

МИНОБРНАУКИ РОССИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»  
(ФГБОУ ВО «ВГУ»)

Факультет прикладной математики, информатики и механики  
Кафедра математического обеспечения ЭВМ

**Разработка приложения для создания воксельных объектов**

Бакалаврская работа

Направление 02.03.02 Фундаментальная информатика и информационные  
технологии

Профиль Инженерия программного обеспечения

Зав. кафедрой \_\_\_\_\_ д.т.н., проф. Г.В. Абрамов \_\_\_\_\_ 2022г

Обучающийся \_\_\_\_\_ А.И. Баклашов

Руководитель \_\_\_\_\_ к.ф.-м.н., доц. Е.В. Трофименко

Воронеж 2022

## Оглавление

Введение.....	4
1. Обзор существующих решений.....	5
2. Постановка и анализ задачи.....	6
2.1. Постановка задачи.....	6
2.2. Анализ задачи.....	7
3. Средства разработки.....	10
4. Реализация.....	12
4.1. Триангуляция Делоне.....	12
4.2. Решение задачи в пространстве.....	17
4.3. Модуль геометрии.....	19
4.4. Модуль адаптеров геометрии для OpenGL.....	23
4.5. Модуль трансформаций.....	25
4.6. Модуль октодерева.....	29
4.7. Модуль импорта файлов.....	37
4.8. Модуль материала.....	40
4.9. Модуль камеры.....	42
4.10. Модуль трассировщика.....	43
4.11. Модуль приложения.....	46
5. Интерфейс.....	47
Заключение.....	52
Список использованных источников.....	53
Приложения.....	55
Приложение 1. Диаграмма классов Октодерева.....	55
Приложение 2. Диаграмма классов, отвечающих за импорт.....	56
Приложение 3. Диаграмма классов структуры геометрии.....	57
Приложение 4. Диаграмма классов Transformation3D и Camera3D.....	58
Приложение 5. Диаграмма классов Material и Voxel.....	59
Приложение 6. Листинг класса Camera3D.....	60
Приложение 7. Листинг классов геометрий.....	63

Приложение 8. Листинг класса GeomTracer.....	70
Приложение 9. Листинг класса Iterator.....	73
Приложение 10. Листинг класса Material.....	74
Приложение 11. Листинг классов Parser, ObjPraser, MtlParser.....	77
Приложение 12. Листинг класса OctoTreeIterator.....	86
Приложение 13. Листинг класса OctoKey.....	89
Приложение 14. Листинг классов Октодерева.....	92
Приложение 15. Листинг класса Transformation3D.....	101
Приложение 16. Листинг класса Voxel.....	107

## **Введение**

Несколько лет назад, использование вокселей в компьютерной графике было очень ограничено. Это связывалось с требованиями к аппаратным ресурсам компьютеров, необходимые для обработки состояний огромного количества вокселей на одной сцене. В связи с этим, замена полигональных моделей на воксельные без потери качества и производительности, была невозможной, при их использовании в режиме реального времени. Поэтому их применяли в узконаправленных областях по большей части в медицине, где с их помощью ряд медицинских устройств визуализировали результаты КТ, УЗИ или МРТ.

На данный момент возможности вычислительных аппаратов, возросли, что позволило расширить применение воксельной графики. Не смотря на требовательность к большой мощности вычислительного устройства, ее основными достоинствами являются симуляция с реалистичной разрушаемостью, и простота создания различных объектов и сцен. Это позволило воксельной графике начать развиваться в игровой сфере.

В качестве примеров применения вокселей в компьютерной графике можно привести создание моделей для обнаружения столкновений полигональных моделей, симуляцию детализированных разрушений различных объектов или создание художниками воксель-артов – изображений, стилизованных под воксельную графику.

В перечисленных случаях встает проблема выбора приложения, разработанного специально для создания таких объектов и предоставляющее различные инструменты для работы с воксельными объектами.

Таким образом целью данной работы является разработка программного обеспечения по созданию воксельных объектов. Разрабатываемая архитектура приложения будет предполагать в себе дальнейшие модификации и оптимизацию работы внутренних моделей приложения, в связи с чем в разработке применяются паттерны проектирования.

## 1. Обзор существующих решений

### *MagicaVoxel*

MagicaVoxel – воксельный Open Source редактор. Он содержит в себе, как инструменты для воксельного моделирования, так и качественный рендер движок, который позволяет гибко настроить материалы и экспортировать их вместе с текстурами и тенями. Редактор доступен на системах Windows и Mac OS и позволяет экспортировать результат работы в популярные 3d форматы и даже 2d спрайты. Помимо стандартных инструментов редактирования, реализована поддержка ввода команд, что позволяет создавать действительно сложные структуры. Экспорт модели возможен только в формате .obj.

### *VoxelShop*

VoxelShop - программное обеспечение для различных платформ, обладающее возможностями для создания и изменения воксельных объектов. Разработан в сотрудничестве с художниками. Программа находится в альфа-версии с открытым исходным кодом. Основная особенность - возможность использования вокселем текстуры вместо цвета. Каждая сторона вокселя может иметь различную текстуру, а текстуры можно вращать и зеркально отображать. Это может быть полезно для имитации более высокого разрешения вокселей в определенных разделах, например, глазами персонажа или для тестирования различных художественных стилей. Программа поддерживает формат dae, который позволяет убрать ненужные полигоны и оптимизировать модель. В программе нет инструмента для анимирования, но оптимизированные модели можно импортировать в Blender для удобной работы.

## **2. Постановка и анализ задачи**

### **2.1. Постановка задачи**

Целью ВКР является разработка приложения для создания воксельных объектов.

Предусмотреть предоставление, пользователю таких базовые инструментов для работы с воксельной моделью как:

- добавление вокселя к существующему множеству;
- удаление вокселя из существующего множества;
- изменение параметров конкретного вокселя;
- настройка материала для вокселя;
- осмотр созданного воксельного объекта;
- добавление геометрии в качестве образца для создания объекта.

Разработать пользовательский интерфейс, с помощью которого пользователь будет совершать операции над воксельным объектом.

Разработку вести с учетом дальнейшей модификации приложения, которая заключается добавлением новых возможностей или обновления существующего функционала.

## 2.2. Анализ задачи

Разрабатываемое приложение должно иметь следующие возможности:

- структура хранения вокселей;
- структура хранения геометрии;
- отображение объектов на сцене;
- загрузка данных из файлов.

### *Структура хранения вокселей*

Воксель – это элемент объемного изображения, содержащий значение элемента растра в трехмерном пространстве. Представляет из себя трехмерный объект, состоящий из шести полигонов и имеющий форму куба.

Рассмотрим существующие структуры для хранения сложных геометрических данных.

1. Октодерево – одна из распространенных типов иерархических структур. Вначале строится ограничивающее тело, описанное вокруг всех объектов исходного множества. Далее это тело разбивается на 8 частей, и для каждой из этих частей строится список всех объектов, полностью или частично попадающих в получившуюся часть.
2. kD-дерево - каждый узел такого дерева содержит ограничивающее тело для содержащихся в нем объектов. Однако на каждом шаге оно делится всего на две части плоскостью, параллельной одной из координатных плоскостей. После чего, составляются списки объектов, хотя бы частично попадающих в каждую из частей. Далее по каждому из этих списков строится следующий узел дерева.
3. BSP-дерево - строится по набору граней. BSP-дерево позволяет осуществить упорядочение граней, при этом для гарантии корректности такого упорядочивания некоторые грани будут разбиваться на части.
4. R-дерево - разбивает многомерное пространство на множество иерархически вложенных и, возможно, пересекающихся,

прямоугольников. В случае трехмерного или многомерного пространства это будут прямоугольные параллелепипеды.

Для хранения вокселей в поставленной задаче будет использоваться структура – октодерево, так как оно использует разбиение пространства на одинаковые по размеру кубы. Воксель, имеющий схожую форму, будет храниться в одном листе октодерева, что дает возможность не хранить точные размеры вокселя в каждом листе октодерева.

Необходимо разработать следующий набор модулей, которые будут отвечать за действия, выполняемые над структурой данных.

- модуль октодерева – инкапсулирует в себе взаимодействия со структурой октодерева: нодами и объектами в контейнере;
- модуль октонод – содержат методы для изменения параметров одной ноды, а также хранят отведенные её данные об объектах в контейнере;
- модуль вокселя – содержит структуру хранения данных описывающий воксель;
- модуль материала – содержит список параметров для отображения объекта.

### *Структура хранения геометрии*

Геометрия состоит набора графических примитивов и полигонов, которые являются совокупностью этих примитивов. Основным геометрическим примитивом, который использует любая геометрия является вершина, вместе с которой может быть определён текстель (текстурная координата) или нормаль.

Модуль для хранения и обработки геометрии должен иметь возможности для реализации различных структур хранения и отображения геометрий. При этом все основные операции, связанные с геометрией, должны выполняться независимо от реализованной структуры.

В рамках данной работы от структуры геометрии требуется её последующее использование при отображении всей сцены.



Необходимо разработать следующий набор модулей, которые будут отвечать за действия, выполняемые над геометрией.

- модуль геометрий;
- модуль триангуляции;
- модуль адаптеров геометрии к OpenGL.

#### *Отображение объектов на сцене*

Для коррекной отрисовки объектов на сцене требуется реализация следующих возможностей.

- хранение местоположения, поворота, масштаба объекта;
- хранение информации о камере;
- загрузка данных в OpenGL буферы.

#### *Загрузка данных из файлов*

Модуль импорта данных для различных форматов, будет реализован с помощью паттерна проектирования шаблонный метод. С его помощью будет обеспечена возможность чтения и записи данных в соответствующую структуру данных из следующих форматов.

- Obj – хранение геометрии;
- MtlLib – хранение данных о материалах.

### 3. Средства разработки

#### *OpenGL*

OpenGL спецификация, определяющая кроссплатформенный программный интерфейс для написания приложений, использующих двумерную и трёхмерную компьютерную графику [1].

Библиотеки OpenGL написаны на языке программирования Си, но при этом допускают множество ответвлений с использованием других языков программирования. Поскольку многие языковые конструкции языка Си не очень хорошо переводятся в другие высокоуровневые языки, то OpenGL был разработан с учетом нескольких абстракций.

Отличительной особенностью OpenGL является поддержка расширений. Всякий раз, когда графическая компания выкатывает новую методику или новую большую оптимизацию для рендеринга, это часто встречается в расширении, реализованном в драйверах. Если оборудование, на котором работает приложение, поддерживает такое расширение, то разработчик может использовать функционал, предоставляемый этим расширением, для более продвинутой или эффективной графики.

Ядро OpenGL поддерживает язык шейдеров GLSL (OpenGL Shading Language), который позволяет заменить фиксированный программный конвейер (Fixed function pipeline) небольшими подпрограммами, написанными на С-подобном языке. Такая возможность позволяет достичь большей гибкости при создании таких эффектов, как освещение или наложение текстур.

#### *Фреймворк Qt*

Qt – это кроссплатформенный фреймворк для разработки программного обеспечения на языке программирования C++ [2]. Данный фреймворк позволяет запускать написанное с его помощью ПО на большинстве современных операционных систем путем обычной компиляции без изменения исходного кода программы. Отличительной особенностью

является использование метаобъектного компилятора, который перед процессом компиляции обрабатывает код. Qt комплектуется IDE Qt Designer, позволяющая создавать элементы графического пользовательского интерфейса.

### *Библиотека Eigen*

Eigen — библиотека линейной алгебры для языка программирования C++ с открытым исходным кодом. Написана на шаблонах и предназначена для векторно-матричных вычислений и связанных с ними операций [3]. В ней есть множество compile-time оптимизаций, библиотека способна выполнять явную векторизацию и поддерживает SSE 2/3/4, ARM и NEON инструкции. А также позволяет реализовывать сложные математические вычисления кратким и выразительным образом.

## 4. Реализация

При импорте полигональных моделей из формата obj, мы сталкиваемся с проблемой описания полигонов. В описании формата obj, описано, что поверхность может быть задана с помощью многоугольника. OpenGL имеет возможность отрисовывать многоугольники путем закрашивания области образуемого выпуклой оболочкой точек. При отрисовке таким образом, возникают проблемы с освещением полигона. В результате чего получаются артефакты.

Отрисовка треугольников имеет ряд преимуществ перед отрисовкой всего многоугольника, это:

- простота определения параметров плоскости, образуемые точками;
- возможность точно задать поверхность полигона, образуемую точками, в следствии того, что треугольник всегда лежит в одной плоскости.

Поэтому для разрабатываемого рендера будем использовать отрисовку с помощью треугольников. Но из-за того, что форматы для хранения полигональных моделей описывают поверхности многоугольниками, возникает задача триангуляции.

Согласно работе [4] триангуляцией называется планарный граф, все внутренние области которого являются треугольниками. Задачей построения триангуляции по заданному набору точек называется задача соединения заданных точек непересекающимися отрезками так, чтобы образовалась триангуляция.

### 4.1. Триангуляция Делоне

Поскольку полученная триангуляция планируется использоваться при вычислении коллизий, будет строиться триангуляция Делоне.

Определение 1: Говорят, что триангуляция удовлетворяет условию Делоне, если внутри окружности, описанной вокруг любого построенного треугольника, не попадает ни одна из заданных точек триангуляции.

Определение 2: Триангуляция называется триангуляцией Делоне, если она является выпуклой и удовлетворяет условию Делоне.

Триангуляция Делоне обладает следующими теоремами:

Теорема 1: Триангуляцию Делоне можно получить из любой другой триангуляции по той же системе точек, последовательно перестраивая пары соседних треугольников  $\triangle ABC$  и  $\triangle BCD$ , не удовлетворяющих условию Делоне, в пары треугольников  $\triangle ABD$  и  $\triangle ACD$ . Такая операция перестроения также называется флипом (рис. 1).

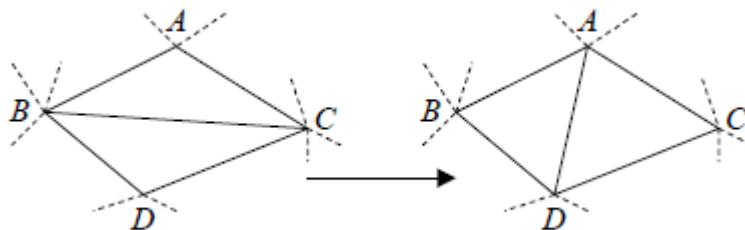


Рис. 1. Операция флипа

Теорема 2. Триангуляция Делоне обладает максимальной суммой минимальных углов всех своих треугольников среди всех возможных триангуляций.

Теорема 3. Триангуляция Делоне обладает минимальной суммой радиусов окружностей, описанных около треугольников, среди всех возможных триангуляций.

В следствии, второй теоремы о триангуляции Делоне, получаем, что треугольники, образованные данной триангуляцией, являются самыми большими по площади, что исключает возможность образования треугольника с точками на одной линии.

Алгоритм пошаговой триангуляции Делоне, на основе первой теоремы:

1. Строим некоторую триангуляцию.

2. Проходим по всем смежным парам треугольников. Если пара треугольников не удовлетворяет условию Делоне (Определение 1 и Теоремы 1,). Производим флип описанный в теореме 1.

Для построения некоторой триангуляции используется алгоритм с использованием минимальной выпуклой оболочки.

Определение 3: Пусть, на плоскости задано конечное множество точек  $A$ . Оболочкой этого множества называется любая замкнутая линия  $H$  без самопересечений такая, что все точки из  $A$  лежат внутри этой кривой. Если кривая  $H$  является выпуклой, то соответствующая оболочка также называется выпуклой. Минимальной выпуклой оболочкой (сокращенно МВО) называется выпуклая оболочка, имеющая минимальный периметр (рис. 2).

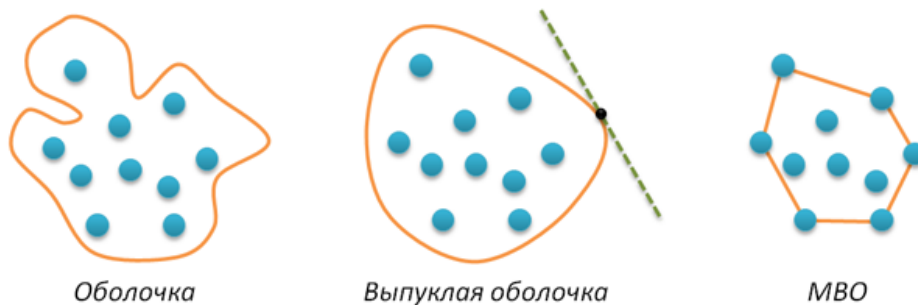


Рис. 2. Типы оболочек

Для построения МВО используется алгоритм Грэхема. Он имеет логарифмическую сложность в следствии применения быстрой сортировки.

Алгоритм Грэхема для построения МВО:

1. Выбираем точку гарантированно входящей в МВО. Ею будет точка с наименьшей координатой по одной из осей, например  $X$ .
2. Сортируем всех точек по степени их левизны относительно стартовой точки. Левизна определяется углом, образуемым между прямой проведенной через стартовую точку и параллельную координатной оси  $Y$ .
3. Проходим по сортированным вершинам и удаляем те, в которых выполняется правый поворот.

Алгоритм триангуляции с помощью МВО (рис. 3).

1. Отсортируем все точки по координате X.
2. Построим треугольник на первых 3 точках.
3. Для каждой последующей точки, добавляем треугольники, образованные между данной точкой и видимым ребром МВО существующей триангуляции.

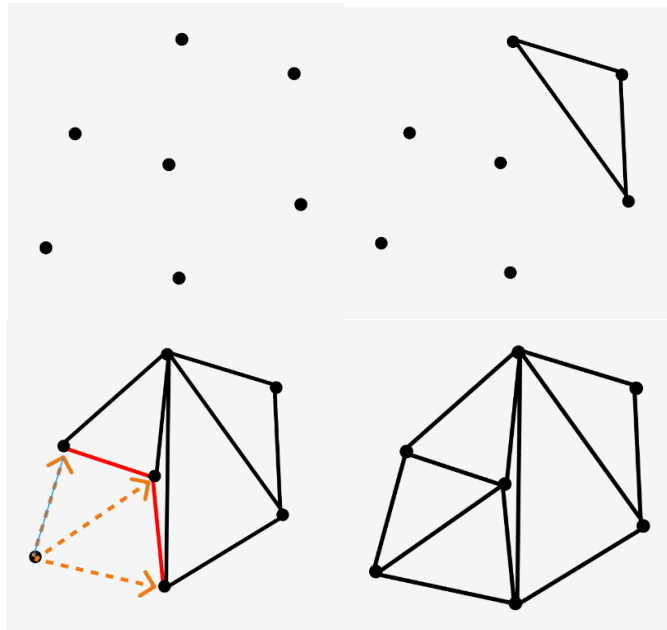


Рис. 3. Алгоритм триангуляции по шагам

Поиск видимых ребер осуществляется через последовательный обход ребер МВО уже найденной ранее триангуляции. Для каждого ребра находится нормаль, направленная от середины многоугольника, образованного МВО. Если скалярное произведение между нормалью и вектором, образованным выбранной точкой и одной из точек ребра, является положительным, то ребро является видимым (рис. 4).

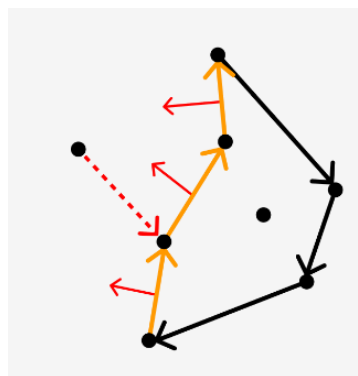


Рис. 4. Видимые ребра

На данном этапе у нас уже есть некая триангуляция, последним шагом является последовательный перебор пар образованных треугольников и проверка их на условие Делоне (рис. 5).

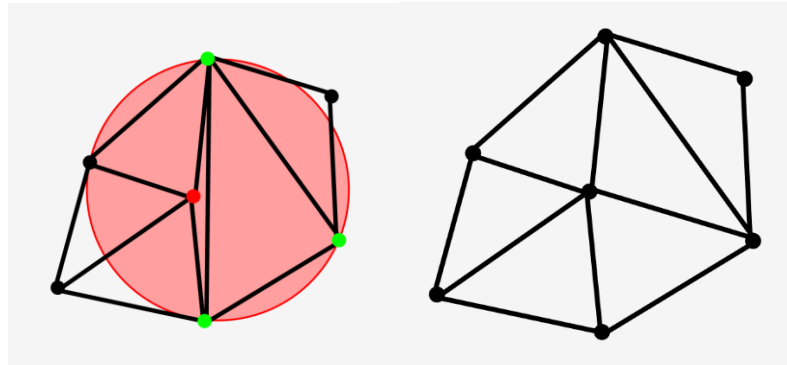


Рис. 5. Проверка на условие Делоне

Для того чтобы проверить условие Делоне для пары треугольников, составляется уравнение окружности с помощью данных о центре окружности и её радиусу, которые находятся с помощью следующей системы линейных уравнений:

$$\begin{bmatrix} 2x_1 & 2y_1 & 1 \\ 2x_2 & 2y_2 & 1 \\ 2x_3 & 2y_3 & 1 \end{bmatrix} * \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} -(x_1^2 + y_1^2) \\ -(x_2^2 + y_2^2) \\ -(x_3^2 + y_3^2) \end{bmatrix}$$

После решения данной системы линейных уравнений с помощью библиотеки линейной алгебры Eigen и применив следующие формулы, получаются необходимые параметры окружности.

$$x_c = -a, y_c = -b, \\ r = \sqrt{x_c^2 + y_c^2 - c},$$

где:

$(x_c, y_c)$  – центр окружности,  $r$  – радиус окружности.

В результате вычисляется триангуляция Делоне для заданных вершин многоугольника.



## 4.2. Решение задачи в пространстве

Реализованный алгоритм триангуляции решает задачу на плоскости, в то время как поставленной задачей является триангуляция в пространстве.

Самым простым методом решения данной проблемы является определение нового базиса и проецирование всех точек, участвующих в триангуляции в данный базис. Этот подход имеет следующие достоинства и недостатки:

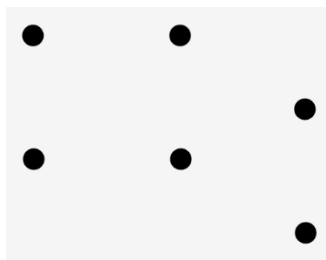
Достоинства:

- скорость преобразования – матрица преобразований для базиса рассчитывается единственный раз после чего применяется для всех точек;
- приведение задачи к двумерной для которой уже есть решение.

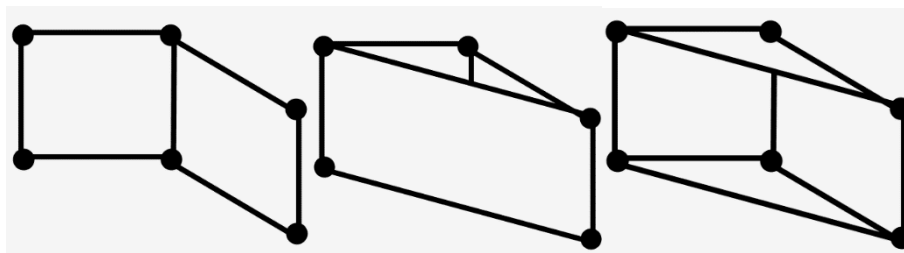
Недостаток:

- точки в пространстве могут лежать не на одной плоскости, тем самым триангуляция будет выполнена неправильно.

Данный недостаток нельзя исправить из-за того, что на вход триангуляции подается только набор вершин, из-за чего однозначно определить, как триангуляция должна быть выполнена и в какой из возможных плоскостей могут лежать точки не представляется возможным (рис. 6-7).



*Рис. 6. Спорный случай набора точек для триангуляции*



*Рис. 7. Различные варианты представления полигонов на заданном наборе точек*

Переход к новому базису позволит решить поставленную задачу с помощью алгоритма, разработанного для точек, определенных на плоскости.

Чтобы перейти к новому базису необходимо определить матрицу перехода, для которой в свою очередь необходимо определить три линейно-независимых вектора и точку начала отсчета.

Чтобы найти линейно независимые вектора, необходимо найти любые три из заданных точек, которые будут образовывать треугольник, в этом случае точно два вектора будут линейно независимы, а третий будет найден через векторное произведение этих векторов.

Алгоритм поиска трёх точек.

1. В качестве первой точки берется крайняя по одной из осей.
2. В качестве второй точки берется любая точка, координата которой, не равна первой точке, иначе говоря, первая и вторая точка образуют прямую.
3. В качестве третьей точки берется любая такая точка, которая не лежит на прямой образованной первой и второй точкой.

После этого берется первая точка в качестве начала отсчета, а также вычисляются два вектора:  $\vec{v}_1, \vec{v}_2$ , которые будут являться основой для вычисления базиса. После чего по следующим формулам, найдем тройку линейно-независимых векторов.

$$\vec{v}_{b1} = \vec{v}_1, \vec{v}_{b2} = \vec{v}_1 \times \vec{v}_2, \vec{v}_{b3} = \vec{v}_1 \times \vec{v}_{b2}$$

Полученные данные используем для вычисления следующей матрицы перехода  $T$ .

$$T = \begin{pmatrix} \overrightarrow{v_{b1x}} & \overrightarrow{v_{b1y}} & \overrightarrow{v_{b1z}} & 0 \\ \overrightarrow{v_{b3x}} & \overrightarrow{v_{b3y}} & \overrightarrow{v_{b3z}} & 0 \\ \overrightarrow{v_{b2x}} & \overrightarrow{v_{b2y}} & \overrightarrow{v_{b2z}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & -\overrightarrow{v_{b1x}} \\ 0 & 1 & 0 & -\overrightarrow{v_{b1y}} \\ 0 & 0 & 1 & -\overrightarrow{v_{b1z}} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Полученная матрица применяется для трехмерных точек полигона после чего выполняется триангуляция.

### 4.3. Модуль геометрии

Структура, применяемая для хранения полигонов и примитивов, может быть разнообразная. Например, в одном случае хранение компонентов в виде массивов может быть оправдано, если геометрия необходима только для отрисовки. Однако в ситуации, когда необходимо провести трассировку или обнаружить коллизию между двумя геометриями, лучше всего хранить компоненты в древовидных структурах данных. И наоборот, затрачивать дополнительное время работы программы на постройку древовидной структуры для геометрии, которая будет только отрисовываться на экране рендеринга неэффективно.

В следствии этого был создан интерфейс, от которого будут наследоваться все классы, хранящие геометрию. Через этот интерфейс реализуется возможность, загрузки геометрии из любого формата или её экспорт, не вдаваясь в подробности реализации структуры хранения.

Следующие методы интерфейса IGeometry, отвечают за выполнение базовых операций над геометрическими примитивами:

- 1) void addVerticle(QVector3D) – добавляет вершину;
- 2) int findVecrticle(QVector3D) – возвращает первый индекс вершины с заданным значением;
- 3) bool deleteVerticle(int) – удаляет вершину с заданным индексом. Если вершины с таким индексом нет возвращает false;
- 4) bool changeVerticle(int, QVector3D) – изменяет значение вершины с заданным индексом;

- 5) `QVector3D verticle(int)` – возвращает вершину под заданным индексом;
- 6) `QVector<QVector3D> vertices()` – возвращает массив вершин;
- 7) `void addTexel(QVector2D)` – добавляет текстель;
- 8) `int findTexel(QVector2D)` – возвращает первый индекс текстеля с заданным значением;
- 9) `bool deleteTexel(int)` – удаляет текстель с заданным индексом. Если текстеля с таким значением нет возвращает `false`;
- 10) `bool changeTexel(int, QVector2D)` – изменяет значение текстеля с заданным индексом;
- 11) `QVector2D texel(int)` – возвращает текстель под заданным индексом;
- 12) `QVector<QVector2D> textels()` – возвращает массив текстелей;
- 13) `void addNormal(QVector3D)` – добавляет нормаль;
- 14) `int findNormal(QVector3D)` – возвращает первый индекс нормали с заданным значением;
- 15) `bool deleteNormal(int)` – удаляет нормаль с заданным индексом. Если нормали с таким индексом нет возвращает `false`;
- 16) `bool changeNormal(int, QVector3D)` – изменяет значение нормали с заданным индексом;
- 17) `QVector3D normal(int)` – возвращает нормаль находящуюся под заданным индексом;
- 18) `QVector<QVector3D> normals()` – возвращает массив нормалей.

Информация о полигоне, хранится в структуре `PoligonInd`, которая содержит в себе следующие данные:

- `m_vertexInd` – индексы вершин, использованных в полигоне;
- `m_texelInd` – индексы текстелей, использованных в полигоне;
- `m_normalInd` – индексы нормалей, использованных в полигоне;
- `m_subPolygons` – описание примитивов субполигонов, составленных из индексов вышеперечисленных списков.

В качестве субполигонов описываются треугольники, которые описывают весь полигон.

Интерфейс содержит следующие методы для получения данных о полигонах.

- 1) `bool addPolygon(QVector<int> vertexs, QVector<int> textelIndexes, QVector<int> normal, QString &errors)` – добавляет полигон с заданными индексами вершин, текстелей и нормалей. Если возникает ошибка – возвращает `false` и пишет в `errors` ошибку, возникшую при попытке добавления;
- 2) `QVector<int> findPolygonsByVerticles(QVector<int>)` – ищет все полигоны в которых присутствуют все индексы вершин переданные в качестве параметра;
- 3) `QVector<int> findPolygonsByTextels(QVector<int>)` – ищет все полигоны в которых присутствуют все индексы текстелей переданные в качестве параметра;
- 4) `QVector<int> findPolygonsByNormals(QVector<int>)` – ищет все полигоны в которых присутствуют все индексы нормалей переданные в качестве параметра;
- 5) `bool deletePolygon(int)` – удаляет полигон с заданным индексом, если такого индекса нет, возвращает `false`;
- 6) `bool changePolygon(int, PolygonInd )` – изменяет полигон с заданным индексом, если полигона с таким индексом не существует возвращает `false`;
- 7) `PolygonInd polygon(int)` – возвращает полигон с заданным индексом.

В отличии от поиска примитивов, поиск полигонов осуществляется по поиску тех примитивов, которые содержатся в полигоне.

Интерфейс `IGeometry` обеспечивает полноценную работу с геометрическими примитивами и полигонами, для всех классов наследников. Однако для корректной работы нужно учитывать следующие условия:

1. Каждому типу примитива и полигону соответствует свой уникальный индекс. Иначе говоря, между индексом и примитивом существует взаимно-однозначное соответствие.

## 2. Структуры хранения, реализуется в классах наследниках.

Таким образом интерфейс IGeometry обеспечивает полноценную работу с геометрией вне зависимости от реализованной структурами хранения данных.

### *Класс GeometryPolygonsSimple*

Класс применяется для хранения геометрии необходимой только для отображения в сцене. По этой причине, структура хранения данных реализована в виде массивов для примитивов и полигонов, но при добавлении полигона дополнительно проводится триангуляция.

Поля:

1. QVector<QVector3D> m\_vertices – вершины геометрии.
2. QVector<QVector2D> m\_textels – текстели геометрии.
3. QVector<QVector3D> m\_normals – нормали геометрии.
4. QVector<PolygonInd> m\_polygons – полигоны геометрии.

Выполнение первого условия достигается путем индекса массива, в котором не может находиться несколько элементов одновременно.

Особенности реализации:

- добавление примитивов не требует каких-либо дополнительных проверок и осуществляется расширением массива и записью в его конечную ячейку;
- поиск примитивов происходит линейно, так как все данные находятся в одномерном массиве;
- удаление примитивов требует также дополнительного удаления полигонов, содержащих этот примитив, а также изменение индексов примитивов у других полигонов в следствии изменения индексов у массива. Из-за этого данная операция является самой трудоёмкой и долгой;
- получение и изменение примитивов происходят за константное время доступа по индексу к массиву;

- добавление полигона требует проверок индексов на присутствие их в массивах примитивов, а также проведение процедуры триангуляции;
- изменение и получение полигона происходит за константное время доступа по индексу к массиву;
- удаление полигона происходит за время удаления из массива одного элемента.

#### 4.4. Модуль адаптеров геометрии для OpenGL

Для отрисовки загруженной геометрии в OpenGL необходимо определить буферы для последующей их передачи на видеокарту. Передача в буферы производится через массив последовательно определенных данных, без ячеек с отсутствующими данными.

OpenGL считывает данные необходимые для отрисовки последовательно, из-за этого происходит перерасход памяти и затрачивается дополнительное время при отрисовке полигонов состоящие из нескольких треугольников. Решение этой проблемы – использование индексного буфера, хранящего индексы на данные из буфера. Такой буфер может быть создан только один. В разработанных классах геометрий присутствуют методы получения всех геометрических примитивов через массив, однако их нельзя использовать ввиду того, что индексы между примитивами не согласованы, так как индексирование каждого из них производится независимо друг от друга.

Для решения данной проблемы создан класс – адаптер. Адаптер – это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Класс GeometryAdapter – использует данные, описывающие геометрию, и формирует на их основе массивы примитивов и индексов, таким образом, чтобы примитивы, находящиеся под одним индексом, образовывали кортеж данных для построения графического примитива в OpenGL.

Класс GeometryAdapter имеет следующие поля:

1. QVector<QVector3D>\* m\_vertices – указатель на массив вершин, готовых к загрузке в буфер OpenGL.
2. QVector<QVector2D>\* m\_textels – указатель на массив текстелей, готовых к загрузке в буфер OpenGL.
3. QVector<QVector3D>\* m\_normals – указатель на массив нормалей, готовых к загрузке в буфер OpenGL.
4. QVector<int>\* m\_indices – указатель на массив индексов, готовых к загрузке в буфер OpenGL.
5. const IGeometry \*m\_geom – константный указатель на класс геометрии.

Методы:

- 1) void setGeom(const IGeometry \*) – передает ссылку на объект геометрии для которого будут строиться данные для буферов;
- 2) QVector<QVector3D>\* vertices() – возвращает указатель на данные вершин для буфера OpenGL;
- 3) QVector<QVector2D>\* textels() – возвращает указатель на данные текстелей для буфера OpenGL;
- 4) QVector<QVector3D>\* normals() – возвращает указатель на данные нормалей для буфера OpenGL;
- 5) QVector<int>\* indices() – возвращает указатель на индексы для кортежей геометрических примитивов;
- 6) void collect() – удаляет предыдущие данные и просчитывает новые данные для кортежей примитивов. Если указатель уже был передан, удаление данных не происходит;
- 7) void clear() – очищает данные, если указатель на данные не был передан для последующей привязке к буферу;
- 8) void deleteAllData() – очищает данные вне зависимости были ли переданы данные или нет.

Класс адаптер, перед составлением кортежей удаляет данные, которые в нем хранились, только если до этого было произведено составление



кортежа и указатель на эти данные не был передан для последующего использования.

Диаграмма разработанных классов представлены в приложении 3.

#### 4.5. Модуль трансформаций

Класс трансформации инкапсулирует в себе базовые аффинные преобразования, применяемые к объекту. А также сохраняет их результат в полях.

Поддерживаемые аффинные преобразования:

- сдвиг;
- поворот;
- масштабирование.

Трансформация реализована в двух классах, разработанных для плоскости и пространства.

##### *Класс Transformation2D*

Данный класс предназначен для хранения аффинных преобразований для элементов интерфейса, изображений, спрайтов и других объектов на плоскости.

Класс содержит в себе следующие поля:

1. QVector2D m\_position – хранит позицию на плоскости.
2. QVector2D m\_scale – хранит коэффициенты масштаба по осям X и Y.
3. float m\_rotationAngle – хранит угол поворота по часовой стрелке.

Также содержит следующие методы:

- 1) void setPosition(QVector2D) – устанавливает позицию на плоскости;
- 2) void setScale(QVector2D) – устанавливает параметры масштаба;
- 3) void setRotationAngle(float rotationAngle);
- 4) void rotate(float) – увеличивает угол поворота на заданное число градусов;

- 5) `QMatrix3x3 getModelMatrix()` – вычисляет матрицу преобразований на основе хранящихся данных;
- 6) `QVector2D getUpVector()` – возвращает вектор координатной оси Y, для данного объекта;
- 7) `QVector2D getRightVector()` – возвращает вектор координатной оси X, для данного объекта;
- 8) `QMatrix3x3 getScaleMatrix()` – возвращает матрицу масштаба следующего вида:

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

где:  $(s_x, s_y)$  – коэффициенты масштаба по осям;

- 9) `QMatrix3x3 getPositionMatrix() const`:

$$\begin{bmatrix} 1 & 0 & p_x \\ 0 & 1 & p_y \\ 0 & 0 & 1 \end{bmatrix},$$

где:  $(p_x, p_y)$  – точка местоположения;

- 10) `QMatrix3x3 getRotationMatrix() const`:

$$\begin{bmatrix} \cos a & -\sin a & 0 \\ \sin a & \cos a & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

где:  $a$  – угол поворота.

### *Класс Transformation3D*

Класс предназначен для хранения информации позиционирования в пространстве объемных объектов, например, геометрии.

Поля:

- 1) `QVector3D m_position` – переменная, хранящая местоположение;
- 2) `QQuaternion m_rotation` – переменная, хранящая поворот объекта в виде кватерниона;
- 3) `QVector3D m_scale` – переменная, хранящая параметры масштаба по трем осям.

Все операции, связанные с поворотом, выполняются через кватернионы. Кватернионы предоставляют удобное математическое обозначение ориентации пространства и вращения объектов в этом пространстве. В сравнении с углами Эйлера, кватернионы позволяют проще комбинировать вращения, а также избежать проблемы, связанной с невозможностью поворота вокруг оси, независимо от совершённого вращения по другим осям. В сравнении с матрицами они обладают большей вычислительной устойчивостью и могут быть более эффективными.

Для работы с кватернионами, используется класс `QQuaternion`, объявленный во фреймворке Qt. Он содержит все операции преобразований. В связи с чем нет необходимости самостоятельно описывать вращение.

Методы:

- 1) `void setPosition(QVector3D)` – задать местоположение объекта;
- 2) `void movePosition(QVector3D)` – сдвинуть объект на заданный вектор;
- 3) `void setRotation` – задает поворот объекта. Имеет следующие перегрузки:
  - a. `void setRotation(float &pitch, float &yaw, float &roll)` – задает вращение в углах Эйлера;
  - b. `void setRotation(QQuaternion)` – задает поворот объекта через кватернион;
  - c. `void setRotation(QVector3D up, QVector3D front, QVector3D right)` – задает поворот объекта тройкой линейно независимых векторов;
- 4) `QVector3D getUpVector()` – возвращает вектор оси Y при текущем повороте;
- 5) `QVector3D getFrontVector()` – возвращает вектор оси Z при текущем повороте;
- 6) `QVector3D getRightVector()` – возвращает вектор оси X при текущем повороте;
- 7) `void getEulersAngles(float &pitch, float &yaw, float &roll)` – возвращает углы Эйлера, эквивалентные текущему повороту;

- 8) void rotateEulerAngles(const float &addPitch, const float &addYaw, const float &addRoll) – совершает поворот по заданным углам Эйлера относительно текущего поворота;
- 9) void getAxisAngle(float &angle, QVector3D &axis) – возвращает ось поворота и угол эквивалентные текущему повороту;
- 10) void rotate() – поворачивает объект относительно текущего поворота.

Имеет следующие перегрузки:

- a. void rotate(QQuaternion) – сложение поворота заданный кватернионом с текущим поворотом объекта;
- b. void rotate(float angle, QVector3D vector) – сложение поворота заданный осью поворота и углом с текущим поворотом объекта;
- 11) void setScale(QVector3D scale) – устанавливает значения коэффициентов масштабирования;
- 12) void addScale(QVector3D scale) – добавляет масштаб по осям;
- 13) QMatrix4x4 getModelMatrix() – возвращает матрицу аффинных преобразований для данного объекта;
- 14) QMatrix4x4 getRotationMatrix() – возвращает матрицу поворота следующего вида:

$$\begin{bmatrix} 1 - 2q_y^2 - 2q_z^2 & 2q_xq_y - 2q_zq_w & 2q_xq_z - 2q_yq_w & 0 \\ 2q_xq_y + 2q_zq_w & 1 - 2q_x^2 - 2q_z^2 & 2q_zq_y - 2q_zq_w & 0 \\ 2q_xq_z - 2q_yq_w & 2q_zq_y - 2q_xq_w & 1 - 2q_x^2 - 2q_y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

где: q(x, y, z, w) – кватернион описывающий поворот;

- 15) QMatrix4x4 getTranslationMatrix() – возвращает матрицу сдвига следующего вида:

$$\begin{bmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

где: p(x, y, z) – точка местоположения;

16) QMatrix4x4 getScaleMatrix() – возвращает матрицу масштабирования следующего вида:

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

где:  $s(x, y, z)$  – коэффициенты масштаба.

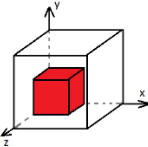
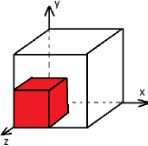
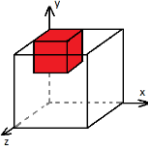
#### 4.6. Модуль октодерева

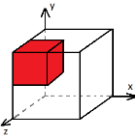
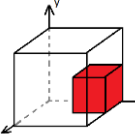
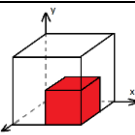
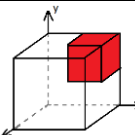
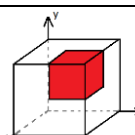
В качестве структуры хранения вокселей, используется октодерев. При его разработке возникают следующие проблемы.

- идентификации нод октодерева;
- память, затрачиваемая на хранение одной ноды;
- очистка памяти и балансировка;
- проход по всем листья дерева;
- распределение обязанностей между выполнением операций над одной нодой или над всем деревом.

Для идентификации ноды, будет использоваться значение от 0 до 7. Каждое значение означает физическое расположение памяти (Таблица 1).

Таблица 1. Соответствие индексов нод октодерева

Значение	Предполагаемое расположение	Знак координаты
0		$X < 0, Y < 0, Z < 0$
1		$X < 0, Y < 0, Z > 0$
2		$X < 0, Y > 0, Z < 0$

3		$X < 0, Y > 0, Z > 0$
4		$X > 0, Y < 0, Z < 0$
5		$X > 0, Y < 0, Z > 0$
6		$X > 0, Y > 0, Z < 0$
7		$X > 0, Y > 0, Z > 0$

Чтобы сэкономить память каждая нода будет хранить только индекс, обозначающий ее расположение в памяти родителя. Исключением будет, только, если октодерево состоит из одной ноды-листа. В следствии того, что для идентификаций ноды требуется одно из девяти значений (восемь значений для определения положение у родителя и одно для обозначения единственной ноды), используется тип `unsigned char` для хранения индентификатора, так как он занимает один байт памяти.

Полный адрес конкретной ноды, будет составлен из массива индентификаторов, который далее будем называть октоключом.

*Класс октоключа*

Определим класс, хранящий индентификаторы к одной ноде.

Поле:

`QVector<unsigned char> m_key` – массив индентификаторов для нод октодерева.

Методы:

- 1) `QVector<unsigned char> key()` – возвращает массив идентификаторов ключа;
- 2) `unsigned char last()` – возвращает последний идентификатор;
- 3) `int depth()` – возвращает длину ключа;
- 4) `bool down()` – переходит к идентификатору потомка;
- 5) `bool up()` – переходит к идентификатору родителя;
- 6) `bool right()` – переходит к идентификатору соседней ноды;
- 7) `bool left()` – возвращается к идентификатору предыдущей ноды;

Методы `down()`, `up()`, `right()`, `left()` необходимы для прохода по дереву (рис. 8).

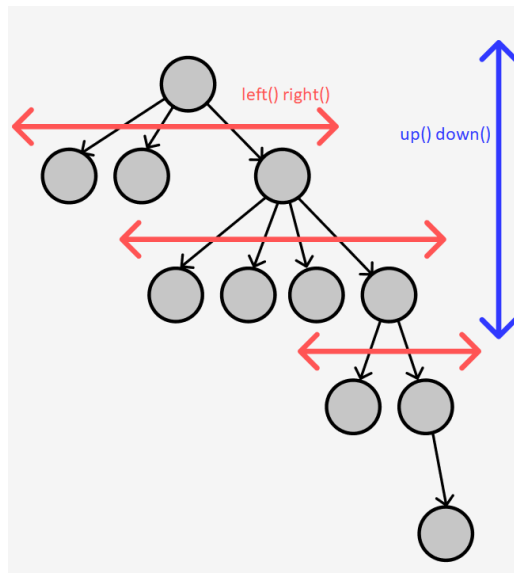


Рис. 8. Методы октоключа для обхода дерева

Данный класс совмещает в себе хранилище ключа, а также способы прохода по октодереву. Стоит учитывать, что при совершении операций перехода к идентификатору соседей, он не может перемещаться к идентификатору родителя или потомка, и наоборот.

*Классы `OctoNode`, `OctoBranch`, `OctoLeaf`*

Реализация проведена с помощью паттерна проектирования – Компоновщик.

Класс `OctoNode` определяет интерфейс для класс хранящий ссылки на другие ноды и класс хранящий ссылку на объект. Таким образом обращение

классам происходит через единый интерфейс. Методы каждой ноды разработаны для работы только с данными текущей ноды, либо создание или удаление ссылок на своих потомкой 1-го поколения.

Класс OctoNode.

Поле:

`const unsigned char m_id` – неизменяемый идентификатор ноды, который задается при инициализации и в последствии не меняется.

Методы:

- 1) `virtual void clear()` – переопределяемый метод для очистки памяти октодерева;
- 2) `virtual bool canSplit()` – переопределяемый метод для проверки возможности создания потомков ноды.;
- 3) `virtual bool canUnite()` – переопределяемый метод для проверки возможности удаления потомков ноды;
- 4) `bool split(unsigned char id)` – метод создания ноды-ветки в качестве потомка;
- 5) `bool unite(unsigned char id)` – метод создания ноды-листа в качестве потомка;
- 6) `virtual OctoNode *getChild(unsigned char id)` – переопределяемый метод для получения потомка по его айдишнику;
- 7) `virtual T* value()` – переопределяемый метод получения объекта;
- 8) `virtual bool setValue(T value)` – переопределяемый метод присваивания объекта;
- 9) `virtual int getType()` – переопределяемый метод для получения типа ноды, которые определены в перечислении `NodeType`;
- 10) `unsigned char id()` – получение идентификатора;
- 11) `virtual QVector<OctoNode<T> *> *getChildren()` – переопределяемый метод получения потомков;

Различия между реализациями `OctoBranch` и `OctoLeaf`, являются типы указателей и реализация методов для работы с ними. `OctoBranch` – содержит в



качестве поля, массив указатель на дочерние объекты, а OctoLeaf – содержит в качестве поля указатель на объект.

Описание работы методов приведено в таблице 2.

Таблица 2. Сравнение методы OctoBranch и OctoLeaf

Метод	OctoBranch	OctoLeaf
void clear()	Вызывает метод clear() у всех потомков с ненулевым указателем, затем удаляет область память на которую указывает указатель.	Удаляет область памяти на которую указывает указатель на объект, если он ненулевой.
bool canSplit()	Возвращает false, так как данная нода уже является OctoBranch.	Возвращает true, если указатель на объект нулевой.
bool canUnite()	Возвращает true, если все указатели на потомки являются нулевыми.	Возвращает false так как, данная нода уже является OctoLeaf.
OctoNode *getChild(unsigned char id)	Возвращает значение указателя указывающий на потомка с указанным индексом.	Возвращает пустой указатель.
T* value()	Возвращает пустой указатель.	Возвращает указатель на объект.

<code>bool setValue(T value)</code>	Возвращает <code>false</code> .	Присваивает новое значение области памяти указывающей на объект, либо выделяет память под объект.
<code>int getType()</code>	Возвращает <code>NodeType::Branch</code>	Возвращает <code>NodeType::Leaf</code>
<code>QVector&lt;OctoNode&lt;T&gt; *&gt; *getChildren()</code>	Возвращает указатель на массив указателей потомков.	Возвращает пустой указатель.

### Класс *OctoTree*

Раждающая нода в разработанной структуре отвечает сама за себя, они не могут отвечать работу над октодеревом в целом. Для этих целей разработан класс *OctoTree*, который будет иметь ту же самую логику что и ноды, но при этом совершать операции над всем деревом. Таким образом мы используем паттерн проектирования фасад.

Использование данного подхода в сравнении с возложением обязанностей выполнения операций над всем деревом на ноды, помогает избежать рекурсии, которую достаточно сложно контролировать, а также разделить обязанности между классами с разными предназначениями.

Поле:

`OctoNode<T>* m_mainNode` – указатель на первый элемент октодерева.

Методы:

- 1) `void clear()` – очищает все дерево, и приводит его к состоянию после инициализации;

- 2) `void balance()` – балансирует все дерево, удаляя ноды с нулевыми ссылками;
- 3) `bool setValue(OctoKey &key, T &value)` – устанавливает значение в дерево по заданному ключу, если ноды типа лист по данному индексу найдено не было возвращает `false`;
- 4) `bool exist(OctoKey &key, int type)` – ищет и проверяет существует ли нода с заданным ключом и заданным типом;
- 5) `bool get(OctoKey &key, T &value)` – передает ссылку на значение хранимое в ноде по заданному ключу, если нода существует и является листом;
- 6) `bool remove(const OctoKey &key)` – очищает ноду по заданному ключу, возвращает `false`, если такой ноды не существует;
- 7) `bool canSplit(OctoKey &key)` – проверяет ноду с заданным ключом на возможность преобразования её в `OctoBranch`;
- 8) `bool canUnite(const OctoKey &key)` – проверяет ноду с заданным ключом на возможность преобразования её в `OctoLeaf`;
- 9) `bool splitNode(OctoKey &key)` – преобразует, если это возможно, ноду с заданным ключом в ноду типа `OctoBranch`;
- 10) `bool uniteNode(OctoKey &key)` – преобразует, если это возможно, ноду с заданным ключом в ноду типа `OctoLeaf`;
- 11) `int getMaxDepth()` – возвращает глубину дерева;
- 12) `bool findCurrentNode(OctoKey &key, OctoNode<T> *&node)` – ищет ноду с заданным ключом, и если она существует, передает на нее указатель.

### *Класс итератора на октодерево*

Октодерево является контейнером, поэтому для перебора всех его элементов, например для поиска все вхождений какого-либо объекта, разработан класс итератора.

Итератор — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

Класс итератор использует методы октоключа для прохода по октодереву (рис. 9). Класс октоключа не может перемещаться вертикально (методы `up()`, `down()`), при выполнении горизонтального перехода (методы `right()`, `left()`). Поэтому реализация перехода между листьями на различных уровнях реализуется в классе итератора.

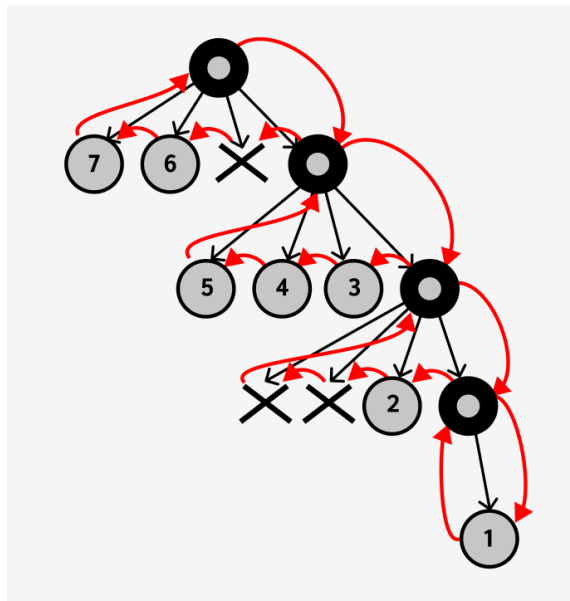


Рис. 9. Схема обхода дерева итератором

Поля:

1. `const OctoTree<T> *m_tree` – ссылка на объект октодерева;
2. `OctoKey m_currentKey` – ключ текущей ноды;
3. `int m_currentStatus` – текущий статус: `Iterator::End` – значит, что итератор находится в конце, `Iterator::NotEnd` – значит, что не дошел до конца.

Методы:

- 1) `bool hasNext()` – метод проверяет, есть ли следующая лист-нода с хранящимся значением в октодереве;
- 2) `bool next()` – ищет ключ от следующей лист-нода с хранящимся значением в октодереве, если таковой не найдено ставит в статус

итератора флаг о конце прохода и возвращает false, если итератор уже в конце возвращает false;

3) bool operator++() – аналогичен методу next();

4) T operator\*() – возвращает значение в текущей ноде-листе;

5) OctoKey currentKey() – возвращает текущий ключ ноды на которой остановился итератор.

Диаграмма разработанных классов представлены в Приложении 1.

### *Класс Вокселей*

Класс хранит размер и ссылку на материал, разработан с использованием паттерн проектирования Легковес.

Легковес — это структурный паттерн проектирования, который позволяет вместить бóльшее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.

Воксель предназначен для хранения информации об одном элементе воксельного объекта, в связи с этим его память ограничена.

## 4.7. Модуль импорта файлов

Для добавления возможности чтения из, необходимых нам, форматов, обеспечивая при этом расширяемость модуля, реализуем классы импорта с помощью паттерна проектирования Шаблонный метод.

Шаблонный метод — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

Класс Parser имеет следующие поля:

1. QFile m\_file – объект файла для считывания;
2. QTextStream m\_stream – текстовый поток, содержащий содержимое из файла.

Методы:

- 1) `bool openFile(QString &filename)` – метод открытия файла по указанному пути, если файл открыть не удалось возвращает `false`;
- 2) `void closeFile()` – закрытие файла;
- 3) `bool parseInt(QString &sValue, int &value)` – преобразует строку в целое число, если преобразование не удалось, возвращает `false`;
- 4) `bool parseFloat(QString &sValue, float &value)` – преобразует строку в число с плавающей точкой, если преобразование не удалось, возвращает `false`;
- 5) `bool parseDouble(const QString &sValue, double &value)` – преобразует строку в число с плавающей точкой двойной точности, если преобразование не удалось, возвращает `false`;
- 6) `bool parseQVector3D(QStringList &sValue, QVector3D &value)` – преобразует массив строк в вектор с тремя значениями, если преобразование не удалось, возвращает `false`;
- 7) `bool parseQVector2D(QStringList &sValue, QVector2D &value)` – преобразует массив строк в вектор с двумя значениями, если преобразование не удалось, возвращает `false`;
- 8) `virtual bool parseFromFile(const QString &filename, QString *errors = nullptr)` – переопределяемый метод, для считывания данных из файла.

На данный момент реализованы следующие наследники класса `Parser`:

- `ObjParser` – считывает геометрию из файлов формата `obj`;
- `MtlParser` – считывает материал, описанный в формате `mtllib`.

Алгоритм для считывания файлов выглядит следующим образом:

1. Открытие файла;
2. Построчное считывание данных, записанных в файл:
  - a. Если встречен токен комментария, пропуск строки;
  - b. Если токен не распознан, строка также пропускается;
  - c. Если встречен один из токенов, происходит распознавание его значения и запись в объект если значение токена не содержит ошибок;

- d. Если при считывании токена встретилась ошибка, происходит запись в строку с ошибками;

### 3. Заккрытие файла.

В таблице 3 представлен список токенов поддерживаемый классом ObjParser.

Таблица 3. Токены формата Obj

Тэг	Описание
#	Комментарий
v	Вершина
vt	Текстель
vn	Нормаль
f	Описание полигона

ObjParser содержит поле класса IGeometry, поэтому при считывании геометрических примитивов и полигонов, они сразу добавляются в класс геометрии.

В таблице 4 представлен список токенов поддерживаемый классом MtlParser.

Таблица 4. Токены формата Mtl

Тэг	Описание
#	Комментарий
newmtl	Название материала
Ka	Цвет фонового освещения
Kd	Цвет рассеянного освещения
Ks	Цвет отраженного освещения
Ke	Цвет излучения
map_Ka	Имя файла текстуры фонового освещения
map_Kd	Имя файла текстуры рассеянного освещения
map_Ks	Имя файла текстуры отраженного освещения

map_Ke	Имя файла текстуры излучения
map_Ns	Имя файла текстуры с альфа-каналом для отраженного освещения
map_d	Имя файла текстуры с альфа-каналом
map_Bump	Имя файла карты нормалей
d	Коэффициент прозрачности
Ni	Коэффициент отражения
Ns	Коэффициент блеска отраженного освещения

Класс MtlParser содержит класс Material, в который при считывании токенов записываются данные о материале.

Диаграмма разработанных классов представлены в Приложении 2.

#### 4.8. Модуль материала

Для описания цветов и текстуры как самих вокселей, так и геометрий, используется класс материала. Он описывает различные параметры, использующиеся при расчете освещения.

В данном классе могут храниться следующие типы параметров.

1. Цвета в трёхканальном формате;
2. Различные карты;
3. Коэффициенты.

В таблице 5, представлены поддерживаемые параметры материала.

Таблица 5. Список поддерживаемых параметров материала

Цвет	Текстура	Коэффициент
Фоновый	Фонового цвета	Прозрачность
Рассеянный	Рассеянного цвета	Отражения
Отраженный	Отраженного цвета	Блеск отраженного освещения.
Излучения	Цвета излучения	



	Карта нормалей	
	Карта альфа канала	
	Карта - альфа-канала для отраженного освещения	

Все параметры хранятся в ассоциативных контейнерах – словарях. Сделано это для сохранения памяти, так как в одном случае, материал состоит, только из цвета, а в другом – из текстур и коэффициентов.

Поля:

1. QString name – имя материала.
2. int flags – флаги материала.
3. QMap<int, QVector3D> m\_colors – контейнер цветов.
4. QMap<int, float> m\_coefficients – контейнер с коэффициентами.
5. QMap<int, QString> m\_maps – контейнер с путями к текстурным файлам.

Методы:

- 1) bool existColor(Types type) – проверяет есть ли в контейнере, цвет с заданным типом;
- 2) bool getColor(Types type, QVector3D &color) – возвращает цвет с заданным типом, если такого, нет возвращает false;
- 3) void setColor(Types type, QVector3D color) – добавляет, или заменяет цвет с заданным типом;
- 4) bool existMap(Types type) – проверяет есть ли в контейнере, текстура с заданным типом;
- 5) bool getMap(Types type, QString &mapPath) – возвращает путь к текстуре с заданным типом, если такого, нет возвращает false;
- 6) void setMap(Types type, QString mapPath) – добавляет, или заменяет путь к текстуре с заданным типом;
- 7) bool existCoefficient(Types type) – проверяет есть ли в контейнере, коэффициент с заданным типом;

- 8) `bool getCoefficient(Types type, float &coef)` – возвращает коэффициент с заданным типом, если такого, нет возвращает `false`;
- 9) `void setCoefficient(Types type, const float coef)` – добавляет, или заменяет коэффициент с заданным типом;
- 10) `int getFlags()` – возвращает все флаги которые были проставлены у материала;
- 11) `void setFlag(Flags type)` – добавляет флаг;
- 12) `void disableFlag(Flags type)` – убирает флаг;
- 13) `void clear()` – очищает все контейнеры материалов и флаги.

Дополнительно в классе материала определены флаги и типы, необходимые для параметров освещения они указаны в Приложении 9.

Диаграмма разработанных классов представлены в Приложении 5.

#### 4.9. Модуль камеры

Класс камеры, описывает параметры камеры и содержит методы по её управлению. Камера наследует класс `Transformation3D`, так как она должна иметь позицию и ориентацию, как и остальные объекты на сцене.

Поля:

1. `float m_fovY` – угол поля зрения выраженный в градусах.
2. `float m_near` – расстояние до ближней грани усеченной пирамиды.
3. `float m_far` – расстояние до дальней грани усеченной пирамиды.
4. `QVector2D principalPoint` – смещение оптической оси.
5. `View m_type` – параметр просмотра камеры (от первого или от третьего лица).

Методы:

- 1) `void setFovY(float fovY)` – устанавливает угол обзора камеры;
- 2) `QMatrix4x4 lookAt()` – вычисляет матрицу вида с типом камеры(от первого лица, от третьего лица);
- 3) `QMatrix4x4 getPerspectiveMatrix()` – возвращает перспективную матрицу для камеры;

- 4) `QMatrix4x4 getOrtograficMatrix()` – возвращает ортогональную матрицу для камеры;
- 5) `void setNear(float near)` – устанавливает расстояние до ближней плоскости усеченной пирамиды;
- 6) `void setFar(float far)` – устанавливает расстояние до дальней плоскости усеченной пирамиды.

В классе предусмотрены рендер с помощью двух видов камер: от первого лица и от третьего лица. Для корректной работы шейдеров, были переопределены методы класса `Transformation3D`:

- 1) `QVector3D getPosition();`
- 2) `void movePosition(QVector3D &move);`
- 3) `QMatrix4x4 getModelMatrix();`

Разница между вычитыванием данных для видов камеры – в порядке умножения матриц. Для вида от первого лица используется порядок матриц 1, для вида от третьего лица используется матрица 2.

$$M_{fps} = scaleMatrix * rotationMatrix * translationMatrix$$

$$M_{tps} = scaleMatrix * translationMatrix * rotationMatrix$$

`Camera3D` являясь наследником `Transformation3D`, наследует все его методы по вращению и перемещению в пространстве.

Диаграмма разработанных классов представлены в приложении 4.

#### 4.10. Модуль трассировщика

Для приложения необходимо, вычислять пересечение луча и полигонов геометрии. Для этих целей разработан небольшой модуль для трассировки над одним объектом геометрии.

Поле:

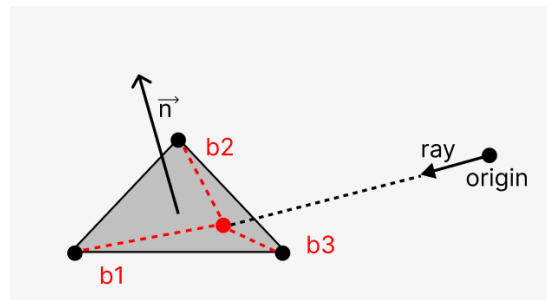
`const IGeometry *m_geom` – константная ссылка на класс геометрии.

Метод:

`bool nearestToRaySurfaceData(QVector3D rayOrigin, QVector3D rayDirection, QVector3D &nearestPoint, QVector3D &normal, double &distance)` – поиск ближайшей точки пересечения по направлению луча, и вычисление данных о плоскости, которую пересек луч.

Вычисление выполняется по следующему алгоритму:

1. Проверка, на перпендикулярность луча и нормали плоскости.
2. Вычисление точки пересечения.
3. Проверка, что точка находится в положительном направлении луча направления.
4. Нахождение барицентрических координат.
5. Проверка на нахождение точки пересечения внутри треугольника (рис. 11).



*Рис. 10. Геометрическое представление работы трассировщика*

Для вычисления точки пересечения луча и плоскости, образованной полигоном, данные о луче выражаются через параметрическое уравнение прямой, а с помощью точек выражается плоскость через уравнение плоскости.

$$\begin{cases} x = \lambda N_x + d_x \\ y = \lambda N_y + d_y, \\ z = \lambda N_z + d_z \end{cases}$$

где:  $N$  – направление луча,  $p$  – точка начала луча.

$$N_x(x - p_x) + N_y(y - p_y) + N_z(z - p_z) = 0,$$

где  $N$  – нормаль плоскости,  $p$  – точка на плоскости.

Подставив параметрическое уравнение прямой в уравнение плоскости.

$$N_x(\lambda N_x + d_x - p_x) + N_y(\lambda N_y + d_y - p_y) + N_z(\lambda N_z + d_z - p_z) = 0$$

Через вычисленный параметр  $\lambda$ , вычисляется точка пересечения через параметрическое уравнение прямой. Если  $\lambda < 0$ , то точка пересечения находится за точкой начала луча и тем самым она не подходит.

Следующим шагом является вычисление барицентрических координат.

$$\begin{bmatrix} b_0 \\ b_1 \end{bmatrix} = \begin{bmatrix} P_{1x} - P_{0x} & P_{1y} - P_{0y} \\ P_{2x} - P_{0x} & P_{2y} - P_{0y} \end{bmatrix}^{-1} * \overrightarrow{(A - P_0)},$$

где:

$P_1, P_2, P_3$  – точки на плоскости, описывающие барицентрическую систему координат.

$A$  – точка для которой вычисляются координаты в барицентрической системе координат.

$b_0, b_1$  – барицентрические координаты.

Последняя барицентрическая координата  $b_3$ , вычисляется следующим образом:

$$b_3 = 1 - b_1 - b_2$$

Одним из свойств барицентрических координат, является то, что, если все координаты находятся в диапазоне  $[0, 1]$ , то точка, описываемая барицентрическими координатами, находится внутри области, ограниченной точками. С помощью этого свойства, определяется, что луч пересекает полигон.

#### 4.11 Модуль приложения

Класс напрямую работает с интерфейсом и содержит методы, взаимодействующие с внутренней структурой, содержащей вышеописанные модули.

Поля:

1. `QOpenGLShaderProgram m_program` – шейдерная программа.
2. `QOpenGLBuffer m_ibo` – индексный буфер.
3. `QOpenGLBuffer buffers[3]` – буферы для графических примитивов.

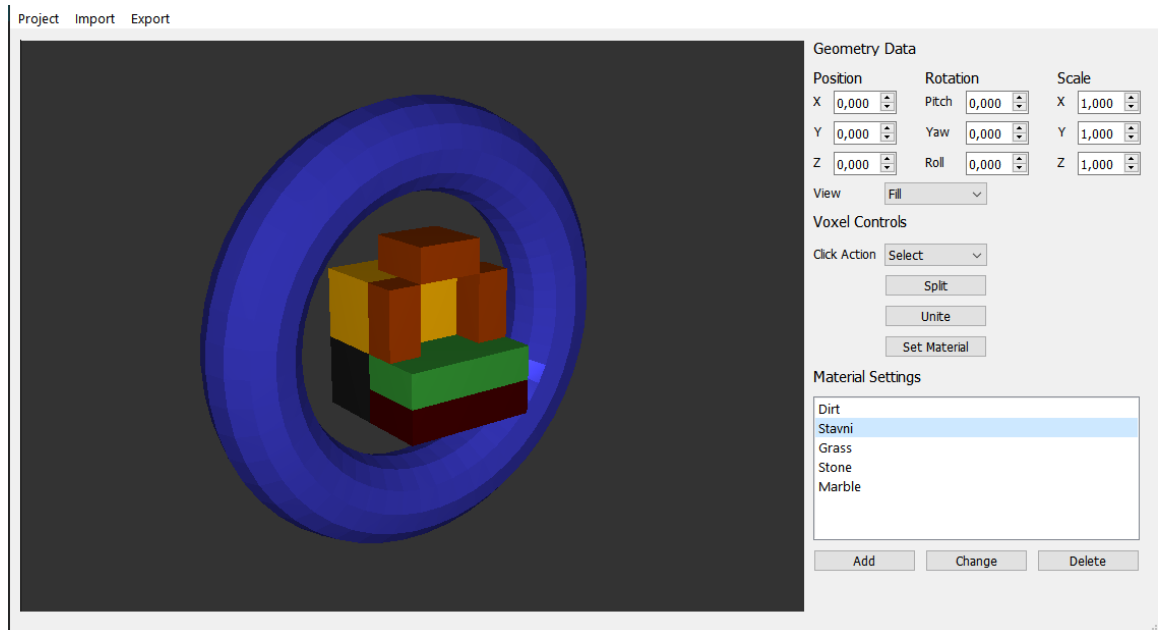
4. OctoTree<Voxel> m\_octotree – октодерево вокселей.
5. Camera3D camera – камера.
6. GeometryPolygonsSimple geom – геометрия.
7. QVector3D backgroundColor – цвет фона.
8. QTime m\_lastMoveTime – время последнего перемещения по сцене пользователем.
9. QTime m\_lastRotateTime – время последнего вращения камеры пользователем.
10. QPoint m\_lastPosition – последняя позиция на экране.
11. bool m\_isRotating – флаг вращения камеры.
12. bool m\_isMoving – флаг перемещения камеры.
13. bool refreshingBuffers – флаг обновления буферов.

Методы:

- 1) void mousePressEvent(QMouseEvent \*event) – метод отслеживания, нажатия кнопки мыши;
- 2) void mouseReleaseEvent(QMouseEvent \*event) – метод отслеживания, отпуска кнопки мыши;
- 3) void keyPressEvent(QKeyEvent \*event) – метод отслеживания, нажатия кнопки клавиатуры;
- 4) void keyReleaseEvent(QKeyEvent \*event) – метод отслеживания, нажатия кнопки клавиатуры;
- 5) void mouseMoveEvent(QMouseEvent \*event) – метод отслеживания, перемещения мыши;
- 6) bool loadGeom(QString filepath) – загрузка геометрии из файла;
- 7) bool loadMaterial(QString filepath) – загрузка материала из файла;
- 8) void paintGL() – метод для отрисовки с помощью OpenGL;
- 9) void initShaders() – метод инициализации шейдеров;
- 10) void initGeom() – метод загрузки геометрии;
- 11) void initializeGL() – инициализация данных для OpenGL;
- 12) void initBuffers() – инициализация буферов.

## 5. Интерфейс

В результате работы был сформирован следующий интерфейс (рис.11).

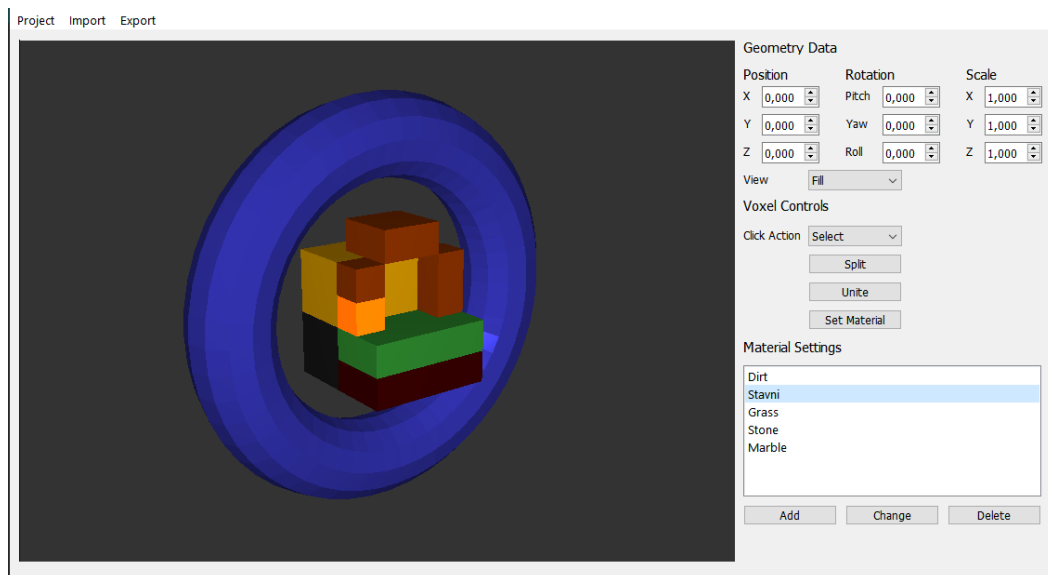


*Рис. 11. Интерфейс главного меню*

В середине окна находится виджет, отображающий текущее состояние сцены: загруженную геометрию и воксельную модель. Пользователь может осматривать сцену путем зажатия кнопки мыши и перемещения ее в соответствующую сторону. Камера по умолчанию имеет вид от третьего лица. Для того чтобы переместить камеру на сцене нужно зажать соответствующую клавишу со стрелками на клавиатуре.

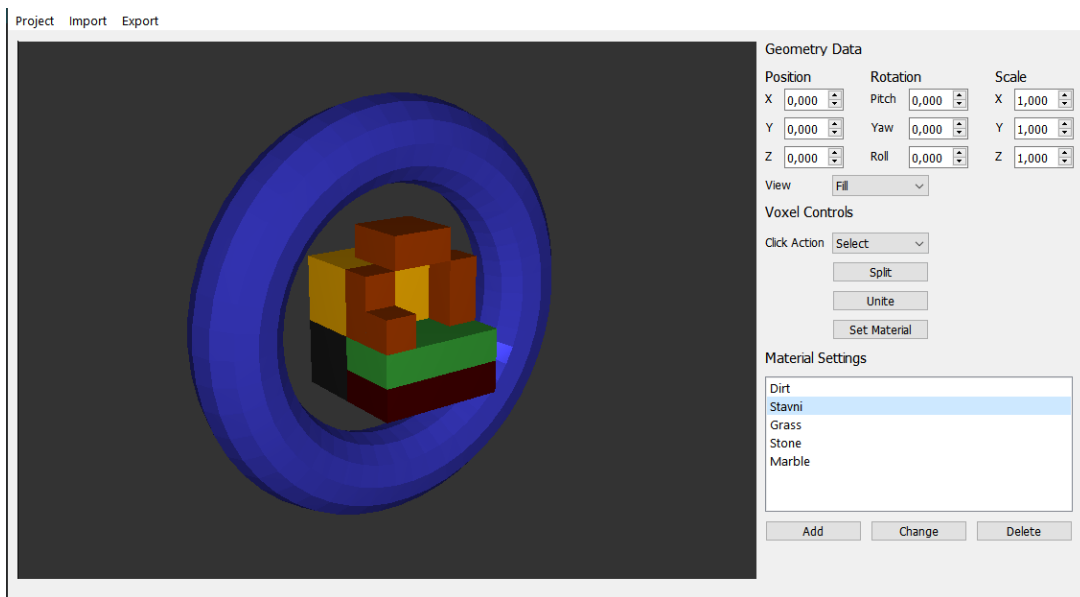
При нажатии левой клавиши мыши на виджет сцены, произойдет одно из перечисленных действий, связанных с ближайшим вокселем по направлению взгляда камеры.

1. Выделение вокселя – воксель модели станет светлее тем самым, сообщая пользователю, что все последующие операции будут выполняться именно над ним (рис. 12).



*Рис. 12. Подсвечивание вокселя при его выборе*

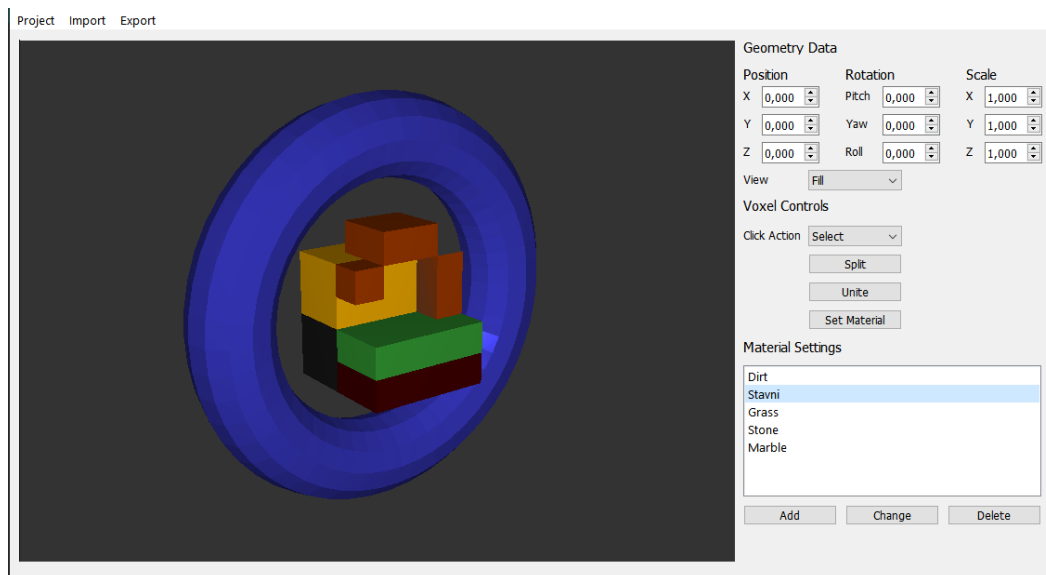
2. Добавление вокселя – добавиться воксель, являющийся смежным к грани вокселя, на который был направлен взгляд камеры (рис. 13).



*Рис. 13. Добавление вокселя*

3. Удаление вокселя – выбранный воксель будет удален со сцены (рис. 14).



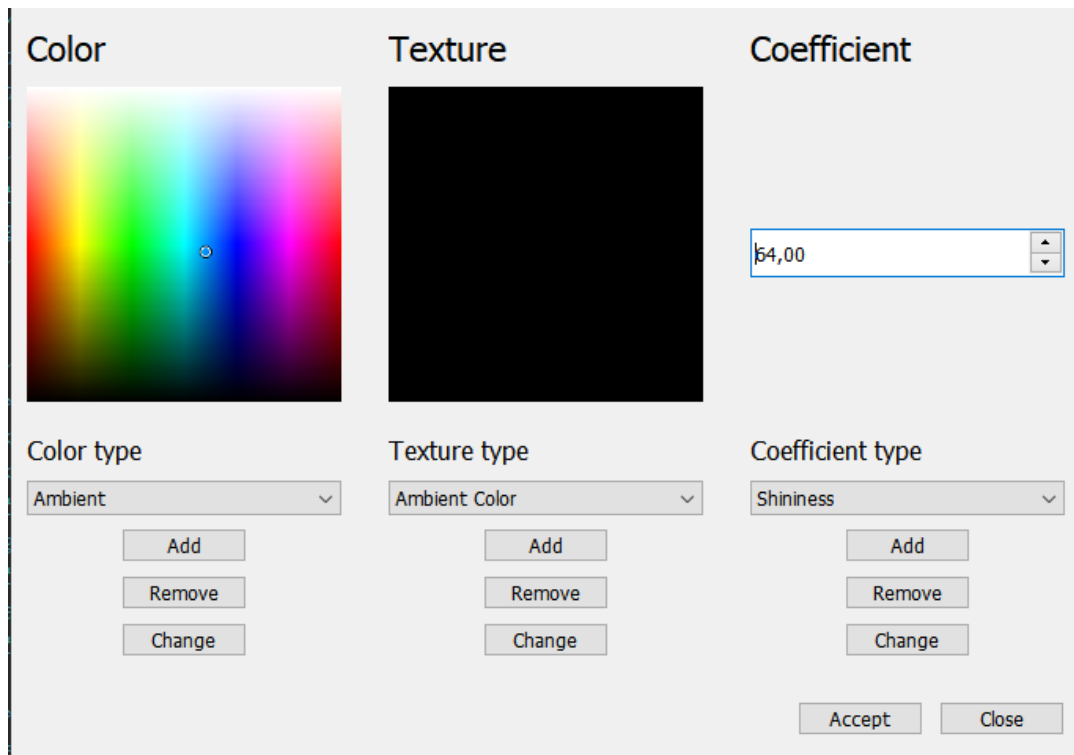


*Рис. 14. Удаление вокселя*

В правой части окна находятся параметры о текущем состоянии сцены и объектов на ней. Они делятся на следующие блоки:

1. Параметры геометрии. В данном блоке, находятся данные о точке местоположения, углах вращения и коэффициентов масштаба для загруженной геометрии. Все параметры можно изменять, нажав левой кнопкой мыши на параметр, вращая колесико мыши, либо вручную вводя необходимые значения с помощью цифр на клавиатуре;
2. Кнопки для управления вокселем. В данном блоке расположено несколько кнопок, для изменения состояния выделенных воксенного вокселя:
  - а. Разбиение вокселя на восемь других, имеющих меньший размер и составленных в форме куба;
  - б. Объединение ячейки, в которой расположен воксель. Объединение может произойти в двух случае, если воксели в заданной ячейке либо имеют одинаковый размер и материал, либо отсутствуют;
  - с. Изменение материала для вокселя. Вокселю будет присвоен текущий активный материал;
3. Блок управления материалами. Данный блок содержит список все существующих в программе материалов, а также кнопки для

добавления, удаления и изменения материала. При нажатии кнопки изменения материала пользователю показывается диалог с возможностью поменять параметры материалов (рис. 15).

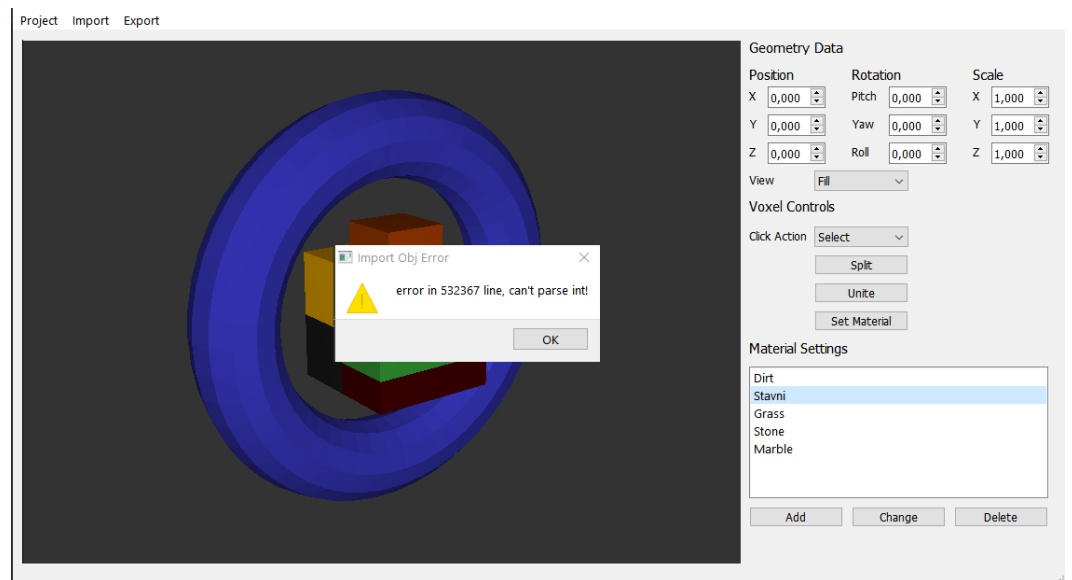


*Рис. 15. Диалог настройки материала*

В данном диалоге, можно выбрать соответствующий цвет, загрузить текстуру из файла, либо задать необходимый коэффициент.

В верхней части окна расположен блок с кнопками, отвечающими за работу с проектом:

1. Выполнение операций над всем проектом: создание нового, сохранение текущего и загрузка уже существующего;
2. Импорт геометрии или материала из реализованных форматов. Если импорт файла по какой-либо причине не удался на экране появится предупреждение пользователю с описанием возникших при импорте ошибок (Рис. 16).



*Рис. 16. Оповещение пользователя об ошибке при импорте*

## **Заключение**

В ходе выполнения ВКР было разработано программное обеспечение для создания воксельных объектов. При этом были реализованы различные модули для структурированного хранения данных о моделях, отображения их пользователю, а также методы по изменению структуры данных в соответствии с действиями пользователя через разработанный интерфейс. Реализация программного обеспечения была выполнена с помощью языка программирования C++, фреймворка Qt и спецификации OpenGL.

Разработанная архитектура приложения предполагает дальнейшие модификации и оптимизацию работы внутренних моделей приложения, в следствии применения паттернов проектирования.

В дальнейшем планируется расширение функционала приложения для добавления новых инструментов работы с вокселями, добавлению новых форматов для импорта и экспорта, а также оптимизации процессов отрисовки.

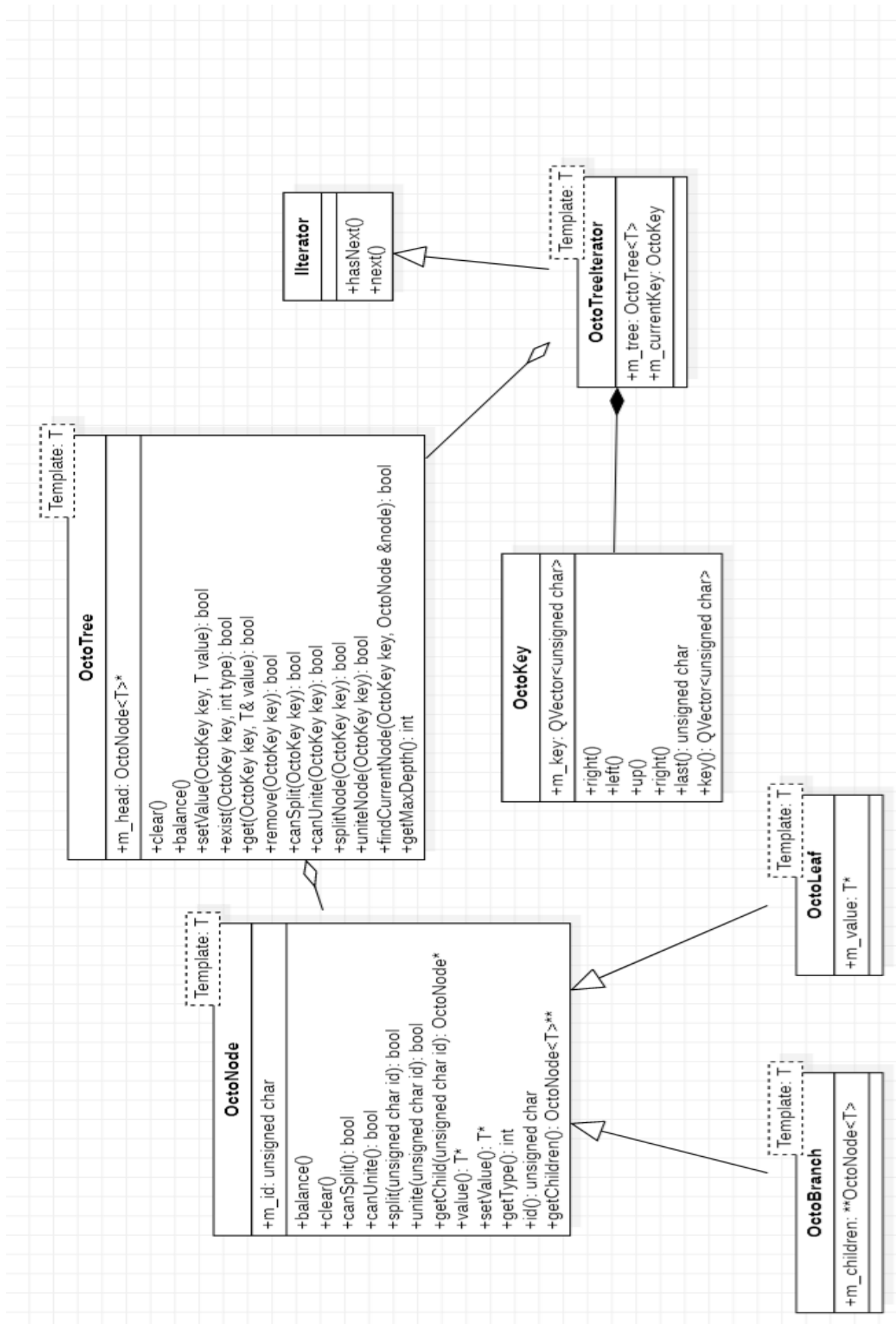
## Список использованных источников

- 1) Боресков А.В. Программирование компьютерной графики. Современный OpenGL. М.: ДМК Пресс, 2019. – 372 с.;
- 2) Qt фреймворк [Электронный ресурс]. – Режим доступа: <https://www.qt.io/> - ы(дата обращения 10.08.2021);
- 3) Библиотека Eigen [Электронный ресурс]. – Режим доступа: [https://eigen.tuxfamily.org/index.php?title=Main\\_Page](https://eigen.tuxfamily.org/index.php?title=Main_Page) - (дата обращения 24.08.2021);
- 4) Скворцов А.В. - Триангуляция Делоне и её применение. – Томск: Изд-во Том. ун-та. 2002. – 128 с. ;
- 5) Алгоритм триангуляции Делоне методом замещающей прямой [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/post/445048/> - (дата обращения 10.01.2022);
- 6) Построение минимальных выпуклых оболочек [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/post/144921/> - (дата обращения 12.01.2022);
- 7) Формат Obj [Электронный ресурс]. – Режим доступа: <https://ru.wikipedia.org/wiki/Obj> - (дата обращения 4.03.2022);
- 8) Преобразование базис [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/post/339968/> - (дата обращения 17.02.2022);
- 9) Переход к новому базису и к новой системе координат [Электронный ресурс]. – Режим доступа: [http://www.mathprofi.ru/perehod\\_k\\_novomu\\_bazisu.html](http://www.mathprofi.ru/perehod_k_novomu_bazisu.html) - (дата обращения 17.02.2022);
- 10) Паттерн проектирования адаптер [Электронный ресурс – Режим доступа: <https://refactoringguru.ru/design-patterns/adapter> - (дата обращения 21.01.2022);

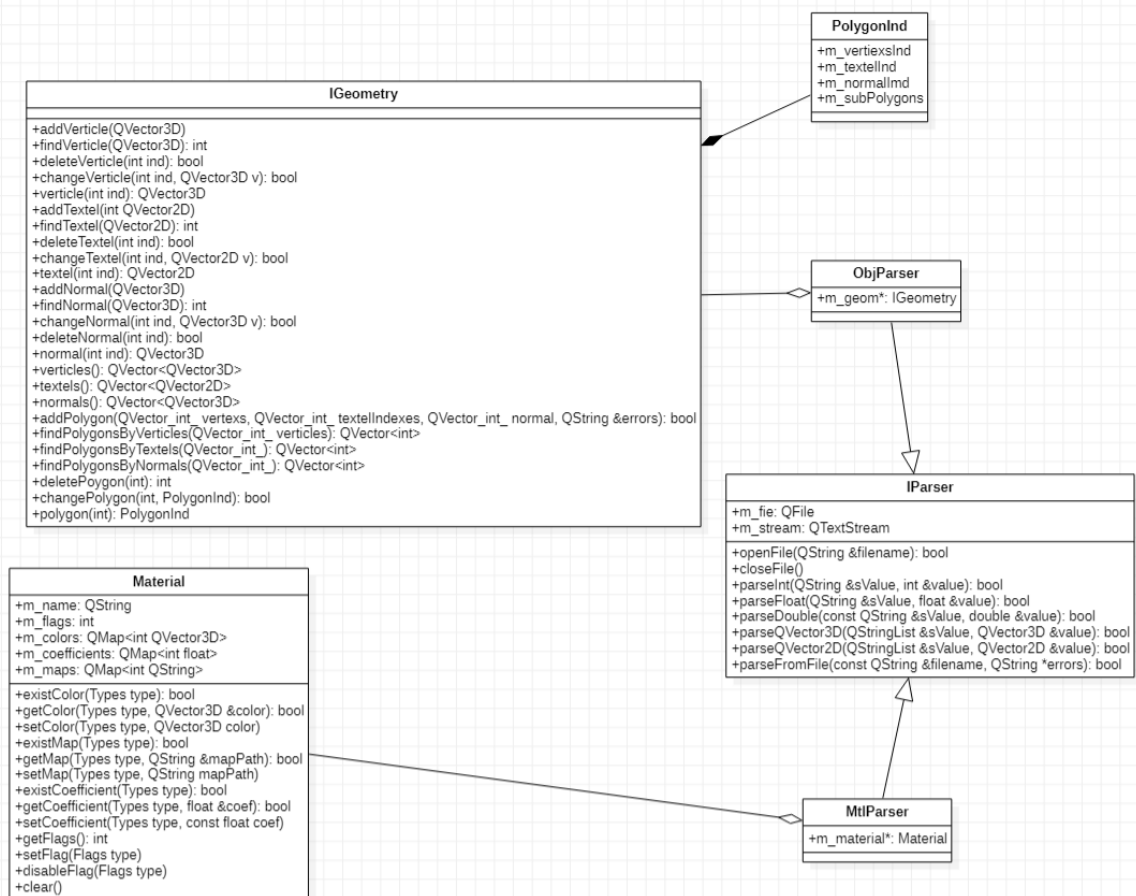
- 11) Паттерн проектирования компоновщик [Электронный ресурс]. – Режим доступа: <https://refactoringguru.cn/design-patterns/composite> - (дата обращения 21.01.2022);
- 12) Паттерн проектирования фасад [Электронный ресурс]. – Режим доступа: <https://refactoringguru.ru/design-patterns/facade> - (дата обращения 21.01.2022);
- 13) Паттерн проектирования итератор [Электронный ресурс]. – Режим доступа: <https://refactoringguru.ru/design-patterns/iterator> - (дата обращения 21.01.2022);
- 14) Паттерн проектирования шаблонный метод [Электронный ресурс]. – Режим доступа: <https://refactoringguru.ru/design-patterns/template-method> - (дата обращения 21.01.2022);
- 15) Паттерн проектирования легковес [Электронный ресурс]. – Режим доступа: <https://refactoringguru.ru/design-patterns/flyweight> - (дата обращения 21.01.2022);
- 16) Индексный буфер в OpenGL [Электронный ресурс]. – Режим доступа: [http://vbomesh.blogspot.com/2012/02/vbo-opengl\\_117.html](http://vbomesh.blogspot.com/2012/02/vbo-opengl_117.html) - (дата обращения 5.04.2022);
- 17) Структура данных Октодеревы [Электронный ресурс]. – Режим доступа: <https://russianblogs.com/article/4896740001/> - (дата обращения 4.02.2022);
- 18) Октодерево [Электронный ресурс]. – Режим доступа: <https://web.archive.org/web/20140605161956/http://www.microsoft.com/msj/archive/S3F1.aspx>: - (дата обращения 14.05.2022).

## Приложения

### Приложение 1. Диаграмма классов Октодерева.

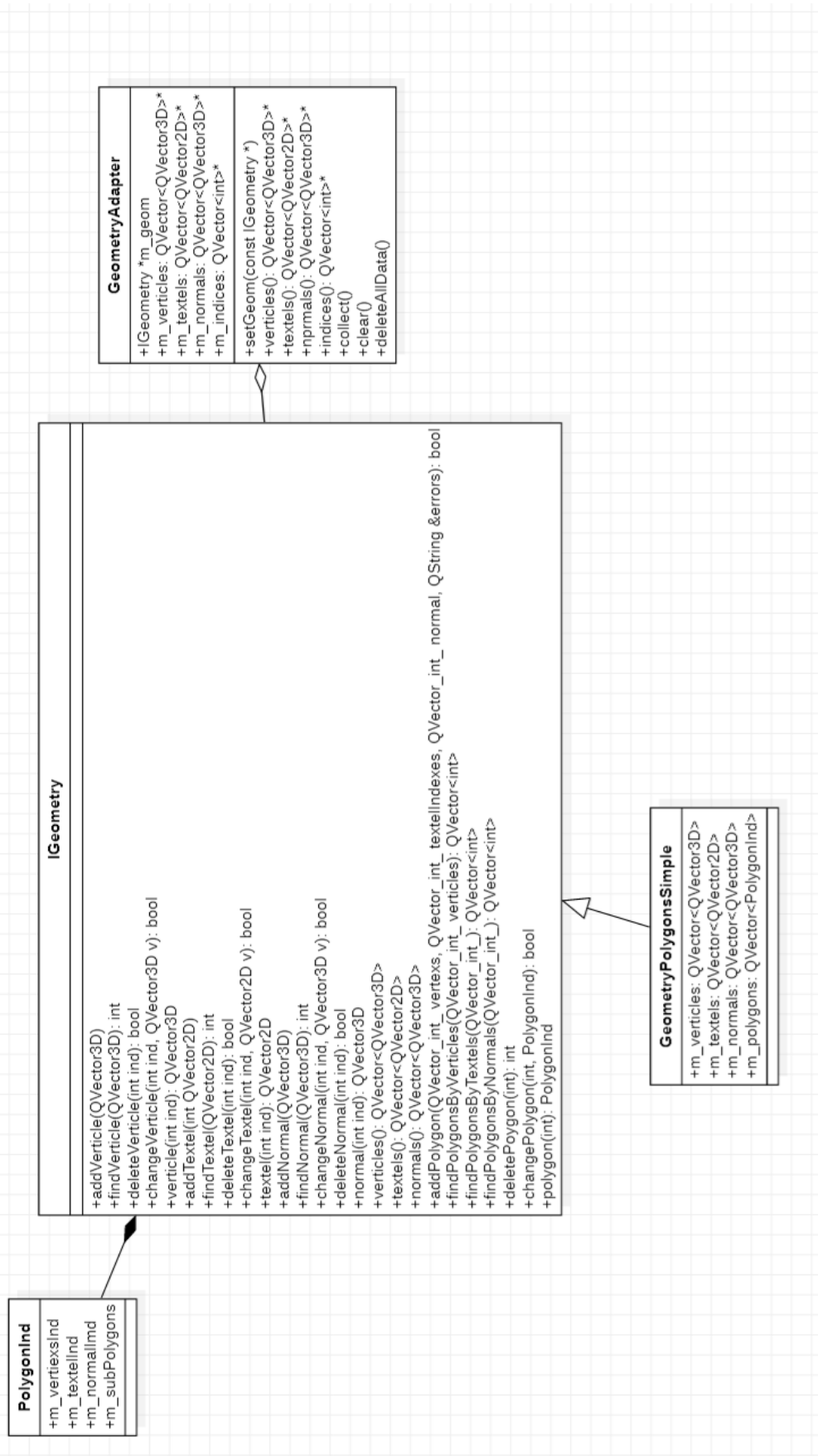


### Приложение 2. Диаграмма классов, отвечающих за импорт.

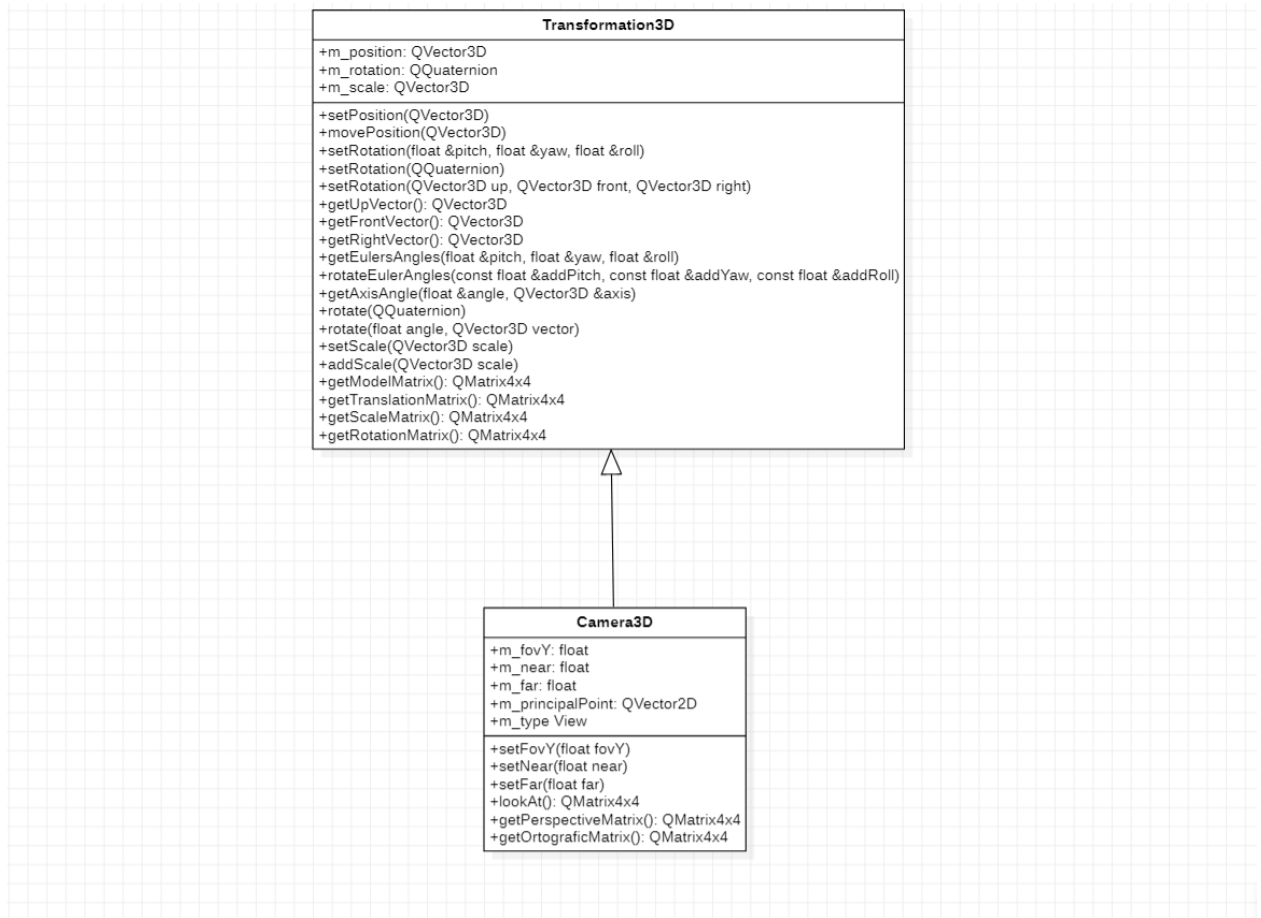




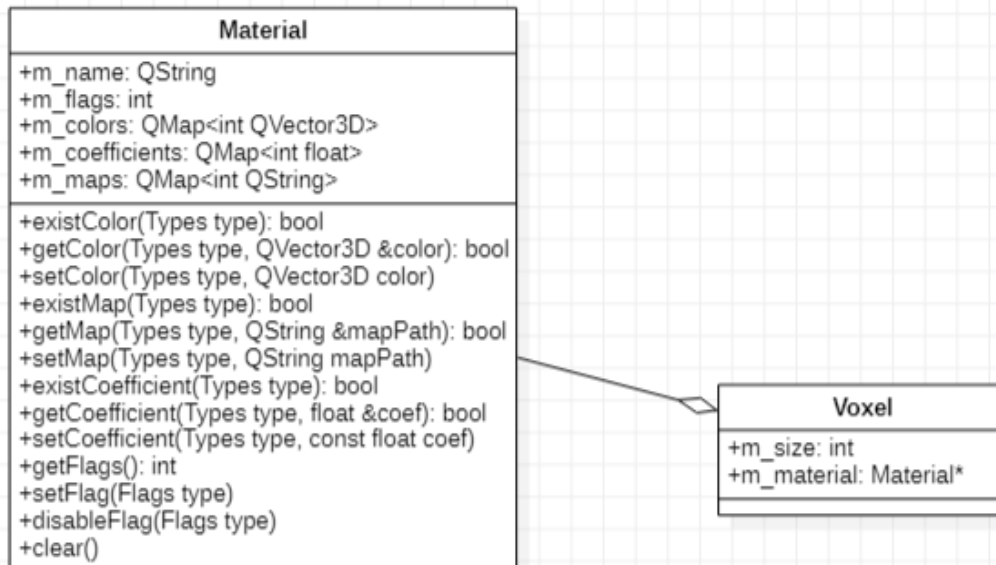
### Приложение 3. Диаграмма классов структуры геометрии.



#### Приложение 4. Диаграмма классов Transformation3D и Camera3D.



## Приложение 5. Диаграмма классов Material и Voxel.



## Приложение 6. Листинг класса Camera3D.

```
class Camera3D : public Transformation3D
{
public:
    enum View{
        FPS = 1,
        TPS = 2
    };
    Camera3D();
    float fovY() const;
    void setFovY(float fovY);
    QVector3D getPosition() const;
    void movePosition(const QVector3D &move);
    QMatrix4x4 lookAt() const;
    QMatrix4x4 getPerspectiveMatrix() const;
    QMatrix4x4 getOrtograficMatrix() const;
    float getNear() const;
    void setNear(float near);
    float getFar() const;
    void setFar(float far);
    View getType() const;
    void setType(const View &type);
protected:
    View m_type = TPS;
    float m_fovY = 30;
    float m_near = 0.1f;
    float m_far = 100.0f;
};
Camera3D::Camera3D()
{}
float Camera3D::fovY() const
{
    return m_fovY;
}
void Camera3D::setFovY(float fovY)
{
    m_fovY = fovY;
}
QVector3D Camera3D::getPosition() const
{
    if(m_type == TPS) {
        QMatrix4x4 modelviewMatrix = (getScaleMatrix() * getRotationMatrix().transposed() *
getTranslationMatrix());
        QVector3D result(modelviewMatrix(0, 3), modelviewMatrix(1, 3), modelviewMatrix(2,
3));
        return -result;
    } else
        return m_position;
}
```

```

void Camera3D::movePosition(const QVector3D &move)
{
    if (m_type == TPS) {
        QVector3D right = getRightVector() * move[0];
        QVector3D up = getUpVector() * move[1];
        QVector3D front = getFrontVector() * move[2];
        m_position += right + up + front;
    } else
        m_position += move;
}

QMatrix4x4 Camera3D::lookAt() const
{
    if (m_type == TPS)
        return getScaleMatrix() * getTranslationMatrix() * getRotationMatrix();
    if (m_type == FPS)
        return getScaleMatrix() * getRotationMatrix() * getTranslationMatrix();

    return getModelMatrix();
}

QMatrix4x4 Camera3D::getPerspectiveMatrix() const
{
    QMatrix4x4 projectionMx;
    projectionMx.perspective(m_fovY, (float)m_width / m_height, m_near, m_far);
    return projectionMx;
}

QMatrix4x4 Camera3D::getOrtograficMatrix() const
{
    QMatrix4x4 projectionMx;
    const float aspectRatio = (float)m_width / m_height;
    if (aspectRatio > 1)
        projectionMx.ortho(-m_near, m_near, -m_near / aspectRatio, m_near / aspectRatio, m_near,
m_far);
    else
        projectionMx.ortho(-m_near * aspectRatio, m_near * aspectRatio, -m_near, m_near,
m_near, m_far);
    return projectionMx;
}

float Camera3D::getNear() const
{
    return m_near;
}

void Camera3D::setNear(float near)
{
    m_near = near;
}

float Camera3D::getFar() const
{
    return m_far;
}

void Camera3D::setFar(float far)
{
    m_far = far;
}

```

```
}  
Camera3D::View Camera3D::getType() const  
{  
    return m_type;  
}  
void Camera3D::setType(const View &type)  
{  
    m_type = type;  
}
```

## Приложение 7. Листинг классов геометрии.

```
struct PolygonInd
{
public:
    QVector<int> m_verticleIndices;
    QVector<int> m_textelIndices;
    QVector<int> m_normalIndices;

    QVector<QVector<int>> m_subPolygons;
};

class IGeometry
{
public:
    IGeometry() {};
    virtual void clear() = 0;
    virtual void addVerticle(const QVector3D v) = 0;
    virtual int findVerticle(const QVector3D v) const = 0;
    virtual QVector3D verticle(const int ind) const = 0;
    virtual bool changeVerticle(const int ind, const QVector3D v);
    virtual bool deleteVerticle(const int ind);
    virtual void addTextel(const QVector2D v) = 0;
    virtual int findTextel(const QVector2D v) const = 0;
    virtual QVector2D textel(const int ind) const = 0;
    virtual bool changeTextel(const int ind, const QVector2D v);
    virtual bool deleteTextel(const int ind);
    virtual void addNormal(const QVector3D v) = 0;
    virtual int findNormal(const QVector3D v) const = 0;
    virtual QVector3D normal(const int ind) const = 0;
    virtual bool changeNormal(const int ind, const QVector3D v);
    virtual bool deleteNormal(const int ind);
    virtual bool addPolygon(const QVector<int> &vertexIndexes,
                           const QVector<int> &textelIndexes,
                           const QVector<int> &normalIndexes,
                           QString &errors) = 0;
    virtual bool deletePolygon(int);
    virtual bool changePolygon(int ind, PolygonInd polInd);
    virtual PolygonInd polygon(int ind);
    virtual QVector<int> findPolygonsByVertices(const QVector<int> &verticleIndices) const =
0;
    virtual QVector<int> findPolygonsByTextels(const QVector<int> &textelIndices) const = 0;
    virtual QVector<int> findPolygonsByNormals(const QVector<int> &normalIndices) const =
0;
    virtual QVector<QVector3D> verticles() const = 0;
    virtual int sizeVerticles() const = 0;
    virtual QVector<QVector2D> textels() const = 0;
    virtual int sizeTextels() const = 0;
    virtual QVector<QVector3D> normals() const = 0;
    virtual int sizeNormals() const = 0;
    virtual QVector<PolygonInd> polygons() const = 0;
```

```

virtual int sizePolygons() const = 0;
int getType() const
{
    return type;
};
void setType(int value)
{
    if((value & Vertex) == 0)
        return;
    type = value;
};
class GeometryPolygonsSimple : public IGeometry
{
public:
    GeometryPolygonsSimple();
    ~GeometryPolygonsSimple();
    void clear() override;
    void addVerticle(const QVector3D v) override;
    int findVerticle(const QVector3D v) const override;
    QVector3D verticle(const int ind) const override;
    bool changeVerticle(const int ind, const QVector3D v) override;
    bool deleteVerticle(const int ind) override;
    void addTextel(const QVector2D v) override;
    int findTextel(const QVector2D v) const override;
    QVector2D textel(const int ind) const override;
    bool changeTextel(const int ind, const QVector2D v) override;
    bool deleteTextel(const int ind) override;
    void addNormal(const QVector3D v) override;
    int findNormal(const QVector3D v) const override;
    QVector3D normal(const int ind) const override;
    bool changeNormal(const int ind, const QVector3D v) override;
    bool deleteNormal(const int ind) override;
    bool addPolygon(const QVector<int> &vertexIndexes,
                   const QVector<int> &textelIndexes,
                   const QVector<int> &normalIndexes,
                   QString &errors) override;
    bool deletePolygon(int);
    bool changePolygon(int ind, PolygonInd polInd);
    PolygonInd polygon(int ind);
    QVector<int> findPolygonsByVerticles(const QVector<int> &verticleIndices) const override;
    QVector<int> findPolygonsByTextels(const QVector<int> &textelIndices) const override;
    QVector<int> findPolygonsByNormals(const QVector<int> &normalIndices) const override;
    QVector<QVector3D> verticles() const override;
    int sizeVerticles() const override;
    QVector<QVector2D> textels() const override;
    int sizeTextels() const override;
    QVector<QVector3D> normals() const override;
    int sizeNormals() const override;
    QVector<PolygonInd> polygons() const override;
    int sizePolygons() const override;
    QVector<QVector<int> > getPolygonsVertexIndices() const;
    QVector<QVector<int> > getPolygonsTextelIndices() const;

```



```

    QVector<QVector<int> > getPolygonsNormalIndices() const;
    void operator=(const GeometryPolygonsSimple &geom);
protected:
    QVector<QVector3D> m_vertices;
    QVector<QVector2D> m_textels;
    QVector<QVector3D> m_normals;
    QVector<PolygonInd> m_polygons;
};
GeometryPolygonsSimple::GeometryPolygonsSimple()
{}
bool firstExistInSecondArray(const QVector<int> &first, const QVector<int> &second)
{
    for(int f_i = 0; f_i < first.size(); f_i++) {
        bool exist = false;
        for(int s_i = 0; s_i < second.size() && !exist; s_i++) {
            if(first[f_i] == second[s_i])
                exist = true;
        }
        if(!exist)
            return false;
    }
    return true;
}
GeometryPolygonsSimple::~GeometryPolygonsSimple()
{
    clear();
}
void GeometryPolygonsSimple::clear()
{
    m_vertices.clear();
    m_textels.clear();
    m_normals.clear();
    m_polygons.clear();
}
void GeometryPolygonsSimple::addVerticle(const QVector3D v)
{
    m_vertices.push_back(v);
}
int GeometryPolygonsSimple::findVerticle(const QVector3D v) const
{
    return m_vertices.indexOf(v);
}
QVector3D GeometryPolygonsSimple::verticle(const int ind) const
{
    if (ind >=0 && ind < m_vertices.size())
        return m_vertices[ind];
    else
        return QVector3D();
}
void GeometryPolygonsSimple::addTextel(const QVector2D v)
{
    m_textels.push_back(v);
}

```

```

}
int GeometryPolygonsSimple::findTexel(const QVector2D v) const
{
    return m_textels.indexOf(v);
}
QVector2D GeometryPolygonsSimple::texel(const int ind) const
{
    if (ind >=0 && ind < m_textels.size())
        return m_textels[ind];
    else
        return QVector2D();
}
void GeometryPolygonsSimple::addNormal(const QVector3D v)
{
    m_normals.push_back(v);
}
int GeometryPolygonsSimple::findNormal(const QVector3D v) const
{
    return m_normals.indexOf(v);
}
QVector3D GeometryPolygonsSimple::normal(const int ind) const
{
    if (ind >=0 && ind < m_normals.size())
        return m_normals[ind];
    else
        return QVector3D();
}
bool GeometryPolygonsSimple::addPolygon(const QVector<int> &vertexIndexes, const
QVector<int> &texelIndexes, const QVector<int> &normalIndexes, QString &errors)
{
    if ((vertexIndexes.size() != texelIndexes.size() && (type & Texel)) ||
        (vertexIndexes.size() != normalIndexes.size() && (type & Normal))) {
        errors += "incorrect indecs vectors sizes\n";
        return false;
    }
    if (vertexIndexes.size() < 3) {
        errors += "indexes size < 3\n";
        return false;
    }
    for(int i = 0; i < vertexIndexes.size(); i++) {
        if (vertexIndexes[i] >= m_vertices.size()) {
            errors = "Wrong vertex index\n";
            return false;
        }
        if (texelIndexes[i] >= m_textels.size()) {
            errors = "Wrong texel index\n";
            return false;
        }
        if (normalIndexes[i] >= m_normals.size()) {
            errors = "Wrong normal index\n";
            return false;
        }
    }
}

```

```

    }
    auto triangles = Triangulation::triangulationSimple3D(m_vertices, vertexIndexes);
    PolygonInd polygon;
    polygon.m_verticleIndices = vertexIndexes;
    if (type & Textel)
        polygon.m_textelIndices = textelIndexes;
    if (type & Normal)
        polygon.m_normalIndices = normalIndexes;
    for(int i = 0; i < triangles.size(); i++)
        polygon.m_subPolygons.push_back({ triangles[i].a,
                                           triangles[i].b,
                                           triangles[i].c });
    m_polygons.push_back(polygon);
    return true;
}

QVector<int> GeometryPolygonsSimple::findPolygonsByVertices(const QVector<int>
&verticleIndices) const
{
    QVector<int> polygonInds;
    for(int ind = 0; ind < m_polygons.size(); ind++) {
        if (firstExistInSecondArray(verticleIndices, m_polygons[ind].m_verticleIndices)) {
            polygonInds.push_back(ind);
        }
    }
    return polygonInds;
}

QVector<int> GeometryPolygonsSimple::findPolygonsByTextels(const QVector<int>
&textelIndices) const
{
    QVector<int> polygonInds;
    for(int ind = 0; ind < m_polygons.size(); ind++) {
        if (firstExistInSecondArray(textelIndices, m_polygons[ind].m_textelIndices)) {
            polygonInds.push_back(ind);
        }
    }
    return polygonInds;
}

QVector<int> GeometryPolygonsSimple::findPolygonsByNormals(const QVector<int>
&normalIndices) const
{
    QVector<int> polygonInds;
    for(int ind = 0; ind < m_polygons.size(); ind++) {
        if (firstExistInSecondArray(normalIndices, m_polygons[ind].m_normalIndices)) {
            polygonInds.push_back(ind);
        }
    }
    return polygonInds;
}

QVector<int> GeometryPolygonsSimple::getPolygonVerticleIndices(const int ind) const
{
    if (ind >=0 && ind < m_polygons.size())
        return m_polygons[ind].m_verticleIndices;
}

```

```

        else
            return QVector<int>();
    }
    QVector<int> GeometryPolygonsSimple::getPolygonTexelIndices(const int ind) const
    {
        if (ind >=0 && ind < m_polygons.size())
            return m_polygons[ind].m_texelIndices;
        else
            return QVector<int>();
    }
    QVector<int> GeometryPolygonsSimple::getPolygonNormalIndices(const int ind) const
    {
        if (ind >=0 && ind < m_polygons.size())
            return m_polygons[ind].m_normalIndices;
        else
            return QVector<int>();
    }
    QVector<QVector3D> GeometryPolygonsSimple::vertices() const
    {
        return m_vertices;
    }
    int GeometryPolygonsSimple::sizeVertices() const
    {
        return m_vertices.size();
    }
    QVector<QVector2D> GeometryPolygonsSimple::textels() const
    {
        return m_textels;
    }
    int GeometryPolygonsSimple::sizeTextels() const
    {
        return m_textels.size();
    }
    QVector<QVector3D> GeometryPolygonsSimple::normals() const
    {
        return m_normals;
    }
    int GeometryPolygonsSimple::sizeNormals() const
    {
        return m_normals.size();
    }
    QVector<PolygonInd> GeometryPolygonsSimple::polygons() const
    {
        return m_polygons;
    }
    int GeometryPolygonsSimple::sizePolygons() const
    {
        return m_polygons.size();
    }

    QVector<QVector<int> > GeometryPolygonsSimple::getPolygonsVertexIndices() const
    {
        QVector<QVector<int> > result;

```

```

        for(int i = 0; i < m_polygons.size(); i++)
            result.push_back(m_polygons[i].m_verticleIndices);
        return result;
    }
    QVector<QVector<int> > GeometryPolygonsSimple::getPolygonsTextelIndices() const
    {
        QVector<QVector<int> > result;
        for(int i = 0; i < m_polygons.size(); i++)
            result.push_back(m_polygons[i].m_textelIndices);

        return result;
    }
    QVector<QVector<int> > GeometryPolygonsSimple::getPolygonsNormalIndices() const
    {
        QVector<QVector<int> > result;
        for(int i = 0; i < m_polygons.size(); i++)
            result.push_back(m_polygons[i].m_normalIndices);
        return result;
    }
    void GeometryPolygonsSimple::operator=(const GeometryPolygonsSimple &geom)
    {
        clear();
        QVector<QVector3D> cvertices = geom.vertices();
        QVector<QVector2D> ctextels = geom.textels();
        QVector<QVector3D> cnormals = geom.normals();
        QVector<PolygonInd> cPolygons = geom.polygons();
        m_vertices.resize(cvertices.size());
        m_textels.resize(ctextels.size());
        m_normals.resize(cnormals.size());
        m_polygons.resize(cPolygons.size());
        for(int i = 0; i < cvertices.size(); i++)
            m_vertices[i] = cvertices[i];
        for(int i = 0; i < ctextels.size(); i++)
            m_textels[i] = ctextels[i];
        for(int i = 0; i < cnormals.size(); i++)
            m_normals[i] = cnormals[i];
        for(int i = 0; i < cPolygons.size(); i++)
            m_polygons[i] = cPolygons[i];
    }
}

```

## Приложение 8. Листинг класса GeomTracer.

```

class GeomTracer
{
public:
    GeomTracer(const GeometryPolygonsSimple *geom);
    void setGeom(const GeometryPolygonsSimple *geom);
    bool nearestToRaySurfaceData(
        const QVector3D rayOrigin,
        const QVector3D rayDirection,
        QVector3D &nearestPoint,
        QVector3D &normal,
        double &distance
    );
protected:
    const IGeometry *m_geom = nullptr;
}
GeomTracer::GeomTracer(const GeometryPolygonsSimple *geom)
{
    m_geom = geom;
}
void GeomTracer::setGeom(const GeometryPolygonsSimple *geom)
{
    m_geom = geom;
}
bool GeomTracer::nearestToRaySurfaceData(const QVector3D rayOrigin, const QVector3D
rayDirection, QVector3D &nearestPoint, QVector3D &normal, double &distance)
{
    distance = 1e15;
    nearestPoint = QVector3D(0, 0, 0);
    bool foundAtLeastOneIntersection = false;
    QVector<PolygonInd> polygons = m_geom->polygons();
    for(int i = 0; i < polygons.size(); i++) {
        bool hasIntersection = false;
        for(int j = 0; j < polygons[i].m_subPolygons.size() && !hasIntersection; j++) {
            QVector3D p1 = m_geom-
>vertice(polygons[i].m_verticeIndices[polygons[i].m_subPolygons[j][0]]);
            QVector3D p2 = m_geom-
>vertice(polygons[i].m_verticeIndices[polygons[i].m_subPolygons[j][1]]);
            QVector3D p3 = m_geom-
>vertice(polygons[i].m_verticeIndices[polygons[i].m_subPolygons[j][2]]);
            QVector3D surfaceNormal = QVector3D::crossProduct((p2 - p1).normalized(), (p3 -
p1).normalized()).normalized();
            if (std::abs(QVector3D::dotProduct(rayDirection, surfaceNormal)) < 1e-5)
                continue;
            QVector3D intersectionPoint = intersectRaySurface(rayOrigin, rayDirection,
surfaceNormal, p1);
            QVector3D baricentrics = calculateBaricentrics2D(p1, p2, p3, intersectionPoint);
            bool firstInside = std::abs(baricentrics[0] - std::clamp(baricentrics[0], 0.0f, 1.0f)) < 1e-4;

```

```

    bool secondInside = std::abs(baricentrics[1] - std::clamp(baricentrics[1], 0.0f, 1.0f)) < 1e-
4;
    bool thirdInside = std::abs(baricentrics[2] - std::clamp(baricentrics[2], 0.0f, 1.0f)) < 1e-4;
    if(firstInside && secondInside && thirdInside) {
        float currentDistance = (rayOrigin - intersectionPoint).length();
        if (distance > currentDistance) {
            distance = currentDistance;
            nearestPoint = intersectionPoint;
            normal = surfaceNormal;
            hasIntersection = true;
        }
    }
    if (hasIntersection)
        foundAtLeastOneIntersection = true;
}
return foundAtLeastOneIntersection;
}
QVector3D GeomTracer::calculateBaricentrics2D(const QVector3D p1, const QVector3D p2,
const QVector3D p3, const QVector3D pb) const
{
    QMatrix4x4 baseMatrix = calculateBasisMatrix(p1, p2, p3);
    QVector4D bp1 = baseMatrix * p1;
    QVector4D bp2 = baseMatrix * p2;
    QVector4D bp3 = baseMatrix * p3;
    QVector4D iMx1 = bp3 - bp1;
    QVector4D iMx2 = bp2 - bp1;
    QMatrix4x4 invMatrix;
    invMatrix(0, 0) = iMx1[0];
    invMatrix(0, 1) = iMx1[1];
    invMatrix(1, 0) = iMx2[0];
    invMatrix(1, 1) = iMx2[1];
    invMatrix = invMatrix.inverted();
    QVector4D baric = invMatrix * (pb - bp1);
    return QVector3D(baric[0], baric[1], 1 - baric[0] - baric[1]);
}
QMatrix4x4 GeomTracer::calculateBasisMatrix(
    const QVector3D p1,
    const QVector3D p2,
    const QVector3D p3) const
{
    const QVector3D base1 = (p3 - p1).normalized();
    const QVector3D base2 = QVector3D::crossProduct(base1, (p2 - p1).normalized());
    const QVector3D base3 = QVector3D::crossProduct(base1, base2).normalized();
    QMatrix4x4 baseMx, trMx;
    for(int i = 0; i < 3; i++) {
        baseMx(0, i) = base1[i];
        baseMx(1, i) = base2[i];
        baseMx(2, i) = base3[i];
        trMx(i, 3) = -p1[i];
    }
    return baseMx * trMx;
}

```

```

}

QVector3D GeomTracer::intersectRaySurface(QVector3D rayOrigin, QVector3D rayDirection,
QVector3D surfaceNormal, QVector3D surfacePoint) const
{
    rayDirection.normalize();
    surfaceNormal.normalize();
    if(std::abs(QVector3D::dotProduct(rayDirection, surfaceNormal)) < 1e-5) {
        Q_ASSERT("dot is null" == "error");
        return QVector3D();
    }
    double t = 0, del = 0;
    for(int i = 0; i < 3; i++) {
        t += surfaceNormal[i] * (rayOrigin[i] - surfacePoint[i]);
        del += surfaceNormal[i] * rayDirection[i];
    }
    t /= del;
    return rayOrigin + rayDirection * t;
}

```



## Приложение 9. Листинг класса Iterator.

```
enum IteratorStatus {  
    NotEnd = 128,  
    End = 0  
};  
template <typename T>  
class Iterator  
{  
public:  
    virtual bool hasNext() const = 0;  
    virtual bool next() = 0;  
};
```

## Приложение 10. Листинг класса Material.

```
namespace Material {
enum class Flags {
    AmbientColor = 1,
    AmbientMap = 2,
    AmbientMapAsDiffuse = 4,
    DiffuseColor = 8,
    DiffuseMap = 16,
    SpecularColor = 32,
    SpecularMap = 64,
    SpecularMapAsDiffuse = 128,
    SpecularAlphaMap = 256,
    EmissionColor = 512,
    EmissionMap = 1024,
    NormalMap = 2048,
    Reflection = 4096,
    Refraction = 4096 * 2,
    Transparency = 4096 * 4,
    Intensity = 4096 * 8,
    Shininess = 4096 * 16,
    AlphaMap = 4096 * 32
};
enum class Types {
    Ambient = -1,
    Diffuse = -2,
    Specular = -3,
    Emission = -4,
    Alpha = -5,
    Normal = -6,
    SpecularAlpha = -7,
    Transparency = -8,
    Reflection = -9,
    Shininess = -10,
    Intensity = -11
};
class Material
{
public:
    QString name;
    bool existColor(const Types type) const;
    bool getColor(const Types type, QVector3D &color) const;
    void setColor(const Types type, const QVector3D color);
    bool existMap(const Types type) const;
    bool getMap(const Types type, QString &mapPath) const;
    void setMap(const Types type, const QString mapPath);
    bool existCoefficient(const Types type) const;
    bool getCoefficient(const Types type, float &coef) const;
    void setCoefficient(const Types type, const float coef);
    int getFlags() const;
```

```

    void setFlag(const Flags type);
    void disableFlag(const Flags type);
    void clear();
protected:
    int flags = 0;
    QMap<int, QVector3D> m_colors;
    QMap<int, float> m_coefficients;
    QMap<int, QString> m_maps;
};
bool Material::existColor(const Types type) const
{
    return m_colors.find((int)type) != m_colors.end();
}
bool Material::getColor(const Types type, QVector3D &color) const
{
    if(!existColor(type))
        return false;
    color = m_colors[(int)type];
    return true;
}
void Material::setColor(const Types type, const QVector3D color)
{
    m_colors[(int)type] = color;
}
bool Material::existMap(const Types type) const
{
    return m_maps.find((int)type) != m_maps.end();
}
bool Material::getMap(const Types type, QString &mapPath) const
{
    if(!existColor(type))
        return false;
    mapPath = m_maps[(int)type];
    return true;
}
void Material::setMap(const Types type, const QString mapPath)
{
    m_maps[(int)type] = mapPath;
}
bool Material::existCoefficient(const Types type) const
{
    return m_coefficients.find((int)type) != m_coefficients.end();
}
bool Material::getCoefficient(const Types type, float &coef) const
{
    if(!existColor(type))
        return false;
    coef = m_coefficients[(int)type];
    return true;
}
void Material::setCoefficient(const Types type, const float coef)
{

```

```

        m_coefficients[(int)type] = coef;
    }
    int Material::getFlags() const
    {
        return flags;
    }
    void Material::setFlag(const Flags type)
    {
        flags = flags | (int)type;
    }
    void Material::disableFlag(const Flags type)
    {
        if (flags | (int)type) {
            flags -= (int)type;
        }
    }
    void Material::clear()
    {
        flags = 0;
        m_colors.clear();
        m_coefficients.clear();
        m_maps.clear();
    }
}

```

## Приложение 11. Листинг классов Parser, ObjPraser, MtlParser.

```
class Parser
{
public:
    Parser();
    virtual bool parseFromFile(const QString &filename, QString *errors = nullptr);
protected:
    QFile m_file;
    QTextStream m_stream;
    bool openFile(const QString &filename);
    void closeFile();
    bool parseInt(const QString &sValue, int &value);
    bool parseFloat(const QString &sValue, float &value);
    bool parseDouble(const QString &sValue, double &value);
    bool parseQVector3D(const QStringList &sValue, QVector3D &value);
    bool parseQVector2D(const QStringList &sValue, QVector2D &value);
};
Parser::Parser()
{}
bool Parser::openFile(const QString &filename)
{
    m_file.setFileName(filename);
    if (m_file.open(QIODevice::ReadOnly | QIODevice::Text)) {
        m_stream.setDevice(&m_file);
        return true;
    } else {
        return false;
    }
}
void Parser::closeFile()
{
    m_file.close();
}
bool Parser::parseInt(const QString &sValue, int &value)
{
    bool ok = false;
    value = sValue.toInt(&ok);
    return ok;
}
bool Parser::parseFloat(const QString &sValue, float &value)
{
    bool ok = false;
    value = sValue.toFloat(&ok);
    return ok;
}
bool Parser::parseDouble(const QString &sValue, double &value)
{
    bool ok = false;
```

```

        value = sValue.toDouble(&ok);
        return ok;
    }
    bool Parser::parseQVector3D(const QStringList &sValue, QVector3D &value)
    {
        if (sValue.size() != 3)
            return false;
        for(int i = 0; i < 3; i++) {
            if (!parseFloat(sValue[i],value[i]))
                return false;
        }
        return true;
    }
    bool Parser::parseQVector2D(const QStringList &sValue, QVector2D &value)
    {
        if (sValue.size() != 2)
            return false;
        for(int i = 0; i < 2; i++) {
            if (!parseFloat(sValue[i],value[i]))
                return false;
        }
        return true;
    }
}

class MtlParser : public Parser
{
public:
    MtlParser();
    bool parseFromFile(const QString &filename, QString *errors = nullptr);
    Material::Material *getMaterial() const;
    void setMaterial(Material::Material *value);
    void clear();
protected:
    Material::Material *m_material = nullptr;
};

MtlParser::MtlParser()
{}

bool MtlParser::parseFromFile(const QString &filename, QString *errors)
{
    if (m_material == nullptr) {
        if (errors != nullptr)
            *errors += "empty geom pointer";
        return false;
    }
    if (!openFile(filename)) {
        if (errors != nullptr)
            *errors += "file not open \n";
        return false;
    }
    m_material->clear();
    QString line;
    int countLine = 0;

```

```

while(!m_stream.atEnd()) {
    countLine++;
    line = m_stream.readLine();
    QStringList tokens = line.split(' ');
    //parse tokens
    if (tokens[0] == "#")
        continue;
    if (tokens[0] == "newmtl") {
        if (tokens.size() != 2) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
            return false;
        }
        m_material->name = tokens[1];
    }
    if (tokens[0] == "Ka") {
        QVector3D vec;
        if (tokens.size() != 4) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
            return false;
        }
        if (!parseQVector3D({tokens[1],tokens[2], tokens[3]},vec)) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, cant parse vector \n";
            return false;
        }
        m_material->setColor(Material::Types::Ambient, vec);
        m_material->setFlag(Material::Flags::AmbientColor);
    }
    if (tokens[0] == "Kd") {
        QVector3D vec;
        if (tokens.size() != 4) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
            return false;
        }
        if (!parseQVector3D({tokens[1],tokens[2], tokens[3]},vec)) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, cant parse vector \n";
            return false;
        }
        m_material->setColor(Material::Types::Diffuse, vec);
        m_material->setFlag(Material::Flags::DiffuseColor);
    }
    if (tokens[0] == "Ks") {
        QVector3D vec;
        if (tokens.size() != 4) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
            return false;
        }
    }
}

```

```

        if (!parseQVector3D({tokens[1],tokens[2], tokens[3]},vec)) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, cant parse vector \n";
            return false;
        }
        m_material->setColor(Material::Types::Specular, vec);
        m_material->setFlag(Material::Flags::SpecularColor);
    }
    if (tokens[0] == "Ke") {
        QVector3D vec;
        if (tokens.size() != 4) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
            return false;
        }
        if (!parseQVector3D({tokens[1],tokens[2], tokens[3]},vec)) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, cant parse vector \n";
            return false;
        }
        m_material->setColor(Material::Types::Emission, vec);
        m_material->setFlag(Material::Flags::EmissionColor);
    }
    if (tokens[0] == "map_Ka") {
        if (tokens.size() != 2) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
            return false;
        }
        m_material->setMap(Material::Types::Ambient, tokens[1]);
        m_material->setFlag(Material::Flags::AmbientMap);
    }
    if (tokens[0] == "map_Kd") {
        if (tokens.size() != 2) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
            return false;
        }
        m_material->setMap(Material::Types::Diffuse, tokens[1]);
        m_material->setFlag(Material::Flags::DiffuseMap);
    }
    if (tokens[0] == "map_Ks") {
        if (tokens.size() != 2) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
            return false;
        }
        m_material->setMap(Material::Types::Specular, tokens[1]);
        m_material->setFlag(Material::Flags::SpecularMap);
    }
    if (tokens[0] == "map_Ke") {
        if (tokens.size() != 2) {

```



```

        if (errors != nullptr)
            *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
        return false;
    }
    m_material->setMap(Material::Types::Emission, tokens[1]);
    m_material->setFlag(Material::Flags::EmissionMap);
}
if (tokens[0] == "map_Ns") {
    if (tokens.size() != 2) {
        if (errors != nullptr)
            *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
        return false;
    }
    m_material->setMap(Material::Types::SpecularAlpha, tokens[1]);
    m_material->setFlag(Material::Flags::SpecularAlphaMap);
}
if (tokens[0] == "map_d") {
    if (tokens.size() != 2) {
        if (errors != nullptr)
            *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
        return false;
    }
    m_material->setMap(Material::Types::Alpha, tokens[1]);
    m_material->setFlag(Material::Flags::AlphaMap);
}
if (tokens[0] == "map_Bump") {
    if (tokens.size() != 2) {
        if (errors != nullptr)
            *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
        return false;
    }
    m_material->setMap(Material::Types::Normal, tokens[1]);
    m_material->setFlag(Material::Flags::NormalMap);
}
if (tokens[0] == "d") {
    float value;
    if (tokens.size() != 2) {
        if (errors != nullptr)
            *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
        return false;
    }
    if (!parseFloat(tokens[1], value)) {
        if (errors != nullptr)
            *errors += "error in " + QString::number(countLine) + " line, cant parse vector \n";
        return false;
    }
    m_material->setCoefficient(Material::Types::Transparency, value);
    m_material->setFlag(Material::Flags::Transparency);
}
if (tokens[0] == "Ni") {
    float value;
    if (tokens.size() != 2) {

```

```

        if (errors != nullptr)
            *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
        return false;
    }
    if (!parseFloat(tokens[1], value)) {
        if (errors != nullptr)
            *errors += "error in " + QString::number(countLine) + " line, cant parse vector \n";
        return false;
    }
    m_material->setCoefficient(Material::Types::Reflection, value);
    m_material->setFlag(Material::Flags::Reflection);
}
if (tokens[0] == "Ns") {
    float value;
    if (tokens.size() != 2) {
        if (errors != nullptr)
            *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
        return false;
    }
    if (!parseFloat(tokens[1], value)) {
        if (errors != nullptr)
            *errors += "error in " + QString::number(countLine) + " line, cant parse vector \n";
        return false;
    }
    m_material->setCoefficient(Material::Types::Shininess, value);
    m_material->setFlag(Material::Flags::Shininess);
}
}
return true;
}
Material::Material *MtlParser::getMaterial() const
{
    return m_material;
}
void MtlParser::setMaterial(Material::Material *value)
{
    m_material = value;
}
void MtlParser::clear()
{
    if (m_material != nullptr)
        m_material->clear();
}

class ObjParser : public Parser
{
public:
    bool parseFromFile(const QString &filename, QString *errors = nullptr) override;

    IGeometry *getGeom() const;
    void clear();
    void setGeom(IGeometry *value);

```

```

protected:
    IGeometry *geom = nullptr;

};

bool ObjParser::parseFromFile(const QString &filename, QString *errors)
{
    if (geom == nullptr) {
        if (errors != nullptr)
            *errors += "empty geom pointer";
        return false;
    }
    if (!openFile(filename)) {
        if (errors != nullptr)
            *errors += "file not open \n";
        return false;
    }
    geom->clear();
    QString line;
    int countLine = 0;
    while(!m_stream.atEnd()) {
        countLine++;
        line = m_stream.readLine();
        QStringList tokens = line.split(' ');
        //parse tokens
        if (tokens[0] == "#")
            continue;
        if (tokens[0] == "v") {
            QVector3D vec;
            if (tokens.size() != 4) {
                if (errors != nullptr)
                    *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
                return false;
            }
            if (!parseQVector3D({tokens[1],tokens[2], tokens[3]},vec)) {
                if (errors != nullptr)
                    *errors += "error in " + QString::number(countLine) + " line, cant parse vector \n";
                return false;
            }
            geom->addVerticle(vec);
        }
        if (tokens[0] == "vt") {
            QVector2D vec;
            if (tokens.size() != 3) {
                if (errors != nullptr)
                    *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
                return false;
            }
            if (!parseQVector2D({tokens[1],tokens[2]},vec)) {
                if (errors != nullptr)
                    *errors += "error in " + QString::number(countLine) + " line, cant parse vector \n";
                return false;
            }
        }
    }
}

```

```

    }
    geom->addTextel(vec);
}
if (tokens[0] == "vn") {
    QVector3D vec;
    if (tokens.size() != 4) {
        if (errors != nullptr)
            *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
        return false;
    }
    if (!parseQVector3D({tokens[1],tokens[2], tokens[3]},vec)) {
        if (errors != nullptr)
            *errors += "error in " + QString::number(countLine) + " line, cant parse vector \n";
        return false;
    }
    geom->addNormal(vec);
}
if (tokens[0] == "f") {
    QVector<int> vertexes, textels, normals;
    if (tokens.size() <= 4) {
        if (errors != nullptr)
            *errors += "error in " + QString::number(countLine) + " line, not enough values \n";
        return false;
    }
    for (int i = 1; i < tokens.size(); i++) {
        QStringList polygonTokens = tokens[i].split('/');
        if (polygonTokens.size() != 3) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, not enough values
\n";
            return false;
        }
        int ind;
        if (!parseInt(polygonTokens[0],ind)) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, cant parse int \n";
            return false;
        }
        vertexes.push_back(ind - 1);
        if (!parseInt(polygonTokens[1],ind)) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, cant parse int \n";
            return false;
        }
        textels.push_back(ind - 1);
        if (!parseInt(polygonTokens[2],ind)) {
            if (errors != nullptr)
                *errors += "error in " + QString::number(countLine) + " line, cant parse int \n";
            return false;
        }
        normals.push_back(ind - 1);
    }
}

```

```

        QString polygonErrors;
        geom->addPolygon(vertexes,textels,normals, polygonErrors);
        if (errors != nullptr)
            *errors += polygonErrors;
    }
}
return true;
}
IGeometry *ObjParser::getGeom() const
{
    return geom;
}
void ObjParser::clear()
{
    geom->clear();
}
void ObjParser::setGeom(IGeometry *value)
{
    geom = value;
}

```

## Приложение 12. Листинг класса OctoTreeIterator.

```
namespace Octo {

template <typename T>
class OctoTreeIterator : public Iterator<T>
{
public:
    OctoTreeIterator(const OctoTree<T> *tree);
    bool hasNext()const override;
    bool next() override;
    bool operator==(const OctoTreeIterator *another) const;
    bool operator!=(const OctoTreeIterator *another) const;
    bool operator++();
    T operator*() const;
    OctoKey currentKey() const;
protected:
    const OctoTree<T> *m_tree;
    OctoKey m_currentKey;
    int m_currentStatus = NotEnd;
};

template<typename T>
OctoTreeIterator<T>::OctoTreeIterator(const OctoTree<T> *tree) : m_tree(tree)
{
    if (!tree->exist(m_currentKey)) {
        if (hasNext())
            next();
        else
            m_currentStatus = End;
    }
}

template<typename T>
bool OctoTreeIterator<T>::hasNext() const
{
    return m_currentStatus != End;
}

template<typename T>
bool OctoTreeIterator<T>::next()
{
    if (m_currentStatus == End)
        return false;
    OctoKey newKey(m_currentKey);
    if (newKey[0] == Main) {
        newKey.down();
    } else {
        bool canRight;
        bool canUp;
        do {
            canRight = newKey.right();
            canUp = true;
        } while (canRight && canUp);
    }
}
```

```

        if (!canRight)
            canUp = newKey.up();
    } while (!canRight && canUp);

    if (!canRight && !canUp) {
        m_currentStatus = End;
        return true;
    }
}
do {
    const bool existNode = m_tree->exist(newKey, Node);
    const bool existLeaf = m_tree->exist(newKey, Leaf);
    if (existLeaf) {
        m_currentKey = newKey;
        return true;
    }
    if (existNode)
        newKey.down();
    else {
        bool canRight;
        bool canUp;
        do {
            canRight = newKey.right();
            canUp = true;
            if (!canRight)
                canUp = newKey.up();
        } while (!canRight && canUp);

        if (!canRight && !canUp) {
            m_currentStatus = End;
            return true;
        }
    }
} while (true);
return true;
}
template<typename T>
bool OctoTreeIterator<T>::operator==(const OctoTreeIterator *another) const
{
    return m_tree == another->m_tree && m_currentKey == another->m_currentKey &&
        another->m_currentStatus == m_currentStatus;
}
template<typename T>
bool OctoTreeIterator<T>::operator!=(const OctoTreeIterator *another) const
{
    return m_tree != another->m_tree || m_currentKey != another->m_currentKey || another-
        >m_currentStatus != m_currentStatus;
}
template<typename T>
bool OctoTreeIterator<T>::operator++()
{
    if (hasNext())

```

```

        return next();
    else
        return false;
}
template<typename T>
T OctoTreeIterator<T>::operator*() const
{
    Q_ASSERT(m_currentStatus != End);
    T value;
    m_tree->get(m_currentKey, value);
    return value;
}
template<typename T>
OctoKey OctoTreeIterator<T>::currentKey() const
{
    return m_currentKey;
}
}

```



### Приложение 13. Листинг класса OctoKey.

```

namespace Octo {
enum Id {
    pX = 4,
    nX = 0,
    pY = 2,
    nY = 0,
    pZ = 1,
    nZ = 0,
    pXpYpZ = pX | pY | pZ,
    pXpYnZ = pX | pY | nZ,
    pXnYpZ = pX | nY | pZ,
    pXnYnZ = pX | nY | nZ,
    nXpYpZ = nX | pY | pZ,
    nXpYnZ = nX | pY | nZ,
    nXnYpZ = nX | nY | pZ,
    nXnYnZ = nX | nY | nZ,
    Main = 8
};
class OctoKey : public IKey
{
public:
    OctoKey();
    OctoKey(QVector<unsigned char> vkey);
    bool right() override;
    bool left() override;
    bool down() override;
    bool up() override;
    int depth() const override;
    unsigned char last() const;
    int operator[](int ind) const;
    bool operator==(const OctoKey &another) const;
    bool operator!=(const OctoKey &another) const;
    QVector<unsigned char> key() const;
protected:
    QVector<unsigned char> m_key = {Main};
};
OctoKey::OctoKey()
{}
OctoKey::OctoKey(QVector<unsigned char> vkey)
{
    if (vkey.size() == 0)
        return;
    if (vkey[0] != Main) {
        for(int i = 0; i < vkey.size(); i++)
            if ((int)vkey[i] < 0 || (int)vkey[i] > 7)
                return;
    }
    m_key = vkey;
}

```

```

}
bool OctoKey::right()
{
    if (depth() == 1 && m_key[0] == Main)
        return false;
    if (m_key[depth() - 1] == 7)
        return false;
    m_key[depth() - 1]++;
    return true;
}
bool OctoKey::left()
{
    if (depth() == 1 && m_key[0] == Main)
        return false;
    if (m_key[depth() - 1] == 0)
        return false;
    m_key[depth() - 1]--;
    return true;
}
bool OctoKey::down()
{
    if (depth() == 1 && m_key[0] == Main)
        m_key[0] = 0;
    else
        m_key.push_back(0);
    return true;
}
bool OctoKey::up()
{
    if (depth() == 1 && m_key[0] == Main)
        return false;
    if (depth() == 1 && m_key[0] != Main)
        m_key[0] = Main;
    else
        m_key.pop_back();

    return true;
}
int OctoKey::depth() const
{
    return m_key.size();
}
unsigned char OctoKey::last() const
{
    return m_key.last();
}
int OctoKey::operator[](int ind) const
{
    Q_ASSERT(ind < depth());
    return (int)m_key[ind];
}

```

```

bool OctoKey::operator==(const OctoKey &another) const
{
    if (another.m_key.size() != m_key.size())
        return false;
    for(int i = 0; i < m_key.size(); i++) {
        if (another.m_key[i] != m_key[i])
            return false;
    }
    return true;
}
bool OctoKey::operator!=(const OctoKey &another) const
{
    return !(*this == another);
}
QVector<unsigned char> OctoKey::key() const
{
    return m_key;
}
}

```

## Приложение 14. Листинг классов октодерева.

```

template <typename T>
class OctoTree : public ITree<T>
{
public:
    OctoTree();
    OctoTree(const int depth);
    ~OctoTree();
    void clear();
    void balance();
    bool setValue(const OctoKey &key, const T &value);
    bool exist(const OctoKey &key, const int type = Leaf) const;
    bool get(const OctoKey &key, T &value) const;
    bool remove(const OctoKey &key);
    bool canSplit(const OctoKey &key) const;
    bool canUnite(const OctoKey &key) const;
    bool splitNode(const OctoKey &key);
    bool uniteNode(const OctoKey &key);
    int getMaxDepth() const;
protected:
    bool findCurrentNode(const OctoKey &key, OctoNode<T> *&node) const;
    int calculateMaxDepth();
    OctoNode<T>* m_mainNode = nullptr;
    int m_maxDepth = 0;
};
template<typename T>
OctoTree<T>::OctoTree()
{
    m_mainNode = new OctoNode<T>(Main, 0);
}
template<typename T>
OctoTree<T>::OctoTree(const int depth)
{
    if (depth == 0) {
        m_mainNode = new OctoLeaf<T>(Main, 0);
    } else if (depth >= 1) {
        m_mainNode = new OctoBranch<T>(Unknown, 0);
        QQueue<OctoNode<T>*> nextNodes;
        nextNodes.enqueue(m_mainNode);
        while (nextNodes.size() != 0) {
            OctoNode<T>* currentNode = nextNodes.dequeue();
            currentNode->split();
            if (currentNode->getCurrentDepth() + 1 < depth)
                for(int i = 0; i < 8; i++)
                    nextNodes.enqueue(currentNode->getChild(i));
        }
    }
}
template<typename T>

```

```

OctoTree<T>::~~OctoTree()
{
    clear();
    if (m_mainNode != nullptr)
        delete m_mainNode;
}
template<typename T>
void OctoTree<T>::clear()
{
    if (m_mainNode == nullptr) {
        m_mainNode = new OctoLeaf<T>(Main, 0);
        return;
    }
    m_mainNode->clear();
    m_maxDepth = 0;
}
template<typename T>
void OctoTree<T>::balance()
{
    if (m_mainNode == nullptr)
        return;
    m_mainNode->balance();
    m_maxDepth = getMaxDepth();
}
template<typename T>
bool OctoTree<T>::setValue(const OctoKey &key, const T &value)
{
    if (key.depth() < 1)
        return false;
    if (key[0] == Main) {
        if (m_mainNode->getType() == Leaf)
            m_mainNode->setValue(value);
        else
            return false;
    }
    OctoNode<T> *previousNode = nullptr;
    OctoNode<T> *currentNode = m_mainNode;
    for(int i = 0; i < key.depth(); i++) {
        Q_ASSERT(key[i] >= 0 && key[i] < 8);
        if (currentNode->getType() == Leaf)
            return false;
        if (currentNode == nullptr) {
            previousNode->split();
            currentNode = previousNode->getChild(key[i - 1]);
        }
        previousNode = currentNode;
        currentNode = currentNode->getChild(key[i]);
    }
    if (currentNode == nullptr){
        previousNode->split();
        currentNode = previousNode->getChild(key[key.depth() - 1]);
    }
}

```

```

const bool ok = currentNode->setValue(value);
if (ok) {
    m_maxDepth = std::max(key.depth(), m_maxDepth);
    return true;
} else
    return false;
}
template<typename T>
bool OctoTree<T>::exist(const OctoKey &key, const int type) const
{
    OctoNode<T> *node = nullptr;
    const bool existNode = findCurrentNode(key, node);
    if (existNode) {
        return node->getType() == type;
    } else
        return false;
}
template<typename T>
bool OctoTree<T>::get(const OctoKey &key, T &value) const
{
    OctoNode<T> *node = nullptr;
    const bool existNode = findCurrentNode(key, node);
    if (existNode && node->getType() == Leaf && node->value() != nullptr) {
        value = *(node->value());
        return true;
    } else
        return false;
}
template<typename T>
bool OctoTree<T>::remove(const OctoKey &key)
{
    OctoNode<T> *node = nullptr;
    const bool existNode = findCurrentNode(key, node);
    if (existNode) {
        node->clear();
        return true;
    } else
        return false;
}
template<typename T>
bool OctoTree<T>::splitNode(const OctoKey &key)
{
    OctoNode<T> *node = nullptr;
    const bool existNode = findCurrentNode(key, node);
    if (existNode) {
        const bool result = node->split();
        if (result) {
            m_maxDepth = std::max(key.depth() + 1, m_maxDepth);
            return true;
        } else
            return false;
    } else
        return false;
}

```

```

        return false;
    }
template<typename T>
bool OctoTree<T>::uniteNode(const OctoKey &key)
{
    OctoNode<T> *node = nullptr;
    const bool existNode = findCurrentNode(key, node);
    if (existNode) {
        const bool result = node->unite();
        if (result) {
            calculateMaxDepth();
            return true;
        } else
            return false;
    } else
        return false;
}
template<typename T>
int OctoTree<T>::getMaxDepth() const
{
    return m_maxDepth;
}
template<typename T>
bool OctoTree<T>::findCurrentNode(const OctoKey &key, OctoNode<T> *&node) const
{
    if (key.depth() < 1)
        return false;

    if (key[0] == Main) {
        if (m_mainNode->getType() == Leaf)
            return true;
        else
            return false;
    }
    OctoNode<T> *previousNode = nullptr;
    OctoNode<T> *currentNode = m_mainNode;
    for(int i = 0; i < key.depth(); i++) {
        Q_ASSERT(key[i] >= 0 && key[i] < 8);
        if (currentNode == nullptr)
            return false;
        previousNode = currentNode;
        currentNode = currentNode->getChild(key[i]);
    }
    if (currentNode == nullptr)
        return false;
    node = currentNode;
    return true;
}
template<typename T>
int depthOfTree(OctoNode<T> *node)
{
    if (node == nullptr)

```

```

        return 0;
    int c_maxDepth = 0;
    for(int i = 0; i < 8; i++)
        c_maxDepth = std::max(depthOfTree(node->getChild(0)), c_maxDepth);
    return c_maxDepth + 1;
}
template<typename T>
int OctoTree<T>::calculateMaxDepth()
{
    m_maxDepth = depthOfTree(m_mainNode);
    return m_maxDepth;
}
template <typename T>
class OctoNode : public ITreeNode<T>
{
public:
    OctoNode(const unsigned char id);
    virtual ~OctoNode() = 0;
    virtual void balance();
    virtual void clear();
    virtual bool canSplit() const;
    virtual bool canUnite() const;
    virtual bool split(const unsigned char id);
    virtual bool unite(const unsigned char id);
    virtual OctoNode *getChild(const unsigned char id) const;
    virtual T* value() const;
    virtual bool setValue(const T value);
    virtual int getType() const;
    unsigned char id() const;
    virtual QVector<OctoNode<T> *> *getChildren() const;
protected:
    const unsigned char m_id;
};
template<typename T>
OctoNode<T>::OctoNode(const unsigned char id, const int currentDepth) : m_id(id),
m_currentDepth(currentDepth)
{}
template<typename T>
OctoNode<T>::~~OctoNode()
{
    clear();
}
template<typename T>
bool OctoNode<T>::split(unsigned int id)
{
    delete m_children[id];
    m_children[id] = nullptr;
    m_children[id] = new OctoBranch(id);
}
template<typename T>
bool OctoNode<T>::unite()
{

```



```

        delete m_children[id];
        m_children[id] = nullptr;
        m_children[id] = new OctoLeaf(id);
    }
template<typename T>
unsigned char OctoNode<T>::id() const
{
    return m_id;
}
template<typename T>
QVector<OctoNode<T>*> *OctoBranch<T>::getChildren() const
{
    return m_children;
}
template <typename T>
class OctoBranch : public OctoNode<T>
{
    ~OctoBranch();
    void balance();
    void clear();
    bool canSplit() const;
    bool canUnite() const;
    bool split(const unsigned char id);
    bool unite(const unsigned char id);
    OctoNode *getChild(const unsigned char id) const;
    T* value() const;
    bool setValue(const T value);
    int getType() const;
    QVector<OctoNode<T>*> *getChildren() const;
protected:
    QVector<OctoNode<T>*> *m_children = nullptr;
}
template<typename T>
void OctoBranch<T>::clear()
{
    if (m_children != nullptr) {
        for (int i = 0; i < m_children->size(); i++) {
            if ((*m_children)[i] != nullptr) {
                (*m_children)[i]->clear();
                delete (*m_children)[i];
            }
        }
        m_children->clear();
        delete m_children;
        m_children = nullptr;
    }
}
template<typename T>
bool OctoBranch<T>::canSplit() const
{
    return false;
}

```

```

template<typename T>
bool OctoBranch<T>::canUnite() const
{
    for (int i = 0; i < 8; i++) {
        if (!((*m_children)[i] == nullptr || (*m_children)[i]->getType() == Unknown))
            return false;
    }
    return true;
}
template<typename T>
OctoNode<T> *OctoNode<T>::getChild(const unsigned char id) const
{
    Q_ASSERT(id >= 0 && id < 8);

    return (*m_children)[id];
}
template<typename T>
T *OctoNode<T>::value() const
{
    return nullptr;
}
template<typename T>
bool OctoNode<T>::setValue(const T value)
{
    return false;
}
template<typename T>
int OctoNode<T>::getType() const
{
    return Branch;
}
template<typename T>
int OctoNode<T>::getCurrentDepth() const
{
    return m_currentDepth;
}
template<typename T>
QVector<OctoNode<T>*> *OctoLeaf<T>::getChildren() const
{
    return m_children;
}
template <typename T>
class OctoLeaf : public OctoNode<T>
{
    ~OctoLeaf();
    void balance();
    void clear();
    bool canSplit() const;
    bool canUnite() const;
    bool split(const unsigned char id);
    bool unite(const unsigned char id);
    OctoNode *getChild(const unsigned char id) const;

```

```

    T* value() const;
    bool setValue(const T value);
    int getType() const;
    QVector<OctoNode<T>*> *getChildren() const;
protected:
    T *m_value = nullptr;
}
template<typename T>
void OctoLeaf<T>::balance()
{}
template<typename T>
void OctoLeaf<T>::clear()
{
    if (m_value != nullptr) {
        delete m_value;
        m_value = nullptr;
    }
}
template<typename T>
bool OctoLeaf<T>::canSplit() const
{
    return m_value != nullptr;
}
template<typename T>
bool OctoLeaf<T>::canUnite() const
{
    return false;
}
template<typename T>
OctoNode<T> *OctoNode<T>::getChild(const unsigned char id) const
{
    Q_ASSERT(id >= 0 && id < 8);
    if (m_type != Node)
        return nullptr;

    return (*m_children)[id];
}
template<typename T>
T *OctoNode<T>::value() const
{
    return m_value;
}
template<typename T>
bool OctoNode<T>::setValue(const T value)
{
    if (m_type == Node)
        return false;
    if (m_type == Unknown)
        m_type = Leaf;
    if (m_value == nullptr)
        m_value = new T(value);
    else

```

```
        *m_value = value;
    return true;
}
template<typename T>
int OctoNode<T>::getType() const
{
    return Leaf;
}
```

## Приложение 15. Листинг класса Transformation3D.

```
class Transformation3D
{
public:
    Transformation3D();
    QVector3D getPosition() const;
    void setPosition(const QVector3D &position);
    void movePosition(const QVector3D &move)
    void setRotation(const float &pitch, const float &yaw, const float &roll);
    void setRotation(const QQuaternion &quat);
    void setRotation(
        const QVector3D &up,
        const QVector3D &front = QVector3D(0, 0, 0),
        const QVector3D &right = QVector3D(0, 0, 0));
    void setUpVector(const QVector3D &up);
    void setFrontVector(const QVector3D &front);
    void setRightVector(const QVector3D &right);
    QVector3D getUpVector() const;
    QVector3D getFrontVector() const;
    QVector3D getRightVector() const;
    void getEulersAngles(float &pitch, float &yaw, float &roll) const;
    float getPitch() const;
    float getYaw() const;
    float getRoll() const;
    void rotatePitch(const float angle);
    void rotateYaw(const float angle);
    void rotateRoll(const float angle);
    void rotateEulerAngles(const float &addPitch, const float &addYaw, const float &addRoll);
    void getAxisAngle(float &angle, QVector3D &axis) const;
    void rotate(const float &angle);
    void rotate(const QQuaternion quat);
    void rotate(const float &angle, const QVector3D &vector);
    void rotate(const QVector3D &vector, const QVector3D oldVector, const QVector3D
&normal);
    void rotateUp(const float &angle);
    void rotateRight(const float &angle);
    void rotateFront(const float &angle);
    QVector3D getScale() const;
    void setScale(const QVector3D &scale);
    void addScale(const float scale);
    void addScale(const QVector3D &scale);
    QMatrix4x4 getModelMatrix() const;
    QMatrix4x4 getRotationMatrix() const;
    QMatrix4x4 getTranslationMatrix() const;
    QMatrix4x4 getScaleMatrix() const;
    QQuaternion getQuaternion() const;
protected:
    QVector3D m_position = QVector3D(0,0,0);
    QQuaternion m_rotation = QQuaternion(1,0,0,0);
```

```

    QVector3D m_scale = QVector3D(1,1,1);
}
Transformation3D::Transformation3D()
{}
QVector3D Transformation3D::getPosition() const
{
    return m_position;
}
void Transformation3D::setPosition(const QVector3D &position)
{
    m_position = position;
}
void Transformation3D::movePosition(const QVector3D &move)
{
    m_position += move;
}
void Transformation3D::setRotation(const float &pitch, const float &yaw, const float &roll)
{
    m_rotation = QQuaternion::fromEulerAngles(pitch, yaw, roll);
}
void Transformation3D::setRotation(const QQuaternion &quat)
{
    m_rotation = quat;
}
void Transformation3D::setRotation(const QVector3D &up, const QVector3D &front, const
QVector3D &right)
{
    m_rotation = QQuaternion::fromAxes(right.normalized(), up.normalized(),
front.normalized());
}
void Transformation3D::setUpVector(const QVector3D &up)
{
    rotate(up.normalized(), getUpVector(),
QVector3D::crossProduct(up.normalized(),getUpVector()));
}
void Transformation3D::setFrontVector(const QVector3D &front)
{
    rotate(front.normalized(), getUpVector(),
QVector3D::crossProduct(front.normalized(),getUpVector()));
}
void Transformation3D::setRightVector(const QVector3D &right)
{
    rotate(right.normalized(),
getUpVector(),QVector3D::crossProduct(right.normalized(),getUpVector()));
}
QVector3D Transformation3D::getUpVector() const
{
    return (getRotationMatrix() * QVector3D(0, 1, 0)).normalized();
}
QVector3D Transformation3D::getFrontVector() const
{
    return (getRotationMatrix() * QVector3D(0, 0, 1)).normalized();
}

```

```

}
QVector3D Transformation3D::getRightVector() const
{
    return (getRotationMatrix() * QVector3D(1, 0, 0)).normalized();
}
void Transformation3D::getEulersAngles(float &pitch, float &yaw, float &roll) const
{
    m_rotation.getEulerAngles(&pitch,&yaw,&roll);
}
float Transformation3D::getPitch() const
{
    float pitch, yaw, roll;
    m_rotation.getEulerAngles(&pitch,&yaw,&roll);
    return pitch;
}
float Transformation3D::getYaw() const
{
    float pitch, yaw, roll;
    m_rotation.getEulerAngles(&pitch,&yaw,&roll);
    return yaw;
}
float Transformation3D::getRoll() const
{
    float pitch, yaw, roll;
    m_rotation.getEulerAngles(&pitch,&yaw,&roll);
    return roll;
}
void Transformation3D::rotatePitch(const float angle)
{
    float pitch, yaw, roll;
    getEulersAngles(pitch, yaw, roll);
    pitch += angle;
    setRotation(pitch, yaw, roll);
}
void Transformation3D::rotateYaw(const float angle)
{
    float pitch, yaw, roll;
    getEulersAngles(pitch, yaw, roll);
    yaw += angle;
    setRotation(pitch, yaw, roll);
}
void Transformation3D::rotateRoll(const float angle)
{
    float pitch, yaw, roll;
    getEulersAngles(pitch, yaw, roll);
    roll += angle;
    setRotation(pitch, yaw, roll);
}
void Transformation3D::rotateEulerAngles(const float &addPitch, const float &addYaw, const
float &addRoll)
{
    float pitch, yaw, roll;

```

```

    getEulersAngles(pitch, yaw, roll);
    pitch += addPitch;
    yaw += addYaw;
    roll += addRoll;
    setRotation(pitch, yaw, roll);
}
void Transformation3D::getAxisAngle(float &angle, QVector3D &axis) const
{
    m_rotation.getAxisAndAngle(&axis, &angle);
}
void Transformation3D::rotate(const float &angle)
{
    QVector3D vector;
    float currAngle;
    m_rotation.getAxisAndAngle(&vector, &currAngle);
    m_rotation.fromAxisAndAngle(vector, angle + currAngle);
}
void Transformation3D::rotate(const QQuaternion quat)
{
    m_rotation *= quat;
}
void Transformation3D::rotate(const float &angle, const QVector3D &vector)
{
    m_rotation *= QQuaternion::fromAxisAndAngle(vector.normalized(), angle);
}
void Transformation3D::rotate(const QVector3D &vector, const QVector3D oldVector, const
QVector3D &normal)
{
    if (!qFuzzyCompare(vector.normalized(), normal.normalized()) &&
!qFuzzyCompare(normal.normalized(), oldVector.normalized()))
        return;
    const QVector3D rightV = QVector3D::crossProduct(vector.normalized(),
normal.normalized());
    const QVector3D projVector =
QVector3D::crossProduct(rightV.normalized(), normal.normalized());
    const QVector3D rightOldV = QVector3D::crossProduct(oldVector.normalized(),
normal.normalized());
    const QVector3D projOldVector =
QVector3D::crossProduct(rightOldV.normalized(), normal.normalized());
    const float angle = QVector3D::dotProduct(projVector, projOldVector);
    if (isLSC(projVector.normalized(), normal.normalized(), projOldVector.normalized()))
        rotate(angle, normal);
    else
        rotate(-angle, normal);
    m_rotation.normalize();
}
void Transformation3D::rotateUp(const float &angle)
{
    QQuaternion rotation = QQuaternion::fromAxisAndAngle(getUpVector(), angle);
    rotation.normalize();
    m_rotation = m_rotation * rotation;
    m_rotation.normalize();
}

```



```

}
void Transformation3D::rotateRight(const float &angle)
{
    QQuaternion rotation = QQuaternion::fromAxisAndAngle(getRightVector(), angle);
    rotation.normalize();
    m_rotation = rotation * m_rotation;
    m_rotation.normalize();
}
void Transformation3D::rotateFront(const float &angle)
{
    QQuaternion rotation = QQuaternion::fromAxisAndAngle(getFrontVector(),angle);
    rotation.normalize();
    m_rotation = rotation * m_rotation;
    m_rotation.normalize();
}
QVector3D Transformation3D::getScale() const
{
    return m_scale;
}
void Transformation3D::setScale(const QVector3D &scale)
{
    m_scale = scale;
}
void Transformation3D::addScale(const float scale)
{
    m_scale *= scale;
}
void Transformation3D::addScale(const QVector3D &scale)
{
    m_scale[0] *= scale[0];
    m_scale[1] *= scale[1];
    m_scale[2] *= scale[2];
}
QMatrix4x4 Transformation3D::getModelMatrix() const
{
    return getRotationMatrix() * getTranslationMatrix() * getScaleMatrix();
}
QMatrix4x4 Transformation3D::getRotationMatrix() const
{
    return QMatrix4x4(m_rotation.toRotationMatrix().transposed());
}
QMatrix4x4 Transformation3D::getTranslationMatrix() const
{
    QMatrix4x4 matrix;
    matrix.translate(-m_position);
    return matrix;
}
QMatrix4x4 Transformation3D::getScaleMatrix() const
{
    QMatrix4x4 matrix;
    matrix.scale(m_scale);
    return matrix;
}

```

```

}
bool isLSC(const QVector3D &right, const QVector3D &up, const QVector3D &front) const
{
    const float first = right[0] * (up[1] * front[2] - up[2] * front[1]);
    const float second = right[1] * (up[0] * front[2] - up[2] * front[0]);
    const float third = right[2] * (up[0] * front[1] - up[1] * front[0]);

    return first - second + third > 0;
}
Quaternion Transformation3D::getQuaternion() const
{
    return m_rotation;
}

```

## Приложение 16. Листинг класса Voxel.

```
class Voxel
{
public:
    QVector3D position;
    int size = 1;
    Material::Material *material = nullptr;
};
```