использованием матрицы `Frustum`;
  - **D** — смещение камеры строго вправо на одну единицу в системе координат наблюдателя с сохранением ракурса (с сохранением направления всех осей системы координат наблюдателя);
  - **Shift-W, Shift-S, Shift-A, Shift-D** — «медленное» смещение камеры: смещение, аналогичное имеющемуся, но не на единицу, а на 0.1;
  - **Y** — поворот камеры относительно оси  $Oz$  по часовой стрелке на угол 0.1 радиан;
  - **G/Shift-G** — разворот камеры на 0.1 радиан по часовой стрелке относительно оси, параллельной  $Ox$  и проходящей через начало координат/условную точку  $P$ ;
  - **F/Shift-F, H/Shift-H** — разворот камеры на 0.1 радиан против и по часовой стрелке относительно оси, параллельной  $Oy$  и проходящей через начало координат/условную точку  $P$ ;
  - **K/Shift-K, L/Shift-L** — изменение значений параметров `b` и `r` на единицу, приводящее к увеличению/уменьшению окна наблюдения;
  - **U/Shift-U** — увеличение уменьшение параметра `n` на 0.2, ограничив его снизу значением 0.1 и сверху значением (`f - 0.1`);
  - **O/Shift-O** — увеличение уменьшение параметра `f` на 0.2, ограничив его снизу значением (`n + 0.1`);
  - **B/Shift-B** — увеличение уменьшение параметра `dist` на 0.2, ограничив его

снизу значением 0.1;

- **Z/Shift-Z** — увеличение уменьшение параметра `fovy_work`, ограничив его значения диапазоном от 0.3 до 3 радиан;
- **X/Shift-X** — увеличение уменьшение параметра `aspect_work` на 0.05, ограничив его снизу значением 0.01;

3. Обработчик панелей прокрутки отслеживает, в том числе, колесо прокрутки мыши. Обработчик события *Scroll* имеет следующий вид

```
// обработчик панелей прокрутки
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset) {
}
```

Параметры `xoffset` и `yoffset` — значения по осям  $x$  и  $y$ , соответственно, на которые произошел сдвиг прокрутки.

Добавьте процедуру `scroll_callback` в проект. В этой процедуре меняйте значение `dist` на полученное значение сдвига по оси  $y$ . Назначьте процедуру обработчиком события с помощью `glfwSetScrollCallback`.

4. Обработчик события *MouseButton* имеет следующую форму.

```
// обработчик нажатия кнопки мыши
void mouse_button_callback(GLFWwindow* window, int button, int action, int mods) {
}
```

Данный обработчик событий срабатывает, когда нажимается или отжимается какая-то кнопка мыши. С помощью `glfwSetMouseButtonCallback` такой обработчик может быть назначен.

Требуется реализовать вращение изображения с помощью мыши вокруг условной точки  $P$  при включенном курсоре: при нажатой левой кнопке мыши передвижение курсора должно вращать изображение вокруг  $P$ . Для этого можно завести глобальную переменную, принимающую `true` в момент нажатия (`action == GLFW_PRESS`) левой кнопки мыши (`button == GLFW_MOUSE_BUTTON_LEFT`) и `false` — в момент её отжатия (`action == GLFW_RELEASE`). В зависимости от значения этой переменной, в процедуре `cursorPosSave_callback` следует добавлять или не добавлять необходимое вращение к общему преобразованию. Вращение следует организовать с помощью перехода к новой системе координат наблюдателя (подобно тому, как оно организовывалось в процедуре `cursorPos_callback` и в обработчике нажатия клавиш при осуществлении поворота вокруг  $P$ ).

5. В качестве результата выполнения задания должен быть загружен архив получившегося проекта со стандартным именем.

## 1.8 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1. Как в нашем проекте установить, чтобы при запуске программы соотношение сторон окна приложения было 1:2? Продемонстрируйте такое изменение проекта.
2. Если загрузить в приложение описание из файла `Geometric.txt` (полученного

- в шестом уроке) и осуществить поворот, назначенный на клавишу **Shift-F**, то пропадет половина изображения. Почему?
3. Если запустить приложение, загрузить в него описание из файла `car.txt` (из 6-го задания), включить прямоугольную проекцию и некоторое время осуществлять движение назад, назначенное на клавишу **S** (не выполняя других преобразований), машина будет в окне оставаться на месте, но она будет исчезать, начиная с задней части. Объясните, почему это происходит.
  4. Какое предназначение библиотеки GLFW?
  5. Укажите в своем проекте все используемые функции и процедуры библиотеки GLFW.
  6. Укажите место в процедуре `main`, где происходит обработка нажатия клавиш.
  7. Какое предназначение библиотеки GLM?
  8. Что произойдет, если у вызова процедуры `glViewport` первые два аргумента указать отличными от нуля?
  9. В итоговом проекте команда `glfwSwapBuffers` стоит в конце цикла `while`. Что произойдет, если поместить её в конец внешнего цикла `for` (по `k`)?
  10. Укажите в своем проекте все используемые функции и процедуры библиотеки GLM.
  11. Что такое вершинный массив, вершинный буфер? Для чего и в каком порядке создаются эти объекты?
  12. Почему шейдеры описываются в программе в виде строки?
  13. Что такое шейдерная программа?
  14. Покажите фрагменты программы, в которых вершинные данные передаются в память GPU.
  15. Что такое `uniform`-переменная? Как присвоить значение `uniform`-переменной?
  16. В проекте 6-го задания отрисовка трехмерной сцены организовывалась с помощью трех вложенных циклов `for`. В этом проекте осталось только два цикла `for`. Почему?





## 2. Освещение

### 2.1 Предмет разработки

За основу нового проекта возьмем проект, полученный при выполнении задания 7 и внесем в него некоторые изменения.

Для упрощения работы с кодом шейдеров мы сначала организуем модуль, который будет текст шейдера считывать из файла, а не из массива символов, как это происходило в предыдущем проекте.

В ходе этого урока мы будем изображать трехмерную сцену, составленную из треугольников. Поэтому, в очередной раз, мы изменим формат входного файла, с тем, чтобы модели задавались не набором ломаных, а набором треугольников. Поменяем код приложения для корректного отображения таких моделей.

Наконец, мы реализуем модель освещения, в которой цвет каждой точки модели будет зависеть от наличия источников света над её поверхностью.

### 2.2 Шейдеры

Описание шейдера в символьном массиве вносит неудобства на этапе его разработки. Намного удобнее редактировать код шейдера, представленный в отдельном файле, а в программе подгружать содержимое такого файла в символьный массив для дальнейшей компиляции и включения в шейдерную программу.

Добавим в проект новый заголовочный файл `Shader.h`. В нем опишем классы для шейдеров и шейдерных программ.

Шейдер будем создавать на основе содержимого текстового файла с кодом. Поэтому добавим в новый файл подключение библиотек для работы со строками, файловыми и строковыми потоками, библиотеку GLAD.

```
#include <iostream>
#include <fstream>
#include <sstream>
```



```
#include <string>
#include <glad\glad.h>
```

Теперь опишем класс шейдера.

```
class shader {
}
```

Сначала в этом классе опишем приватную функцию `readTextFile`, которой в качестве аргумента передается имя текстового файла, и результатом которой получается строка содержимого этого файла.

```
private:
    std::string readTextFile(const char* fileName) {
    }
```

Здесь, сначала опишем потоковую переменную.

```
std::ifstream shaderFile;
```

После чего организуем блок `try — catch` в котором организуем чтение из файла с помощью описанной потоковой переменной.

```
try {
}
catch (std::ifstream::failure e) {
}
```

В части `catch` выведем сообщение об ошибке чтения из файла.

```
std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ:" << fileName << std::endl;
```

В части `try` сначала откроем файл.

```
shaderFile.open(fileName); // файл открывается для чтения
```

Все, что есть в файле, выведем в строковый поток. Для этого опишем соответствующую переменную.

```
std::stringstream shaderStream; // строковый поток для вывода содержимого файла
```

В описанный поток выводим содержимое файла.

```
shaderStream << shaderFile.rdbuf(); // выводим все из файла в строковый поток
```

Закрываем файл и возвращаем в виде строки полученное содержимое строкового потока.

```
shaderFile.close(); // закрываем файл
return shaderStream.str(); // возвращаем строку из строкового потока
```

В результате получим функцию в следующем виде.

```

1 private:
2     std::string readTextFile(const char* fileName) {
3         std::ifstream shaderFile;
4         shaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
5         try {
6             shaderFile.open(fileName); // файл открывается для чтения
7             std::stringstream shaderStream; // строковый поток для вывода содержимого файла
8             shaderStream << shaderFile.rdbuf(); // выводим все из файла в строковый поток
9             shaderFile.close(); // закрываем файл
10            return shaderStream.str(); // возвращаем строку из строкового потока
11        }
12        catch (std::ifstream::failure e) {
13            std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ:" << fileName << std::endl;
14        }
15    }

```

Теперь в классе `shader` опишем `public`-поле `shaderID`

```

public:
    GLuint shaderID; // идентификатор шейдера

```

и конструктор, аргументом которого будет имя файла и тип шейдера, заданный тем же значением, что и аргумент функции `glCreateShader`.

```

shader(const char* fileName, GLenum shaderType) {
}

```

В теле конструктора сначала считаем содержимое файла с помощью `readTextFile`.

```

std::string fileStd = readTextFile(fileName); // содержимое файла в виде строки

```

Превратим полученную строку в массив символов.

```

const char* shaderSource = fileStd.c_str(); // содержимое файла в виде массива символов

```

Зная код шейдера, заданный в виде массива символов, и тип шейдера мы можем теперь повторить те действия, что мы выполняем в файле `Main.cpp` для создания вершинного шейдера. Этот фрагмент выглядит так.

```

1 vertexShader = glCreateShader(GL_VERTEX_SHADER); // создаем объект
2 // привязываем исходный код к шейдерному объекту
3 glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
4 glCompileShader(vertexShader); // компилируем шейдер
5 GLint success; // результат компиляции
6 // запрашиваем статус компиляции шейдера в переменную success
7 glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
8 if (!success) { // если компиляция прошла с ошибкой
9     GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
10    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog); // запрашиваем сообщение
11    // выводим сообщение об ошибке на экран
12    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
13 }

```

Включим этот фрагмент в наш конструктор, заменив вхождения `vertexShader` на `shaderID`, значение `GL_VERTEX_SHADER` — на имя параметра `shaderType`. Кроме того, вместо слова

VERTEX в команде вывода сообщения об ошибке следует выводить название типа шейдера, которое зависит от значения shaderType. Поэтому перед этим выводом определим тип шейдера в виде строки.

```
std::string strShaderType; // тип шейдера в строке
switch (shaderType) {
case GL_VERTEX_SHADER: strShaderType = "VERTEX"; break;
case GL_GEOMETRY_SHADER: strShaderType = "GEOMETRY"; break;
case GL_FRAGMENT_SHADER: strShaderType = "FRAGMENT"; break;
}
```

Теперь изменим команду вывода сообщения об ошибке, чтобы тип шейдера выводился корректно.

```
std::cout << "ERROR::SHADER::" << strShaderType << " ::COMPILATION_FAILED\n" << infoLog << std::endl;
```

Получим конструктор shader в следующем виде.

```
1 shader(const char* fileName, GLenum shaderType) {
2     std::string fileStd = readTextFile(fileName); // содержимое файла в виде строки
3     const char* shaderSource = fileStd.c_str(); // содержимое файла в виде массива символов
4     shaderID = glCreateShader(shaderType); // создаем объект
5     // привязываем исходный код к шейдерному объекту
6     glShaderSource(shaderID, 1, &shaderSource, NULL);
7     glCompileShader(shaderID); // компилируем шейдер
8     GLint success; // результат компиляции
9     // запрашиваем статус компиляции шейдера в переменную success
10    glGetShaderiv(shaderID, GL_COMPILE_STATUS, &success);
11    if (!success) { // если компиляция прошла с ошибкой
12        GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
13        glGetShaderInfoLog(shaderID, 512, NULL, infoLog); // запрашиваем сообщение
14        // выводим сообщение об ошибке на экран
15        std::string strShaderType; // тип шейдера в строке
16        switch (shaderType) {
17            case GL_VERTEX_SHADER: strShaderType = "VERTEX"; break;
18            case GL_GEOMETRY_SHADER: strShaderType = "GEOMETRY"; break;
19            case GL_FRAGMENT_SHADER: strShaderType = "FRAGMENT"; break;
20        }
21        std::cout << "ERROR::SHADER::" << strShaderType << " ::COMPILATION_FAILED\n" << infoLog << std::endl;
22    }
23 }
```

Еще опишем метод clear для очистки шейдера.

```
void clear() { // очистка шейдера
}
```

В нем добавим лишь команду удаления шейдерного объекта с заданным идентификатором. Получим.

```
1 void clear() { // очистка шейдера
2     // удаление шейдерного объекта
3     glDeleteShader(shaderID);
4 }
```

Опробуем полученный код на деле. В папке проекта (в той же папке, где лежит Main.cpp) создадим два файла: Vertex.glsl и Fragment.glsl. В эти файлы скопируйте текст, соответственно, вершинного и фрагментного шейдера. То есть, содержимое файла Vertex.glsl должно быть

```

1 #version 330 core
2 layout (location = 0) in vec3 position;
3 uniform mat4 clipView;
4 void main() {
5     gl_Position = clipView * vec4(position, 1.0);
6 }

```

Содержимое Fragment.glsl

```

1 #version 330 core
2 out vec4 color;
3 uniform vec3 pathColor;
4 void main() {
5     color = vec4(pathColor, 1.0f);
6 }

```

Теперь в файле Main.cpp в конце блока команд `#include` подключим только что созданный файл.

```
#include "Shader.h"
```

Закомментируем в процедуре `main` строки, относящиеся к определению и компиляции шейдеров (строки 34–86 листинга функции `main`, приведенного в конце раздела 1.3.5), за исключением строки с описанием переменной `success` (строка 51 того же листинга). Вместо них опишем два объекта класса `shader`, заданных именами соответствующих файлов.

```

shader vertexShader("Vertex.glsl", GL_VERTEX_SHADER); // вершинный шейдер
shader fragmentShader("Fragment.glsl", GL_FRAGMENT_SHADER); // фрагментный шейдер

```

Теперь, при присоединении шейдеров к шейдерной программе, следует на них ссылаться через упоминание поля `shaderID` соответствующих объектов. То есть в строках 91–92 листинга функции `main`, приведенного в конце раздела 1.3.5, следует заменить имена `vertexShader` и `fragmentShader` на `vertexShader.shaderID` и `fragmentShader.shaderID`.

Команды удаления шейдерных объектов (строки 104–105 того же листинга) следует заменить на вызов метода `clear`.

```

// удаление шейдерных объектов
vertexShader.clear();
fragmentShader.clear();

```

Проект должен компилироваться и функционировать без изменений.



В случае, когда программа ведет себя не так, как Вы ожидаете, обращайтесь внимание на вывод в окно консоли (второе окно, которое появляется при запуске приложения и обычно находится на заднем плане). Вывод сообщений об ошибках, как и любой вывод в консоль, производится в этом окне.

Вернемся к изменению `Shader.h`. Добавим класс `program` для шейдерной программы.

```
class program {
public:
}
```

Добавим поле `programID` типа `GLuint` и конструктор, принимающий в качестве аргументов два объекта класса `shader`

```
GLuint programID; // идентификатор шейдерной программы
program(shader vertexShader, shader fragmentShader) {
}
```

В тело конструктора скопируем фрагмент кода, отвечающий за компиляцию шейдерной программы в функции `main` файла `Main.cpp`.

```
1 shaderProgram = glCreateProgram(); // создаем программный объект
2 glAttachShader(shaderProgram, vertexShader.shaderID); // присоединяем вершинный шейдер
3 glAttachShader(shaderProgram, fragmentShader.shaderID); // и фрагментный шейдер
4 glLinkProgram(shaderProgram); // компоновка программы
5 // запрашиваем статус компоновки шейдерной программы в переменную success
6 glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
7 if (!success) { // если компоновка прошла с ошибкой
8     GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
9     glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog); // запрашиваем сообщение
10    // выводим сообщение об ошибке на экран
11    std::cout << "ERROR::SHADER::PROGRAM::LINK_FAILED\n" << infoLog << std::endl;
12 }
```

Здесь необходимо только заменить все вхождения имени `shaderProgram` на `programID`. Получим конструктор в следующем виде.

```
1 program(shader vertexShader, shader fragmentShader) {
2     programID = glCreateProgram(); // создаем программный объект
3     glAttachShader(programID, vertexShader.shaderID); // присоединяем вершинный шейдер
4     glAttachShader(programID, fragmentShader.shaderID); // и фрагментный шейдер
5     glLinkProgram(programID); // компоновка программы
6     GLint success; // результат компиляции
7     // запрашиваем статус компоновки шейдерной программы в переменную success
8     glGetProgramiv(programID, GL_LINK_STATUS, &success);
9     if (!success) { // если компоновка прошла с ошибкой
10        GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
11        glGetProgramInfoLog(programID, 512, NULL, infoLog); // запрашиваем сообщение
12        // выводим сообщение об ошибке на экран
13        std::cout << "ERROR::SHADER::PROGRAM::LINK_FAILED\n" << infoLog << std::endl;
14    }
15 }
```

Добавим метод `use` для активизации шейдерной программы. В этом методе используем только вызов `glUseProgram`.

```
void use() { // активизация шейдерной программы
    glUseProgram(programID);
}
```

Организуем в классе передачу значений `uniform`-переменным. Опишем метод `useUniform`, в котором будем регистрировать `uniform`-переменные, с которыми будем

работать в программе, а также семейство методов `setUniform`, с помощью которых будем загружать значения `uniform`-переменных. Обращаться к `uniform`-переменным будем по имени, используемому в шейдерах. Для сопоставления имени `uniform`-переменной и его расположения будем использовать ассоциативный массив, организуемый с помощью контейнера `map`.

В блоке директив `#include` добавим

```
#include <map>
#include <glm\glm.hpp>
#include <glm\gtc\type_ptr.hpp>
```

Опишем в классе `program` ассоциативный массив `uniforms`, сопоставляющий именам `uniform`-переменных их расположения.

```
std::map<std::string, GLint> uniforms; // uniform-переменные
```

Добавим метод `useUniform`, которому передается имя `uniform`-переменной.

```
void useUniform(std::string uniformName) { // регистрация переменной
}
```

В методе запросим расположение переменной в программе и запишем результат запроса в массив `uniforms`.

```
// имя представляем в виде массива символов, запрашиваем расположение
// записываем расположение в ассоциированный массив
uniforms[uniformName] = glGetUniformLocation(programID, uniformName.c_str());
```

Получим метод в общем виде.

```
1 void useUniform(std::string uniformName) { // регистрация переменной
2   // имя представляем в виде массива символов, запрашиваем расположение
3   // записываем расположение в ассоциированный массив
4   uniforms[uniformName] = glGetUniformLocation(programID, uniformName.c_str());
5 }
```

Теперь опишем методы `setUniform`. В нашем проекте мы передаем в шейдерную программу матрицы  $4 \times 4$  и трехмерные векторы. Опишем сначала метод `setUniform`, в котором `uniform`-переменной устанавливается значение трехмерного вектора.

```
void setUniform(std::string uniformName, glm::vec3 value) {
}
```

В тело метода поместим вызов процедуры `glUniform3fv`, первый аргумент для которой извлечем по имени переменной из ассоциативного массива `uniforms`.

```
glUniform3fv(uniforms[uniformName], 1, glm::value_ptr(value));
```

Получим метод в общем виде.

```
void setUniform(std::string uniformName, glm::vec3 value) {
    glUniform3fv(uniforms[uniformName], 1, glm::value_ptr(value));
}
```

По аналогии добавим метод, в котором в uniform-переменную будет загружаться матрица четвертого порядка.

```
void setUniform(std::string uniformName, glm::mat4 value) {
    glUniformMatrix4fv(uniforms[uniformName], 1, GL_FALSE, glm::value_ptr(value));
}
```

На данном этапе описание класса `program` примет следующий вид.

```
1 class program {
2 public:
3     GLuint programID; // идентификатор шейдерной программы
4     std::map<std::string, GLint> uniforms; // uniform-переменные
5
6     program(shader vertexShader, shader fragmentShader) {
7         programID = glCreateProgram(); // создаем программный объект
8         glAttachShader(programID, vertexShader.shaderID); // присоединяем вершинный шейдер
9         glAttachShader(programID, fragmentShader.shaderID); // и фрагментный шейдер
10        glLinkProgram(programID); // компоновка программы
11        GLint success; // результат компиляции
12        // запрашиваем статус компоновки шейдерной программы в переменную success
13        glGetProgramiv(programID, GL_LINK_STATUS, &success);
14        if (!success) { // если компоновка прошла с ошибкой
15            GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
16            glGetProgramInfoLog(programID, 512, NULL, infoLog); // запрашиваем сообщение
17            // выводим сообщение об ошибке на экран
18            std::cout << "ERROR::SHADER::PROGRAM::LINK_FAILED\n" << infoLog << std::endl;
19        }
20    }
21    void use() { // активизация шейдерной программы
22        glUseProgram(programID);
23    }
24    void useUniform(std::string uniformName) { // регистрация переменных
25        // имя представляем в виде массива символов, запрашиваем расположение
26        // записываем расположение в ассоциированный массив
27        uniforms[uniformName] = glGetUniformLocation(programID, uniformName.c_str());
28    }
29    void setUniform(std::string uniformName, glm::vec3 value) {
30        glUniform3fv(uniforms[uniformName], 1, glm::value_ptr(value));
31    }
32    void setUniform(std::string uniformName, glm::mat4 value) {
33        glUniformMatrix4fv(uniforms[uniformName], 1, GL_FALSE, glm::value_ptr(value));
34    }
35 };
```

Внедрим разработанный код в наше приложение. Перейдем к изменению функции `main` файла `Main.cpp`.

Закомментируем код, отвечающий за компоновку шейдерной программы:

```
1 // Шейдерная программа
2 GLuint shaderProgram; // идентификатор шейдерной программы
3 shaderProgram = glCreateProgram(); // создаем программный объект
4 glAttachShader(shaderProgram, vertexShader.shaderID); // присоединяем вершинный шейдер
5 glAttachShader(shaderProgram, fragmentShader.shaderID); // и фрагментный шейдер
6 glLinkProgram(shaderProgram); // компоновка программы
```

```

7 // запрашиваем статус компоновки шейдерной программы в переменную success
8 glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
9 if (!success) { // если компоновка прошла с ошибкой
10     GLchar infoLog[512]; // объявим контейнер для сообщения об ошибке
11     glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog); // запрашиваем сообщение
12     // выводим сообщение об ошибке на экран
13     std::cout << "ERROR::SHADER::PROGRAM::LINK_FAILED\n" << infoLog << std::endl;
14 }

```

Вместо него опишем объект класса `program`.

```

// Шейдерная программа
program shaderProgram(vertexShader, fragmentShader);

```

Закомментируем описание переменных `pathColorLocation` и `clipViewLocation`. Вместо них объявим, что в шейдерной программе мы используем соответствующие `uniform`-переменные.

```

// декларируем использование uniform-переменных
shaderProgram.useUniform("pathColor");
shaderProgram.useUniform("clipView");

```

В цикле `while` заменим команду активизации шейдерной программы на

```

shaderProgram.use(); // шейдерную программу shaderProgram делаем активной

```

Загрузку матрицы `TM` в `uniform`-переменную `clipView` с помощью `glUniformMatrix4fv` заменим на вызов `setUniform`.

```

shaderProgram.setUniform("clipView", TM);

```

По аналогии заменим вызов `glUniform3fv`.

```

shaderProgram.setUniform("pathColor", figure[i].color);

```

Проект можно запустить. Он должен работать без изменений.

Функция `main` проекта на данном этапе примет вид.

```

1 int main () {
2     glfwInit(); // Инициализация GLFW
3     // Проведение начальных установок GLFW
4     // Задаётся минимальная требуемая версия OpenGL.
5     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3); // Номер до десятичной точки
6     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3); // Номер после десятичной точки
7     // Используем только средства указанной версии без совместимости с более ранними
8     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
9
10    // Создаем окно
11    GLFWwindow* window = glfwCreateWindow(800, 600, "Task 8. Ivanov", NULL, NULL);
12    if (window == NULL) { // если ссылка на окно не создана
13        std::cout << "Вызов glfwCreateWindow закончился неудачей." << std::endl;
14        glfwTerminate(); // завершить работу GLFW
15        return -1; // завершить программу
16    }
17    glfwMakeContextCurrent(window); // делаем окно window активным (текущим)
18    // Назначение обработчика события Resize
19    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
20    // Назначение обработчика нажатия клавиш
21    glfwSetKeyCallback(window, key_callback);
22    // назначение обработчика положения курсора
23    glfwSetCursorPosCallback(window, cursorPosSave_callback);

```



```

24 glwSetScrollCallback(window, scroll_callback);
25 glwSetMouseButtonCallback(window, mouse_button_callback);
26
27 // Инициализация GLAD
28 if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
29     std::cout << "Не удалось загрузить GLAD" << std::endl;
30     glfwTerminate(); // завершить работу GLFW
31     return -1;      // завершить программу
32 }
33
34 // сообщаем диапазон координат в окне
35 // (0, 0) - координаты левого нижнего угла, 800x600 - размеры окна в пикселах
36 glViewport(0, 0, 800, 600);
37
38 shader vertexShader("Vertex.glsl", GL_VERTEX_SHADER); // вершинный шейдер
39 shader fragmentShader("Fragment.glsl", GL_FRAGMENT_SHADER); // фрагментный шейдер
40 // Шейдерная программа
41 program shaderProgram(vertexShader, fragmentShader);
42
43 // удаление шейдерных объектов
44 vertexShader.clear();
45 fragmentShader.clear();
46
47 // декларируем использование uniform-переменных
48 shaderProgram.useUniform("pathColor");
49 shaderProgram.useUniform("clipView");
50
51 readFromFile("triangle.txt");
52
53 while (!glfwWindowShouldClose(window)) { // пока окно window не должно закрыться
54     glClearColor(0.2f, 0.3f, 0.3f, 1.0f); // назначаем цвет заливки
55     glClear(GL_COLOR_BUFFER_BIT); // очищаем буфер заданным цветом
56
57     shaderProgram.use(); // шейдерную программу shaderProgram делаем активной
58
59     glm::mat4 proj; // матрица перехода в пространство отсечения
60     switch (pType) {
61     case Ortho: // прямоугольная проекция
62         proj = glm::ortho(l, r, b, t, n, f);
63         break;
64     case Frustum: // перспективная проекция с Frustum
65         proj = glm::frustum(l, r, b, t, n, f);
66         break;
67     case Perspective: // перспективная проекция с Perspective
68         proj = glm::perspective(fovy_work, aspect_work, n, f);
69         break;
70     }
71     glm::mat4 C = proj * T; // матрица перехода от мировых координат в пространство отсечения
72     for (int k = 0; k < models.size(); k++) { // цикл по моделям
73         std::vector<path> figure = models[k].figure; // список ломаных очередной модели
74         glm::mat4 TM = C * models[k].modelM; // матрица общего преобразования модели
75         // пересылка пересылка матрицы TM в переменную clipView шейдерной программы
76         shaderProgram.setUniform("clipView", TM);
77         for (int i = 0; i < figure.size(); i++) {
78             // пересылка цвета линии в переменную pathColor шейдерной программы
79             shaderProgram.setUniform("pathColor", figure[i].color);
80             glBindVertexArray(figure[i].vertexArray); // делаем активным вершинный массив i-й ломаной
81             glLineWidth(figure[i].thickness); // устанавливаем толщину линии
82             glDrawArrays(GL_LINE_STRIP, 0, figure[i].vertices.size()); // отрисовка ломаной
83             glBindVertexArray(0); // отключаем вершинный массив
84         }
85     }
86
87     glfwSwapBuffers(window); // поменять местами буферы изображения
88     glfwPollEvents(); // проверить, произошли ли какие-то события
89 }
90
91 glfwTerminate(); // завершить работу GLFW
92 return 0;
93 }

```

## 2.3 Изменение формата представления объектов сцены

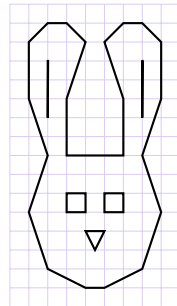
### 2.3.1 Формат входного файла

В этом уроке модели, составляющие изображение, будут представлены наборами треугольников. В предыдущем уроке мы рассматривали возможность изображать набор треугольников с помощью OpenGL. Но здесь будем придерживаться иной стратегии.

Если произвольный многоугольник представлять набором треугольников, неизбежно окажется, что в какие-то из вершин исходного многоугольника будут являться вершинами сразу нескольких треугольников. Таким образом, если каждый треугольник представлять тремя тройками координат точек, это приведет к повторению одной и той же тройки вещественных чисел среди вершинных данных.

Более приемлемый вариант — использовать отдельно набор точек грани (или нескольких граней, в которых вершины имеют одинаковые свойства), а треугольники составлять из номеров точек в этом наборе.

Рассмотрим, например, часть изображения зайца, представленного в заданиях 2 и 3.



Эта часть изображения была описана в виде трехмерной ломаной

```
# голова
path 23
0.5 3. 0. # от левой щеки вверх до уха
1. 4.5 0. # левое ухо слева снизу вверх
0.5 6. 0. # левое ухо слева
0.5 7.5 0. # левое ухо верх слева
1. 8. 0. # левое ухо верх середина
1.5 8. 0. # левое ухо верх справа
2. 7.5 0. # левое ухо справа сверху вниз
1.5 6. 0. # левое ухо справа до макушки
1.5 4.5 0. # макушка
3. 4.5 0. # правое ухо слева снизу вверх
3. 6. 0. # правое ухо слева
2.5 7.5 0. # правое ухо верх слева
3. 8. 0. # правое ухо верх середина
3.5 8. 0. # правое ухо верх справа
4. 7.5 0. # правое ухо сверху вниз
4. 6. 0. # правое ухо справа
3.5 4.5 0. # от правого уха вниз до щеки
4. 3. 0. # правая скула
3.5 1.5 0. # подбородок справа
2.5 1. 0. # подбородок снизу
2. 1. 0. # подбородок слева
1. 1.5 0. # левая скула
0.5 3. 0.

# глаза
# левый глаз
```

```
path 5
1.5 3.5 0. # левый глаз слева сверху вниз
1.5 3. 0. # левый глаз низ
2. 3. 0. # левый глаз справа
2. 3.5 0. # левый глаз верх
1.5 3.5 0.
```

```
# правый глаз
```

```
path 5
2.5 3.5 0. # правый глаз слева
2.5 3. 0. # правый глаз снизу
3. 3. 0. # правый глаз справа
3. 3.5 0. # правый глаз сверху
2.5 3.5 0.
```

```
# ушные раковины
```

```
# левая ушная раковина
```

```
path 2
1. 5.5 0.
1. 7. 0.
```

```
# правая ушная раковина
```

```
path 2
3.5 5.5 0.
3.5 7. 0.
```

```
# нос
```

```
path 4
2. 2.5 0. # нос сверху
2.5 2.5 0. # нос справа
2.25 2. 0. # нос слева
2. 2.5 0.
```

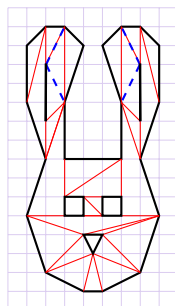
Внесем поправки в эту модель: ушные раковины, представляющие отрезки, представим в виде треугольников очень маленькой ширины.

```
# ушные раковины
# левая ушная раковина
path 2
1. 5.5 0.
1. 7. 0.
1.01 7. 0.
1. 5.5 0.

# правая ушная раковина
path 2
3.5 5.5 0.
3.5 7. 0.
3.49 7. 0.
3.5 5.5 0.
```

На изображении это практически не отразится, но зато теперь ушные раковины можно представить самостоятельными треугольниками.

Добавим в изображение дополнительные линии, чтобы морда и уши зайца были разбиты на треугольники.



Синим пунктиром на рисунке начерчены линии, исходящие из новых вершин ушных раковин.



Разбиение многоугольника на треугольники обычно не является единственным.

Перенумеруем точки, участвующие в формировании фрагмента, в порядке их упоминания (номера точек здесь укажем в комментарии).

```
# голова
0.5 3. 0. #0 левая щека
1. 4.5 0. #1 левое ухо слева снизу
0.5 6. 0. #2 левое ухо слева середина
0.5 7.5 0. #3 левое ухо верх слева
1. 8. 0. #4 левое ухо верх середина слева
1.5 8. 0. #5 левое ухо верх середина справа
2. 7.5 0. #6 левое ухо крайняя правая
1.5 6. 0. #7 левое ухо середина справа
1.5 4.5 0. #8 макушка слева
3. 4.5 0. #9 макушка справа
3. 6. 0. #10 правое ухо середина слева
2.5 7.5 0. #11 правое ухо крайняя левая
3. 8. 0. #12 правое ухо верх середина слева
3.5 8. 0. #13 правое ухо верх середина справа
4. 7.5 0. #14 правое ухо сверху справа
4. 6. 0. #15 правое ухо середина справа
3.5 4.5 0. #16 правое ухо низ справа
4. 3. 0. #17 правая щека
3.5 1.5 0. #18 правая скула
2.5 1. 0. #19 подбородок справа
2. 1. 0. #20 подбородок слева
1. 1.5 0. #21 левая скула
# глаза
# левый глаз
1.5 3.5 0. #22 левый глаз слева верх
1.5 3. 0. #23 левый глаз слева низ
2. 3. 0. #24 левый глаз справа низ
2. 3.5 0. #25 левый глаз справа верх
# правый глаз
2.5 3.5 0. #26 правый глаз слева верх
2.5 3. 0. #27 правый глаз слева низ
3. 3. 0. #28 правый глаз справа низ
3. 3.5 0. #29 правый глаз справа верх
# ушные раковины
# левая ушная раковина
1. 5.5 0. #30 нижняя точка
```

```

1. 7. 0. #31 верхняя левая
1.01 7. 0. #32 верхняя правая (дополнительная)
# правая ушная раковина
3.5 5.5 0. #33 нижняя точка
3.5 7. 0. #34 верхняя правая
3.49 7. 0. #35 верхняя левая (дополнительная)
# нос
2. 2.5 0. #36 сверху слева
2.5 2.5 0. #37 сверху справа
2.25 2. 0. #38 низ

```

Получилось 39 точек (пронумерованных от нуля).

Теперь перечислим все треугольники, составляющие морду и уши зайца, тройками номеров точек вершин.

```

0 7 23 # левый большой треугольник от щеки до глаза
1 30 7 # нижняя часть левого уха
1 2 4 # большой треугольник левого уха
2 3 4 # левая вертушка левого уха
4 5 31 # вертушка левого уха
5 31 32 # тонкий треугольник сверху
5 32 7 # с двумя пунктирными ребрами
5 6 7 # правая вертушка левого уха
32 7 30 # середина левого уха
8 9 22 # лоб сверху
9 22 29 # лоб справа
24 25 27 # между глаз левый
25 27 26 # между глаз правый
0 17 36 # большой под глазами
36 37 17 # от носа до правой скулы
0 36 21 # левая щека
36 38 21 # слева от носа
38 20 21 # под носом слева
38 19 20 # самый нижний
38 18 19 # под носом вправо
37 18 38 # справа от носа
37 17 18 # правая щека
17 28 10 # большой треугольник справа
10 33 16 # нижняя часть правого уха
16 13 15 # большой треугольник правого уха
13 14 15 # правая вертушка правого уха
12 13 34 # вертушка правого уха
12 34 35 # тонкий треугольник сверху
12 35 10 # правый треугольник с синим пунктиром
10 11 12 # левая вертушка правого уха
10 35 33 # средняя часть правого уха

```

Преимущество такого представления — вместо упоминания трех вещественных чисел на точку нам достаточно указать один целочисленный номер в списке исходных точек.

Возможность такого представления предоставляется OpenGL. Для этого с вершинным массивом нужно связать два буфера: вершинный буфер, представляющий набор атрибутов точек (как и раньше), плюс буфер индексов элементов вершинного буфера.

Но сначала создадим пример входного файла для описания трехмерной сцены в новом формате. Пусть вместо команды `path` во входном файле будет команда

```
mesh N K
```

где  $N$  и  $K$  — целые числа:  $N$  — число точек для вершинного буфера,  $K$  — количество треугольников в наборе. Будем считать, что после этой команды во входном файле следуют  $N$  строк с координатами точек, после которых идут  $K$  строк с тройками целых чисел, представляющих треугольники.

Таким образом, определим входной файл `hare.txt`.

```
camera 0 0 10 0 0 0 0 1 0
screen 90 1.333333 2 2000 # соотношение сторон 800x600
model 2.25 4.5 0 4.5 8 2
color 128 128 128 # серый цвет

mesh 39 31
# голова
0.5 3. 0. #0 левая щека
1. 4.5 0. #1 левое ухо слева снизу
0.5 6. 0. #2 левое ухо слева середина
0.5 7.5 0. #3 левое ухо верх слева
1. 8. 0. #4 левое ухо верх середина слева
1.5 8. 0. #5 левое ухо верх середина справа
2. 7.5 0. #6 левое ухо крайняя правая
1.5 6. 0. #7 левое ухо середина справа
1.5 4.5 0. #8 макушка слева
3. 4.5 0. #9 макушка справа
3. 6. 0. #10 правое ухо середина слева
2.5 7.5 0. #11 правое ухо крайняя левая
3. 8. 0. #12 правое ухо верх середина слева
3.5 8. 0. #13 правое ухо верх середина справа
4. 7.5 0. #14 правое ухо сверху справа
4. 6. 0. #15 правое ухо середина справа
3.5 4.5 0. #16 правое ухо низ справа
4. 3. 0. #17 правая щека
3.5 1.5 0. #18 правая скула
2.5 1. 0. #19 подбородок справа
2. 1. 0. #20 подбородок слева
1. 1.5 0. #21 левая скула
# глаза
# левый глаз
1.5 3.5 0. #22 левый глаз слева верх
1.5 3. 0. #23 левый глаз слева низ
2. 3. 0. #24 левый глаз справа низ
2. 3.5 0. #25 левый глаз справа верх
# правый глаз
2.5 3.5 0. #26 правый глаз слева верх
2.5 3. 0. #27 правый глаз слева низ
3. 3. 0. #28 правый глаз справа низ
3. 3.5 0. #29 правый глаз справа верх
# ушные раковины
# левая ушная раковина
1. 5.5 0. #30 нижняя точка
1. 7. 0. #31 верхняя левая
1.01 7. 0. #32 верхняя правая (дополнительная)
# правая ушная раковина
3.5 5.5 0. #33 нижняя точка
3.5 7. 0. #34 верхняя правая
3.49 7. 0. #35 верхняя левая (дополнительная)
# нос
2. 2.5 0. #36 сверху слева
2.5 2.5 0. #37 сверху справа
2.25 2. 0. #38 низ
# треугольники
```

```

0 7 23 # левый большой треугольник от щеки до глаза
1 30 7 # нижняя часть левого уха
1 2 4 # большой треугольник левого уха
2 3 4 # левая вертушка левого уха
4 5 31 # вертушка левого уха
5 31 32 # тонкий треугольник сверху
5 32 7 # с двумя пунктирными ребрами
5 6 7 # правая вертушка левого уха
32 7 30 # середина левого уха
8 9 22 # лоб сверху
9 22 29 # лоб справа
24 25 27 # между глаз левый
25 27 26 # между глаз правый
0 17 36 # большой под глазами
36 37 17 # от носа до правой скулы
0 36 21 # левая щека
36 38 21 # слева от носа
38 20 21 # под носом слева
38 19 20 # самый нижний
38 18 19 # под носом вправо
37 18 38 # справа от носа
37 17 18 # правая щека
17 28 10 # большой треугольник справа
10 33 16 # нижняя часть правого уха
16 13 15 # большой треугольник правого уха
13 14 15 # правая вертушка правого уха
12 13 34 # вертушка правого уха
12 34 35 # тонкий треугольник сверху
12 35 10 # правый треугольник с синим пунктиром
10 11 12 # левая вертушка правого уха
10 35 33 # средняя часть правого уха

figure

```

По аналогии переопределим входной файл `triangle.txt`, который используется при загрузке приложения.

```

camera 0 0 10 0 0 0 0 1 0
screen 90 1 3 200 # соотношение сторон 1x1
model 0 0 0 2 2 2
color 155 200 0
mesh 3 1 # три точки - один треугольник
-0.5 -0.5 0.0
0.5 -0.5 0.0
0.0 0.5 0.0
0 1 2
figure

```

Внесем изменения в файл `Geometric.txt`, полученный в ходе выполнения задания 4. Изменения состоят только в замене блока команды `path` на блок команды `mesh`.

```

# установка камеры в точку (10,5,5) направленной в точку (10,5,0)
# с направлением вверх (0,1,0)
camera 10 5 5 10 5 0 0 1 0
# установка окна с углом обзора 90 градусов
# с соотношением сторон 2:1 на расстоянии 5 от наблюдателя (от камеры)
# расстояние до горизонта - 20
screen 90 2 5 20
# первый рисунок

```

```

model 1.5 1 0 3 2 1 # центр в точке (1.5, 1), размеры 3x2
color 0 255 0 # цвет зеленый
thickness 3 # толщина линии 3
mesh 4 2 # набор из 4-х точек для двух треугольников
0.5 0.5 0 # левый нижний угол
0.5 1.5 0 # левый верхний угол
2.5 1.5 0 # правый верхний угол
2.5 0.5 0 # правый нижний угол
0 1 2 # индексы точек первого треугольника
2 3 0 # индексы точек второго треугольника
# преобразования и размещения по описанию
pushTransform # сохранить отправную точку
rotate -45 0 0 1 # поворот на -45 градусов
pushTransform # сохранить преобразование поворота
scale 2.25 # масштабирование до большого прямоугольника
translate 5 5 0 # перенос центра рисунка в точку (5,5)
figure # запомнить положение и ракурс первого рисунка
popTransform # откатились к преобразованию поворота
scale 0.75 # масштабирование до малого прямоугольника
translate 15 1 0 # установить в позицию нижнего малого прямоугольника
figure # запомнить положение и ракурс второго экземпляра рисунка
translate 0 8 0 # передвинуться в позицию (15,9) из (15,1)
figure # запомнить положение и ракурс третьего экземпляра рисунка
popTransform # откатились к стартовой позиции
# второй рисунок
model 1 1.25 0 2 2.5 1 # параметры рисунка с треугольником
color 255 0 0 # цвет красный
mesh 3 1 # три точки для одного треугольника
0.5 0.5 0 # нижний левый угол
1 2 0 0 # верхний угол
1.5 0.5 0 # нижний правый угол
0 1 2 # порядок точек в треугольнике
# преобразования и размещения по описанию
rotate 90 0 0 1 # поворот на 90 градусов
scale 1.5 # масштабирование до синего прямоугольника
translate 15 5 0 # сдвиг в нужную позицию
figure # запомнить положение и ракурс рисунка

```

Создадим файл `colorCube.txt`, описывающий куб с разноцветными гранями. Так как грани имеют разный цвет, то придется каждую грань описывать отдельно, несмотря на то, что в них используются одни и те же вершины. Каждую грань представим двумя треугольниками.

```

camera 3 3 10 0 0 0 0 1 0
screen 90 1.333333 1 2000 # соотношение сторон 800x600
model 0 0 0 2 2 2

# нижняя грань красная
color 255 0 0
mesh 4 2
-0.5 -0.5 -0.5
-0.5 -0.5 0.5
0.5 -0.5 0.5
0.5 -0.5 -0.5
0 1 2 # первый треугольник нижней грани
0 2 3 # второй треугольник нижней грани

# верхняя грань синяя
color 0 0 255

```



```

mesh 4 2
-0.5 0.5 -0.5
-0.5 0.5 0.5
0.5 0.5 0.5
0.5 0.5 -0.5
0 1 3 # первый треугольник верхней грани
1 2 3 # второй треугольник верхней грани

# левая грань зеленая
color 0 255 0
mesh 4 2
-0.5 -0.5 -0.5
-0.5 -0.5 0.5
-0.5 0.5 -0.5
-0.5 0.5 0.5
0 1 2 # первый треугольник левой грани
1 2 3 # второй треугольник левой грани

# правая грань желтая
color 255 255 0
mesh 4 2
0.5 -0.5 0.5
0.5 -0.5 -0.5
0.5 0.5 0.5
0.5 0.5 -0.5
2 0 1 # первый треугольник правой грани
2 1 3 # второй треугольник правой грани

# передняя грань фиолетовая
color 255 0 255
mesh 4 2
-0.5 -0.5 0.5
0.5 -0.5 0.5
-0.5 0.5 0.5
0.5 0.5 0.5
0 2 3 # первый треугольник передней грани
0 3 1 # второй треугольник передней грани

# задняя грань морской волны
color 0 255 255
mesh 4 2
-0.5 -0.5 -0.5
0.5 -0.5 -0.5
-0.5 0.5 -0.5
0.5 0.5 -0.5
1 3 2 # первый треугольник задней грани
1 2 0 # второй треугольник задней грани

figure

```

Создадим копию получившегося файла, которую назовем `cube.txt`. Изменим в описании модели цвет граней. Объявим, что все грани оранжевого цвета. Для этого удалим все команды `color` и добавим перед описанием модели одну, устанавливающую оранжевый цвет.

```
color 255 140 0
```

Получим файл со следующим содержимым.

```
camera 3 3 10 0 0 0 0 1 0
screen 90 1.333333 1 2000 # соотношение сторон 800x600

# оранжевый куб
color 255 140 0
model 0 0 0 2 2 2

# нижняя грань
mesh 4 2
-0.5 -0.5 -0.5
-0.5 -0.5 0.5
0.5 -0.5 0.5
0.5 -0.5 -0.5
0 1 2 # первый треугольник нижней грани
0 2 3 # второй треугольник нижней грани

# верхняя грань
mesh 4 2
-0.5 0.5 -0.5
-0.5 0.5 0.5
0.5 0.5 0.5
0.5 0.5 -0.5
0 1 3 # первый треугольник верхней грани
1 2 3 # второй треугольник верхней грани

# левая грань
mesh 4 2
-0.5 -0.5 -0.5
-0.5 -0.5 0.5
-0.5 0.5 -0.5
-0.5 0.5 0.5
0 1 2 # первый треугольник левой грани
1 2 3 # второй треугольник левой грани

# правая грань
mesh 4 2
0.5 -0.5 0.5
0.5 -0.5 -0.5
0.5 0.5 0.5
0.5 0.5 -0.5
2 0 1 # первый треугольник правой грани
2 1 3 # второй треугольник правой грани

# передняя грань
mesh 4 2
-0.5 -0.5 0.5
0.5 -0.5 0.5
-0.5 0.5 0.5
0.5 0.5 0.5
0 2 3 # первый треугольник передней грани
0 3 1 # второй треугольник передней грани

# задняя грань
mesh 4 2
-0.5 -0.5 -0.5
0.5 -0.5 -0.5
-0.5 0.5 -0.5
0.5 0.5 -0.5
1 3 2 # первый треугольник задней грани
1 2 0 # второй треугольник задней грани
```

figure



Можно было бы одноцветный куб описать в виде одного меша. Но куб в той форме, что получился на текущий момент, нам понадобится для последующих изменений.

### 2.3.2 Структуры данных

**Класс** mesh

Теперь переопределим структуры данных, представляющие трехмерную сцену. Перейдем к изменению файла `Figure.h`.

Добавим в него описание нового класса `mesh`, представляющий набор точек и их использующие треугольники<sup>1</sup>. За основу возьмем описание класса `path`. Сделаем его копию.

```

1 class path {
2 public:
3     std::vector<glm::vec3> vertices; // последовательность точек
4     glm::vec3 color; // цвет, разбитый на составляющие RGB
5     float thickness; // толщина линии
6     GLuint vertexArray; // вершинный массив (объект OpenGL)
7     path(std::vector<glm::vec3> verts, glm::vec3 col, float thickn) {
8         vertices = verts;
9         color = col;
10        thickness = thickn;
11        setupPath();
12    }
13 private:
14     GLuint vertexBuffer; // вершинный буфер (объект OpenGL)
15     void setupPath() {
16         // создаем вершинный массив и вершинный буфер
17         glGenVertexArrays(1, &vertexArray); // создаем вершинный массив
18         glGenBuffers(1, &vertexBuffer); // создаем вершинный буфер
19         glBindVertexArray(vertexArray); // делаем вершинный массив активным
20         // связываем vertexBuffer с GL_ARRAY_BUFFER
21         glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
22         // копируем содержимое vertices в вершинный буфер vertexBuffer
23         glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec3), &vertices[0], GL_STATIC_DRAW);
24         // описание расположения параметра вершинного шейдера в вершинном буфере
25         glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(glm::vec3), (GLvoid*)0);
26         glEnableVertexAttribArray(0); // включение параметра 0 для шейдера
27         glBindBuffer(GL_ARRAY_BUFFER, 0); // отвязка буферного объекта
28         glBindVertexArray(0); // отключение вершинного массива
29     }
30 };

```

В копии переименуем класс и конструктор в `mesh`, а процедуру `setupPath` и ее вызов в конструкторе переименуем в `setupMesh`. Добавим дополнительное публичное поле `indices` — список (вектор) индексов элементов в вершин, составляющих треугольники.

```
std::vector<GLuint> indices; // последовательность индексов в наборе точек
```

Так как изображение будет состоять из треугольников, а не из линий, то поле `thickness` становится ненужным. Удалим его.

В конструкторе удалим последний аргумент, отвечающий за толщину линии и строку кода `thickness = thickn;`. Но добавим дополнительным аргументом аналог поля `indices`.

<sup>1</sup>Термин «Меш» используется для множества точек с набором заданных на них многоугольников, определяющих трехмерную поверхность

```
mesh(std::vector<glm::vec3> verts, std::vector<GLuint> inds, glm::vec3 col) {
```

В теле конструктора добавим соответствующее присваивание.

```
indices = inds;
```

В общем виде конструктор примет вид.

```
1 mesh(std::vector<glm::vec3> verts, std::vector<GLuint> inds, glm::vec3 col) {
2     vertices = verts;
3     indices = inds;
4     color = col;
5     setupMesh();
6 }
```

Добавим новое приватное поле `elementBuffer` : дополнительный буферный объект, в который загрузим содержимое списка `indices` .

```
GLuint elementBuffer; // буфер индексов вершин (объект OpenGL)
```

В процедуре `setupMesh` после инициализации объекта `vertexBuffer` , создадим второй буфер.

```
glGenBuffers(1, &elementBuffer); // создаем буфер индексов
```

После того, как будет загружена информация в вершинный буфер (после вызова `glEnableVertexAttribArray` ) свяжем буфер индексов с контекстом `GL_ELEMENT_ARRAY_BUFFER` .

```
// связываем elementBuffer с GL_ELEMENT_ARRAY_BUFFER
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementBuffer);
```

Теперь через этот контекст загрузим в этот буфер информацию из списка в поле `indices` также, как раньше загружали вершинную информацию.

```
// копируем содержимое indices в буфер индексов elementBuffer
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLuint), &indices[0], GL_STATIC_DRAW);
```

Можно отвязать `elementBuffer` от `GL_ELEMENT_ARRAY_BUFFER` , но только после того, как вершинный массив станет неактивным. В противном случае буфер индексов будет отвязан и от вершинного массива. Мы проводить отвязку не будем.

Класс `mesh` вместе с методом `setupMesh` на текущем этапе примет вид.

```
1 class mesh {
2 public:
3     std::vector<glm::vec3> vertices; // последовательность точек
4     std::vector<GLuint> indices; // последовательность индексов в наборе точек
5     glm::vec3 color; // цвет, разбитый на составляющие RGB
6     GLuint vertexArray; // вершинный массив (объект OpenGL)
7     mesh(std::vector<glm::vec3> verts, std::vector<GLuint> inds, glm::vec3 col) {
8         vertices = verts;
9         indices = inds;
10        color = col;
11        setupMesh();
12    }
13
14 private:
```

```

15 GLuint vertexBuffer; // вершинный буфер (объект OpenGL)
16 GLuint elementBuffer; // буфер индексов вершин (объект OpenGL)
17 void setupMesh() {
18     // создаем вершинный массив и вершинный буфер
19     glGenVertexArrays(1, &vertexArray); // создаем вершинный массив
20     glGenBuffers(1, &vertexBuffer); // создаем вершинный буфер
21     glGenBuffers(1, &elementBuffer); // создаем буфер индексов
22
23     glBindVertexArray(vertexArray); // делаем вершинный массив активным
24
25     // связываем vertexBuffer с GL_ARRAY_BUFFER
26     glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
27     // копируем содержимое vertices в вершинный буфер vertexBuffer
28     glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec3), &vertices[0], GL_STATIC_DRAW);
29     // описание расположения параметра вершинного шейдера в вершинном буфере
30     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(glm::vec3), (GLvoid*)0);
31     glEnableVertexAttribArray(0); // включение параметра 0 для шейдера
32
33     // связываем elementBuffer с GL_ELEMENT_ARRAY_BUFFER
34     glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementBuffer);
35     // копируем содержимое indices в буфер индексов elementBuffer
36     glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLuint), &indices[0], GL_STATIC_DRAW);
37
38     glBindBuffer(GL_ARRAY_BUFFER, 0); // отвязка буферного объекта
39     glBindVertexArray(0); // отключение вершинного массива
40 }
41 };

```

### Класс model

Внесем изменения в класс `model`. Все что нужно сделать здесь — заменить упоминания класса `path` на `mesh`. Класс `model` в общем виде примет вид.

```

1 class model {
2 public:
3     std::vector<mesh> figure; // составляющие рисунка
4     glm::mat4 modelM; // модельная матрица
5     model(std::vector<mesh> fig, glm::mat4 mat) {
6         figure = fig;
7         modelM = mat;
8     }
9 };

```

### 2.3.3 Чтение файла

Теперь внесем изменения в файл `Main.cpp`. Модифицируем процедуру `readFromFile` так, чтобы она была пригодной для чтения файла в новом формате.

Сначала изменим описание переменной `figure` — списка, в котором накапливались считанные ломаные.

```
std::vector<path> figure; // список ломаных очередного рисунка
```

Теперь наши модели будут состоять не из ломаных, а из мешей. Поэтому изменим описание на следующее.

```
std::vector<mesh> figure; // список мешей очередной модели
```

Дальнейшие изменения коснутся блока, соответствующего команде `path`.

```

1 else if (cmd == "path") { // набор точек
2     std::vector<glm::vec3> vertices; // список точек ломаной
3     int N; // количество точек
4     s >> N;

```

```

5  std::string str1; // дополнительная строка для чтения из файла
6  while (N > 0) { // пока не все точки считали
7      std::getline(in, str1); // считываем в str1 из входного файла очередную строку
8          // так как файл корректный, то на конец файла проверять не нужно
9      if ((str1.find_first_not_of(" \t\r\n") != std::string::npos) && (str1[0] != '#')) {
10         // прочитанная строка не пуста и не комментарий
11         // значит в ней тройка координат
12         float x, y, z; // переменные для считывания
13         std::stringstream s1(str1); // еще один строковый поток из строки str1
14         s1 >> x >> y >> z;
15         vertices.push_back(glm::vec3(x, y, z)); // добавляем точку в список
16         N--; // уменьшаем счетчик после успешного считывания точки
17     }
18 }
19 // все точки считаны, генерируем ломаную (path) и кладем ее в список figure
20 figure.push_back(path(vertices, glm::vec3(r, g, b) / 255.f, thickness));
21 }

```

Изменим имя команды на **"mesh"**. Так как у команды `mesh` не один, а два аргумента, опишем дополнительную переменную `K` и произведем её считывание.

```

else if (cmd == "mesh") { // набор мешей
    std::vector<glm::vec3> vertices; // список точек
    int N, K; // количество точек и треугольников
    s >> N >> K;
}

```

Последующий цикл наполнения списка `vertices` нас вполне устраивает. Оставим его без изменений. После этого цикла необходимо организовать считывание информации о `K` треугольниках. Опишем список `indices`, в который будем сохранять считанную информацию.

```

std::vector<GLuint> indices; // список индексов вершин треугольников

```

Теперь организуем цикл для считывания, подобный предыдущему, но с изменением счетчика `K`.

```

while (K > 0) { // пока не считали все треугольники
    std::getline(in, str1); // считываем в str1 из входного файла очередную строку
        // так как файл корректный, то на конец файла проверять не нужно
    if ((str1.find_first_not_of(" \t\r\n") != std::string::npos) && (str1[0] != '#')) {
        // прочитанная строка не пуста и не комментарий
        // значит в ней тройка индексов вершин треугольника

        K--; // уменьшаем счетчик после успешного считывания точки
    }
}

```

В отличие от предыдущего цикла, три индекса вершин треугольника будут составлять три отдельных элемента списка `indices`. Поэтому для считывания тройки индексов воспользуемся одной вспомогательной переменной и циклом, трижды выполняющим чтение значения и его запись в список `indices`.

```

GLuint x; // переменная для считывания
std::stringstream s1(str1); // еще один строковый поток из строки str1
for (int i = 0; i < 3; i++) { // три раза
    s1 >> x; // считываем индекс
    indices.push_back(x); // добавляем индекс в список indices
}

```

Наконец, в команде добавления элемента в список `figure` следует заменить вызов конструктора `path` на `mesh` и соответствующим образом исправить набор его параметров.

```
// все точки и индексы считаны, генерируем меш и кладем его в список figure
figure.push_back(mesh(vertices, indices, glm::vec3(r, g, b) / 255.f));
```

Фрагмент, соответствующий команде `mesh` на текущем этапе примет вид.

```
1 else if (cmd == "mesh") { // набор мешей
2     std::vector<glm::vec3> vertices; // список точек
3     int N, K; // количество точек и треугольников
4     s >> N >> K;
5     std::string str1; // дополнительная строка для чтения из файла
6     while (N > 0) { // пока не все точки считали
7         std::getline(in, str1); // считываем в str1 из входного файла очередную строку
8         // так как файл корректный, то на конец файла проверять не нужно
9         if ((str1.find_first_not_of(" \t\r\n") != std::string::npos) && (str1[0] != '#')) {
10            // прочитанная строка не пуста и не комментарий
11            // значит в ней тройка координат
12            float x, y, z; // переменные для считывания
13            std::stringstream s1(str1); // еще один строковый поток из строки str1
14            s1 >> x >> y >> z;
15            vertices.push_back(glm::vec3(x, y, z)); // добавляем точку в список
16            N--; // уменьшаем счетчик после успешного считывания точки
17        }
18    }
19    std::vector<GLuint> indices; // список индексов вершин треугольников
20    while (K > 0) { // пока не считали все треугольники
21        std::getline(in, str1); // считываем в str1 из входного файла очередную строку
22        // так как файл корректный, то на конец файла проверять не нужно
23        if ((str1.find_first_not_of(" \t\r\n") != std::string::npos) && (str1[0] != '#')) {
24            // прочитанная строка не пуста и не комментарий
25            // значит в ней тройка индексов вершин треугольника
26            GLuint x; // переменная для считывания
27            std::stringstream s1(str1); // еще один строковый поток из строки str1
28            for (int i = 0; i < 3; i++) { // три раза
29                s1 >> x; // считываем индекс
30                indices.push_back(x); // добавляем индекс в список indices
31            }
32            K--; // уменьшаем счетчик после успешного считывания точки
33        }
34    }
35    // все точки и индексы считаны, генерируем меш и кладем его в список figure
36    figure.push_back(mesh(vertices, indices, glm::vec3(r, g, b) / 255.f));
37 }
```

### 2.3.4 Отрисовка

Перейдем к изменению цикла отрисовки в процедуре `main`.

Первым делом, изменим тип используемой здесь переменной `figure`, заменив тип элементов списка `path` на `mesh`.

Строка кода, определяющая толщину линии стала ненужной. Удалим её.

Теперь следует заменить вызов процедуры отрисовки ломаной на отрисовку набора треугольников. Но теперь наши треугольники в вершинном массиве определяются буфером индексов, поэтому, вместо `glDrawArrays` будем использовать процедуру `glDrawElements`.

```
// отрисовка набора треугольников по буферу индексов
glDrawElements(GL_TRIANGLES, figure[i].indices.size(), GL_UNSIGNED_INT, 0);
```

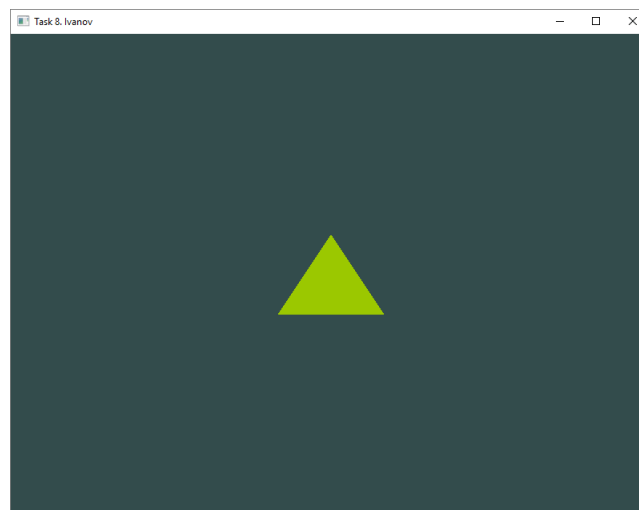
Первый аргумент, как и в `glDrawArrays` — тип изображаемых примитивов, определяемых буфером. Второй аргумент — количество индексов в буфере. Третий аргумент определяет

тип значений индексов. Четвертый аргумент — номер индекса в буфере индексов, с которого необходимо начать отрисовку.

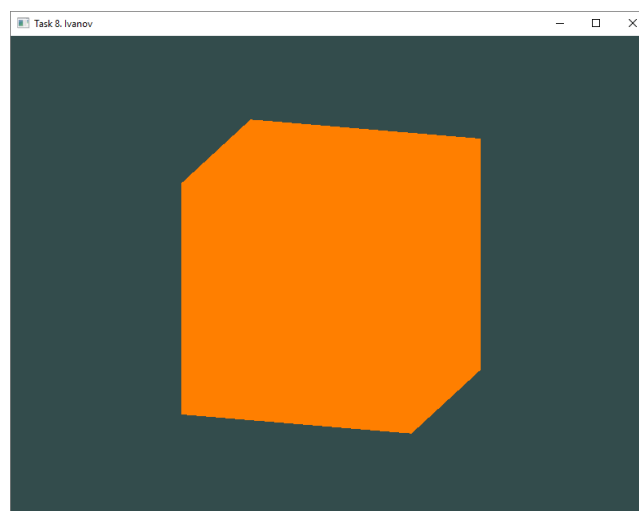
Цикл `for`, пробегающий по всем моделям трехмерной сцены, примет вид

```
1 for (int k = 0; k < models.size(); k++) { // цикл по моделям
2     std::vector<mesh> figure = models[k].figure; // список мешей очередной модели
3     glm::mat4 TM = C * models[k].modelM; // матрица общего преобразования модели
4     // пересылка пересылка матрицы TM в переменную clipView шейдерной программы
5     shaderProgram.setUniform("clipView", TM);
6     for (int i = 0; i < figure.size(); i++) {
7         // пересылка цвета линии в переменную pathColor шейдерной программы
8         shaderProgram.setUniform("pathColor", figure[i].color);
9         glBindVertexArray(figure[i].vertexArray); // делаем активным вершинный массив i-го меша
10        // отрисовка набора треугольников по буферу индексов
11        glDrawElements(GL_TRIANGLES, figure[i].indices.size(), GL_UNSIGNED_INT, 0);
12        glBindVertexArray(0); // отключаем вершинный массив
13    }
14 }
```

Запуск приложения приведет к появлению треугольника в окне.

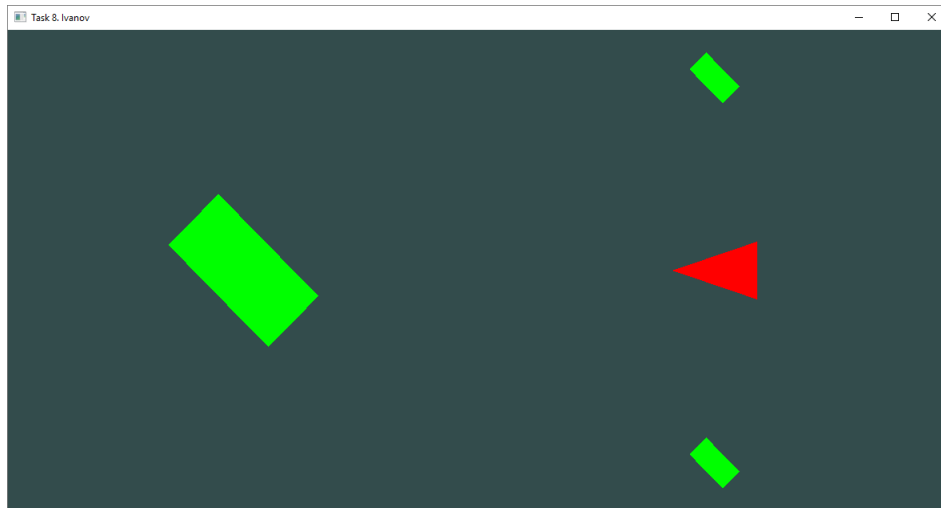


Загрузка файла `cube.txt` выведет куб в следующем виде.

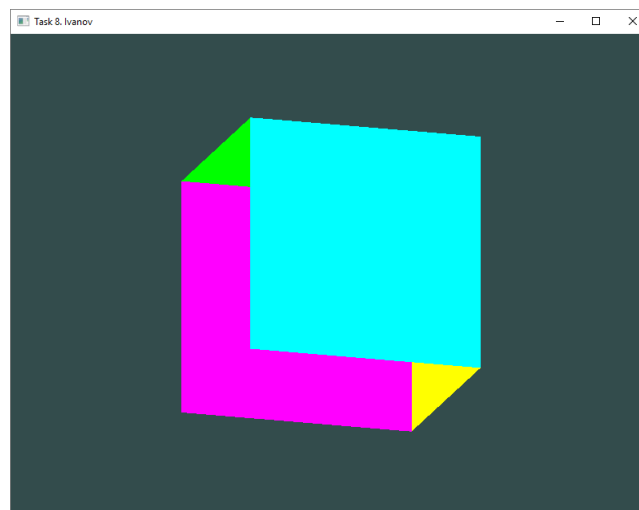


Загрузка файла `Geometric.txt` выведет следующую сцену.





Но загрузка `colorCube.txt` приведет к появлению следующего изображения.



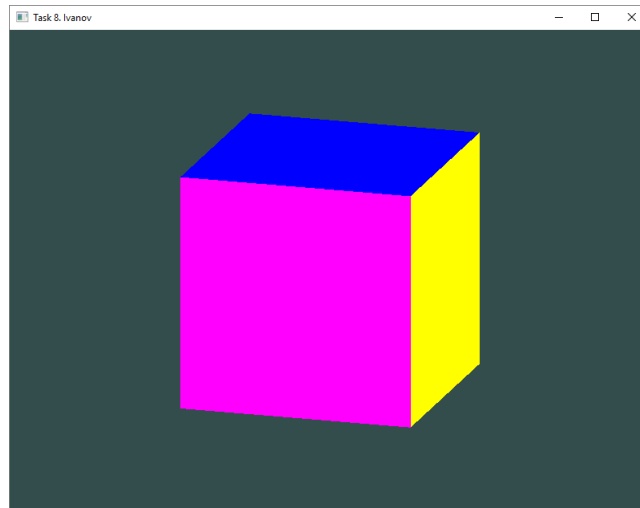
Здесь задняя грань изображена поверх передней. Дело в том, что OpenGL по умолчанию не анализирует глубину точек изображаемых примитивов трехмерной сцены, а просто изображает их в том порядке, в котором они следуют в вершинном буфере. Для того, чтобы такой анализ проводился, необходимо включить «тест глубины». Для этого, перед циклом отрисовки в процедуре `main` добавим вызов

```
glEnable(GL_DEPTH_TEST);
```

Тем самым мы включаем использование Z-буфера для анализа глубины закрашиваемых пикселей. Этот буфер необходимо очищать (заполнять значениями максимальной глубины) перед каждой отрисовкой. Поэтому внутри цикла `while` изменим вызов `glClear` на следующий.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

После проведенных изменений изображение цветного куба станет корректным.



В дальнейшем, при определении триангуляции собственных моделей, может быть полезным отключение закрашки треугольников, чтобы треугольники изображались лишь своими контурами. Для этого перед циклом отрисовки следует добавить вызов.

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

## 2.4 Источник света

Источник света в OpenGL — гипотетический объект в некоторой позиции пространства. Его характеристики участвуют в вычислении освещенности объектов сцены, но сам источник света не отображается. Если мы хотим отметить позицию источника света в пространстве, то мы должны сделать это самостоятельно.

Отметим точку, в которой будет находиться источник света, отдельно заданным объектом, позиция которого не зависит от трехмерной сцены, считанной из входного файла. Будем изображать источник света в виде белой звезды, состоящей из тринадцати отрезков, соединяющих 26 точек.

В процедуре `main` после блока объявления `uniform`-переменных опишем массив вершинных данных для этого набора отрезков (на месте закомментированного массива вершинных данных для треугольника).

```
//=====
//   НАБОР ИСХОДНЫХ ДАННЫХ ДЛЯ ОТРИСОВКИ ИСТОЧНИКА СВЕТА
//=====
GLfloat lightVertices[] = {
    -0.1f, 0.f, 0.f,
    0.1f, 0.f, 0.f,
    0.f, 0.1f, 0.f,
    0.f, -0.1f, 0.f,
    0.f, 0.f, 0.1f,
    0.f, 0.f, -0.1f,
    0.07071f, 0.07071f, 0.f,
    -0.07071f, -0.07071f, 0.f,
    -0.07071f, 0.07071f, 0.f,
    0.07071f, -0.07071f, 0.f,
    0.07071f, 0.f, 0.07071f,
    -0.07071f, 0.f, -0.07071f,
```

```

-0.07071f, 0.f, 0.07071f,
0.07071f, 0.f, -0.07071f,
0.f, 0.07071f, 0.07071f,
0.f, -0.07071f, -0.07071f,
0.f, -0.07071f, 0.07071f,
0.f, 0.07071f, -0.07071f,
0.05774f, 0.05774f, 0.05774f,
-0.05774f, -0.05774f, -0.05774f,
-0.05774f, -0.05774f, 0.05774f,
0.05774f, 0.05774f, -0.05774f,
-0.05774f, 0.05774f, 0.05774f,
0.05774f, -0.05774f, -0.05774f,
0.05774f, -0.05774f, 0.05774f,
-0.05774f, 0.05774f, -0.05774f
};

```

После этого описания организуем вершинный массив `lightVertexArray`.

```

GLuint lightVertexArray; // объект вершинного массива
// создаем вершинный массив, идентификатор которого присваиваем vertexArray
glGenVertexArrays(1, &lightVertexArray);
glBindVertexArray(lightVertexArray); // делаем активным вершинный массив

```

Опишем вершинный буфер `lightVertexBuffer` и загрузим в него содержимое массива `lightVertices`.

```

GLuint lightVertexBuffer; // идентификатор буферного объекта
// создаем буферный объект, идентификатор которого присваиваем vertexBuffer
glGenBuffers(1, &lightVertexBuffer);
// привязка vertexBuffer к GL_ARRAY_BUFFER
glBindBuffer(GL_ARRAY_BUFFER, lightVertexBuffer);
// в буфер, привязанный к GL_ARRAY_BUFFER копируем содержимое vertices
glBufferData(GL_ARRAY_BUFFER, sizeof(lightVertices), lightVertices, GL_STATIC_DRAW);
// описание расположения параметра вершинного шейдера в вершинном буфере
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0); // включение параметра 0 для шейдера
glBindBuffer(GL_ARRAY_BUFFER, 0); // отвязка буферного объекта
glBindVertexArray(0); // отключение вершинного массива

```

Сейчас модель, изображающая источник света, описана, как звездочка с центром в начале координат. Создадим для этого источника света отдельное модельное преобразование. В блоке описания глобальных переменных опишем переменную `lightM` для модельной матрицы.

```

glm::mat4 lightM; // модельная матрица для источника света

```

Установим источник света в точку  $(0, 0, 5)$ . Для этого, после формирования вершинного массива для источника света, присвоим переменной `lightM` соответствующую матрицу переноса.

```

// перемещение источника света из начала координат в точку (0, 0, 5)
lightM = glm::translate(glm::vec3(0, 0, 5));

```

Для отрисовки источника света будем использовать отдельную шейдерную программу.

Сделайте копии файлов `Vertex.glsl` и `Fragment.glsl`, которым дайте имена `LightVertex.glsl` и `LightFragment.glsl`, соответственно. Оба шейдера оставим без изменений.



Нам понадобились копии шейдеров, так как в ходе этого урока мы изменим шейдеры `Vertex.glsl` и `Fragment.glsl`.

После создания шейдерной программы `shaderProgram` и удаления шейдерных объектов `vertexShader` и `fragmentShader` создадим шейдерную программу `lightShaderProgram`, скомпоновав в ней два новых шейдера (с использованием тех же переменных для шейдерных объектов).

```
// вершинный шейдер для источника света
fragmentShader = shader("LightVertex.glsl", GL_VERTEX_SHADER);
// фрагментный шейдер для источника света
fragmentShader = shader("LightFragment.glsl", GL_FRAGMENT_SHADER);
// шейдерная программа для источника света
program lightShaderProgram(vertexShader, fragmentShader);

// удаление шейдерных объектов
vertexShader.clear();
fragmentShader.clear();
```

Заделикарируем `uniform`-переменные для новой шейдерной программы.

```
lightShaderProgram.useUniform("clipView");
lightShaderProgram.useUniform("pathColor");
```

Теперь в цикле отрисовки добавим отрисовку источника света. Для этого после цикла `for` с отрисовкой объектов сцены сделаем активной шейдерную программу для источника света

```
lightShaderProgram.use(); // делаем активной программу для источника света
```

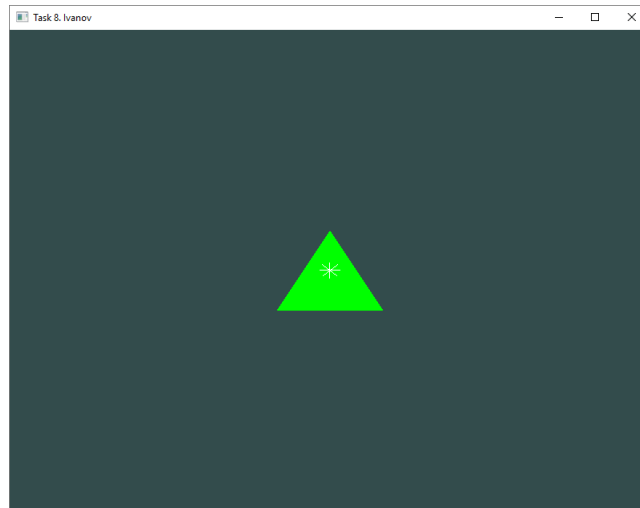
Задаем значения `uniform`-переменным. Модельную матрицу источника света домножаем на ту же матрицу `C`, а цвет зададим трехмерным вектором из единиц, что соответствует белому цвету.

```
// матрица перехода в пространство отсечения
lightShaderProgram.setUniform("clipView", C * lightM);
// белый цвет
lightShaderProgram.setUniform("pathColor", glm::vec3(1.0f));
```

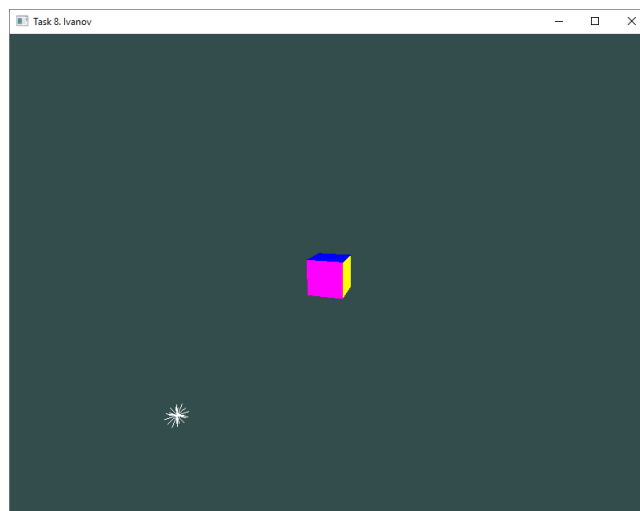
Сама отрисовка звезды — по-старинке, с использованием `glDrawArrays`. Только в качестве примитивов возьмем отрезки, что указывается с помощью константы `GL_LINES`.

```
glBindVertexArray(lightVertexArray); // делаем активным вершинный массив
glDrawArrays(GL_LINES, 0, 26); // рисуем 13 отрезков (в массиве 26 точек)
glBindVertexArray(0); // отключаем вершинный массив
```

Запуск программы приведет к появлению звездочки перед треугольником.



Если загрузить `colorCube`, то в первоначальном ракурсе изображения звезда не видна. Она становится видимой если перейти к перспективной проекции. Если после этого подойти несколько вперед к кубу, то получим изображение

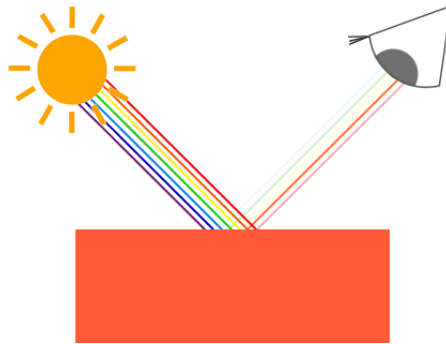


Понятно, что та звезда, которая у нас выводится, это всего лишь дополнительная модель, изображаемая в трехмерной сцене.

## 2.5 Модель освещения

### 2.5.1 Освещение, общие принципы

Цвета, видимые нами в реальной жизни, это цвета не самих объектов, а цвет отраженного ими света; т. е. мы воспринимаем цвета, которые остаются непоглощёнными (отраженными) объектом. Таким образом, если мы посветим белым светом на синюю игрушку, то она будет поглощать все составляющие белый свет цвета, кроме синего. Так как игрушка не поглощает синий цвет, он отразится, и этот отраженный свет, воздействуя на наши органы зрения, создаёт впечатление, что игрушка окрашена в синий цвет. Следующий рисунок иллюстрирует этот феномен на примере объекта кораллового цвета, отражающего исходные цвета с разной интенсивностью.



Эти правила отражения света непосредственно применяются в компьютерной графике. Когда в OpenGL мы создаем источник света, то указываем его цвет. Если затем мы умножим цвет источника света на цвет объекта, то полученное значение будет отраженным цветом объекта (и, следовательно, тем цветом, в каком мы воспринимаем объект). Вернемся к нашей игрушке (на этот раз кораллового цвета) и посмотрим, как графическими средствами вычислить её цвет, воспринимаемый наблюдателем. Для получения нужного нам цвета, произведем покомпонентное умножение двух цветовых векторов:

```
glm::vec3 lightColor(1.0f, 1.0f, 1.0f); // белый цвет источника света
glm::vec3 toyColor(1.0f, 0.5f, 0.31f); // коралловый цвет объекта
// результат получается перемножением соответствующих компонент цвета
glm::vec3 result = lightColor * toyColor; // = (1.0f, 0.5f, 0.31f);
```

Объект поглощает наибольшую часть белого света, а оставшееся количество красного, зеленого и синего излучения она отражает соответственно цвету своей поверхности. Это демонстрация того, как цвета ведут себя в реальном мире. Таким образом, мы можем задать цвет объекта вектором, характеризующим величины отражения цветовых компонент, поступающих от источника света. А что бы произошло, если бы для освещения мы использовали зеленый свет?

```
glm::vec3 lightColor(0.0f, 1.0f, 0.0f); // зеленый цвет источника света
glm::vec3 toyColor(1.0f, 0.5f, 0.31f); // коралловый цвет объекта
// результат - темно зеленый цвет
glm::vec3 result = lightColor * toyColor; // = (0.0f, 0.5f, 0.0f);
```

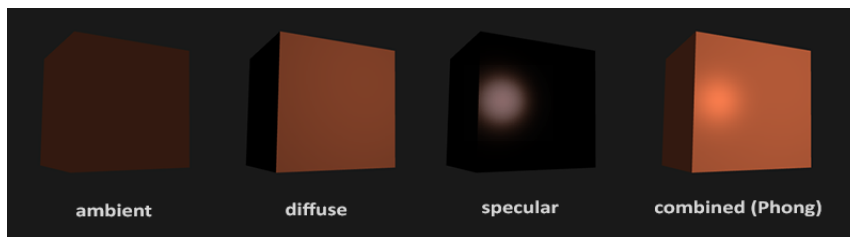
Как мы видим, в источнике света нет красной и синей составляющей, поэтому объект не будет их поглощать и/или отражать. Одну половину всего количества зеленого света игрушка поглощает, а вторую отражает. Поэтому цвет объекта, который мы увидим будет темно-зеленый. Таким образом, если мы используем зеленый источник света, то отражаться и восприниматься будут только зеленые компоненты; никакие красные и синие оттенки не будут видны. В результате объект кораллового цвета неожиданно становится темно-зеленоватым. Давайте проведем еще один эксперимент с темным оливково-зеленым светом:

```
glm::vec3 lightColor(0.33f, 0.42f, 0.18f); // оливково-зеленый цвет источника света
glm::vec3 toyColor(1.0f, 0.5f, 0.31f); // коралловый цвет объекта
// результат - темно зеленый цвет
glm::vec3 result = lightColor * toyColor; // = (0.33f, 0.21f, 0.06f);
```

Перед вами пример получения странной окраски объекта из-за использования разноцветного освещения.

Распространение света в реальном мире это чрезвычайно сложное явление, зависящее от слишком многих факторов, и, располагая ограниченными вычислительными ресурсами, мы не можем себе позволить учитывать в расчетах все нюансы. Поэтому освещение в OpenGL основано на использовании приближенных к реальности упрощенных математических моделей, которые выглядят достаточно похожими, но рассчитываются гораздо проще. Эти модели освещения описывают физику света исходя из нашего понимания его природы. Одна из этих моделей называется моделью освещения по Фонгу (Phong).

Цвет объекта в модели Фонга состоит из трех главных компонентов: фонового (ambient), рассеянного/диффузного (diffuse) и бликового (specular). Ниже вы можете видеть, что они из себя представляют:



### Фоновый цвет

Компонента ambient моделирует свет, который попадает на объект не от источника света напрямую, а как свет, отраженный от других объектов. Даже в самой темной сцене обычно всегда есть хоть какой-нибудь свет (луна, дальний свет), поэтому объекты почти никогда не бывают абсолютно чёрными. Чтобы имитировать это, мы используем константу окружающего освещения, которая всегда будет придавать объекту некоторый оттенок. Компонента ambient цвета объекта не зависит от положения объекта относительно источника света или наблюдателя.

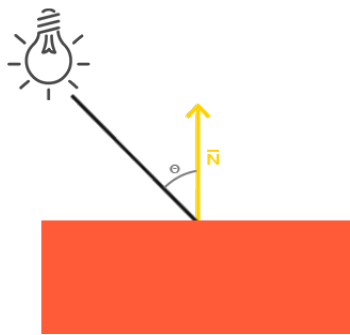
Вычисление фоновой составляющей происходит по вышеприведенному принципу: если есть источник света, то для него должна быть определена составляющая ambient — интенсивность фоновой составляющей для каждой компоненты цвета. Например, пусть объект, как и раньше, кораллового цвета, а фоновая составляющая источника света — представлена трехмерным вектором  $(0.01f, 0.2f, 0.1f)$ .

```
// интенсивность фоновой составляющей для каждой компоненты цвета
glm::vec3 lightAmbient(0.01f, 0.2f, 0.1f);
glm::vec3 toyColor(1.0f, 0.5f, 0.31f); // коралловый цвет объекта
// результат получается перемножением соответствующих компонент цвета
glm::vec3 result = lightAmbient * toyColor; // = (0.01f, 0.1f, 0.031f);
```

### Рассеянный цвет

Компонента diffuse имитирует воздействие на объект направленного источника света. Это наиболее визуально значимый компонент модели освещения. Чем круче угол падения света на объект, тем больше объект освещен. При этом считаем, что цвет точки объекта не зависит от положения наблюдателя.

Рассмотрим рисунок, поясняющий диффузное освещение.



Здесь  $\vec{N}$  — нормаль к поверхности в точке, для которой высчитывается освещение,  $\theta$  — угол, между нормалью и направлением на источник света.

Как уже сказали, чем круче угол падения, т.е. чем меньше угол  $\theta$ , тем больше объект освещен. Для моделирования этого поведения освещенность вычисляется пропорционально косинусу угла  $\theta$ : косинус положительный при  $-90^\circ \leq \theta \leq 90^\circ$ , достигает на этом промежутке своего максимума при  $\theta = 0^\circ$  и минимума при  $\theta = -90^\circ$  и  $\theta = 90^\circ$ . Понятно, что если косинус будет отрицательным, то компонента подобного освещения равна нулю.

Косинус угла  $\theta$  можно вычислить, найдя скалярное произведение между единичным вектором  $\vec{N}$  и единичным вектором, направленным из точки к источнику света.

Таким образом, будем вычислять компоненту diffuse таким же образом, что и фоновый цвет, но итоговый результат домножать на значение косинуса угла  $\theta$ .

Например, предположим, что заданы все необходимые значения, для вычисления диффузной составляющей цвета.

```
// исходные данные для расчета
// интенсивность рассеянного освещения для каждой компоненты цвета
glm::vec3 lightDiffuse(0.9f, 0.9f, 0.9f);
glm::vec3 lightPosition(23.0f, 24.0f, 2.0f); // координаты источника света
glm::vec3 toyColor(1.0f, 0.5f, 0.31f); // коралловый цвет объекта
glm::vec3 position(17.0f, 21.0f, -4.0f); // координаты точки
glm::vec3 normal(0.0f, 1.0f, 0.0f); // вектор нормали в точке

// вектор направления на источник света
// (6, 3, 6), после нормализации - (2/3, 1/3, 2/3)
glm::vec3 lightDir = glm::normalize(lightPosition - position);
// косинус угла, если косинус положительный, или ноль
// для нашего случая - 1/3
float cosTheta = max(glm::dot(lightDir, normal), 0.0f);
// результат получается перемножением соответствующих компонент цвета
// и домножается на косинус угла
// = (0.3, 0.15, 0.093)
glm::vec3 result = cosTheta * lightDiffuse * toyColor;
```

### Бликовый цвет

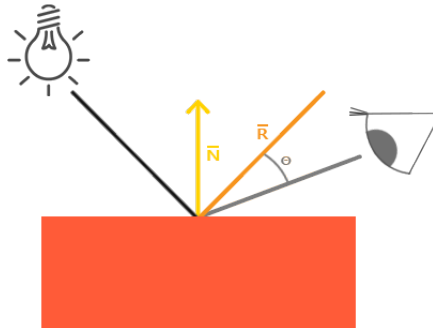
Компонента specular имитирует яркое пятно света (блик), которое появляется на блестящих объектах. По цвету зеркальные блики часто ближе к цвету источника света, чем к цвету объекта. Положение блика на объекте зависит как от положения источника света, так и от положения наблюдателя.

Освещение зеркальных бликов, так же как и рассеянное освещение, основано на векторе направления источника света и нормали поверхности объекта, но кроме этого в вычислениях учитывается и позиция наблюдателя, то есть направление, в котором игрок смотрит на фрагмент. Зеркальное освещение основано на отражательных свойствах



света. Если представить поверхность объекта в виде зеркала, то освещение бликов будет наибольшим в том месте, где бы мы увидели отраженный от поверхности свет источника.

Рассмотрим рисунок, поясняющий бликовое освещение.



Здесь  $\bar{R}$  — отраженный от поверхности луч света,  $\theta$  — угол между отраженным лучом и направлением к наблюдателю. В этой ситуации, чем меньше угол  $\theta$  — тем ярче свет видит наблюдатель (при  $\theta = 0^\circ$  наблюдатель видит отраженный луч). Поэтому, составляющая specular будет зависеть от косинуса этого угла.

Размер блика зависит от степени «глянцевости» поверхности: чем больше глянцевость, тем меньше по размерам блик. Значение косинуса — от 0 до 1. При возведении любого значения между нулем и единицей в некоторую степень мы получим результат не превышающий исходное значение. Этим воспользуемся для увеличения глянцевости: косинус угла  $\theta$  будем возводить в степень, где показатель степени — коэффициент глянцевости (shininess).

Таким образом, получим следующий порядок вычисления бликовой составляющей цвета точки, на примере, при наличии всех исходных данных.

```
// исходные данные для расчета
// интенсивность бликового освещения для каждой компоненты цвета
glm::vec3 lightSpecular(0.4f, 0.4f, 0.5f);
glm::vec3 lightPosition(23.0f, 24.0f, 2.0f); // координаты источника света
glm::vec3 toyColor(1.0f, 0.5f, 0.31f); // коралловый цвет объекта
float shininess = 2.0; // коэффициент глянцевости поверхности
glm::vec3 position(17.0f, 21.0f, -4.0f); // координаты точки
glm::vec3 normal(0.0f, 1.0f, 0.0f); // вектор нормали в точке
glm::vec3 ViewPosition(9.0f, 27.0f, -4.0f); // координаты точки наблюдения

// вектор направления на источник света
// (6, 3, 6), после нормализации - (2/3, 1/3, 2/3)
glm::vec3 lightDir = glm::normalize(lightPosition - position);
// косинус угла, если косинус положительный, или ноль
// для нашего случая - 1/3
float cosTheta = max(glm::dot(lightDir, normal), 0.0f);

float powOfCos; // коэффициент бликовой освещенности
// если cosTheta <= 0, свет на точку не падает и бликовая освещенность равна 0
if (cosTheta > 0.0)
    // вектор направления к наблюдателю = (-0.8, 0.6, 0)
    glm::vec3 viewDir = glm::normalize(ViewPosition - position)
    // вектор отраженного света от точки = (-2/3, 1/3, -2/3)
    glm::vec3 lightDirR = glm::reflect(lightDir, normal);
    // в нашем случае = 0.25333333 ^ 2 = 0.06418
    powOfCos = pow(max(glm::dot(viewDir, lightDirR), 0), shininess);
else
    powOfCos = 0.0;
// результат получается перемножением соответствующих компонент цвета
```

```
// и домножается на коэффициент блеклости
// = (0.025672, 0.012836, 0.0099479)
glm::vec3 result = powOfCos * lightSpecular * toyColor;
```

### 2.5.2 Добавление информации о нормалях

Чтобы вычислить освещенность, нам понадобятся нормали к поверхностям.

Как описано выше, нормаль влияет на освещенность точки. Поэтому, можно имитировать рельеф поверхности, изменяя направление векторов нормалей на ней.

Наши поверхности задаются набором треугольников. Будем задавать нормаль к поверхности в каждой вершине каждого треугольника. Таким образом нормаль будет являться частью вершинной информации.

В связи с этим, опять изменим представление трехмерной сцены в файле, дополнив её векторами нормалей. После этого изменим структуры данных, описанные в `Figure.h` с целью передачи информации о нормалях в состав вершинного массива.

#### Изменение формата входного файла

Добавим сначала нормали к вершинам сцены в файле `triangle.txt`. Для простоты описаний, каждую строку с тройкой координат вершины пополним тройкой координат вектора нормали. Пусть нормали к поверхности треугольника направлены в положительную сторону оси  $Oz$ , т.е. задаются вектором  $(0, 0, 1)$ . Тогда получим файл в следующем виде.

```
camera 0 0 10 0 0 0 0 1 0
screen 90 1 3 200 # соотношение сторон 1x1
model 0 0 0 2 2 2
color 155 200 0
mesh 3 1 # три точки - один треугольник
-0.5 -0.5 0.0 0.0 0.0 1.0 # координаты и нормаль
0.5 -0.5 0.0 0.0 0.0 1.0 # нормали для всех трех точек
0.0 0.5 0.0 0.0 0.0 1.0 # одинаковые
0 1 2
figure
```

По такому же принципу (с тем же вектором) пополним описания «плоских» трехмерных сцен в файлах `Geometric.txt`, `hare.txt`.

Добавим нормали к точкам модели куба в файле `colorCube.txt`. Здесь для каждой из 6-ти граней должна быть своя нормаль. После изменений получим содержимое файла.

```
camera 3 3 10 0 0 0 0 1 0
screen 90 1.333333 1 2000 # соотношение сторон 800x600
model 0 0 0 2 2 2

# нижняя грань красная
color 255 0 0
mesh 4 2
-0.5 -0.5 -0.5 0.0 -1.0 0.0 # нормаль направлена вниз
-0.5 -0.5 0.5 0.0 -1.0 0.0
0.5 -0.5 0.5 0.0 -1.0 0.0
0.5 -0.5 -0.5 0.0 -1.0 0.0
0 1 2 # первый треугольник нижней грани
0 2 3 # второй треугольник нижней грани

# верхняя грань синяя
color 0 0 255
mesh 4 2
```

```

-0.5 0.5 -0.5    0.0 1.0 0.0 # нормаль направлена вверх
-0.5 0.5 0.5     0.0 1.0 0.0
0.5 0.5 0.5      0.0 1.0 0.0
0.5 0.5 -0.5     0.0 1.0 0.0
0 1 3 # первый треугольник верхней грани
1 2 3 # второй треугольник верхней грани

# левая грань зеленая
color 0 255 0
mesh 4 2
-0.5 -0.5 -0.5   -1.0 0.0 0.0 # нормаль направлена влево
-0.5 -0.5 0.5    -1.0 0.0 0.0
-0.5 0.5 -0.5    -1.0 0.0 0.0
-0.5 0.5 0.5     -1.0 0.0 0.0
0 1 2 # первый треугольник левой грани
1 2 3 # второй треугольник левой грани

# правая грань желтая
color 255 255 0
mesh 4 2
0.5 -0.5 0.5      1.0 0.0 0.0 # нормаль направлена вправо
0.5 -0.5 -0.5     1.0 0.0 0.0
0.5 0.5 0.5       1.0 0.0 0.0
0.5 0.5 -0.5      1.0 0.0 0.0
2 0 1 # первый треугольник правой грани
2 1 3 # второй треугольник правой грани

# передняя грань фиолетовая
color 255 0 255
mesh 4 2
-0.5 -0.5 0.5     0.0 0.0 1.0 # нормаль направлена на нас
0.5 -0.5 0.5      0.0 0.0 1.0
-0.5 0.5 0.5      0.0 0.0 1.0
0.5 0.5 0.5       0.0 0.0 1.0
0 2 3 # первый треугольник передней грани
0 3 1 # второй треугольник передней грани

# задняя грань морской волны
color 0 255 255
mesh 4 2
-0.5 -0.5 -0.5    0.0 0.0 -1.0 # нормаль направлена от нас
0.5 -0.5 -0.5     0.0 0.0 -1.0
-0.5 0.5 -0.5     0.0 0.0 -1.0
0.5 0.5 -0.5      0.0 0.0 -1.0
1 3 2 # первый треугольник задней грани
1 2 0 # второй треугольник задней грани

figure

```

Аналогичные изменения следует внести в файл `cube.txt`.

### Изменение структур данных

Внесем изменения в файл `Figure.h`.

Объединим информацию об одной вершине (координаты и нормаль) в структуре `vertex`. Добавим её описание перед описаниями классов.

```

struct vertex { // одна вершина
    glm::vec3 position; // координаты вершины

```

```
glm::vec3 normal; // нормаль в вершине
};
```

Теперь изменим тип поля `vertices` в описании класса `mesh`:

```
std::vector<vertex> vertices; // последовательность точек
```

Такое же изменение проведем в описании аргумента `verts` конструктора.

```
mesh(std::vector<vertex> verts, std::vector<GLuint> inds, glm::vec3 col) {
```

Будем предполагать, что нормаль к вершине будет вторым входным параметром вершинного шейдера. Внесем соответствующие изменения в метод `setupMesh`. Его вид на текущем этапе следующий.

```
1 void setupMesh() {
2     // создаем вершинный массив и вершинный буфер
3     glGenVertexArrays(1, &vertexArray); // создаем вершинный массив
4     glGenBuffers(1, &vertexBuffer); // создаем вершинный буфер
5     glGenBuffers(1, &elementBuffer); // создаем буфер индексов
6
7     glBindVertexArray(vertexArray); // делаем вершинный массив активным
8
9     // связываем vertexBuffer с GL_ARRAY_BUFFER
10    glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
11    // копируем содержимое vertices в вершинный буфер vertexBuffer
12    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec3), &vertices[0], GL_STATIC_DRAW);
13    // описание расположения параметра вершинного шейдера в вершинном буфере
14    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(glm::vec3), (GLvoid*)0);
15    glEnableVertexAttribArray(0); // включение параметра 0 для шейдера
16
17    // связываем elementBuffer с GL_ELEMENT_ARRAY_BUFFER
18    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementBuffer);
19    // копируем содержимое indices в буфер индексов elementBuffer
20    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLuint), &indices[0], GL_STATIC_DRAW);
21
22    glBindBuffer(GL_ARRAY_BUFFER, 0); // отвязка буферного объекта
23    glBindVertexArray(0); // отключение вершинного массива
24 }
```

Во-первых у нас изменился размер списка `vertices`. Поэтому `sizeof(glm::vec3)` нужно заменить в вызовах `glBufferData` и `glVertexAttribPointer` на `sizeof(vertex)`. В процедуре `glVertexAttribPointer` это будет означать, что для перехода к координатам следующей вершины нужно пропустить столько байт, сколько их в одной ячейке типа `vertex`.

После включения параметра 0 для шейдера определим параметр 1. Для этого добавим еще один вызов `glVertexAttribPointer`, в котором параметром будет определяться нормаль к вершине.

```
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(vertex), (GLvoid*)offsetof(vertex, normal));
```

Вызов подобен предыдущему, за двумя исключениями. Номер параметра шейдера — 1 вместо 0. Кроме этого, в последнем аргументе указывается, что нормаль первой вершины отстоит от начала ряда данных в вершинном буфере на то же расстояние, что и поле `normal` в элементе данных типа `vertex` (для этого удобно использовать вызов `offsetof(vertex, normal)`).

Теперь включим параметр 1.

```
glEnableVertexAttribArray(1); // включение параметра 1 для шейдера
```

Метод `setupMesh` примет следующий вид.

```
1 void setupMesh() {
2     // создаем вершинный массив и вершинный буфер
3     glGenVertexArrays(1, &vertexArray); // создаем вершинный массив
4     glGenBuffers(1, &vertexBuffer); // создаем вершинный буфер
5     glGenBuffers(1, &elementBuffer); // создаем буфер индексов
6
7     glBindVertexArray(vertexArray); // делаем вершинный массив активным
8
9     // связываем vertexBuffer с GL_ARRAY_BUFFER
10    glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
11    // копируем содержимое vertices в вершинный буфер vertexBuffer
12    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(vertex), &vertices[0], GL_STATIC_DRAW);
13    // описание расположения параметра вершинного шейдера в вершинном буфере
14    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(vertex), (GLvoid*)0);
15    glEnableVertexAttribArray(0); // включение параметра 0 для шейдера
16    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(vertex), (GLvoid*)offsetof(vertex, normal));
17    glEnableVertexAttribArray(1); // включение параметра 1 для шейдера
18
19    // связываем elementBuffer с GL_ELEMENT_ARRAY_BUFFER
20    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementBuffer);
21    // копируем содержимое indices в буфер индексов elementBuffer
22    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLuint), &indices[0], GL_STATIC_DRAW);
23
24    glBindBuffer(GL_ARRAY_BUFFER, 0); // отвязка буферного объекта
25    glBindVertexArray(0); // отключение вершинного массива
26    //glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0); // отвязка буферного объекта
27 }
```

### Изменение процедуры чтения из файла

Мы добавили в каждую строку с координатами вершин координаты вектора нормали. Изменим код процедуры `readFromFile` так, чтобы информация о нормалях считывалась и записывалась в соответствующие структуры данных.

Изменения коснутся блока, соответствующего команде `mesh`. Он на текущем этапе имеет следующий вид.

```
1 else if (cmd == "mesh") { // набор мешей
2     std::vector<glm::vec3> vertices; // список точек
3     int N, K; // количество точек и треугольников
4     s >> N >> K;
5     std::string str1; // дополнительная строка для чтения из файла
6     while (N > 0) { // пока не все точки считали
7         std::getline(in, str1); // считываем в str1 из входного файла очередную строку
8         // так как файл корректный, то на конец файла проверять не нужно
9         if ((str1.find_first_not_of("\t\r\n") != std::string::npos) && (str1[0] != '#')) {
10            // прочитанная строка не пуста и не комментарий
11            // значит в ней тройка координат
12            float x, y, z; // переменные для считывания
13            std::stringstream s1(str1); // еще один строковый поток из строки str1
14            s1 >> x >> y >> z;
15            vertices.push_back(glm::vec3(x, y, z)); // добавляем точку в список
16            N--; // уменьшаем счетчик после успешного считывания точки
17        }
18    }
19    std::vector<GLuint> indices; // список индексов вершин треугольников
20    while (K > 0) { // пока не считали все треугольники
21        std::getline(in, str1); // считываем в str1 из входного файла очередную строку
22        // так как файл корректный, то на конец файла проверять не нужно
23        if ((str1.find_first_not_of("\t\r\n") != std::string::npos) && (str1[0] != '#')) {
24            // прочитанная строка не пуста и не комментарий
25            // значит в ней тройка индексов вершин треугольника
26            GLuint x; // переменная для считывания
27            std::stringstream s1(str1); // еще один строковый поток из строки str1
28            for (int i = 0; i < 3; i++) { // три раза
```

```

29     s1 >> x; // считываем индекс
30     indices.push_back(x); // добавляем индекс в список indices
31 }
32 K--; // уменьшаем счетчик после успешного считывания точки
33 }
34 }
35 // все точки и индексы считаны, генерируем меш и кладем его в список figure
36 figure.push_back(mesh(vertices, indices, glm::vec3(r, g, b) / 255.f));
37 }

```

Здесь необходимо изменить тип элементов списка `vertices` на `vertex`.

```
std::vector<glm::vec3> vertices; // список точек
```

Кроме переменных `x`, `y`, `z` для координат вершины опишем временные переменные `nx`, `ny`, `nz` для координат нормали.

```
float x, y, z; // переменные для считывания координат вершины
float nx, ny, nz; // переменные для считывания координат нормали
```

Теперь добавим считывание координат нормали

```
s1 >> x >> y >> z;
s1 >> nx >> ny >> nz;
```

Наконец, заменим элемент, добавляемый в список `vertices`, на структуру из двух векторов.

```
// добавляем точку в список
vertices.push_back({ glm::vec3(x, y, z), glm::vec3(nx, ny, nz) });
```

Получим блок команды `mesh` в следующем виде.

```

1  else if (cmd == "mesh") { // набор мешей
2      std::vector<glm::vec3> vertices; // список точек
3      int N, K; // количество точек и треугольников
4      s >> N >> K;
5      std::string str1; // дополнительная строка для чтения из файла
6      while (N > 0) { // пока не все точки считали
7          std::getline(in, str1); // считываем в str1 из входного файла очередную строку
8              // так как файл корректный, то на конец файла проверять не нужно
9          if ((str1.find_first_not_of("\t\r\n") != std::string::npos) && (str1[0] != '#')) {
10             // прочитанная строка не пуста и не комментарий
11             // значит в ней тройка координат
12             float x, y, z; // переменные для считывания координат вершины
13             float nx, ny, nz; // переменные для считывания координат нормали
14             std::stringstream s1(str1); // еще один строковый поток из строки str1
15             s1 >> x >> y >> z;
16             s1 >> nx >> ny >> nz;
17             // добавляем точку в список
18             vertices.push_back({ glm::vec3(x, y, z), glm::vec3(nx, ny, nz) });
19             N--; // уменьшаем счетчик после успешного считывания точки
20         }
21     }
22     std::vector<GLuint> indices; // список индексов вершин треугольников
23     while (K > 0) { // пока не считали все треугольники
24         std::getline(in, str1); // считываем в str1 из входного файла очередную строку
25             // так как файл корректный, то на конец файла проверять не нужно
26         if ((str1.find_first_not_of("\t\r\n") != std::string::npos) && (str1[0] != '#')) {
27             // прочитанная строка не пуста и не комментарий
28             // значит в ней тройка индексов вершин треугольника
29             GLuint x; // переменная для считывания
30             std::stringstream s1(str1); // еще один строковый поток из строки str1
31             for (int i = 0; i < 3; i++) { // три раза
32                 s1 >> x; // считываем индекс

```

```

33     indices.push_back(x); // добавляем индекс в список indices
34 }
35 K--; // уменьшаем счетчик после успешного считывания точки
36 }
37 }
38 // все точки и индексы считаны, генерируем меш и кладем его в список figure
39 figure.push_back(mesh(vertices, indices, glm::vec3(r, g, b) / 255.f));
40 }

```

### Изменение кода вершинного шейдера

На текущем этапе вершинный шейдер `Vertex.glsl` имеет следующий вид.

```

1  #version 330 core
2  layout (location = 0) in vec3 position;
3
4  uniform mat4 clipView;
5
6  void main() {
7      gl_Position = clipView * vec4(position, 1.0);
8  }

```

Вторым входным параметром вершинного шейдера мы объявили нормаль к вершине. Объявим этот параметр в шейдере после объявления параметра `position`.

```
layout (location = 1) in vec3 normal;
```

Взаимное расположение точек объекта и источников света будем анализировать в мировой системе координат, т. е. после выполнения модельных преобразований.

Цвет будем вычислять во фрагментном шейдере. Мы будем передавать во фрагментный шейдер из вершинного координаты вершины и координаты вектора нормали. Для этого опишем выходные параметры вершинного шейдера.

```

out vec3 fragPos; // координаты точки
out vec3 fragNorm; // координаты нормали в точке

```

Эти параметры должны быть теперь описаны во фрагментном шейдере в качестве входных. Сделаем это позже.

Координаты точек и нормалей в вершинном массиве — координаты до выполнения модельного преобразования. Поэтому будем передавать в вершинный шейдер матрицу модельного преобразования для соответствующего объекта в виде значения `uniform`-переменной `modelView`. Опишем её.

```
uniform mat4 modelView;
```

Как Вы можете помнить, если задано матричное преобразование  $M$  для точек в виде матрицы  $4 \times 4$ , то нормаль, полученная для поверхности после преобразования  $M$  вычисляется как

$$\vec{n}' = (M^{-1})_{3 \times 3}^T \vec{n},$$

где  $(M^{-1})_{3 \times 3}^T$  — обратная матрица  $M$  после транспонирования, у которой отброшены последняя строка и последний столбец.

Чтобы не вычислять в шейдере для каждой точки обратную матрицу, будем передавать значение уже вычисленной транспонированной обратной матрицы в виде значения `uniform`-переменной `modelInv`. Опишем её.

```
uniform mat4 modelInv;
```

В теле функции `main` шейдера добавим вычисление значения `fragPos`

```
fragPos = vec3(modelView * vec4(position, 1.0));
```

Здесь координаты точки переводятся в однородную систему координат и умножаются на модельную матрицу, после чего отбрасывается последняя координата.

Добавим вычисление преобразованной нормали.

```
fragNorm = mat3(modelInv) * normal;
```

Здесь, напротив, отбрасывается последний столбец и последняя строка полученной матрицы, после чего вычисляются координаты нормали.

Вершинный шейдер примет следующий вид.

```
1 #version 330 core
2 layout (location = 0) in vec3 position;
3 layout (location = 1) in vec3 normal;
4
5 out vec3 fragPos; // координаты точки
6 out vec3 fragNorm; // координаты нормали в точке
7
8 uniform mat4 clipView;
9 uniform mat4 modelView;
10 uniform mat4 modelInv;
11
12 void main() {
13     fragPos = vec3(modelView * vec4(position, 1.0));
14     fragNorm = mat3(modelInv) * normal;
15     gl_Position = clipView * vec4(position, 1.0);
16 }
```

### Изменение фрагментного шейдера

Фрагментный шейдер на текущем этапе имеет следующий вид.

```
1 #version 330 core
2 out vec4 color;
3 uniform vec3 pathColor;
4 void main() {
5     color = vec4(pathColor, 1.0f);
6 }
```

Необходимо описать в нем два входных параметра, соответствующих выходным вершинного шейдера. Добавим описание перед описанием выходного параметра.

```
in vec3 fragPos; // координаты точки
in vec3 fragNorm; // координаты нормали в точке
```

Чтобы описать свойства источника света, нужно описать составляющие его компоненты. Эти компоненты представляются тремя векторами, описывающими интенсивность компонент `ambient`, `diffuse` и `specular`. Представим в шейдере эти компоненты света в виде единой структуры `Light`, которую опишем перед описанием входных параметров.

```
struct Light {
    vec3 ambient;
    vec3 diffuse;
```



```
vec3 specular;
};
```

Источник света для всех точек сцены один, поэтому информацию о нем будем передавать в виде значения uniform-переменной `light` типа `Light`. Опишем её.

```
uniform Light light; // компоненты света
```

Кроме того, следует сюда передать мировые координаты источника света

```
uniform vec3 lightPos; // координаты источника света
```

и мировые координаты точки наблюдения

```
uniform vec3 viewPos; // координаты точки наблюдения
```

Теперь, в теле процедуры `main` приступим к вычислению составляющих цвета фрагмента.

Сначала вычислим косинус угла между направлением на источник света и нормалью.

```
// нормализуем полученный вектор нормали
vec3 norm = normalize(fragNorm);
// получаем нормализованный вектор направления на источник света
vec3 lightDir = normalize(lightPos - fragPos);
// вычисляем косинус угла между полученными векторами
float cosTheta = max(dot(norm, lightDir), 0.0);
```



Обратите внимание, что полученный вектор `fragNorm` необходимо нормализовать, так как это вектор, полученный в результате процедуры растеризации, и его длина в общем случае может быть отлична от единицы.

Вычислим коэффициент бликовой освещенности `powOfCos`.

```
float powOfCos; // коэффициент бликовой освещенности
if (cosTheta > 0.0) { // если cosTheta <= 0, бликовая освещенность равна 0
    // получаем нормализованный вектор направления в точку наблюдения
    vec3 viewDir = normalize(viewPos - fragPos);
    // получаем вектор отраженного света
    vec3 lightDirR = reflect(-lightDir, norm);
    powOfCos = pow(max(dot(viewDir, lightDirR), 0.0), 32);
}
else
    powOfCos = 0.0;
```



Обратите внимание, что мы инвертировали вектор `lightDir`. Функция `reflect` ожидает, что первый вектор будет указывать направление от источника света к положению фрагмента, но вектор `lightDir` в настоящее время указывает в обратную сторону, то есть от фрагмента к источнику света. Предполагается, что второй аргумент должен быть единичной длины, и мы передаем нормализованный вектор `norm`.

Здесь мы временно взяли степень глянцевого равную 32. В ходе дальнейшего урока мы сделаем это значение более универсальным.

Теперь можем вычислить все три составляющих цвета пиксела.

```
vec3 ambient = light.ambient * pathColor;
vec3 diffuse = light.diffuse * cosTheta * pathColor;
vec3 specular = light.specular * powOfCos * pathColor;
```

Итоговый цвет вычисляем как сумму этих компонент.

```
vec3 result = ambient + diffuse + specular;
```

Наконец, вычисляем выходное значения шейдера, добавив к трехмерному вектору четвертую компоненту, равную единице.

```
color = vec4(result, 1.0);
```

На текущем этапе получим следующий вид фрагментного шейдера.

```
1  #version 330 core
2
3  struct Light {
4      vec3 ambient;
5      vec3 diffuse;
6      vec3 specular;
7  };
8
9  in vec3 fragPos;
10 in vec3 fragNorm;
11
12 out vec4 color;
13
14 uniform vec3 pathColor;
15 uniform Light light;
16 uniform vec3 lightPos;
17 uniform vec3 viewPos;
18
19 void main() {
20     // нормализуем полученный вектор нормали
21     vec3 norm = normalize(fragNorm);
22     // получаем нормализованный вектор направления на источник света
23     vec3 lightDir = normalize(lightPos - fragPos);
24     // вычисляем косинус угла между полученными векторами
25     float cosTheta = max(dot(norm, lightDir), 0.0);
26
27     float powOfCos; // коэффициент бликовой освещенности
28     if (cosTheta > 0.0) { // если cosTheta <= 0, бликовая освещенность равна 0
29         // получаем нормализованный вектор направления в точку наблюдения
30         vec3 viewDir = normalize(viewPos - fragPos);
31         // получаем вектор отраженного света
32         vec3 lightDirR = reflect(-lightDir, norm);
33         powOfCos = pow(max(dot(viewDir, lightDirR), 0.0), 32);
34     }
35     else
36         powOfCos = 0.0;
37
38     vec3 ambient = light.ambient * pathColor;
39     vec3 diffuse = light.diffuse * cosTheta * pathColor;
40     vec3 specular = light.specular * powOfCos * pathColor;
41
42     vec3 result = ambient + diffuse + specular;
43     color = vec4(result, 1.0);
44 }
```

### Изменения в шейдерной программе

В наших шейдерах Vertex.glsl и Fragment.glsl добавилось достаточно много uniform-переменных. Обеспечим загрузку их значений из основной программы. В процедуре main файла Main.cpp после деклараций

```
shaderProgram.useUniform("clipView");
shaderProgram.useUniform("pathColor");
```

объявим, что мы используем дополнительные uniform-значения.

```
shaderProgram.useUniform("modelView");
shaderProgram.useUniform("modelInv");
shaderProgram.useUniform("lightPos");
shaderProgram.useUniform("viewPos");
shaderProgram.useUniform("light.ambient");
shaderProgram.useUniform("light.diffuse");
shaderProgram.useUniform("light.specular");
```



Следует обратить внимание, что мы объявляем каждое поле uniform-переменной `light` по отдельности.

Теперь проведем загрузку значений этих переменных в цикле отрисовки. Добавим соответствующие вызовы `shaderProgram.setUniform` перед циклом `for` основной отрисовки (сразу после инициализации матрицы `C`).

Во первых загрузим характеристики света. Пусть, к примеру, они будут следующими.

```
shaderProgram.setUniform("light.ambient", glm::vec3(0.2f, 0.2f, 0.2f));
shaderProgram.setUniform("light.diffuse", glm::vec3(0.5f, 0.5f, 0.5f));
shaderProgram.setUniform("light.specular", glm::vec3(1.f, 1.f, 1.f));
```

Позиция источника света в мировой системе координат определяется соответствующей модельной матрицей. То есть координаты источника света можно вычислить, если домножить слева на эту матрицу вектор  $(0, 0, 0, 1)$ .

```
shaderProgram.setUniform("lightPos", glm::vec3(lightM * glm::vec4(0, 0, 0, 1)));
```

Позиция наблюдателя в системе координат наблюдателя — в начале координат. Переход из мировой системы координат в систему координат наблюдателя осуществляется матрицей  $T$ , следовательно преобразование из системы координат наблюдателя в мировые координаты можно произвести с помощью матрицы  $T^{-1}$ . Таким образом, позицию наблюдателя в мировой системе координат можно получить перемножением этой матрицы с вектором  $(0, 0, 0, 1)$ .

```
shaderProgram.setUniform("viewPos", glm::vec3(glm::inverse(T) * glm::vec4(0, 0, 0, 1)));
```

Осталось загрузить матрицы для модельного преобразования, которые для каждой модели свои, а потому их загрузку будем производить уже в цикле `for`. После загрузки uniform-переменной `clipView`

```
shaderProgram.setUniform("clipView", TM);
```

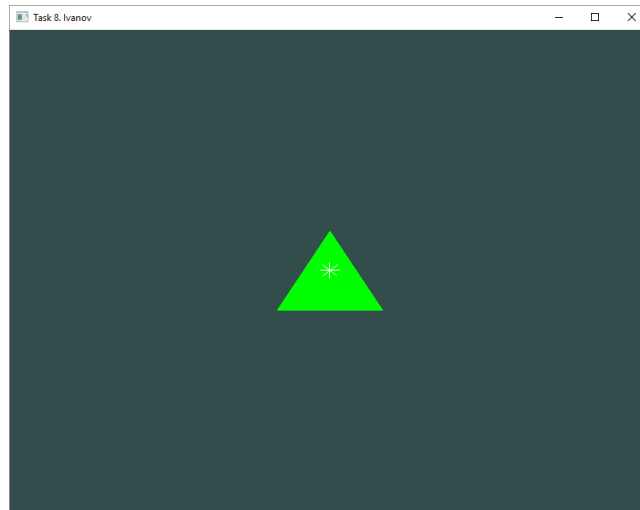
загрузим модельную матрицу

```
shaderProgram.setUniform("modelView", models[k].modelM);
```

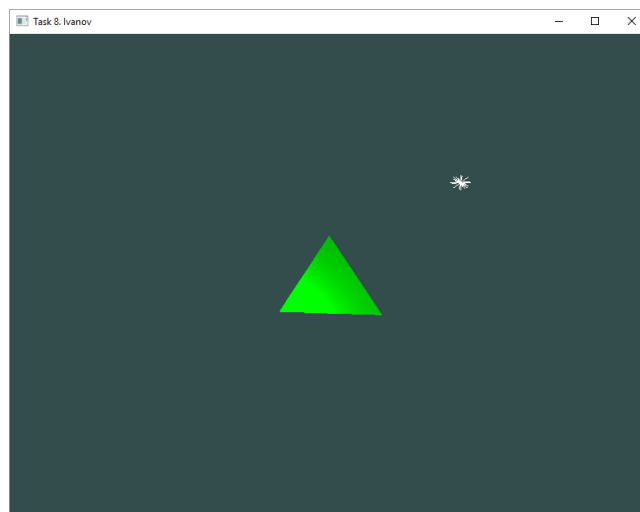
и транспонированную обратную модельную матрицу

```
shaderProgram.setUniform("modelInv", glm::transpose(glm::inverse(models[k].modelM)));
```

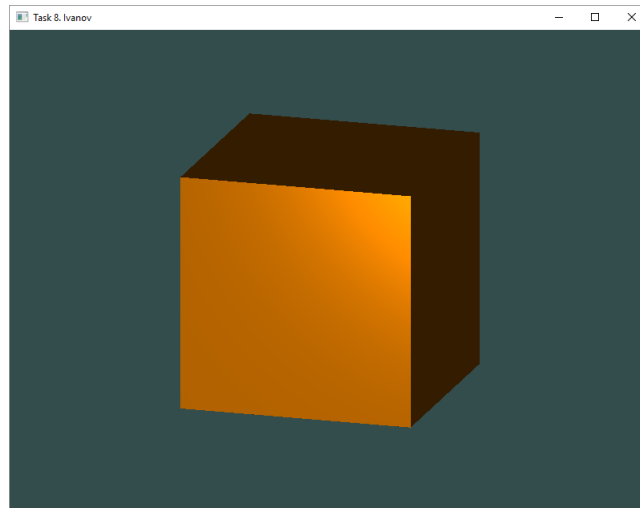
Проект можно запустить. Запуск приведет к появлению треугольника.



Если его несколько повернуть, то можно заметить на треугольнике широкий блик от источника света.

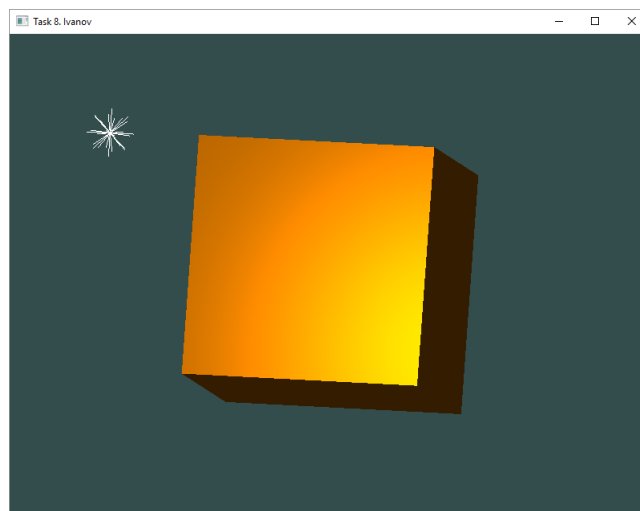


Некоторый блик сразу заметен при загрузке файла `cube.txt`.



Так как источник света расположен прямо перед передней гранью, остальные грани остаются неосвещенными и для них составляющие цвета *diffuse* и *specular* равны нулю, и цвет состоит только из значения компоненты *ambient* (20% от исходного значения цвета).

Если производить вращение куба, то можно наблюдать бликовый эффект более явно.



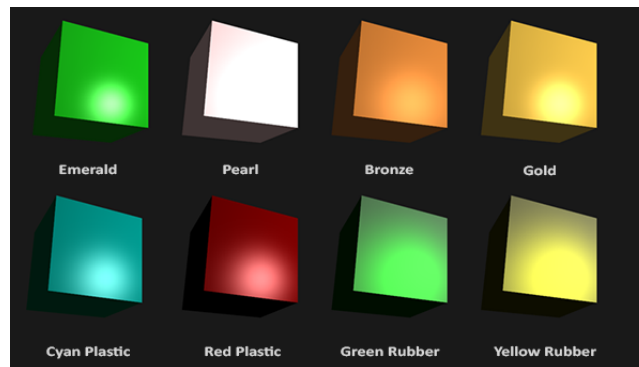
### 2.5.3 Материалы

В реальном мире каждый объект по-разному реагирует на свет. Железные объекты обычно сверкают сильнее, чем, например, глиняная ваза. А деревянный контейнер реагирует на свет не так же, как стальной. Каждый объект имеет разную отражающую способность. Некоторые объекты отражают свет без сильного рассеивания, у других радиус отражения довольно большой. Если мы хотим имитировать разные типы поверхностей в OpenGL, то нам нужно определить свойства материала, специфичные для каждого объекта.

Описывая объекты, мы можем установить цвет материала для всех трех компонентов освещения: окружающего (*ambient*), рассеянного (*diffuse*) и бликового (*specular*). После этого мы получим детальный контроль над результирующим цветом объекта. Теперь добавим силу блеска (*shininess*) к нашим трем цветам и получим все свойства материала, которые нам нужны.

Используя эти четыре компонента, мы можем имитировать множество реальных материалов. Таблица на сайте <http://devernay.free.fr/cours/opengl/materials.html>

содержит свойства некоторых материалов, которые мы можем видеть в реальной жизни. Следующее изображение показывает куб, с разными свойствами материалов.



### Изменения в формате входного файла

Чтобы внедрить такую модель в нашу реализацию, мы изменим формат входного файла: вместо одной команды `color` во входном файле будем задавать 4 команды: `ambient`, `diffuse`, `specular` и `shininess`.

Начнем изменения с файла `triangle.txt`. На текущем этапе он выглядит так.

```
camera 0 0 10 0 0 0 1 0
screen 90 1 3 200 # соотношение сторон 1x1
model 0 0 0 2 2 2
color 155 200 0
mesh 3 1 # три точки - один треугольник
-0.5 -0.5 0.0 0.0 0.0 1.0 # координаты и нормаль
0.5 -0.5 0.0 0.0 0.0 1.0 # нормали для всех трех точек
0.0 0.5 0.0 0.0 0.0 1.0 # одинаковые
0 1 2
figure
```

Сделаем треугольник изумрудным (emerald). То есть вместо команды `color` добавим четыре команды

```
ambient 0.0215 0.1745 0.0215
diffuse 0.07568 0.61424 0.07568
specular 0.633 0.727811 0.633
shininess 76.8 # значение 0.6 домноженное на 128
```



Обратите внимание, что значение `shininess`, заданное в таблице на вышеуказанном сайте, необходимо домножать на 128.

Файл `triangle.txt` примет вид

```
camera 0 0 10 0 0 0 1 0
screen 90 1 3 200 # соотношение сторон 1x1
model 0 0 0 2 2 2
ambient 0.0215 0.1745 0.0215
diffuse 0.07568 0.61424 0.07568
specular 0.633 0.727811 0.633
shininess 76.8 # значение 0.6 домноженное на 128
mesh 3 1 # три точки - один треугольник
-0.5 -0.5 0.0 0.0 0.0 1.0 # координаты и нормаль
```

```
0.5 -0.5 0.0  0.0 0.0 1.0 # нормали для всех трех точек
0.0  0.5 0.0  0.0 0.0 1.0 # одинаковые
0 1 2
figure
```

По аналогии, внесем изменения в файл `cube.txt` : сделаем наш куб рубиновым (ruby). Заменяем команду `color` на

```
ambient 0.1745 0.01175 0.01175
diffuse 0.61424 0.04136 0.04136
specular 0.727811 0.626959 0.626959
shininess 76.8
```

### Изменения структур данных

Как и прежде, изменения формата входного файла приводит к необходимости изменения структур данных.

Перейдем к изменению файла `Figure.h`.

Перед описанием классов добавим структуру `Material`, составляющими которой будут все компоненты цвета материала.

```
struct Material {
    glm::vec3 ambient;
    glm::vec3 diffuse;
    glm::vec3 specular;
    float shininess;
};
```

Цвет являлся свойством объекта класса `mesh` (поле `color`). Заменяем это свойство на структуру `material` типа `Material`.

```
Material material;
```

Кроме того, соответствующее изменение необходимо внести в конструктор `mesh`: изменить тип последнего аргумента на `Material` и имя поля, которому присваивается значение этого аргумента, на `material`.

Файл `Figure.h` (его используемые в проекте части) примет следующий вид.

```
1  #pragma once
2  #include <glad\glad.h>
3  #include <glm\glm.hpp>
4
5  #include <vector>
6
7  struct vertex {
8      glm::vec3 position;
9      glm::vec3 normal;
10 };
11
12 struct Material {
13     glm::vec3 ambient;
14     glm::vec3 diffuse;
15     glm::vec3 specular;
16     float shininess;
17 };
18
19 class mesh {
20 public:
21     std::vector<vertex> vertices; // последовательность точек
22     std::vector<GLuint> indices; // последовательность индексов в наборе точек
```

```

23 GLuint vertexArray; // вершинный массив (объект OpenGL)
24 Material material;
25 mesh(std::vector<vertex> verts, std::vector<GLuint> inds, Material col) {
26     vertices = verts;
27     indices = inds;
28     material = col;
29     setupMesh();
30 }
31
32 private:
33 GLuint vertexBuffer; // вершинный буфер (объект OpenGL)
34 GLuint elementBuffer; // буфер индексов вершин (объект OpenGL)
35 void setupMesh() {
36     // создаем вершинный массив и вершинный буфер
37     glGenVertexArrays(1, &vertexArray); // создаем вершинный массив
38     glGenBuffers(1, &vertexBuffer); // создаем вершинный буфер
39     glGenBuffers(1, &elementBuffer); // создаем буфер индексов
40
41     glBindVertexArray(vertexArray); // делаем вершинный массив активным
42
43     // связываем vertexBuffer с GL_ARRAY_BUFFER
44     glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
45     // копируем содержимое vertices в вершинный буфер vertexBuffer
46     glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(vertex), &vertices[0], GL_STATIC_DRAW);
47     // описание расположения параметра вершинного шейдера в вершинном буфере
48     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(vertex), (GLvoid*)0);
49     glEnableVertexAttribArray(0); // включение параметра 0 для шейдера
50     glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(vertex), (GLvoid*)offsetof(vertex, normal));
51     glEnableVertexAttribArray(1); // включение параметра 1 для шейдера
52
53     // связываем elementBuffer с GL_ELEMENT_ARRAY_BUFFER
54     glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementBuffer);
55     // копируем содержимое indices в буфер индексов elementBuffer
56     glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLuint), &indices[0], GL_STATIC_DRAW);
57
58     glBindBuffer(GL_ARRAY_BUFFER, 0); // отвязка буферного объекта
59     glBindVertexArray(0); // отключение вершинного массива
60 }
61 };
62
63 class model {
64 public:
65     std::vector<mesh> figure; // составляющие рисунка
66     glm::mat4 modelM; // модельная матрица
67     model(std::vector<mesh> fig, glm::mat4 mat) {
68         figure = fig;
69         modelM = mat;
70     }
71 };

```

### Чтение входных данных

Внесем изменение в процедуру `readFromFile` чтения входного файла.

Здесь за цвет модели отвечают три переменные

```

float r, g, b; // составляющие цвета
r = g = b = 0; // значение составляющих цвета по умолчанию (черный)

```

Теперь для полного набора компонент цвета нам понадобятся три вектора и вещественное число. Опишем их вместо переменных `r`, `g` и `b`, присвоив им значения по умолчанию.

```

// составляющие цвета модели со значениями по умолчанию
glm::vec3 ambient = glm::vec3(0, 0, 0);
glm::vec3 diffuse = glm::vec3(0, 0, 0);
glm::vec3 specular = glm::vec3(0, 0, 0);
float shininess = 1;

```



Организуем чтение соответствующих команд вместо команды `color`. Блок соответствующий команде `color` имеет вид

```
else if (cmd == "color") { // цвет линии
    s >> r >> g >> b; // считываем три составляющие цвета
}
```

Чтобы можно было загрузить файлы, в которых мы не заменили команду `color` на четверку новых команд, изменим этот блок на следующий.

```
else if (cmd == "color") { // цвет линии
    float r, g, b;
    s >> r >> g >> b; // считываем три составляющие цвета
    ambient = glm::vec3(r, g, b) / 255.f;
    diffuse = ambient;
    specular = ambient;
}
```

Кроме этого блока добавим 4 дополнительных для четырех новых команд

```
else if (cmd == "ambient") { // способность излучать окружающий свет
    float r, g, b;
    s >> r >> g >> b; // считываем три составляющие цвета
    ambient = glm::vec3(r, g, b);
}
else if (cmd == "diffuse") { // способность излучать рассеянный свет
    float r, g, b;
    s >> r >> g >> b; // считываем три составляющие цвета
    diffuse = glm::vec3(r, g, b);
}
else if (cmd == "specular") { // // способность излучать блики
    float r, g, b;
    s >> r >> g >> b; // считываем три составляющие цвета
    specular = glm::vec3(r, g, b);
}
else if (cmd == "shininess") { // степень глянцеваемости
    s >> shininess;
}
```

Кроме этого, необходимо внести изменения в блок команды `mesh`, который сейчас имеет следующий вид.

```
1 else if (cmd == "mesh") { // набор мешей
2     std::vector<glm::vec3> vertices; // список точек
3     int N, K; // количество точек и треугольников
4     s >> N >> K;
5     std::string str1; // дополнительная строка для чтения из файла
6     while (N > 0) { // пока не все точки считали
7         std::getline(in, str1); // считываем в str1 из входного файла очередную строку
8         // так как файл корректный, то на конец файла проверять не нужно
9         if ((str1.find_first_not_of("\t\r\n") != std::string::npos) && (str1[0] != '#')) {
10             // прочитанная строка не пуста и не комментарий
11             // значит в ней тройка координат
12             float x, y, z; // переменные для считывания координат вершины
13             float nx, ny, nz; // переменные для считывания координат нормали
14             std::stringstream s1(str1); // еще один строковый поток из строки str1
15             s1 >> x >> y >> z;
16             s1 >> nx >> ny >> nz;
17             // добавляем точку в список
18             vertices.push_back({ glm::vec3(x, y, z), glm::vec3(nx, ny, nz) });
19             N--; // уменьшаем счетчик после успешного считывания точки
20         }
21     }
```

```

21 }
22 std::vector<GLuint> indices; // список индексов вершин треугольников
23 while (K > 0) { // пока не считали все треугольники
24     std::getline(in, str1); // считываем в str1 из входного файла очередную строку
25     // так как файл корректный, то на конец файла проверять не нужно
26     if ((str1.find_first_not_of(" \t\r\n") != std::string::npos) && (str1[0] != '#')) {
27         // прочитанная строка не пуста и не комментарий
28         // значит в ней тройка индексов вершин треугольника
29         GLuint x; // переменная для считывания
30         std::stringstream s1(str1); // еще один строковый поток из строки str1
31         for (int i = 0; i < 3; i++) { // три раза
32             s1 >> x; // считываем индекс
33             indices.push_back(x); // добавляем индекс в список indices
34         }
35         K--; // уменьшаем счетчик после успешного считывания точки
36     }
37 }
38 // все точки и индексы считаны, генерируем меш и кладем его в список figure
39 figure.push_back(mesh(vertices, indices, glm::vec3(r, g, b) / 255.f));
40 }

```

Изменения здесь касаются только последней строки, в которой при вызове конструктора нужно заменить вектор цвета на структуру, описывающую материал. То есть, вместо

```
figure.push_back(mesh(vertices, indices, glm::vec3(r, g, b) / 255.f));
```

нужно написать

```
figure.push_back(mesh(vertices, indices, {ambient, diffuse, specular, shininess}));
```

Процедура `readFromFile` (её части, используемые в проекте: исключены блоки и переменные, соответствующие командам `path`, `thickness`) примет вид.

```

1 void readFromFile(const char* fileName) { // чтение сцены из файла fileName
2     // объявление и открытие файла
3     std::ifstream in;
4     in.open(fileName);
5     if (in.is_open()) {
6         // файл успешно открыт
7         models.clear(); // очищаем имеющийся список рисунков
8         // временные переменные для чтения из файла
9         glm::mat4 M = glm::mat4(1.f); // матрица для получения модельной матрицы
10        glm::mat4 initM; // матрица для начального преобразования каждого рисунка
11        std::vector<glm::mat4> transforms; // стек матриц преобразований
12        std::vector<mesh> figure; // список мешей очередной модели
13        glm::vec3 diffuse = glm::vec3(0, 0, 0);
14        glm::vec3 ambient = glm::vec3(0, 0, 0);
15        glm::vec3 specular = glm::vec3(0, 0, 0); // составляющие цвета
16        float shininess = 1;
17        std::string cmd; // строка для считывания имени команды
18        // непосредственно работа с файлом
19        std::string str; // строка, в которую считываем строки файла
20        std::getline(in, str); // считываем из входного файла первую строку
21        while (in) { // если очередная строка считана успешно
22            // обрабатываем строку
23            if ((str.find_first_not_of(" \t\r\n") != std::string::npos) && (str[0] != '#')) {
24                // прочитанная строка не пуста и не комментарий
25                std::stringstream s(str); // строковый поток из строки str
26                s >> cmd;
27                if (cmd == "camera") { // положение камеры
28                    float x, y, z;
29                    s >> x >> y >> z; // координаты точки наблюдения
30                    S = glm::vec3(x, y, z);
31                    s >> x >> y >> z; // точка, в которую направлен вектор наблюдения
32                    P = glm::vec3(x, y, z);
33                    s >> x >> y >> z; // вектор направления вверх
34                    u = glm::vec3(x, y, z);
35                }

```

```

36     else if (cmd == "screen") { // положение окна наблюдения
37         s >> fovy_work >> aspect >> near_view >> far_view; // параметры команды
38         fovy = glm::radians(fovy_work); // перевод угла из градусов в радианты
39     }
40     else if (cmd == "color") { // цвет линии
41         float r, g, b;
42         s >> r >> g >> b; // считываем три составляющие цвета
43         ambient = glm::vec3(r, g, b) / 255.f;
44         diffuse = ambient;
45         specular = ambient;
46     }
47     else if (cmd == "diffuse") { // цвет линии
48         float r, g, b;
49         s >> r >> g >> b; // считываем три составляющие цвета
50         diffuse = glm::vec3(r, g, b);
51     }
52     else if (cmd == "ambient") { // цвет линии
53         float r, g, b;
54         s >> r >> g >> b; // считываем три составляющие цвета
55         ambient = glm::vec3(r, g, b);
56     }
57     else if (cmd == "specular") { // цвет линии
58         float r, g, b;
59         s >> r >> g >> b; // считываем три составляющие цвета
60         specular = glm::vec3(r, g, b);
61     }
62     else if (cmd == "shininess") { // толщина линии
63         s >> shininess; // считываем значение толщины
64     }
65     else if (cmd == "mesh") { // набор мешей
66         std::vector<vertex> vertices; // список точек
67         int N, K; // количество точек и треугольников
68         s >> N >> K;
69         std::string str1; // дополнительная строка для чтения из файла
70         while (N > 0) { // пока не все точки считали
71             std::getline(in, str1); // считываем в str1 из входного файла очередную строку
72             // так как файл корректный, то на конец файла проверять не нужно
73             if ((str1.find_first_not_of("\t\r\n") != std::string::npos) && (str1[0] != '#')) {
74                 // прочитанная строка не пуста и не комментарий
75                 // значит в ней тройка координат
76                 float x, y, z; // переменные для считывания
77                 float nx, ny, nz;
78                 std::stringstream s1(str1); // еще один строковый поток из строки str1
79                 s1 >> x >> y >> z;
80                 s1 >> nx >> ny >> nz;
81                 // добавляем точку в список
82                 vertices.push_back({ glm::vec3(x, y, z), glm::vec3(nx, ny, nz) });
83                 N--;
84             }
85         }
86         std::vector<GLuint> indices; // список индексов вершин треугольников
87         while (K > 0) { // пока не считали все треугольники
88             std::getline(in, str1); // считываем в str1 из входного файла очередную строку
89             // так как файл корректный, то на конец файла проверять не нужно
90             if ((str1.find_first_not_of("\t\r\n") != std::string::npos) && (str1[0] != '#')) {
91                 // прочитанная строка не пуста и не комментарий
92                 // значит в ней тройка индексов вершин треугольника
93                 GLuint x; // переменная для считывания
94                 std::stringstream s1(str1); // еще один строковый поток из строки str1
95                 for (int i = 0; i < 3; i++) { // три раза
96                     s1 >> x; // считываем индекс
97                     indices.push_back(x); // добавляем индекс в список indices
98                 }
99                 K--; // уменьшаем счетчик после успешного считывания точки
100             }
101         }
102         // все точки и индексы считаны, генерируем меш и кладем его в список figure
103         figure.push_back(mesh(vertices, indices, {ambient, diffuse, specular, shininess}));
104     }
105     else if (cmd == "model") { // начало описания нового рисунка
106         float mVcx, mVcy, mVcz, mVx, mVy, mVz; // параметры команды model
107         s >> mVcx >> mVcy >> mVcz >> mVx >> mVy >> mVz; // считываем значения переменных
108         float S = mVx / mVy < 1 ? 2.f / mVy : 2.f / mVx;
109         // сдвиг точки привязки из начала координат в нужную позицию

```

```

110         // после которого проводим масштабирование
111         initM = glm::scale(glm::vec3(S)) * glm::translate(glm::vec3(-mVcx, -mVcy, -mVcz));
112         figure.clear();
113     }
114     else if (cmd == "figure") { // формирование новой модели
115         models.push_back(model(figure, M * initM));
116     }
117     else if (cmd == "translate") { // перенос
118         float Tx, Ty, Tz; // параметры преобразования переноса
119         s >> Tx >> Ty >> Tz; // считываем параметры
120         M = glm::translate(glm::vec3(Tx, Ty, Tz)) * M; // добавляем перенос к общему преобразованию
121     }
122     else if (cmd == "scale") { // масштабирование
123         float S; // параметр масштабирования
124         s >> S; // считываем параметр
125         M = glm::scale(glm::vec3(S)) * M; // добавляем масштабирование к общему преобразованию
126     }
127     else if (cmd == "rotate") { // поворот
128         float theta; // угол поворота в градусах
129         float nx, ny, nz; // координаты направляющего вектора оси вращения
130         s >> theta >> nx >> ny >> nz; // считываем параметры
131         // добавляем вращение к общему преобразованию
132         M = glm::rotate(glm::radians(theta), glm::vec3(nx, ny, nz)) * M;
133     }
134     else if (cmd == "pushTransform") { // сохранение матрицы в стек
135         transforms.push_back(M); // сохраняем матрицу в стек
136     }
137     else if (cmd == "popTransform") { // откат к матрице из стека
138         M = transforms.back(); // получаем верхний элемент стека
139         transforms.pop_back(); // выкидываем матрицу из стека
140     }
141 }
142 // считываем очередную строку
143 std::getline(in, str);
144 }
145 initWorkPars();
146 }
147 }

```

### Изменение шейдеров

Цвет модели мы передавали в качестве значения uniform-переменной. Теперь же вместо цвета нам необходимо загружать во фрагментный шейдер компоненты материала.

Внесем изменения в файл `Fragment.glsl`.

После описания структуры `Light` добавим описание структуры `Material`, подобное описанию `Material` в файле `Fragment.h`.

```

struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};

```

Вместо uniform-переменной `pathColor` объявим uniform-переменную `material` типа `Material`.

```

uniform Material material;

```

При вычислении значения `powOfCos` вместо «магического» числа 32 используем значение поля `shininess` структуры `material`.

```
powOfCos = pow(max(dot(viewDir, lightDirR), 0.0), material.shininess);
```

При вычислении значений `ambient`, `diffuse` и `specular` вместо `pathColor` будем использовать соответствующие компоненты материала.

```
vec3 ambient = light.ambient * material.ambient;
vec3 diffuse = light.diffuse * cosTheta * material.diffuse;
vec3 specular = light.specular * powOfCos * material.specular;
```

Файл шейдера `Fragment.glsl` примет следующий вид.

```
1  #version 330 core
2
3  struct Light {
4      vec3 ambient;
5      vec3 diffuse;
6      vec3 specular;
7  };
8
9  struct Material {
10     vec3 ambient;
11     vec3 diffuse;
12     vec3 specular;
13     float shininess;
14 };
15
16 in vec3 fragPos;
17 in vec3 fragNorm;
18
19 out vec4 color;
20
21 uniform Material material;
22 uniform Light light;
23 uniform vec3 lightPos;
24 uniform vec3 viewPos;
25
26 void main() {
27     // нормализуем полученный вектор нормали
28     vec3 norm = normalize(fragNorm);
29     // получаем нормализованный вектор направления на источник света
30     vec3 lightDir = normalize(lightPos - fragPos);
31     // вычисляем косинус угла между полученными векторами
32     float cosTheta = max(dot(norm, lightDir), 0.0);
33
34     float powOfCos; // коэффициент бликовой освещенности
35     if (cosTheta > 0.0) { // если cosTheta <= 0, бликовая освещенность равна 0
36         // получаем нормализованный вектор направления в точку наблюдения
37         vec3 viewDir = normalize(viewPos - fragPos);
38         // получаем вектор отраженного света
39         vec3 lightDirR = reflect(-lightDir, norm);
40         powOfCos = pow(max(dot(viewDir, lightDirR), 0.0), material.shininess);
41     }
42     else
43         powOfCos = 0.0;
44
45
46     vec3 ambient = light.ambient * material.ambient;
47     vec3 diffuse = light.diffuse * cosTheta * material.diffuse;
48     vec3 specular = light.specular * powOfCos * material.specular;
49
50     vec3 result = ambient + diffuse + specular;
51     color = vec4(result, 1.0);
52 }
```

### Изменения в шейдерной программе

Опять увеличилось количество uniform-переменных. Объявим их в блоке декларации uniform-переменных.

```
shaderProgram.useUniform("material.ambient");  
shaderProgram.useUniform("material.diffuse");  
shaderProgram.useUniform("material.specular");  
shaderProgram.useUniform("material.shininess");
```

Осталось осуществить загрузку значений этих переменных. Но прежде чем делать это, добавим лишнюю перегрузку для метода `setUniform`. Значение поля `shininess` представляет из себя вещественное число, а мы пока не определили варианта метода `setUniform`, которому вторым аргументом было бы передано вещественное число.

Обратимся к файлу `Shader.h`. В конце описания класса `program` добавим дополнительное определение метода `setUniform`.

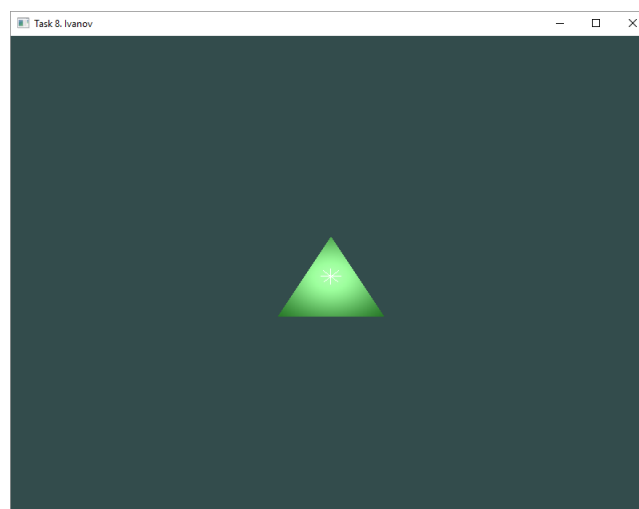
```
void setUniform(std::string uniformName, float value) {  
    // загрузка одного вещественного числа  
    glUniform1f(uniforms[uniformName], value);  
}
```

Теперь вернемся в файл `Main.cpp` к основному циклу в функции `main`.

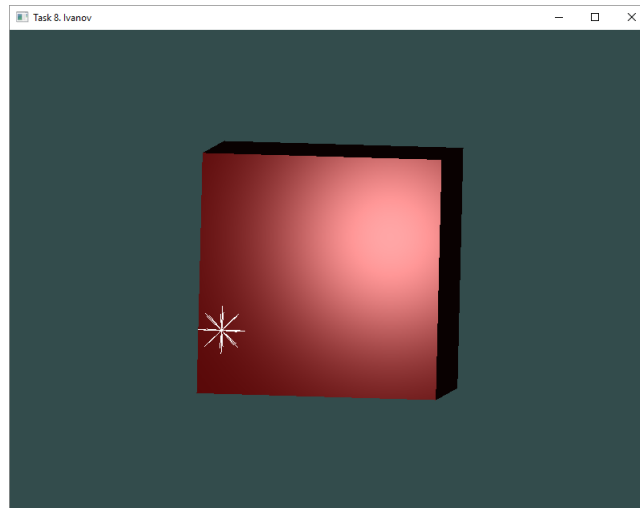
Внутри цикла `for` с параметром `i` вместо загрузки переменной `pathColor` добавим загрузку значений компонент материала `i`-й модели.

```
shaderProgram.setUniform("material.ambient", figure[i].material.ambient);  
shaderProgram.setUniform("material.diffuse", figure[i].material.diffuse);  
shaderProgram.setUniform("material.specular", figure[i].material.specular);  
shaderProgram.setUniform("material.shininess", figure[i].material.shininess);
```

Проект теперь можно запустить. «Изумрудный» треугольник будет выглядеть примерно так.



а «рубиновый» куб (после некоторого поворота) вот так



К заданию 8 приложены обновленные файлы `car.txt` и `engine.txt`, загрузив которые Вы сможете наблюдать (при включенной перспективной проекции) как освещение влияет на восприятие многогранника, если в каждой вершине многогранника задать вектор нормали как среднее между нормальными граней, сходящихся в вершине.

## 2.6 Передвижение источника света

В последнем небольшом разделе этого урока добавим в сцену немного движения.

До сих пор только наблюдатель мог передвигаться по сцене. Но сама сцена оставалась статичной и взаимное расположение объектов сцены не менялось. Чутьочку изменим это состояние дел.

Движение объекта среди других объектов сцены означает изменение его положения в мировой системе координат, т. е. изменение его модельного преобразования, заданного модельной матрицей.

Добавим динамики для источника света: пусть источник света вращается вокруг начала мировой системы координат относительно оси, заданной вектором  $(0, 1, 1)$ . Чтобы организовать это вращение, определим матрицу вращения, зависящую от времени работы приложения.

В цикле отрисовки процедуры `main` файла `Main.cpp` после вызова `glClear` опишем переменную `move`, которой присвоим матрицу вращения вокруг оси, заданной упомянутой выше вектором, на угол  $10^\circ$  умноженный на количество времени работы приложения. Количество времени можно получить с помощью функции `glfwGetTime`.

```
glm::mat4 move = glm::rotate((float)glfwGetTime() * glm::radians(10.0f), glm::vec3(0.0f, 1.0f, 1.0f));
```

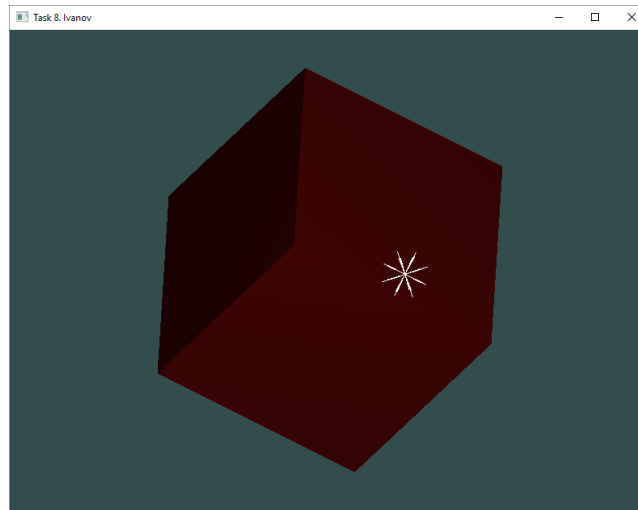
На эту матрицу будем умножать модельную матрицу для источника света. Опишем вспомогательную матрицу, которой присвоим это произведение.

```
glm::mat4 lightM1 = move * lightM;
```

Теперь в выражениях при передаче значений `uniform`-переменной `lightPos` шейдерной программы `shaderProgram` и `clipView` шейдерной программы `lightShaderProgram` следует использовать `lightM1` вместо `lightM`.

Запуск приложения приведет к тому, что источник света будет совершать по сцене круговые движения. Поэтому, если загрузить файл с описанием куба, то теперь, в результате

движения источника света, можем наблюдать ситуацию, когда освещены три грани куба, а не одна.



Процедура main файла Main.cpp примет примерно следующий вид.

```

1  int main () {
2      glfwInit(); // Инициализация GLFW
3      // Проведение начальных установок GLFW
4      // Задается минимальная требуемая версия OpenGL.
5      glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3); // Номер до десятичной точки
6      glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3); // Номер после десятичной точки
7      // Используем только средства указанной версии без совместимости с более ранними
8      glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
9
10     // Создаем окно
11     GLFWwindow* window = glfwCreateWindow(800, 600, "Task 8. Ivanov", NULL, NULL);
12     if (window == NULL) { // если ссылка на окно не создана
13         std::cout << "Вызов glfwCreateWindow закончился неудачей." << std::endl;
14         glfwTerminate(); // завершить работу GLFW
15         return -1;      // завершить программу
16     }
17     glfwMakeContextCurrent(window); // делаем окно window активным (текущим)
18     // Назначение обработчика события Resize
19     glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
20     // Назначение обработчика нажатия клавиш
21     glfwSetKeyCallback(window, key_callback);
22     // назначение обработчика положения курсора
23     glfwSetCursorPosCallback(window, cursorPosSave_callback);
24     glfwSetScrollCallback(window, scroll_callback);
25     glfwSetMouseButtonCallback(window, mouse_button_callback);
26
27     // Инициализация GLAD
28     if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
29         std::cout << "Не удалось загрузить GLAD" << std::endl;
30         glfwTerminate(); // завершить работу GLFW
31         return -1;      // завершить программу
32     }
33
34     // сообщаем диапазон координат в окне
35     // (0, 0) - координаты левого нижнего угла, 800x600 - размеры окна в пикселах
36     glViewport(0, 0, 800, 600);
37
38     shader vertexShader("Vertex.glsl", GL_VERTEX_SHADER); // вершинный шейдер
39     shader fragmentShader("Fragment.glsl", GL_FRAGMENT_SHADER); // фрагментный шейдер
40     // Шейдерная программа
41     program shaderProgram(vertexShader, fragmentShader);
42
43     // удаление шейдерных объектов
44     vertexShader.clear();
45     fragmentShader.clear();

```



```

46
47 // вершинный шейдер для источника света
48 fragmentShader = shader("LightVertex.glsl", GL_VERTEX_SHADER);
49 // фрагментный шейдер для источника света
50 fragmentShader = shader("LightFragment.glsl", GL_FRAGMENT_SHADER);
51 // шейдерная программа для источника света
52 program lightShaderProgram(vertexShader, fragmentShader);
53
54 // удаление шейдерных объектов
55 vertexShader.clear();
56 fragmentShader.clear();
57
58 // декларируем использование uniform-переменных
59 shaderProgram.useUniform("light.ambient");
60 shaderProgram.useUniform("light.diffuse");
61 shaderProgram.useUniform("light.specular");
62 shaderProgram.useUniform("clipView");
63 shaderProgram.useUniform("modelView");
64 shaderProgram.useUniform("modelInv");
65 shaderProgram.useUniform("lightPos");
66 shaderProgram.useUniform("viewPos");
67 shaderProgram.useUniform("material.ambient");
68 shaderProgram.useUniform("material.diffuse");
69 shaderProgram.useUniform("material.specular");
70 shaderProgram.useUniform("material.shininess");
71
72 lightShaderProgram.useUniform("clipView");
73 lightShaderProgram.useUniform("pathColor");
74
75
76 //=====
77 //   НАБОР ИСХОДНЫХ ДАННЫХ ДЛЯ ОТРИСОВКИ ИСТОЧНИКА СВЕТА
78 //=====
79 GLfloat lightVertices[] = {
80     -0.1f, 0.1f, 0.1f,
81     0.1f, 0.1f, 0.1f,
82     0.1f, -0.1f, 0.1f,
83     0.1f, -0.1f, -0.1f,
84     0.1f, 0.1f, -0.1f,
85     0.1f, 0.1f, 0.1f,
86     0.07071f, 0.07071f, 0.1f,
87     -0.07071f, -0.07071f, 0.1f,
88     -0.07071f, 0.07071f, 0.1f,
89     0.07071f, -0.07071f, 0.1f,
90     0.07071f, 0.1f, 0.07071f,
91     -0.07071f, 0.1f, -0.07071f,
92     -0.07071f, 0.1f, 0.07071f,
93     0.07071f, 0.1f, -0.07071f,
94     0.1f, 0.07071f, 0.07071f,
95     0.1f, -0.07071f, -0.07071f,
96     0.1f, -0.07071f, 0.07071f,
97     0.1f, 0.07071f, -0.07071f,
98     0.05774f, 0.05774f, 0.05774f,
99     -0.05774f, -0.05774f, -0.05774f,
100     -0.05774f, -0.05774f, 0.05774f,
101     0.05774f, 0.05774f, -0.05774f,
102     -0.05774f, 0.05774f, 0.05774f,
103     0.05774f, -0.05774f, -0.05774f,
104     0.05774f, -0.05774f, 0.05774f,
105     -0.05774f, 0.05774f, -0.05774f
106 };
107
108 GLuint lightVertexArray; // объект вершинного массива
109 // создаем вершинный массив, идентификатор которого присваиваем vertexArray
110 glGenVertexArrays(1, &lightVertexArray);
111 glBindVertexArray(lightVertexArray); // делаем активным вершинный массив
112
113 GLuint lightVertexBuffer; // идентификатор буферного объекта
114 // создаем буферный объект, идентификатор которого присваиваем vertexBuffer
115 glGenBuffers(1, &lightVertexBuffer);
116 // привязка vertexBuffer к GL_ARRAY_BUFFER
117 glBindBuffer(GL_ARRAY_BUFFER, lightVertexBuffer);
118 // в буфер, привязанный к GL_ARRAY_BUFFER копируем содержимое vertices
119 glBufferData(GL_ARRAY_BUFFER, sizeof(lightVertices), lightVertices, GL_STATIC_DRAW);

```

```

120 // описание расположения параметра вершинного шейдера в вершинном буфере
121 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
122 glEnableVertexAttribArray(0); // включение параметра 0 для шейдера
123 glBindBuffer(GL_ARRAY_BUFFER, 0); // отвязка буферного объекта
124 glBindVertexArray(0); // отключение вершинного массива
125
126 // перемещение источника света из начала координат в точку (0, 0, 5)
127 lightM = glm::translate(glm::vec3(0, 0, 5));
128
129 readFromFile("triangle.txt");
130 glEnable(GL_DEPTH_TEST);
131 //glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
132 while (!glfwWindowShouldClose(window)) { // пока окно window не должно закрыться
133     glClearColor(0.2f, 0.3f, 0.3f, 1.0f); // назначаем цвет заливки
134     // очищаем буфер кадра и буфер глубины (z-буфер)
135     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
136
137     glm::mat4 move = glm::rotate((float)glfwGetTime() * glm::radians(10.0f), glm::vec3(0.0f, 1.0f, 1.0f));
138     glm::mat4 lightM1 = move * lightM;
139
140     shaderProgram.use(); // шейдерную программу shaderProgram делаем активной
141
142     glm::mat4 proj; // матрица перехода в пространство отсечения
143     switch (pType) {
144     case Ortho: // прямоугольная проекция
145         proj = glm::ortho(l, r, b, t, n, f);
146         break;
147     case Frustum: // перспективная проекция с Frustum
148         proj = glm::frustum(l, r, b, t, n, f);
149         break;
150     case Perspective: // перспективная проекция с Perspective
151         proj = glm::perspective(fovy_work, aspect_work, n, f);
152         break;
153     }
154     glm::mat4 C = proj * T; // матрица перехода от мировых координат в пространство отсечения
155     shaderProgram.setUniform("light.ambient", glm::vec3(0.2f, 0.2f, 0.2f));
156     shaderProgram.setUniform("light.diffuse", glm::vec3(0.5f, 0.5f, 0.5f));
157     shaderProgram.setUniform("light.specular", glm::vec3(1.f, 1.f, 1.f));
158     shaderProgram.setUniform("lightPos", glm::vec3(lightM1 * glm::vec4(0, 0, 0, 1)));
159
160     shaderProgram.setUniform("viewPos", glm::vec3(glm::inverse(T) * glm::vec4(0, 0, 0, 1)));
161
162     for (int k = 0; k < models.size(); k++) { // цикл по моделям
163         std::vector<mesh> figure = models[k].figure; // список мешей очередной модели
164         glm::mat4 TM = C * models[k].modelM; // матрица общего преобразования модели
165         // пересылка матриц в переменные шейдерной программы
166         shaderProgram.setUniform("clipView", TM);
167         shaderProgram.setUniform("modelView", models[k].modelM);
168         shaderProgram.setUniform("modelInv", glm::transpose(glm::inverse(models[k].modelM)));
169         for (int i = 0; i < figure.size(); i++) {
170             // пересылка цвета линии в переменную pathColor шейдерной программы
171             shaderProgram.setUniform("material.ambient", figure[i].material.ambient);
172             shaderProgram.setUniform("material.diffuse", figure[i].material.diffuse);
173             shaderProgram.setUniform("material.specular", figure[i].material.specular);
174             shaderProgram.setUniform("material.shininess", figure[i].material.shininess);
175             glBindVertexArray(figure[i].vertexArray); // делаем активным вершинный массив i-го меша
176             // отрисовка набора треугольников по буферу индексов
177             glDrawElements(GL_TRIANGLES, figure[i].indices.size(), GL_UNSIGNED_INT, 0);
178             glBindVertexArray(0); // отключаем вершинный массив
179         }
180     }
181
182     lightShaderProgram.use(); // делаем активной программу для источника света
183     // матрица перехода в пространство отсечения
184     lightShaderProgram.setUniform("clipView", C * lightM1);
185     // белый цвет
186     lightShaderProgram.setUniform("pathColor", glm::vec3(1.0f));
187     glBindVertexArray(lightVertexArray); // делаем активным вершинный массив
188     glDrawArrays(GL_LINES, 0, 26); // рисуем 13 отрезков (в массиве 26 точек)
189     glBindVertexArray(0); // отключаем вершинный массив
190
191     glfwSwapBuffers(window); // поменять местами буферы изображения
192     glfwPollEvents(); // проверить, произошли ли какие-то события
193 }

```

```
194 glwfTerminate(); // завершить работу GLFW
195
196 return 0;
197 }
```

## 2.7 Задание для самостоятельной работы

### Задание 8

1. Создайте проект, включающий в себя все, что было описано выше в качестве примера. Проект должен называться Вашей фамилией, записанной латинскими буквами. Основное окно приложения должно называться *Task 8. Ivanov*, где фамилия *Ivanov* должна быть заменена на Вашу, записанную латинскими буквами.
2. Как уже отмечалось выше, установка

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

приводит к тому, что изображаемые треугольники выводятся только в виде контуров. Вызов

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

включает отрисовку заполненных цветом многоугольников. Добавьте в приложение реакцию на нажатие клавиши **0** (ноль): включение/отключение заполнения отрисовываемых многоугольников.

3. Добавьте в приложение дополнительный источник света с первоначальным положением в точке  $(7, 5, -4)$  мировой системы координат.



Для дополнительного источника света следует ввести свою модельную матрицу и второй набор uniform-переменных, определяющих его свойства. Результирующий цвет фрагмента, освещенного несколькими источниками, должен вычисляться как сумма цветов, полученных в результате освещения каждым из источников.

Для отрисовки положения дополнительного источника света следует воспользоваться уже имеющимся вершинным массивом `lightVertexArray`.

Назначьте реакции на нажатия клавиш для передвижения дополнительного источника света в системе координат наблюдателя

- **4/Shift-4** — сдвиг источника света на 0.3 в положительном/отрицательном направлении оси  $Oz$  текущей системы координат наблюдателя;
- **5/Shift-5** — сдвиг источника света на 0.3 в положительном/отрицательном направлении оси  $Ox$  текущей системы координат наблюдателя;
- **6/Shift-6** — сдвиг источника света на 0.3 в положительном/отрицательном направлении оси  $Oy$  текущей системы координат наблюдателя;



Нажатия клавиш должны приводить к изменению модельной матрицы для источника света (т. е. матрицы перехода из модельной системы координат к мировой). Но так как преобразования должны производиться в системе координат наблюдателя, то они должны складываться из преобразований перехода из мировой системы координат к системе координат наблюдателя, необходимого сдвига и обратного перехода к мировой системе координат.

4. Добавьте в приложение реакцию на нажатие клавиш **7/Shift-7** приводящую к увеличению/уменьшению интенсивности компонент `diffuse` и `specular` (одновременно) источников света в 1.1 раза.
5. Измените цвет отрисовки источников света: он должен равняться сумме компонент `ambient`, `diffuse` и `specular` источника.
6. Исправьте файл `pyramide.txt` : представьте файл в окончательном формате для загрузки в приложение. Вычислите и добавьте векторы нормалей к каждой из граней. В качестве материала назначьте золото (`gold`). Для того, чтобы можно было бродить по сцене без отсечения при приближении, установите начальное значение `near` равным 0.3 и `far` — 200.
7. Исправьте файл `Geometric3D.txt` : представьте файл в окончательном формате для загрузки в приложение. В качестве векторов нормалей установите вектор  $(0, 0, 1)$ . Подберите различные материалы для элементов изображения. Для того, чтобы можно было бродить по сцене без отсечения при приближении, установите начальное значение `near` равным 0.3 и `far` — 200.



Вместо изображения зайца, можно взять описание головы зайца, полученное в этом уроке. Рисунок, соответствующий Вашему варианту, должен быть представлен полностью.

8. Исправьте файл `Geometric3D-2.txt` : представьте файл в окончательном формате для загрузки в приложение, изменив размещаемые модели на куб и пирамиду из окончательных файлов `cube.txt` и `pyramide.txt` . Для того, чтобы можно было бродить по сцене без отсечения при приближении, установите начальное значение `near` равным 0.3 и `far` — 200.
9. В качестве результата выполнения задания должен быть загружен архив получившегося проекта со стандартным именем.



## 2.8 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1. Что такое буфер индексов? Как он используется? Укажите фрагменты программы, в которых осуществляется общение с таким буфером.
2. Откуда вершинный шейдер получает свои входные параметры? Укажите входные параметры для вершинных шейдеров в Вашей программе.
3. Откуда фрагментный шейдер получает свои входные параметры? Укажите входные параметры для фрагментных шейдеров в Вашей программе.
4. В чем смысл вызова метода `useUniform` класса `program` ?
5. Почему источник света отрисовывается с помощью `glDrawArrays` , а отрисовка элементов трехмерной сцены — с помощью `glDrawElements` ? В чем отличие этих процедур?
6. Что произойдет, если каждую модельную матрицу объектов трехмерной сцены домножить на матрицу `move` справа? Слева? Почему?
7. Зачем нам нужны две шейдерные программы?
8. Почему длина вектора нормали, поступающего во фрагментный шейдер, может быть отлична от единицы? Что означает, что вектор нормали получен в результате растеризации?

9. Откуда берется формула  $(M^{-1})_{3 \times 3}^T \bar{n}$  для вычисления вектора нормали после проведенного преобразования, определенного матрицей  $M$ ?
10. Что произойдет, если компоненту ambient источника света взять равной вектору  $(1, 1, 1)$ ?
11. Что произойдет, если компоненту ambient материала поверхности взять равной вектору  $(1, 1, 1)$ ?
12. Что происходит с изображением, когда увеличивается значение свойства материала shininess?
13. Объясните смысл компонент света ambient, diffuse и specular.
14. Объясните смысл компонент материала ambient, diffuse, specular и shininess.
15. Почему во фрагментном шейдере вычисление значения косинуса угла между векторами мы заменили скалярным произведением этих векторов?
16. Что произойдет с окраской куба, если для каждой вершины в качестве вектора нормали взять среднее значение векторов нормали граней, сходящихся в этой вершине?
17. Как нужно поменять файл triangle.txt, чтобы плоский треугольник казался выпуклым в центре?