МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное

образовательное учреждение высшего образования

**«Челябинский государственный университет»**

**(ФГБОУ ВО «ЧелГУ»)**

Математический факультет

Кафедра вычислительной математики

**КУРСОВАЯ РАБОТА**

Построение с помощью OpenGL изображений сферы с определёнными точками на ней

Выполнил студент

Русин Владислав Сергеевич

группы МП-302

очной формы обучения

направления подготовки (специальности)

01.03.02 Прикладная математика и информатика

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_г.

Научный руководитель

Овчинников Михаил Алексеевич

Доцент

Кандидат физ.-мат. наук

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_г.

Челябинск

2023

Содержание

[Введение 3](#_Toc136593800)

[1. Введение в OpenGL 4](#_Toc136593801)

[1.1. Что такое OpenGL 4](#_Toc136593802)

[1.2. Сравнение современного OpenGL и legacy OpenGL 4](#_Toc136593803)

[1.3. Перед началом работы с OpenGL 6](#_Toc136593804)

[1.4. Базовые понятия OpenGL и основной рабочий цикл 8](#_Toc136593805)

[1.4.1. Vertex Array Object и Vertex Buffer Object 8](#_Toc136593806)

[1.4.2. Шейдеры 9](#_Toc136593807)

[1.4.3. Instanced Drawing 10](#_Toc136593808)

[1.4.4. Основной цикл работы с OpenGL 10](#_Toc136593809)

[2. Программа для построения изображений сферы 12](#_Toc136593810)

[2.1. Используемые библиотеки и сборка проекта 12](#_Toc136593811)

[2.2. Алгоритм построения множества точек сферы 13](#_Toc136593812)

[2.3. Построение изображения сферы и её поворотов 15](#_Toc136593813)

[2.3.1. Поля и конструктор класса Sphere 15](#_Toc136593814)

[2.3.2. Метод void Sphere::SetUpRendering() 18](#_Toc136593815)

[2.3.3. Функция отрисовки и шейдеры 22](#_Toc136593816)

[3. Демонстрация работы программы 26](#_Toc136593817)

[Заключение 28](#_Toc136593818)

[Список литературы 29](#_Toc136593819)

[Приложение 31](#_Toc136593820)

# Введение

Данная работа посвящена современным принципам применения библиотеки OpenGL. Целью работы является создание инструмента для построения изображений сферы, которые могут быть применены для изучения поведения формирующих её точек при поворотах вокруг осей, проходящих через центр сферы.

В ходе написания работы были выполнены следующие задачи: изучены возможности OpenGL, разработан алгоритм визуализации сферы, состоящей из множества точек, и написана программа для визуализации сферы с помощью средств, предоставляемых OpenGL.

# 1. Введение в OpenGL

## 1.1. Что такое OpenGL

OpenGL (Open Graphics Library) – это кроссплатформенный межъязыковой программный интерфейс для написания приложений (API), использующих отображение двумерной и трёхмерной компьютерной графики. OpenGL позволяет исполнять указанные программным кодом операции на графическом ядре компьютера, т.е. передавать данные между центральным процессором системы и видеокартой.

Приведённое выше определение и верно, и не верно одновременно. Дело в том, что на самом деле OpenGL – это не более, чем спецификация, определяющая набор из некоторых функций и то, как они должны себя вести, т.е. устанавливающая правила относительно того, какие должны поступать входные данные и какие данные следует ожидать на выходе. Стандарт разрабатывается промышленным консорциумом Khronos Group, реализация же функций спецификации ложиться на плечи разработчиков, имплементирующих её. Такой подход заставляет разные версии OpenGL как API, написанные разными разработчиками с разными целями, следовать единому стандарту, что позволяет программам, написанным с использованием функций OpenGL, быть независимыми от платформы и от языка программирования, на котором их пишут.

## 1.2. Сравнение современного OpenGL и legacy OpenGL

В 2008 году вышла версия OpenGL 3.0, которая объявила устаревшим так называемый Фиксированный Конвейер Функций (Fixed Function Pipeline), а также связанные с ним функции и константы, которые, начиная с версии 3.1 (2009 год), вовсе начали удаляться из спецификации. Эти устаревшие концепты теперь обозначаются как legacy OpenGL. Хотя на некоторых платформах всё ещё сохраняется обратная совместимость с более старыми версиями OpenGL при использовании режима работы, обозначаемого как профиль совместимости (compatibility profile), разработчики программного обеспечения постепенно отказываются от добавления обратной совместимости, концентрируя свои ресурсы на поддержке современных версий стандарта (функции и концепты самых последних версий доступны на современных видеокартах в режиме работы, обозначаемом как центральный профиль (core profile)).

Есть несколько причин, по которым сейчас следует избегать работы с legacy OpenGL, среди них:

* Проблемы с производительностью

Современный стандарт позволяет передать данные на видеокарту один раз и хранить их там столько, сколько потребуется. Благодаря этому у графического ядра всегда есть быстрый доступ к данным, что положительно влияет на скорость выполнения требуемых операций. При применении же такой устаревшей концепции, как мгновенный режим описания атрибутов вершин (immediate-mode vertex attribute specification), данные приходится передавать на графическое ядро каждый раз, когда они должны быть отрисованы, даже если описываемая геометрия статична и не меняется от кадра к кадру.

* Малая гибкость взаимодействия с API при написании программ

Сейчас широкое применение в стандарте OpenGL получили программы, которые выполняются прямо на графическом ядре компьютера – шейдеры (shaders или shading programs). Их пишут на специально для этого разработанном C-подобном языке GLSL (OpenGL Shading Language). Их преимущество по сравнению с Фиксированным Конвейером Функций, который они были призваны заменить, в гибкости, которую они дают при написании программ. С помощью шейдеров можно создавать изображения, которые старыми методами были либо крайне трудно реализуемы, либо вовсе недостижимы. Среди множества примеров того, что можно реализовать, используя шейдеры, стоит отметить: построение комплексных моделей света, составление карт теней (shadow mapping), создание экземпляров геометрии для отрисовки нескольких схожих объектов за один вызов функции отрисовки (instanced rendering или просто instancing), рендеринг фотореалистичной и стилизованной графики, анимация геометрии.

* Постепенный повсеместный отказ от устаревшего стандарта

Как было сказано выше, разработчики программного обеспечения постепенно отказываются от поддержки старых версий OpenGL, поэтому для сохранения кроссплатформенности приложения следует писать без использования legacy функций и техник. Использование только OpenGL версии 3.0 и выше гарантирует, что программа будет корректно работать не только на всех современных платформах, но и на тех, которые появятся в ближайшем будущем, поскольку со слов самих Khronos Group: "На данный момент нет планов по отказу от большего количества функций."[[1]](#footnote-1)

Пожалуй, единственным преимуществом, которым обладает legacy OpenGL по сравнению с современными версиями, является простота в освоении. Legacy OpenGL предоставляет более высокоуровневый интерфейс для взаимодействия с графическим аппаратным обеспечением. С его помощью можно легко создавать простую двумерную и трёхмерную графику, не заботясь о низкоуровневых деталях. Современная же спецификация требует от разработчика лучшего понимания низкоуровневых процессов отрисовки, что требует большего опыта и времени для освоения, но взамен даёт полный контроль над графическим процессом и позволяет создавать сложные и красивые визуальные эффекты.

## 1.3. Перед началом работы с OpenGL

В этом пункте речь пойдёт о вещах, заботиться о которых придётся преимущественно в тех случаях, когда языком программирования выбран C/C++, поскольку большинство библиотек и модулей, позволяющих работать с функциями OpenGL в других языках, уже предоставляют все необходимые для этого условия.

Прежде чем начать использовать OpenGL в программе, его нужно проинициализировать, т.е. создать контекст и загрузить все функции. На разных платформах инициализация OpenGL происходит по-разному, поэтому универсального подхода к этому шагу нет. Из-за этого, чтобы программа могла запускаться на различных платформах, придётся для каждой прописать отдельную инициализацию. К счастью, делать это вручную не придётся. Существуют библиотеки, которые создадут контекст и загрузят все функции OpenGL за нас, причём это займёт всего пару строк кода в программе, и в итоге она будет платформонезависимой.

Некоторыми из самых популярных библиотек для создания контекста являются GLFW (Graphics Library Framework), freeglut и SDL (Simple DirectMedia Layer). При написании программы, описываемой в данной работе далее, применялся GLFW, который позволяет создавать окна, привязывать к ним контекст OpenGL и обрабатывать события, такие как перемещение курсора мыши и нажатие клавиши на клавиатуре. GLFW уступает SDL в обширности функционала, но является более легковесным и простым в использовании, при этом достаточным для написания сложных программ. GLFW и freeglut весьма схожи, но первый получил более широкое распространение среди профессиональных разработчиков, и вследствие этого найти нужную информацию о нём проще, а опыт работы с ним пригодится с большей вероятностью.

Тремя наиболее широко распространёнными загрузчиками функций OpenGL являются glad, gl3w и GLEW (OpenGL Extension Wrangler). Задача этих библиотек заключается в том, чтобы устанавливать указатели на функции как core OpenGL, так и его расширений во время работы программы (расширения – это дополнительный функционал, который могут добавлять отдельные производители графических процессоров; некоторые популярные расширения имеют шанс со временем стать частью набора "стандартных" расширений). Это нужно для получения доступа к функциям OpenGL версий выше 1.1, и, хотя это можно сделать вручную, загрузчики сильно упрощают эту задачу, которая для непрофессионалов может оказаться совсем не тривиальной. При разработке программы, которая описывается в дальнейшем тексте, был использован загрузчик glad. Названные библиотеки имеют незначительные различия, поэтому выбор между ними сводится к индивидуальным предпочтениям и привычкам.

## 1.4. Базовые понятия OpenGL и основной рабочий цикл

### 1.4.1. Vertex Array Object и Vertex Buffer Object

OpenGL подходит для создания сложной трёхмерной графики и поэтому является достаточно гибким API, однако эта гибкость появляется за счёт того, что разработчику необходимо самостоятельно контролировать многие аспекты работы с графическим процессором. Для более удобной работы с OpenGL были разработаны такие концепции, как Vertex Array Object (VAO) и Vertex Buffer Object (VBO), а также множество других, позволяющих оптимизировать работу с графикой. Приведём некоторые базовые понятия, с которыми столкнёмся в дальнейшем при рассмотрении кода программы. Начнём с уже обозначенных VAO и VBO.

Vertex Array Object представляет собой объект, который содержит информацию о состоянии данных, которые нужно передать графическому ядру для отрисовки. VAO хранит в себе информацию обо всех VBO, которые к нему привязаны (но не хранит их копии, а только ссылается на них), а также о том, как данные, содержащиеся в VBO, должны интерпретироваться при рендеринге.

Vertex Buffer Object – это объект, который хранит в себе всю информацию о вершинах (vertices): координаты, цвет, матрицы поворота и так далее. Эти свойства принято называть атрибутами вершин (vertex attributes), они могут содержаться в одном большом VBO или быть разбиты на несколько меньших по размеру. Проще говоря, VBO содержит информацию о том, что нужно нарисовать, а VAO – о том, как это нарисовать.

### 1.4.2. Шейдеры

Уже упомянутые ранее шейдеры делятся на несколько стадий, идущих в строгом порядке, и у каждой из них есть своё enum имя, с помощью которого к соответствующей стадии можно обратиться из программы:

* Вершинный Шейдер (Vertex Shader) – GL\_VERTEX\_SHADER
* Шейдер Контроля Тесселяции (Tessellation Control Shader) – GL\_TESS\_CONTROL\_SHADER
* Шейдер Оценки Тесселяции (Tessellation Evaluation Shader) – GL\_TESS\_EVALUATION\_SHADER
* Шейдер Геометрии (Geometry Shader) – GL\_GEOMETRY\_SHADER
* Фрагментный Шейдер (Fragment Shader) – GL\_FRAGMENT\_SHADER
* Вычислительный Шейдер (Compute Shader) – GL\_COMPUTE\_SHADER

Во многих случаях при разработке программ, использующих шейдеры, достаточно написать только 2 из них – вершинный и фрагментный.

При вызове функции отрисовки вершинному шейдеру поступает поток вершин (т.е. наборов атрибутов вершин), каждую из которых он обрабатывает отдельно. После окончания обработки вершины она передаётся на следующую стадию Конвейера Отрисовки (Rendering Pipeline). Вершинный шейдер предназначен для применения трансформаций, связанных с положением вершины в пространстве, таких как нормализация координат и применение матриц поворотов, и поскольку вершинный шейдер является первой стадией Конвейера, то используется для подготовки вершины к обработке на более поздних этапах.

Фрагментный шейдер – это стадия, на которой обычно происходит больше всего манипуляций непосредственно над внешним видом отображаемой геометрии. Этот шейдер определяет, как будет выглядеть каждый фрагмент, покрытый рисуемым примитивом. Фрагментом называется набор определённых значений, вычисленных для каждого пикселя (или нескольких пикселей), например цвет и прозрачность. Примитив – это интерпретация, используемая OpenGL, чтобы определить, что из себя представляет набор вершин. Примитивами являются точки, линии, треугольники и их вариации.

### 1.4.3. Instanced Drawing

Instanced drawing — это техника оптимизации рендеринга, которая позволяет повысить производительность при отрисовке большого числа однотипных объектов. Вместо отрисовки каждого объекта отдельно, используется механизм инстанцирования, который позволяет на основе одной модели сгенерировать множество её экземпляров с разными параметрами. Обычно за один раз можно вызвать функцию отрисовки только для одного объекта, но применяя технику instanced drawing, можно за один вызов функции отрисовки отобразить сразу несколько объектов, имеющих общие черты, но и индивидуальные свойства, такие как цвет или координаты. Например, если нужно нарисовать несколько одинаковых сфер, но применить к ним разные цвета, то достаточно для каждой из них сложить в общий VBO по одному трёхмерному вектору, содержащему *r*, *g* и *b* значения нужного цвета, указать частоту обновления атрибута "цвет" равной 1 (т.е. цвет будет обновляться после отрисовки одной сферы), и вызвать специальную для instanced drawing функцию рендера.

### 1.4.4. Основной цикл работы с OpenGL

Основной цикл работы с OpenGL сводится к последовательному выполнению следующих операций:

1. Создание VAO и всех VBO, которые будут к этому VAO привязаны.

2. Привязка VAO – знак для программы, что именно над этим VAO нужно проводить операции, которые последуют далее. Одновременно может быть привязан только один VAO.

3. Привязка VBO и прочих объектов OpenGL к соответствующим типам буферов. Буфер, к которому привязывается VBO, обозначается именем GL\_ARRAY\_BUFFER. К одному буферу одновременно может быть привязан только один объект OpenGL, поэтому при наличии нескольких объектов, которые должны представлять одинаковый тип буфера, нужно привязать сначала один из них, выполнить все нужные операции с ним и только потом привязать следующий (предыдущий объект отвяжется автоматически при привязке нового).

4. Выполнение операций над привязанным VBO, таких как загрузка в него данных, установка указателей на атрибуты вершин и указание частоты их обновления при применении instanced drawing.

5. После выполнения всех операций над объектами OpenGL их следует отвязать, чтобы избежать случайного изменения их состояния.

6. Для отрисовки изображения на экране, нужно сначала привязать соответствующие VAO и шейдерную программу, затем вызвать функцию рендера и отвязать VAO.

# 2. Программа для построения изображений сферы

## 2.1. Используемые библиотеки и сборка проекта

Программа написана на языке программирования C++ с применением функций OpenGL версии 3.3. Список сторонних библиотек, которые использовались при написании программы:

* GLFW
* glad
* GLM
* Dear ImGui

Задачи, которые решают GLFW и glad уже были расписаны выше.

GLM (OpenGL Mathematics) – это математическая библиотека, написанная специально для вычислений при работе с графикой. Дизайн классов и функций, предоставляемых GLM, разработан по спецификациям GLSL с соответствующими соглашениями об именованиях, что делает эту библиотеку особенно удобной для работы с OpenGL.

Dear ImGui – это минималистичная библиотека, позволяющая создавать простой пользовательский интерфейс (user interface, сокращённо UI), работа с которой напоминает работу с legacy OpenGL, что делает её очень лёгкой в освоении.

Рассматриваемая далее программа, пусть и не больших размеров, но всё же разбита на несколько файлов и имеет зависимости от сторонних библиотек. К тому же желательно сохранить кроссплатформенность приложения, но тонкости компиляции различаются на разных платформах. По этим причинам для сборки проекта используется специальная утилита - CMake, позволяющая контролировать процесс компиляции приложений с помощью простых конфигурационных файлов, не зависящих от платформы и компилятора. По правилам, описываемым в этих файлах, в процессе сборки проекта создаются так называемые мейкфайлы (makefiles) и рабочие пространства, нацеленные на компиляцию программы для конкретной платформы с помощью выбранного компилятора.

## 2.2. Алгоритм построения множества точек сферы

Сфера – это геометрическое место точек в пространстве, равноудаленных от некоторой заданной точки. Множество точек, образующих сферу, бесконечно, поэтому нарисовать каждую отдельную точку сферы невозможно, однако можно отобразить конечное количество точек, которые в совокупности дадут представление сферы (далее сферой будет называться именно такое представление, состоящее из конечного множества точек).

При большом количестве точек задание координат вручную для каждой из них является крайне времязатратным процессом (хотя и такую возможность нужно предусмотреть на случай, если пользователь захочет проверить, как ведут себя какие-то конкретные точки на сфере), поэтому был разработан алгоритм для процедурной генерации множества точек сферы.

Если предоставить пользователю возможность самостоятельно выбирать точное количество точек, которые должны быть сгенерированы на сфере, то для равномерного их распределения потребуется разработать сложные правила. Вместо этого проще дать пользователю право контролировать переменную, обозначающую уровень детализации сферы, и определять количество точек программно. Это более простой подход, тем не менее, сохраняющий возможность пользователя создавать приблизительно то количество точек, которое ему нужно.

Для представления точки в программе используется тип glm::vec3 – трёхмерный вектор из библиотеки GLM. Рассмотрим функцию, которая генерирует точки сферы и заполняет ими некоторый контейнер:

void CreateUvSphere(float radius, unsigned int detail\_level, std::vector<glm::vec3> &points\_container);

На вход функции поступает желаемый радиус сферы, уровень детализации и ссылка на контейнер, в который она должна поместить сгенерированные точки. По уровню детализации определяется число вертикальных и горизонтальных сегментов, на которые должна быть разбита сфера:

unsigned int v\_segments\_count = detail\_level + 2;

unsigned int h\_segments\_count = detail\_level + 1;

Так, например, при минимальном уровне детализации, равном 1, число вертикальных и горизонтальных сегментов будет равно 3 и 1 соответственно. Вертикальные сегменты – это промежутки между меридианами сферы, а горизонтальные сегменты – это промежутки между её параллелями. Далее определяется ширина (или шаг) каждого вертикального и горизонтального сегмента, а также число точек на сфере:

float v\_step = 180.0f / h\_segments\_count;

float h\_step = 360.0f / v\_segments\_count;

unsigned int points\_count = v\_segments\_count \* (h\_segments\_count - 1) + 2;

Точки устанавливаются на пересечения параллелей и меридиан сферы, кроме того, на её полюсах ставятся ещё две точки, поэтому их общее число равно количеству меридиан (которое равно v\_segments\_count), помноженному на количество параллелей (равное h\_segments\_count - 1), плюс 2 (точки на полюсах).  
Далее идёт изменение размера контейнера points\_container и вставка в него первого элемента – верхнего полюса сферы, который имеет нулевые координаты на осях *X* и *Z*, а его координата *y* равна радиусу сферы:

points\_container.resize(points\_count);

unsigned int ind = 0;

points\_container[ind++] = {0.0f, radius, 0.0f};

Далее идёт заполнение остальной части контейнера. Для каждой точки положение высчитывается по формулам перехода от сферических координат к декартовым с учётом того, что осью, направленной вверх, является ось *Y*:

* *x = r \* cos(φ) \* cos(θ)*
* *y = r \* sin(θ)*
* *z = r \* sin(φ) \* cos(θ)*

где *r* – радиус сферы, *θ* – угол между осью *Y* и отрезком, соединяющим взятую точку с центром сферы (началом координат), *φ* - угол между осью *X* и проекцией отрезка, соединяющего взятую точку с центром сферы, на плоскость *XY*:

for (int i = 1; i < h\_segments\_count; i++)

{

float v\_angle = -v\_step \* i + 90.0f;

for (int j = 0; j < v\_segments\_count; j++)

{

float h\_angle = h\_step \* j;

points\_container[ind++] =

{

radius \* glm::cos(glm::radians(h\_angle)) \* glm::cos(glm::radians(v\_angle)),

radius \* glm::sin(glm::radians(v\_angle)),

radius \* glm::sin(glm::radians(h\_angle)) \* glm::cos(glm::radians(v\_angle))

};

}

}

Здесь *θ* *=* v\_angle, *φ =* h\_angle. Функции glm::cos() и glm::sin(), возвращающие косинус и синус переданного угла соответственно, принимают значения в радианах, поэтому v\_angle и h\_angle нужно предварительно преобразовать с помощью функции glm::radians().

Последняя точка, всё ещё не занесённая в контейнер, – нижний полюс сферы. Её координаты *x* и *z* такие же, как у верхнего полюса, а *y* равен отрицательному радиусу:

points\_container[ind] = {0.0f, -radius, 0.0f};

На этом алгоритм построения множества точек сферы завершается. Функция вызывается каждый раз, когда пользователь изменяет уровень детализации сферы.

## 2.3. Построение изображения сферы и её поворотов

### 2.3.1. Поля и конструктор класса Sphere

Рассмотрим некоторые поля и методы класса Sphere, представляющего сферу.

Начнём с приватных полей:

* std::vector<glm::vec3> \_base\_points;

\_base\_points хранит в себе те точки, которые генерируются в функции CreateUvSphere() или задаются вручную.

* std::vector<std::pair<Rotation, int>> \_rotations;

Первый элемент пар, содержащихся в векторе \_rotations, имеет тип Rotation – структуры, описывающей поворот сферы. Её члены: float Angle (угол), glm::vec3 Axis (ось), glm::vec3 Color (цвет). Второй элемент пар – это индекс первого ребёнка этого поворота в этом же контейнере. Под "детьми" поворота *R* подразумеваются повороты, которые применяются не к изначальным точкам сферы, хранимым в \_base\_points, а к точкам, полученным после применения поворота *R* к точкам из \_base\_points.

* static const unsigned int \_max\_detail\_level = 40;

max\_detail\_level – максимальный допустимый уровень детализации сферы (его разумно ограничить, чтобы избежать проблем с оптимизацией, которые могут возникнуть при очень большом количестве точек).

* ShaderProgram \_shader;

\_shader – это переменная класса, который представляет из себя шейдерную программу, т.е. скомпилированные и связанные вершинный и фрагментный шейдеры. Это единственный шейдер в программе, он используется для отрисовки точек сферы.

* unsigned int \_VAO;

Это поле не хранит в себе данные VAO, а является его идентификатором (или ссылкой на него). Сам же VAO находится в памяти OpenGL.

* unsigned int \_coords\_VBO;
* unsigned int \_colors\_VBO;
* unsigned int \_rotations\_VBO;
* unsigned int \_visibles\_VBO;

Ссылки на разные VBO, которые используются для хранения информации об изначальной сфере и сферах, порождаемых поворотами. Отдельные VBO для координат точек, их цветов, поворотов и значений типа видимый (1) / невидимый (0), чтобы была возможность включать и отключать отображение сфер.

* int Detail\_level;

Публичное поле, содержащее текущий уровень детализации.

Рассмотрим теперь один из конструкторов класса Sphere и методы, которые он вызывает:

Sphere::Sphere(unsigned int level\_of\_detail, ShaderProgram &&shader);

На вход конструктору поступают начальный уровень детализации (его изменение в дальнейшем не требует создания нового экземпляра класса) и шейдерная программа, которая будет использоваться для отрисовки этой сферы. Значения level\_of\_detail и shader сразу записываются в поля Detail\_level и \_shader соответственно. Первым делом, Detail\_level загоняется в рамки между 1 и \_max\_detail\_level, на тот случай, если он в них не попадает:

Detail\_level = std::min(Detail\_level, int(\_max\_detail\_level));

Detail\_level = std::max(Detail\_level, 1);

Далее устанавливается размер контейнера поворотов. Для этого используется статический метод структуры Rotation, высчитывающий значение на основе максимального допустимого количества детей у одного поворота - Rotation::Max\_children - и максимальной глубины поворотов (на глубине 1 находятся первые повороты, на глубине 2 их дети, на глубине 3 дети детей и так далее) – Rotation::Max\_depth:

\_rotations.resize(Rotation::MaxRotations());

В \_rotations повороты хранятся в следующем порядке:

*[R1, R2, …, Rc, R1,1, R1,2, …, R1,c, R2,1, …, Rc,c, R1,1,1, …, R1,1,c, R1,2,1, … Rc,c,c, …]*, где *Ri*– первые повороты (т.е. такие, которые применяются непосредственно к точкам из \_base\_points), *Ri,j* – ребёнок с номером *j* поворота *Ri*, *Ri,j,k* – ребёнок под номером *k* поворота *Ri,j* и так далее до глубины Rotation::Max\_depth (*i*, *j*, *k ∈ [1, c]*); *c =* Rotation::Max\_children.

Для каждого поворота определяется индекс его первого ребёнка для применений родительских матриц поворота к детям в будущем:

unsigned int ind = 0;

for (unsigned int depth = 1; depth < Rotation::Max\_depth; depth++)

{

unsigned int local\_rotations\_num = pow(Rotation::Max\_children, depth);

for (int i = 0; i < local\_rotations\_num; i++)

{

\_rotations[ind].second = ind - i + i \* Rotation::Max\_children + local\_rotations\_num;

ind++;

}

}

У поворотов, находящихся на максимальной глубине, детей нет, поэтому в качестве индекса первого ребёнка устанавливаем для них -1:

for (; ind < \_rotations.size(); ind++)

\_rotations[ind].second = -1;

Вызываем функцию CreateUvSphere() для заполнения вектора \_base\_points и передаём управление методу SetUpRendering(), который реализует процессы, связанные непосредственно с OpenGL:

CreateUvSphere(1.0f, Detail\_level, \_base\_points);

SetUpRendering();

Работа конструктора на этом завершается.

### 2.3.2. Метод void Sphere::SetUpRendering()

Первым делом этот метод подключает шейдерную программу, которая была передана конструктору сферы:

glUseProgram(\_shader.ID());

Поскольку это единственный шейдер в программе, то можно привязать его прямо сейчас и не возвращаться к нему в дальнейшем. Далее, следуя порядку операций, описанному в пункте **1.4.4.**,создаются VAO и все нужные VBO, сразу после чего VAO привязывается:

glGenVertexArrays(1, &\_VAO);

glGenBuffers(1, &\_coords\_VBO);

glGenBuffers(1, &\_colors\_VBO);

glGenBuffers(1, &\_rotations\_VBO);

glGenBuffers(1, &\_visibles\_VBO);

glBindVertexArray(\_VAO);

Единица, передаваемая функции glGenBuffers(), означает, что мы хотим создать 1 объект VBO, идентификатор которого будет записан в указанную переменную типа unsigned int (или GLuint, который, по сути, является тем же unsigned int, но определённым в библиотеке OpenGL). Числа, большие, чем 1, используются в тех случаях, когда передаётся не одна переменная для хранения ссылки на объект, а массив, в который эти ссылки нужно записать.

Далее к GL\_ARRAY\_BUFFER с помощью вызова функции glBindBuffer() привязывается \_coords\_VBO, данные в который пока не заносятся (это будет сделано в самом конце работы метода SetUpRendering()), но разметка атрибутов уже происходит:

glBindBuffer(GL\_ARRAY\_BUFFER, \_coords\_VBO);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0);

glEnableVertexAttribArray(0);

Функции glVertexAttribPointer() на вход поступают следующие параметры:

* GLuint index – индекс атрибута, который размечается.
* GLint size – количество элементов, определяющих атрибут. Так как мы

описываем атрибут, обозначающий положение точки, которое определяется тремя координатами - *x*, *y* и *z*, - то устанавливаем size равным 3.

* GLenum type – тип данных, который будет представлять каждый элемент

атрибута. GL\_FLOAT – это float, определённый в библиотеке OpenGL.

* GLboolean normalized – имеет смысл только при передаче данных типов,

представляющих целые числа. Определяет, как они будут приводиться к типу GL\_FLOAT. Чаще всего достаточно передавать GL\_FALSE.

* GLsizei stride – определяет общий размер данных в байтах, который

занимают атрибуты одной вершины в привязанном VBO. Это нужно для того, чтобы OpenGL при чтении вершин мог понять, где заканчиваются данные одной вершины и начинаются данные другой. Поскольку \_coords\_VBO хранит в себе только один атрибут вершины – её координаты, - то stride будет равен размеру данных, занимаемому этим атрибутом, и вместо того, чтобы считать stride самим, можно передать 0, который укажет OpenGL на то, что данные упакованы плотно, то есть сразу за координатами одной точки идут координаты другой.

* const GLvoid \*pointer – указывает смещение атрибута в байтах внутри

набора данных об одной вершине. Так как в \_coords\_VBO хранится только один атрибут вершины, то смещение равно 0.

Функция glEnableVertexAttribArray() включает атрибут вершин по указанному индексу, т.е. позволяет манипулировать им в шейдере.

Далее идёт заполнение данных о \_colors\_VBO, которое добавляет новый шаг в описанный выше процесс – занесение данных непосредственно в VBO:

unsigned int max\_spheres = \_rotations.size() + 1;

std::vector<glm::vec3> default\_colors(max\_spheres, Base\_color);

glBindBuffer(GL\_ARRAY\_BUFFER, \_colors\_VBO);

glBufferData(GL\_ARRAY\_BUFFER, max\_spheres \* sizeof(glm::vec3), default\_colors.data(), GL\_STATIC\_DRAW);

glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 0, 0);

glEnableVertexAttribArray(1);

glVertexAttribDivisor(1, 1);

Сначала определяется максимальное число сфер, которые могут выведены на экран. Так как каждый поворот, по сути, тоже является сферой, то max\_spheres будет равно числу поворотов (т.е. размеру вектора \_rotations) плюс 1 (изначальная сфера).

Затем создаётся вектор default\_colors размера max\_spheres, который сразу заполняется одинаковыми элементами – базовым цветом сферы. Этот цвет будет применён ко всем сферам на экране по умолчанию.

Далее с помощью функции glBindBuffer() \_colors\_VBO привязывается к GL\_ARRAY\_BUFFER (в этот момент отвязывается \_coords\_VBO), после чего происходит заполнение данных в VBO. Для этого используется функция glBufferData(), имеющая следующие параметры:

* GLenum target – буффер, в который заносятся данные.
* GLsizeiptr size – размер в байтах, который должен быть выделен под

данные. В рассматриваемом случае размер равен числу сфер, умноженному на число байт, занимаемых одним цветом.

* const GLvoid\* data – указатель на данные, которые нужно поместить в

буффер.

* GLenum usage – подсказка для OpenGL, как и для чего эти данные будут

использоваться. GL\_STATIC\_DRAW означает, что данные будут применены много раз для команд отрисовки, при этом меняться почти не будут.

Дальше идут уже знакомые функции glVertexAttribPointer() и glVertexAttribArray(), индекс атрибута установлен равным единице.

Функция glVertexAttribDivisor() принимает индекс атрибута и присваивает ему указанную частоту обновления. Теперь цвет обновляется не для каждой вершины (как это делают координаты), а для каждого нарисованного экземпляра сферы.

Операции над \_rotations\_VBO ничем не отличаются от операций над \_colors\_VBO за одним небольшим исключением – данные, передаваемые в \_rotations\_VBO представляют собой не вектора, а матрицы 3x3, удобные для представления поворотов:

std::vector<glm::mat3> matrices(max\_spheres, glm::mat3(1.0f));

glBindBuffer(GL\_ARRAY\_BUFFER, \_rotations\_VBO);

glBufferData(GL\_ARRAY\_BUFFER, max\_spheres \* sizeof(glm::mat3), matrices.data(), GL\_STATIC\_DRAW);

for (int i = 0; i < 3; i++)

{

glVertexAttribPointer(2 + i, 3, GL\_FLOAT, GL\_FALSE, sizeof(glm::mat3), (void\*)(i\*sizeof(glm::vec3)));

glEnableVertexAttribArray(2 + i);

glVertexAttribDivisor(2 + i, 1);

}

Атрибут может состоять не более, чем из четырёх элементов, но матрицам 3x3 нужно место под 9. В связи с этим, атрибут типа "матрица" следует воспринимать как 3 отдельных атрибута типа "вектор", хранимых в одном VBO. Это значит, что матрицы будут занимать не только индекс атрибутов 2, но также индексы 3 и 4, и что в параметр stride функции glVertexAttribPointer() нужно передавать не 0, а действительный размер матриц, т.е. sizeof(glm::mat3). Также параметр offset у каждого атрибута будет разный: 0 - у первого (так как до него атрибутов нет), sizeof(glm::vec3) – у второго (так как до него стоит первый атрибут) и *2 \** sizeof(glm::vec3) – у третьего (так как до него стоят первый и второй атрибуты).

Последний VBO, требующий настройки - \_visibles\_VBO, который почти ничем особенным не выделяется:

std::vector<int> actives = std::vector<int>(max\_spheres, 0); actives[0] = 1;

glBindBuffer(GL\_ARRAY\_BUFFER, \_visibles\_VBO);

glBufferData(GL\_ARRAY\_BUFFER, max\_spheres \* sizeof(int), actives.data(), GL\_STATIC\_DRAW);

glVertexAttribIPointer(5, 1, GL\_INT, 0, 0);

glEnableVertexAttribArray(5);

glVertexAttribDivisor(5, 1);

Стоит отметить, что в векторе actives для изначальной сферы стоит 1, а для всех поворотов – 0, т.е. при запуске программы будет видна только одна сфера. Также можно заметить, что вместо функции glVertexAttribPointer() используется её вариация – glVertexAttribIPointer(), особенность которой в том, что она работает только с данными целочисленных типов и оставляет их как есть, т.е. не приводит к типу GL\_FLOAT.

После того как все нужные данные обо всех VBO были записаны в VAO, все эти объекты стоит отвязать, для этого достаточно вызвать функции glBindVertexArray() и glBindBuffer() с нулевыми аргументами:

glBindVertexArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

Последняя операция, которая ещё не была выполнена – занесение данных в \_coords\_VBO. Этот функционал вынесен в отдельный небольшой метод, который вызывается каждый раз при изменении уровня детализации сферы:

void Sphere::UpdateCoordsVBO()

{

    std::size\_t points\_size = \_base\_points.size() \* sizeof(glm::vec3);

    glBindBuffer(GL\_ARRAY\_BUFFER, \_coords\_VBO);

    glBufferData(GL\_ARRAY\_BUFFER, points\_size, \_base\_points.data(), GL\_STATIC\_DRAW);

    glBindBuffer(GL\_ARRAY\_BUFFER, 0);

}

При занесении данных в \_coords\_VBO не требуется привязывать \_VAO, поскольку данные в буфере уже размечены, нужно их только туда положить. Вместе с окончанием работы этого метода завершается и выполнение SetUpRendering().

### 2.3.3. Функция отрисовки и шейдеры

Sphere::Draw() – это метод класса Sphere, ответственный за инициирование процесса отрисовки. Он вызывается из функции main() программы внутри цикла, который срабатывает каждый кадр до тех пор, пока не будет совершён выход из программы.

void Sphere::Draw() const

{

    glBindVertexArray(\_VAO);

    glDrawArraysInstanced(GL\_POINTS, 0, \_base\_points.size(), \_rotations.size() + 1);

    glBindVertexArray(0);

}

Чтобы вывести изображение на экран, нужно сначала привязать VAO, в котором содержится вся нужная информация о рисуемом объекте. Функция glDrawArraysInstanced() (слово instanced в названии указывает на применение техники instancing) – это функция отрисовки изображения, её параметры:

* GLenum mode – задаёт тип примитивов, которые будут нарисованы. В

нашем случае нужны точки, поэтому передаём GL\_POINTS.

* GLint first – индекс элементов из привязанных массивов, с которых

должна начинаться отрисовка. Если указать не 0, а, например, 1, то будут нарисованы все точки, кроме первой, так как атрибуты этой вершины будут пропущены.

* GLsizei count – число элементов из привязанных массивов, которые

должны быть отрисованы. Если указать число меньшее, чем количество вершин, то на экране появятся не все точки.

* GLsizei primcount – число экземпляров, которые нужно отрисовать. Так

как мы хотим видеть изначальную сферу, а так же все её повороты, то устанавливаем это значение равным количеству поворотов плюс 1.

После того, как функция отрисовки была вызвана, следует отвязать VAO, чтобы избежать случайных изменений этого объекта.

Инициализация процесса отрисовки означает, что данные из VAO передаются шейдерам. В программе присутствуют два файла, представляющих шейдеры – sphere.vert и sphere.frag для вершинного и фрагментного шейдеров соответственно. Рассмотрим часть функционала первого из них.

Прежде всего обязательно идёт указание о том, какая должна использоваться версия GLSL:

#version 330 core

Версия 330 core соответствует core profile OpenGL 3.3. Следующим шагом идёт чтение атрибутов вершин из буферов с помощью квалификатора layout (location = value):

layout (location = 0) in vec3 coords;

layout (location = 1) in vec3 color;

layout (location = 2) in mat3 rotation\_matrix;

layout (location = 5) in int is\_visible;

location обозначает индекс, по которому нужно искать атрибут вершины. in – ключевое слово, которое говорит о том, что шейдер ожидает получить эти данные на вход. Далее следует тип атрибута: vec3 – это трёхмерный вектор, каждый элемент которого имеет тип float. Каждый элемент матрицы 3x3 mat3 также имеет тип float. Можно получить матрицу, находящуюся под индексом 2, иначе:

layout (location = 2) in vec3 rotation\_row1;

layout (location = 3) in vec3 rotation\_row2;

layout (location = 4) in vec3 rotation\_row3;

Это возможно, поскольку при занесении данных о \_rotations\_VBO в \_VAO атрибут "матрица" был разбит на 3 отдельных атрибута "вектор".

Помимо атрибутов, шейдеру могут поступать специальные переменные, называемые униформами (uniforms). Их особенность в том, что они общие для всех шейдеров и могут быть определены на любой стадии Конвейера Отрисовки, при этом изменять униформы внутри шейдеров нельзя:

uniform mat4 u\_clip\_matrix;

u\_clip\_matrix – это униформа, содержащая в себе матрицу поворота, полученную при повороте камеры в пространстве. Эта матрица, очевидно, общая для всех объектов на экране, поэтому логично передать её в шейдер с помощью униформы, а не как атрибут вершины.

В шейдере могут быть определены переменные с ключевым словом out, значение которых будет передано дальше по цепочке следующему шейдеру при наличии в нём переменной с таким же названием и объявленной с ключевым словом in. Так, в вершинном шейдере есть переменная v\_color:

out vec4 v\_color;

Забегая наперёд, во фрагментном шейдере есть такая переменная:

in vec4 v\_color;

Внутри каждого шейдера есть своя функция void main(), которой передаётся управление при вызове этого шейдера. Не будем рассматривать функцию main() вершинного шейдера целиком, лишь взглянем в её начало:

if (is\_visible == 0)

{

v\_color = vec4(0.0f);

return;

}

is\_visible равно 1, если рисуемая сфера видима, и 0, если она не видима. В том случае, когда сфера невидима, нет смысла проводить над ней какие-либо манипуляции, поэтому просто выходим из функции с помощью оператора return, записав в вектор v\_color нули на все 4 координаты (цвета в шейдерах представляются четырёхмерными векторами с элементами, соответствующими *r*, *g*, *b*, *w* компонентам цвета, которые принимают значения от 0 до 1 включительно).

Координаты точки после применения к ней поворота высчитываются прямо в шейдере (матрица поворота, передаваемая при отрисовке изначальной сферы, равна единичной матрице, поэтому координаты изначальной сферы остаются прежними):

vec3 rotated\_coords = rotation\_matrix \* coords;

gl\_Position = u\_clip\_matrix \* vec4(rotated\_coords, 1.0f);

В gl\_Position записываются окончательные координаты точки в пространстве, полученные применением к ней необходимых матриц поворота.

Всё, что происходит во фрагментном шейдере – это определение цвета фрагмента с помощью некоторых математических операций, останавливаться на которых не будем. Заметим лишь, что в начале шейдера точно также определяется версия GLSL и объявлены переменные с ключевыми словами in и out:

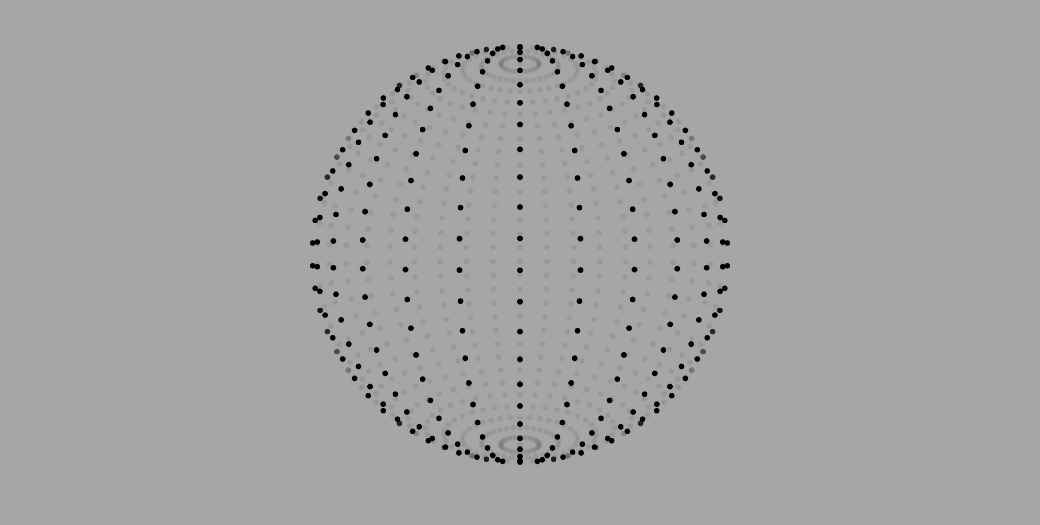
#version 330 core

in vec4 v\_color;

out vec4 f\_color;

# 3. Демонстрация работы программы

Собрав проект и скомпилировав программу, запустим её. Перед нами сфера с уровнем детализации, равным 30, которую с помощью мыши или клавиатуры можно приближать, отдалять, вращать:



: Сфера с уровнем детализации 30

Также доступны два UI окна, написанные с применением библиотеки Dear ImGui (не будем вдаваться в подробности того, как они написаны). Окно слева (окно свойств) позволяет изменять состояние сферы: манипулировать уровнем детализации, цветом, видимостью. Также это окно позволяет изменять свойства поворотов – их угол, ось вращения, цвет и видимость.

Окно справа позволяет увидеть результаты применения каждого поворота к точкам сферы. Если отметить галочкой использование стилизованного текста, то записи в окошке окрасятся в цвет, соответствующий их повороту.

Оба окна можно свободно перемещать по экрану и сворачивать:

Изображение выглядит как текст, снимок экрана, диаграмма, Мультимедийное программное обеспечение

Автоматически созданное описание

: Окна свойств и результатов поворотов

Для примера изменим с помощью окна свойств первый поворот – повернём сферу на 90° вокруг оси *X* и дадим получившейся сфере белый цвет:

Изображение выглядит как текст, снимок экрана, дизайн, искусство

Автоматически созданное описание

: Отображение поворота сферы вокруг оси X на 90°

В окне результатов можем рассмотреть, как меняются точки сферы после применения поворота 1. Так, например, точка *(0, 1, 0)* перешла в положение  
*(0, 0, 1)*.

Применим к результатам поворота 1 другой поворот – 1.1. Его угол равен 90°, а ось вращения совпадает с осью Y. Выключим отображение оригинальной сферы и установим цветом поворота 1.1 – чёрный:

Изображение выглядит как текст, снимок экрана, дизайн, искусство

Автоматически созданное описание

: Применение нового поворота к результатам предыдущего поворота

Окно результатов покажет, как точки, полученные при повороте 1, изменяются после применения поворота 1.1. Так, точка с координатами *(0, 0, 1)* перешла в положение *(1, 0, 0)*.

# Заключение

В итоге была написана программа, являющаяся инструментом для создания трёхмерных изображений сферы и изучения поведения образующих её точек при применении к ним поворотов, оси вращения которых проходят через центр сферы.

Кроме того, в процессе разработки данной программы были изучены основы компьютерной графики и графического программирования с применением возможностей библиотеки OpenGL.

# Список литературы

1. OpenGL // Learn OpenGL URL: https://learnopengl.com/Getting-started/OpenGL (дата обращения: 27.01.2023).
2. Hello Triangle // Learn OpenGL URL: https://learnopengl.com/Getting-started/Hello-Triangle (дата обращения: 27.01.2023).
3. History of OpenGL // khronos.org URL: https://www.khronos.org/opengl/wiki/History\_of\_OpenGL (дата обращения: 28.01.2023).
4. Legacy OpenGL // khronos.org URL: https://www.khronos.org/opengl/wiki/Legacy\_OpenGL (дата обращения: 28.01.2023).
5. Fixed Function Pipeline // khronos.org URL: https://www.khronos.org/opengl/wiki/Fixed\_Function\_Pipeline (дата обращения: 28.01.2023).
6. Shader // OpenGL Wiki URL: https://www.khronos.org/opengl/wiki/Shader (дата обращения: 28.01.2023).
7. Vertex Shader // OpenGL Wiki URL: https://www.khronos.org/opengl/wiki/Vertex\_Shader (дата обращения: 29.01.2023).
8. Fragment Shader // OpenGL Wiki URL: https://www.khronos.org/opengl/wiki/Fragment\_Shader (дата обращения: 29.01.2023).
9. Fragment // OpenGL Wiki URL: https://www.khronos.org/opengl/wiki/Fragment (дата обращения: 29.01.2023).
10. Primitive // OpenGL Wiki URL: https://www.khronos.org/opengl/wiki/Primitive (дата обращения: 29.01.2023).
11. Getting Started // OpenGL Wiki URL: https://www.khronos.org/opengl/wiki/Getting\_Started#Downloading\_OpenGL (дата обращения: 30.01.2023).
12. GLFW // glfw.org URL: https://www.glfw.org/ (дата обращения: 30.01.2023).
13. Related toolkits and APIs // OpenGL Wiki URL: https://www.khronos.org/opengl/wiki/Related\_toolkits\_and\_APIs (дата обращения: 30.01.2023).
14. OpenGL Loading Library // OpenGL Wiki URL: https://www.khronos.org/opengl/wiki/OpenGL\_Loading\_Library (дата обращения: 30.01.2023).
15. Vertex Specification // OpenGL Wiki URL: https://www.khronos.org/opengl/wiki/Vertex\_Specification#Vertex\_Array\_Object (дата обращения: 31.01.2023).
16. Generating Meshes of a Sphere // Daniel Sieger URL: https://www.danielsieger.com/blog/2021/03/27/generating-spheres.html (дата обращения: 31.01.2023).

# Приложение

**main.cpp:**

#include <iostream>

#include "glad/gl.h"

#include "GLFW/glfw3.h"

#include "glm/vec2.hpp"

#include "sphere.hpp"

#include "camera.hpp"

#include "ui.hpp"

static const char \*glsl\_version = "#version 330";

static Sphere sphere;

static bool clip\_update\_needed = true;

static glm::vec2 last\_cursor\_pos;

float delta\_time = 0.0f;

void ErrorCallback(int error\_code, const char \*message)

{

std::cout << "ERROR: " << message << "\nERROR CODE: " << error\_code << std::endl;

}

void KeyCallback(GLFWwindow \*window, int key, int scancode, int action, int mods)

{

if (key == GLFW\_KEY\_ESCAPE && action == GLFW\_PRESS)

glfwSetWindowShouldClose(window, GL\_TRUE);

}

void ScrollCallback(GLFWwindow \*window, double x\_offset, double y\_offset)

{

if (y\_offset == 0)

return;

Camera::Zoom(y\_offset > 0 ? int(Camera::Move::IN) : int(Camera::Move::OUT));

sphere.SetCameraDistanceU(Camera::Distance());

clip\_update\_needed = true;

}

void CursorPosCallback(GLFWwindow \*window, double x\_pos, double y\_pos)

{

if (glfwGetMouseButton(window, GLFW\_MOUSE\_BUTTON\_RIGHT) == GLFW\_PRESS)

{

glm::vec2 move(-(x\_pos - last\_cursor\_pos.x) \* Camera::Drag\_sensitivity, (y\_pos - last\_cursor\_pos.y) \* Camera::Drag\_sensitivity);

Camera::Rotate(move.x, move.y);

clip\_update\_needed = true;

}

last\_cursor\_pos = glm::vec2(x\_pos, y\_pos);

}

void WindowSizeCallback(GLFWwindow \*window, int width, int height)

{

glViewport(0, 0, width, height);

Camera::UpdateProjectionMatrix(width, height);

clip\_update\_needed = true;

}

static bool IsPressed(GLFWwindow \*window, int key)

{

bool pressed = glfwGetKey(window, key) == GLFW\_PRESS;

clip\_update\_needed = clip\_update\_needed || pressed;

return pressed;

}

static void ProcessInput(GLFWwindow \*window)

{

if (IsPressed(window, GLFW\_KEY\_RIGHT))

Camera::Rotate(int(Camera::Move::RIGHT), int(Camera::Move::STAY));

if (IsPressed(window, GLFW\_KEY\_LEFT))

Camera::Rotate(int(Camera::Move::LEFT), int(Camera::Move::STAY));

if (IsPressed(window, GLFW\_KEY\_UP))

{

if (IsPressed(window, GLFW\_KEY\_LEFT\_SHIFT) || IsPressed(window, GLFW\_KEY\_RIGHT\_SHIFT))

{

Camera::Zoom(int(Camera::Move::IN));

sphere.SetCameraDistanceU(Camera::Distance());

}

else

{

Camera::Rotate(int(Camera::Move::STAY), int(Camera::Move::UP));

}

}

if (IsPressed(window, GLFW\_KEY\_DOWN))

{

if (IsPressed(window, GLFW\_KEY\_LEFT\_SHIFT) || IsPressed(window, GLFW\_KEY\_RIGHT\_SHIFT))

{

Camera::Zoom(int(Camera::Move::OUT));

sphere.SetCameraDistanceU(Camera::Distance());

}

else

{

Camera::Rotate(int(Camera::Move::STAY), int(Camera::Move::DOWN));

}

}

}

static void TryUpdateClip()

{

if (clip\_update\_needed)

{

sphere.SetClipMatrixU(Camera::ClipSpaceMatrix());

sphere.SetCameraCoordsU(Camera::Position());

clip\_update\_needed = false;

}

}

int main()

{

glfwSetErrorCallback(ErrorCallback);

if (!glfwInit())

{

std::cout << "LAUNCH ERROR: Glfw initialization failed" << std::endl;

return 1;

}

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);

unsigned int width = 1280;

unsigned int height = 720;

GLFWwindow \*window = glfwCreateWindow(width, height, "coursework-2", NULL, NULL);

if (window == NULL)

{

std::cout << "LAUNCH ERROR: Window creation failed" << std::endl;

glfwTerminate();

return 1;

}

glfwMakeContextCurrent(window);

int gl\_version = gladLoadGL(glfwGetProcAddress);

if (gl\_version == 0)

{

std::cout << "LAUNCH ERROR: Glad initialization failed" << std::endl;

glfwTerminate();

return 1;

}

glfwSetKeyCallback(window, KeyCallback);

glfwSetScrollCallback(window, ScrollCallback);

glfwSetCursorPosCallback(window, CursorPosCallback);

glfwSetWindowSizeCallback(window, WindowSizeCallback);

glfwSwapInterval(1);

glViewport(0, 0, width, height);

glEnable(GL\_PROGRAM\_POINT\_SIZE);

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

Camera::UpdateProjectionMatrix(width, height);

Camera::UpdatePosition();

sphere = Sphere(30, {"../shaders/sphere.vert", "../shaders/sphere.frag"});

sphere.SetCameraDistanceU(Camera::Distance());

UI ui(&sphere, window, glsl\_version);

double last\_time = 0.0f;

while (!glfwWindowShouldClose(window))

{

glClearColor(0.65f, 0.65f, 0.65f, 1.0f);

glClear(GL\_COLOR\_BUFFER\_BIT);

ui.BeginFrame();

ui.DrawPropertiesWindow();

ui.DrawRotationsResultsWindow();

sphere.Draw();

ProcessInput(window);

glfwPollEvents();

TryUpdateClip();

ui.EndFrame();

glfwSwapBuffers(window);

double now = glfwGetTime();

delta\_time = now - last\_time;

last\_time = now;

}

ui.Die();

glfwTerminate();

}

**sphere.hpp:**

#pragma once

#include <vector>

#include "glm/vec3.hpp"

#include "glm/mat4x4.hpp"

#include "glm/mat3x3.hpp"

#include "shader\_program.hpp"

static const glm::vec3 default\_color = glm::vec3(0.2f, 0.2f, 0.2f);

struct Rotation

{

public:

float Angle = 0.0f;

// Ось - это вектор, проходящий через центр сферы (начало координат) и вторую точку,

// координаты которой и будут координатами вектора, поэтому назовём её просто Axis

glm::vec3 Axis = glm::vec3(0.0f, 1.0f, 0.0f);

glm::vec3 Color = default\_color;

bool Is\_visible = false;

const glm::mat3& ParentMatrix() const { return \_parent\_matrix; }

static const unsigned int Max\_depth = 2;

static const unsigned int Max\_children = 3;

static unsigned int MaxRotations()

{

unsigned int m\_rotations = 0;

for (int depth = 1; depth <= Max\_depth; depth++)

m\_rotations += pow(Max\_children, depth);

return m\_rotations;

}

friend class Sphere;

private:

glm::mat3 \_parent\_matrix = glm::mat3(1.0f);

};

class Sphere

{

private:

std::vector<glm::vec3> \_base\_points;

std::vector<std::pair<Rotation, int>> \_rotations; // int - индекс первого потомка поворота

static const unsigned int \_max\_detail\_level = 40;

ShaderProgram \_shader;

unsigned int \_VAO;

unsigned int \_coords\_VBO; // Содержит координаты \_base\_points

unsigned int \_colors\_VBO; // Содержит цвет сферы и цвета всех поворотов

unsigned int \_rotations\_VBO; // Содержит матрицы 3x3, задающие повороты (для сферы содержит единичную матрицу в качестве матрицы поворота)

unsigned int \_visibles\_VBO; // Содержит значения члена Is\_visible поворотов и сферы

void SetUpRendering();

void UpdateCoordsVBO();

void PutDataIntoVBO(unsigned int &VBO, std::size\_t offset, std::size\_t size, const void\* data);

void UpdateChildRotations(unsigned int parent\_ind, const glm::mat3 &parent\_matrix);

void SetChildRotationsVisibility(unsigned int parent\_ind, bool is\_visible);

public:

int Detail\_level;

glm::vec3 Base\_color = default\_color;

bool Is\_visible = true;

Sphere() {}

Sphere(const std::vector<glm::vec3> &points, ShaderProgram &&shader);

Sphere(unsigned int level\_of\_detail, ShaderProgram &&shader);

const std::vector<std::pair<Rotation, int>>& Rotations() const { return \_rotations; }

const std::vector<glm::vec3>& BasePoints() const { return \_base\_points; }

int MaxDetailLevel() const { return int(\_max\_detail\_level); }

Rotation& RotationByIndex(unsigned int ind) { return \_rotations[ind].first; } // Позволяет изменить поворот, но не структуру вектора \_rotations

void ChangeVisibility(bool should\_affect\_rotations);

void SetClipMatrixU(const glm::mat4 &value);

void SetCameraCoordsU(const glm::vec3 &value);

void SetCameraDistanceU(float value);

void UpdateSphereShape();

void UpdateSphereBaseColor();

void UpdateRotation(unsigned int ind, bool rotation\_changed, bool color\_changed, std::pair<bool, bool> visibility\_changed);

void Draw() const;

};

**sphere.cpp:**

#include <vector>

#include "glm/vec3.hpp"

#include "glm/mat4x4.hpp"

#include "glm/mat3x3.hpp"

#include "glm/trigonometric.hpp"

#include "glm/gtc/type\_ptr.hpp"

#include "glad/gl.h"

#include "sphere.hpp"

#include "shader\_program.hpp"

void CreateUvSphere(float radius, unsigned int detail\_level, std::vector<glm::vec3> &points\_container);

Sphere::Sphere(const std::vector<glm::vec3> &points, ShaderProgram &&shader) : \_base\_points(points), \_shader(shader), Detail\_level(0)

{

\_rotations.resize(Rotation::MaxRotations());

unsigned int ind = 0;

for (unsigned int depth = 1; depth < Rotation::Max\_depth; depth++)

{

unsigned int local\_rotations\_num = pow(Rotation::Max\_children, depth);

for (int i = 0; i < local\_rotations\_num; i++)

{

\_rotations[ind].second = ind - i + i \* Rotation::Max\_children + local\_rotations\_num;

ind++;

}

}

// Повороты на максимальной глубине не имеют потомков

for (; ind < \_rotations.size(); ind++)

\_rotations[ind].second = -1;

SetUpRendering();

}

Sphere::Sphere(unsigned int level\_of\_detail, ShaderProgram &&shader) : Detail\_level(level\_of\_detail), \_shader(shader)

{

Detail\_level = std::min(Detail\_level, int(\_max\_detail\_level));

Detail\_level = std::max(Detail\_level, 1);

\_rotations.resize(Rotation::MaxRotations());

unsigned int ind = 0;

for (unsigned int depth = 1; depth < Rotation::Max\_depth; depth++)

{

unsigned int local\_rotations\_num = pow(Rotation::Max\_children, depth);

for (int i = 0; i < local\_rotations\_num; i++)

{

\_rotations[ind].second = ind - i + i \* Rotation::Max\_children + local\_rotations\_num;

ind++;

}

}

// Повороты на максимальной глубине не имеют потомков

for (; ind < \_rotations.size(); ind++)

\_rotations[ind].second = -1;

CreateUvSphere(1.0f, Detail\_level, \_base\_points);

SetUpRendering();

}

void CreateUvSphere(float radius, unsigned int detail\_level, std::vector<glm::vec3> &points\_container)

{

unsigned int v\_segments\_count = detail\_level + 2;

unsigned int h\_segments\_count = detail\_level + 1;

float v\_step = 180.0f / h\_segments\_count;

float h\_step = 360.0f / v\_segments\_count;

unsigned int points\_count = v\_segments\_count \* (h\_segments\_count - 1) + 2;

points\_container.resize(points\_count);

// Верхняя точка сферы

unsigned int ind = 0;

points\_container[ind++] = {0.0f, radius, 0.0f};

for (int i = 1; i < h\_segments\_count; i++)

{

float v\_angle = -v\_step \* i + 90.0f;

for (int j = 0; j < v\_segments\_count; j++)

{

float h\_angle = h\_step \* j;

points\_container[ind++] =

{

radius \* glm::cos(glm::radians(h\_angle)) \* glm::cos(glm::radians(v\_angle)),

radius \* glm::sin(glm::radians(v\_angle)),

radius \* glm::sin(glm::radians(h\_angle)) \* glm::cos(glm::radians(v\_angle))

};

}

}

// Нижняя точка сферы

points\_container[ind] = {0.0f, -radius, 0.0f};

}

void Sphere::SetUpRendering()

{

glUseProgram(\_shader.ID());

glGenVertexArrays(1, &\_VAO);

glGenBuffers(1, &\_coords\_VBO);

glGenBuffers(1, &\_colors\_VBO);

glGenBuffers(1, &\_rotations\_VBO);

glGenBuffers(1, &\_visibles\_VBO);

glBindVertexArray(\_VAO);

glBindBuffer(GL\_ARRAY\_BUFFER, \_coords\_VBO);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0);

glEnableVertexAttribArray(0);

unsigned int max\_spheres = \_rotations.size() + 1;

std::vector<glm::vec3> default\_colors(max\_spheres, Base\_color);

glBindBuffer(GL\_ARRAY\_BUFFER, \_colors\_VBO);

glBufferData(GL\_ARRAY\_BUFFER, max\_spheres \* sizeof(glm::vec3), default\_colors.data(), GL\_STATIC\_DRAW);

glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 0, 0);

glEnableVertexAttribArray(1);

glVertexAttribDivisor(1, 1);

std::vector<glm::mat3> matrices(max\_spheres, glm::mat3(1.0f));

glBindBuffer(GL\_ARRAY\_BUFFER, \_rotations\_VBO);

glBufferData(GL\_ARRAY\_BUFFER, max\_spheres \* sizeof(glm::mat3), matrices.data(), GL\_STATIC\_DRAW);

for (int i = 0; i < 3; i++)

{

glVertexAttribPointer(2 + i, 3, GL\_FLOAT, GL\_FALSE, sizeof(glm::mat3), (void\*)(i\*sizeof(glm::vec3)));

glEnableVertexAttribArray(2 + i);

glVertexAttribDivisor(2 + i, 1);

}

std::vector<int> actives = std::vector<int>(max\_spheres, 0); actives[0] = 1;

glBindBuffer(GL\_ARRAY\_BUFFER, \_visibles\_VBO);

glBufferData(GL\_ARRAY\_BUFFER, max\_spheres \* sizeof(int), actives.data(), GL\_STATIC\_DRAW);

glVertexAttribIPointer(5, 1, GL\_INT, 0, 0);

glEnableVertexAttribArray(5);

glVertexAttribDivisor(5, 1);

glBindVertexArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

UpdateCoordsVBO();

}

void Sphere::UpdateCoordsVBO()

{

std::size\_t points\_size = \_base\_points.size() \* sizeof(glm::vec3);

glBindBuffer(GL\_ARRAY\_BUFFER, \_coords\_VBO);

glBufferData(GL\_ARRAY\_BUFFER, points\_size, \_base\_points.data(), GL\_STATIC\_DRAW);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

}

void Sphere::PutDataIntoVBO(unsigned int &VBO, std::size\_t offset, std::size\_t size, const void\* data)

{

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferSubData(GL\_ARRAY\_BUFFER, offset, size, data);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

}

void Sphere::ChangeVisibility(bool should\_affect\_rotations)

{

int is\_visible = (int)Is\_visible;

if (!should\_affect\_rotations)

{

PutDataIntoVBO(\_visibles\_VBO, 0, sizeof(int), &is\_visible);

return;

}

for (auto &r : \_rotations)

r.first.Is\_visible = is\_visible;

std::vector<int> visible = std::vector(\_rotations.size() + 1, is\_visible);

PutDataIntoVBO(\_visibles\_VBO, 0, visible.size() \* sizeof(int), visible.data());

}

void Sphere::SetClipMatrixU(const glm::mat4 &value)

{

\_shader.SetUniformMatrix4fv("u\_clip\_matrix", glm::value\_ptr(value));

}

void Sphere::SetCameraCoordsU(const glm::vec3 &value)

{

\_shader.SetUniform3fv("u\_cam\_coords", glm::value\_ptr(value));

}

void Sphere::SetCameraDistanceU(float value)

{

\_shader.SetUniform1f("u\_cam\_distance", value);

}

void Sphere::UpdateSphereShape()

{

CreateUvSphere(1.0f, Detail\_level, \_base\_points);

UpdateCoordsVBO();

}

void Sphere::UpdateSphereBaseColor()

{

PutDataIntoVBO(\_colors\_VBO, 0, sizeof(glm::vec3), glm::value\_ptr(Base\_color));

}

void Sphere::UpdateRotation(unsigned int ind, bool rotation\_changed, bool color\_changed, std::pair<bool, bool> visibility\_changed)

{

const Rotation\* rotation = &\_rotations[ind].first;

if (rotation\_changed)

{

glm::mat3 rotation\_matrix = glm::mat3(glm::rotate(glm::mat4(1.0f), glm::radians(rotation->Angle), glm::normalize(rotation->Axis))) \* rotation->\_parent\_matrix;

PutDataIntoVBO(\_rotations\_VBO, (ind + 1) \* sizeof(glm::mat3), sizeof(glm::mat3), glm::value\_ptr(rotation\_matrix));

UpdateChildRotations(ind, rotation\_matrix);

}

if (color\_changed)

PutDataIntoVBO(\_colors\_VBO, (ind + 1) \* sizeof(glm::vec3), sizeof(glm::vec3), glm::value\_ptr(rotation->Color));

if (visibility\_changed.first)

{

int is\_visible = (int)\_rotations[ind].first.Is\_visible;

PutDataIntoVBO(\_visibles\_VBO, (ind + 1) \* sizeof(int), sizeof(int), &is\_visible);

if (visibility\_changed.second)

SetChildRotationsVisibility(ind, \_rotations[ind].first.Is\_visible);

}

}

void Sphere::UpdateChildRotations(unsigned int parent\_ind, const glm::mat3 &parent\_matrix)

{

unsigned int first\_child\_ind = \_rotations[parent\_ind].second;

if (first\_child\_ind == -1)

return;

for (unsigned int i = first\_child\_ind; i < first\_child\_ind + Rotation::Max\_children; i++)

{

\_rotations[i].first.\_parent\_matrix = parent\_matrix;

UpdateRotation(i, true, false, {false, false});

}

}

void Sphere::SetChildRotationsVisibility(unsigned int parent\_ind, bool is\_visible)

{

unsigned int first\_child\_ind = \_rotations[parent\_ind].second;

if (first\_child\_ind == -1)

return;

for (unsigned int i = first\_child\_ind; i < first\_child\_ind + Rotation::Max\_children; i++)

{

\_rotations[i].first.Is\_visible = is\_visible;

UpdateRotation(i, false, false, {true, true});

}

}

void Sphere::Draw() const

{

glBindVertexArray(\_VAO);

glDrawArraysInstanced(GL\_POINTS, 0, \_base\_points.size(), \_rotations.size() + 1);

glBindVertexArray(0);

}

**shader\_program.hpp:**

#pragma once

#include <string>

#include <unordered\_map>

#include "glad/gl.h"

class ShaderProgram

{

private:

unsigned int \_program;

std::unordered\_map<std::string, unsigned int> \_uniforms\_locations;

void CreateShaderProgram(const char \*vertex\_source, const char \*fragment\_source);

unsigned int CompileShader(GLuint type, const char \*source);

void TryGetNewLocation(const GLchar \*name);

public:

ShaderProgram() {}

ShaderProgram(std::string vertex\_shader\_path, std::string fragment\_shader\_path);

unsigned int ID() const { return \_program; }

void SetUniform1i(const GLchar \*name, GLint value);

void SetUniform1f(const GLchar \*name, GLfloat value);

void SetUniform3fv(const GLchar \*name, const GLfloat \*value);

void SetUniformMatrix4fv(const GLchar \*name, const GLfloat \*value);

};

**shader\_program.cpp:**

#include <iostream>

#include <fstream>

#include <sstream>

#include "shader\_program.hpp"

ShaderProgram::ShaderProgram(std::string vertex\_shader\_path, std::string fragment\_shader\_path)

{

std::ifstream v\_shader\_file;

std::ifstream f\_shader\_file;

std::ostringstream v\_shader\_source;

std::ostringstream f\_shader\_source;

v\_shader\_file.exceptions(std::ifstream::badbit);

f\_shader\_file.exceptions(std::ifstream::badbit);

try

{

v\_shader\_file.open(vertex\_shader\_path);

v\_shader\_source << v\_shader\_file.rdbuf();

}

catch(const std::ifstream::failure& e)

{

std::cout << "ERROR: FAILED TO READ SHADER FILE: VERTEX SHADER" << std::endl;

}

try

{

f\_shader\_file.open(fragment\_shader\_path);

f\_shader\_source << f\_shader\_file.rdbuf();

}

catch(const std::ifstream::failure& e)

{

std::cout << "ERROR: FAILED TO READ SHADER FILE: FRAGMENT SHADER" << std::endl;

}

CreateShaderProgram(v\_shader\_source.str().c\_str(), f\_shader\_source.str().c\_str());

}

void ShaderProgram::CreateShaderProgram(const char \*vertex\_source, const char \*fragment\_source)

{

\_program = glCreateProgram();

unsigned int vertex\_shader = CompileShader(GL\_VERTEX\_SHADER, vertex\_source);

unsigned int fragment\_shader = CompileShader(GL\_FRAGMENT\_SHADER, fragment\_source);

glAttachShader(\_program, vertex\_shader);

glAttachShader(\_program, fragment\_shader);

glLinkProgram(\_program);

int linking\_result;

glGetProgramiv(\_program, GL\_LINK\_STATUS, &linking\_result);

if (linking\_result == GL\_FALSE)

{

char info\_log[512];

glGetProgramInfoLog(\_program, 512, NULL, info\_log);

std::cout << "ERROR: FAILED TO LINK PROGRAM\n" << info\_log << std::endl;

}

}

unsigned int ShaderProgram::CompileShader(GLuint type, const char \*source)

{

unsigned int shader = glCreateShader(type);

glShaderSource(shader, 1, &source, NULL);

glCompileShader(shader);

int compilation\_result;

glGetShaderiv(shader, GL\_COMPILE\_STATUS, &compilation\_result);

if (compilation\_result == GL\_FALSE)

{

char info\_log[512];

glGetShaderInfoLog(shader, 512, NULL, info\_log);

std::cout << "ERROR: FAILED TO COMPILE SHADER: "

<< (type == GL\_VERTEX\_SHADER ? "VERTEX" : "FRAGMENT")

<< " SHADER\n" << info\_log << std::endl;

}

return shader;

}

void ShaderProgram::TryGetNewLocation(const GLchar \*name)

{

if (\_uniforms\_locations.find(name) == \_uniforms\_locations.end())

\_uniforms\_locations[name] = glGetUniformLocation(\_program, name);

}

void ShaderProgram::SetUniform1i(const GLchar \*name, GLint value)

{

TryGetNewLocation(name);

glUniform1i(\_uniforms\_locations[name], value);

}

void ShaderProgram::SetUniform1f(const GLchar \*name, GLfloat value)

{

TryGetNewLocation(name);

glUniform1f(\_uniforms\_locations[name], value);

}

void ShaderProgram::SetUniform3fv(const GLchar \*name, const GLfloat \*value)

{

TryGetNewLocation(name);

glUniform3fv(\_uniforms\_locations[name], 1, value);

}

void ShaderProgram::SetUniformMatrix4fv(const GLchar \*name, const GLfloat \*value)

{

TryGetNewLocation(name);

glUniformMatrix4fv(\_uniforms\_locations[name], 1, GL\_FALSE, value);

}

**ui.hpp:**

#pragma once

#include <vector>

#include <string>

#include "glad/gl.h"

#include "glm/vec3.hpp"

#include "sphere.hpp"

class UI

{

private:

Sphere\* \_sphere;

std::vector<std::vector<glm::vec3>> \_rotations\_points;

std::tuple<bool, bool, std::pair<bool, bool>> DisplayRotationContent(Rotation &rotation, const std::string &id);

void DisplayRotationNode(unsigned int ind);

void DisplayRotationPointsNode(unsigned int ind, std::string label, int prev\_ind = -1);

void TryApplyChanges(const std::tuple<bool, bool, std::pair<bool, bool>> &changes, unsigned int rotation\_ind);

void UpdateLevelOfDetail();

void UpdateRotationPoints(unsigned int ind);

public:

UI (Sphere \*sphere, GLFWwindow \*window, const char \*glsl\_version);

void Die();

void BeginFrame();

void EndFrame();

void DrawPropertiesWindow();

void DrawRotationsResultsWindow();

};

**ui.cpp:**

#include <vector>

#include <string>

#include "glm/vec4.hpp"

#define IM\_VEC4\_CLASS\_EXTRA \

constexpr ImVec4(const glm::vec4& f) : x(f.x), y(f.y), z(f.z), w(f.w) {} \

operator glm::vec4() const { return glm::vec4(x,y,z,w); }

#include "glad/gl.h"

#include "imgui.h"

#include "backends/imgui\_impl\_opengl3.h"

#include "backends/imgui\_impl\_glfw.h"

#include "glm/gtc/type\_ptr.hpp"

#include "glm/vec3.hpp"

#include "glm/mat3x3.hpp"

#include "ui.hpp"

#include "sphere.hpp"

UI::UI(Sphere\* sphere, GLFWwindow \*window, const char \*glsl\_version) : \_sphere(sphere)

{

IMGUI\_CHECKVERSION();

ImGui::CreateContext();

ImGui\_ImplGlfw\_InitForOpenGL(window, true);

ImGui\_ImplOpenGL3\_Init(glsl\_version);

ImGui::StyleColorsDark();

ImGuiIO& io = ImGui::GetIO();

io.Fonts->AddFontFromFileTTF("../arial.ttf", 20, NULL, io.Fonts->GetGlyphRangesCyrillic());

\_rotations\_points.resize(Rotation::MaxRotations());

std::fill(\_rotations\_points.begin(), \_rotations\_points.end(), \_sphere->BasePoints());

}

void UI::Die()

{

ImGui\_ImplGlfw\_Shutdown();

ImGui\_ImplOpenGL3\_Shutdown();

ImGui::DestroyContext();

}

void UI::BeginFrame()

{

ImGui\_ImplGlfw\_NewFrame();

ImGui\_ImplOpenGL3\_NewFrame();

ImGui::NewFrame();

}

void UI::EndFrame()

{

ImGui::Render();

ImGui\_ImplOpenGL3\_RenderDrawData(ImGui::GetDrawData());

}

void UI::DrawPropertiesWindow()

{

// ImGui::ShowDemoWindow();

ImGui::SetNextWindowSizeConstraints({400, -1}, {INFINITY, -1});

if (!ImGui::Begin("##UiWindow", nullptr, ImGuiWindowFlags\_AlwaysAutoResize))

{

ImGui::End();

return;

}

ImGui::Text(u8"Свойства сферы:");

if(\_sphere->Detail\_level && ImGui::SliderInt(u8"Уровень детализации", &(\_sphere->Detail\_level), 1, \_sphere->MaxDetailLevel()))

{

\_sphere->UpdateSphereShape();

UpdateLevelOfDetail();

}

if (ImGui::ColorEdit3(u8"Цвет", glm::value\_ptr(\_sphere->Base\_color)))

\_sphere->UpdateSphereBaseColor();

if (ImGui::Checkbox(u8"Видима", &\_sphere->Is\_visible))

\_sphere->ChangeVisibility(ImGui::GetIO().KeyCtrl);

ImGui::Separator();

ImGui::Text(u8"Повороты:");

for (unsigned int i = 0; i < Rotation::Max\_children; i++)

DisplayRotationNode(i);

ImGui::End();

}

static bool stylized\_text = true;

void UI::DrawRotationsResultsWindow()

{

if (!ImGui::Begin("##RotationsResultsWindow"))

{

ImGui::End();

return;

}

ImGui::Checkbox(u8"Использовать стилизованный текст", &stylized\_text);

// Изначальные точки сферы

bool opened = ImGui::TreeNodeEx("##BasePoints", ImGuiTreeNodeFlags\_OpenOnArrow);

ImGui::SameLine();

if (stylized\_text)

ImGui::PushStyleColor(ImGuiCol\_Text, glm::vec4(\_sphere->Base\_color, 1.0f));

else

ImGui::PushStyleColor(ImGuiCol\_Text, ImGui::GetStyle().Colors[ImGuiCol\_Text]);

ImGui::Text(u8"Изначальные точки сферы");

if (opened)

{

for (int i = 0; i < \_sphere->BasePoints().size(); i++)

{

glm::vec3 point = \_sphere->BasePoints()[i];

ImGui::Text("(%.4f, %.4f, %.4f)", point.x, point.y, point.z);

}

ImGui::TreePop();

}

ImGui::PopStyleColor();

ImGui::Separator();

// Точки, полученные из поворотов

for (int i = 0; i < Rotation::Max\_children; i++)

DisplayRotationPointsNode(i, u8"Поворот " + std::to\_string(i + 1));

ImGui::End();

}

void UI::DisplayRotationPointsNode(unsigned int ind, std::string label, int prev\_ind)

{

if (prev\_ind != -1)

ImGui::Indent();

std::string id = "##Node" + std::to\_string(ind);

bool opened = ImGui::TreeNodeEx(id.c\_str(), ImGuiTreeNodeFlags\_OpenOnArrow);

ImGui::SameLine();

glm::vec4 default\_text\_color = ImGui::GetStyle().Colors[ImGuiCol\_Text];

if (stylized\_text)

ImGui::PushStyleColor(ImGuiCol\_Text, glm::vec4(\_sphere->RotationByIndex(ind).Color, 1.0f));

else

ImGui::PushStyleColor(ImGuiCol\_Text, default\_text\_color);

ImGui::Text(label.c\_str());

ImGui::PopStyleColor();

if (opened)

{

for (int i = 0; i < \_rotations\_points[ind].size(); i++)

{

if (stylized\_text)

{

ImGui::PushStyleColor(ImGuiCol\_Text, glm::vec4(\_sphere->RotationByIndex(ind).Color, 1.0f));

ImGui::PushStyleColor(ImGuiCol\_Text, default\_text\_color);

if (prev\_ind == -1)

ImGui::PushStyleColor(ImGuiCol\_Text, glm::vec4(\_sphere->Base\_color, 1.0f));

else

ImGui::PushStyleColor(ImGuiCol\_Text, glm::vec4(\_sphere->RotationByIndex(prev\_ind).Color, 1.0f));

}

else

{

ImGui::PushStyleColor(ImGuiCol\_Text, default\_text\_color);

ImGui::PushStyleColor(ImGuiCol\_Text, default\_text\_color);

ImGui::PushStyleColor(ImGuiCol\_Text, default\_text\_color);

}

glm::vec3 parent = (prev\_ind == -1) ? \_sphere->BasePoints()[i] : \_rotations\_points[prev\_ind][i];

glm::vec3 child = \_rotations\_points[ind][i];

ImGui::Text("(%.4f, %.4f, %.4f)", parent.x, parent.y, parent.z);

ImGui::PopStyleColor();

ImGui::SameLine();

ImGui::Text(" ---> ");

ImGui::PopStyleColor();

ImGui::SameLine();

ImGui::Text("(%.4f, %.4f, %.4f)", child.x, child.y, child.z);

ImGui::PopStyleColor();

}

ImGui::TreePop();

}

unsigned int first\_child = \_sphere->Rotations()[ind].second;

if (first\_child != -1)

{

for (int i = 0; i < Rotation::Max\_children; i++)

{

std::string child\_label = label + '.' + std::to\_string(i + 1);

DisplayRotationPointsNode(first\_child + i, child\_label, ind);

}

}

if (prev\_ind != -1)

ImGui::Unindent();

}

void UI::DisplayRotationNode(unsigned int ind)

{

std::string id = "##Node" + std::to\_string(ind);

if (\_sphere->Rotations()[ind].second != -1)

{

bool opened = ImGui::TreeNodeEx(id.c\_str(), ImGuiTreeNodeFlags\_OpenOnArrow);

TryApplyChanges(DisplayRotationContent(\_sphere->RotationByIndex(ind), std::to\_string(ind)), ind);

if (opened)

{

for (unsigned int i = 0; i < Rotation::Max\_children; i++)

DisplayRotationNode(\_sphere->Rotations()[ind].second + i);

ImGui::TreePop();

}

}

else

{

ImGui::Bullet();

TryApplyChanges(DisplayRotationContent(\_sphere->RotationByIndex(ind), std::to\_string(ind)), ind);

}

}

static float GetPeriodicValue(float value, float period)

{

if (std::fabs(value) < period)

return value;

return value > 0 ? value - period \* std::floor(value / period)

: value + period \* std::floor(std::fabs(value) / period);

}

std::tuple<bool, bool, std::pair<bool, bool>> UI::DisplayRotationContent(Rotation &rotation, const std::string &id)

{

std::tuple<bool, bool, std::pair<bool, bool>> changed = {false, false, {false, false}};

std::string angle\_label = u8"Угол##" + id;

std::string axis\_label = u8"Ось вращения##" + id;

std::string color\_label = u8"Цвет##" + id;

std::string visible\_label = u8"Видимый##" + id;

ImGui::SameLine();

ImGui::SetNextItemWidth(130.0f);

if (std::get<0>(changed) = ImGui::InputFloat(angle\_label.c\_str(), &rotation.Angle, 0.1f, 1.0f, "%.2f"))

rotation.Angle = GetPeriodicValue(rotation.Angle, 360.0f);

ImGui::SameLine();

ImGui::SetNextItemWidth(200.0f);

std::get<0>(changed) = ImGui::InputFloat3(axis\_label.c\_str(), glm::value\_ptr(rotation.Axis), "%.2f") || std::get<0>(changed);

ImGui::SameLine();

std::get<1>(changed) = ImGui::ColorEdit3(color\_label.c\_str(), glm::value\_ptr(rotation.Color), ImGuiColorEditFlags\_NoInputs);

ImGui::SameLine();

std::get<2>(changed).first = ImGui::Checkbox(visible\_label.c\_str(), &rotation.Is\_visible);

std::get<2>(changed).second = ImGui::GetIO().KeyCtrl;

return changed;

}

void UI::TryApplyChanges(const std::tuple<bool, bool, std::pair<bool, bool>> &changes, unsigned int rotation\_ind)

{

if (!std::get<0>(changes) && !std::get<1>(changes) && !std::get<2>(changes).first)

return;

\_sphere->UpdateRotation(rotation\_ind, std::get<0>(changes), std::get<1>(changes), std::get<2>(changes));

if (std::get<0>(changes))

UpdateRotationPoints(rotation\_ind);

}

void UI::UpdateLevelOfDetail()

{

std::fill(\_rotations\_points.begin(), \_rotations\_points.end(), \_sphere->BasePoints());

for (int i = 0; i < \_rotations\_points.size(); i++)

{

Rotation rotation = \_sphere->RotationByIndex(i);

if (rotation.Angle == 0.0f && rotation.ParentMatrix() == glm::mat3(1.0f))

{

continue;

}

else

{

glm::mat3 rotation\_matrix = glm::mat3(glm::rotate(glm::mat4(1.0f), glm::radians(rotation.Angle), glm::normalize(rotation.Axis))) \* rotation.ParentMatrix();

for (int j = 0; j < \_rotations\_points[i].size(); j++)

\_rotations\_points[i][j] = rotation\_matrix \* \_sphere->BasePoints()[j];

}

}

}

void UI::UpdateRotationPoints(unsigned int ind)

{

Rotation rotation = \_sphere->RotationByIndex(ind);

glm::mat3 rotation\_matrix = glm::mat3(glm::rotate(glm::mat4(1.0f), glm::radians(rotation.Angle), glm::normalize(rotation.Axis))) \* rotation.ParentMatrix();

for (int i = 0; i < \_rotations\_points[ind].size(); i++)

\_rotations\_points[ind][i] = rotation\_matrix \* \_sphere->BasePoints()[i];

if (\_sphere->Rotations()[ind].second != -1)

for (int i = 0; i < Rotation::Max\_children; i++)

UpdateRotationPoints(\_sphere->Rotations()[ind].second + i);

}

**camera.hpp:**

#pragma once

#include "glm/vec3.hpp"

#include "glm/mat4x4.hpp"

class Camera

{

private:

constexpr static glm::vec3 \_look\_target = glm::vec3(0.0f, 0.0f, 0.0f);

inline static glm::vec3 \_position;

// Расстояние до \_look\_target

inline static float \_distance = 3.0f;

constexpr static float \_min\_distance = 2.1f;

constexpr static float \_max\_distance = 8.0f;

inline static float \_yaw = 0.0f; // Угол поворота вокруг оси Y

inline static float \_pitch = 0.0f; // Угол поворота вокруг оси X

constexpr static float \_max\_pitch = 89.0f;

constexpr static float \_rotation\_speed = 25.0f;

constexpr static float \_zoom\_speed = 4.5f;

inline static glm::mat4 \_projection;

constexpr static float \_field\_of\_view = 45.0f;

Camera() {}

public:

enum class Move

{

RIGHT = 1, LEFT = -1,

UP = 1, DOWN = -1,

IN = 1, OUT = -1,

STAY = 0

};

constexpr static float Drag\_sensitivity = 0.6f;

static const glm::vec3& Position() { return \_position; }

static float Distance() { return \_distance; }

static glm::mat4 ClipSpaceMatrix();

static void UpdateProjectionMatrix(unsigned int w\_width, unsigned int w\_height);

static void UpdatePosition();

static void Rotate(float yaw, float pitch);

static void Zoom(float zoom);

};

**camera.cpp:**

#include "glm/vec3.hpp"

#include "glm/mat4x4.hpp"

#include "glm/trigonometric.hpp"

#include "glm/gtc/matrix\_transform.hpp"

#include "camera.hpp"

extern float delta\_time;

glm::mat4 Camera::ClipSpaceMatrix()

{

return \_projection \* glm::lookAt(\_position, \_look\_target, glm::vec3(0.0f, 0.1f, 0.0f));

}

void Camera::UpdateProjectionMatrix(unsigned int w\_width, unsigned int w\_height)

{

\_projection = glm::perspective(\_field\_of\_view, float(w\_width) / float(w\_height), 1.0f, 20.0f);

}

void Camera::UpdatePosition()

{

\_position = glm::vec3

(

\_distance \* glm::cos(glm::radians(\_yaw)) \* glm::cos(glm::radians(\_pitch)),

\_distance \* glm::sin(glm::radians(\_pitch)),

\_distance \* glm::sin(glm::radians(\_yaw)) \* glm::cos(glm::radians(\_pitch))

);

}

void Camera::Rotate(float yaw, float pitch)

{

\_yaw -= yaw \* \_rotation\_speed \* delta\_time;

\_pitch += pitch \* \_rotation\_speed \* delta\_time;

if (std::fabs(\_yaw) >= 360.0f)

\_yaw -= (int(\_yaw > 0.0f) \* 2 - 1) \* 360.0f;

if (std::fabs(\_pitch) > \_max\_pitch)

\_pitch = \_pitch > 0.0f ? \_max\_pitch : -\_max\_pitch;

UpdatePosition();

}

void Camera::Zoom(float zoom)

{

\_distance -= zoom \* \_zoom\_speed \* delta\_time;

\_distance = std::max(\_min\_distance, \_distance);

\_distance = std::min(\_max\_distance, \_distance);

UpdatePosition();

}

**GitHub-репозиторий проекта: https://github.com/qqpilla/coursework-2**

1. Legacy OpenGL // khronos.org URL: https://www.khronos.org/opengl/wiki/Legacy\_OpenGL (дата обращения: 30.05.2023) [↑](#footnote-ref-1)