

Содержание

1 Введение	3
2 Обзор литературы	4
3 Функциональные и нефункциональные требования к проекту	4
3.1 Функциональные требования	4
3.2 Нефункциональные требования	5
4 Теоретическая часть	5
4.1 Введение	5
4.2 Геометрические примитивы	5
4.3 Преобразования пространства и камеры	6
4.3.1 Объект камеры	6
4.3.2 Однородные координаты	6
4.3.3 Сдвиг камеры	7
4.3.4 Поворот камеры	7
4.3.5 Перенос объектов в пространство камеры	8
4.3.6 Пирамида зрения и её преобразование в параллелепипед	9
4.3.7 Клиппинг треугольников	10
4.3.8 Область видимости	11
4.3.9 Сдвиг параллелепипеда и масштабирование в куб	12
4.4 Отрисовка на экран	12
4.4.1 Проекция на экран	12
4.4.2 Z-буфер	12
4.4.3 Интерполяция цвета	13
4.4.4 Отрисовка треугольника на экран с помощью сканирующей прямой	13
4.5 Заключение	14
5 Имплементация движка и приложения	15
5.1 GLM	15
5.2 SFML	16
5.3 Vector4	16
5.4 Vertex	16
5.5 Triangle	17
5.6 Object	17
5.7 World	18
5.8 Camera	18
5.9 Screen	19
5.10 Loader	19
5.11 Storage2d	20
5.12 Renderer	20
5.13 Application	20
5.14 Заголовочные файлы global_usings.h и configuraton.h	21
5.15 Общая диаграмма классов с зависимостями	22
6 Тестирование	23
6.1 Unit-тестирование	23
6.2 Ручное тестирование	23
7 Заключение	26

Аннотация

В данном проекте разработан пайплайн для отрисовки трехмерных объектов на двумерный экран с нуля без привлечения внешних графических библиотек. В ходе работы изучены базовые принципы компьютерной графики и необходимые алгоритмы для преобразования трехмерных сцен в двумерное изображение, реализованы и протестированы основные функциональности движка рендера, а также имплементировано GUI-приложение для демонстрации работы движка и загрузчик объектов из файлов формата OFF.

Ключевые слова

Рендеринг, 3D сцены, трёхмерная геометрия, растеризация, компьютерная графика, C++, интерактивные приложения.

1 Введение

В современном мире компьютерная графика играет ключевую роль во множестве приложений, начиная от развлекательных игр и кинематографии до научных и инженерных визуализаций. Одним из фундаментальных аспектов создания реалистичных изображений является процесс рендеринга трехмерных сцен. Целью проекта является разработка 3D-рендерера без использования сторонних графических библиотек, применяя только базовые математические и графические принципы: геометрические примитивы, проективные преобразования, растеризация, буфер глубины, интерполяция и т.д. Это позволяет глубоко понять принципы работы с графикой и каждый этап создания рендера в деталях. Не менее важной задачей проекта является разработка интуитивно понятного графического пользовательского интерфейса для удобства визуализации и демонстрации работы с рендерером.

Реализация проекта разделена на три глобальные части: создание пайплайна для отрисовки 3D-сцен на экране (движка); имплементация интерактивного приложения, визуализирующего для пользователя работу движка, а также загрузчика 3D-моделей из объектных файлов; тестирование разработанного приложения с помощью ручного и юнит-тестирования.

В первой части проекта, посвященной разработке движка, были поставлены и выполнены следующие задачи: в начале проекта ознакомиться с основными концепциями компьютерной графики, такими как трехмерная геометрия, проективные преобразования, растеризация. Далее изучить, на какие этапы должен быть поделён пайплайн, проанализировав соответствующую литературу и существующие решения с открытым исходным кодом; и на основе этого продумать организацию собственного решения. Затем перейти непосредственно к разработке движка на языке программирования C++, начиная с реализации базовых геометрических примитивов и необходимых функций для работы с ними. Далее выделить и имплементировать несколько основных сущностей: мир и объекты в нём, составленные из примитивов, камера (для определения положения наблюдателя), экран (плоскость, на которую отрисовываются трёхмерные объекты), рендерер (непосредственно производит отрисовку сцены, используя всё вышеупомянутое). По ходу работы реализовать все необходимые алгоритмы, такие как: перенос объектов из глобальной системы координат в систему координат камеры; проекция объектов из трёхмерного пространства в двумерную плоскость экрана; корректная обработка объектов, которые частично или полностью выходят за область видимости камеры(клиппинг); растеризация и отрисовка треугольников на экран. В конечном итоге все эти компоненты были объединены в движок (рендерер), способный реалистично отображать трехмерные объекты.

Во второй части проекта, посвященной разработке GUI-приложения для демонстрации работы рендера, основной задачей стала разработка интуитивно понятного интерфейса, который позволит пользователям взаимодействовать с рендерером и просматривать результаты его работы. Сначала были изучены основные фреймворки и библиотеки для создания пользовательских GUI-приложений, среди них была выбрана наиболее подходящий для решения задачи — библиотека SFML [11]. После этого имплементировано само приложение, которое отображает результат работы рендера в реальном времени с возможностью взаимодействия с трехмерными объектами с помощью клавиатуры: позволяет изменять точку обзора, вращать камеру по различным осям и просматривать загруженную сцены с различных углов. Кроме того, реализован загрузчик 3D-моделей из объектных файлов OFF формата, для того, чтобы продемонстрировать рендеринг более сложных моделей.

В заключительной части проекта проведено тестирование приложения двумя способами: ручное и юнит-тестирование с использованием фреймворка googletest [1]. Ручное тестирование включает в себя загрузку ряда различных моделей и ручную проверку их правильного рендеринга и отображения на экран; проверку корректной работы приложения и проецирования объектов на экран при различных сдвигах и поворотах камеры. Юнит-тестирование подразумевает разработку и запуск ряда тестов, проверяющих, что отдельные модули приложения работают должным образом.

Таким образом, итогом курсового проекта стал разработанный практически с нуля движок для 3D-рендеринга и приложение для демонстрации его работы. Это приложение способно в реальном времени отображать трехмерные сцены, обеспечивая пользователей возможностью взаимодействия с объектами и загрузки моделей. Также были проведены тестирования, подтверждающие корректную работу приложения.

Код проекта выложен в открытом доступе на [Github](#).

2 Обзор литературы

За основу работы над проектом было решено взять книгу «Mathematics for 3D game programming and computer graphics» [9], поскольку в ней подробно описаны как общий план построения пайплайна, так и советы по имплементации каждого шага, а также приведены все необходимые для рендеринга математические формулы.

Также для отдельных элементов дизайна были изучены решения одной из наиболее распространённых графических библиотек с открытым исходным кодом OpenGL, пользуясь официальной документацией [8], а также книгой «3-D Computer Graphics A Mathematical Introduction with OpenGL» [4].

Для математических вычислений с трёхмерными векторами и матрицами используется библиотека glm [7]. Другим возможным решением могла стать библиотека Eigen [6], однако было решено использовать glm, поскольку Eigen менее интуитивно понятна и более трудозатратна в освоении, а предлагаемого glm функционала достаточно для решения поставленной задачи.

В качестве возможных библиотек для написания GUI-приложения были рассмотрены Qt [10] и SFML [11] как одни из наиболее популярных для разработки на C++. Было решено остановиться на SFML, поскольку она обладает рядом преимуществ по сравнению с Qt: во-первых, её API более интуитивно и прямолинейно, в нём гораздо легче разобраться и использовать на практике. Во-вторых, она во многом оптимизирована для работы с мультимедийными данными и часто обеспечивает высокую производительность при выполнении графических операций, что хорошо подходит для поставленной задачи.

Также необходимо было выбрать формат данных для загрузчика 3D-моделей. Ознакомившись с различными наиболее популярными форматами файлов моделей — OBJ [5], STL [3], PLY [12] — было решено остановиться на формате OFF [2], поскольку он обладает рядом достоинств: он компактный и лёгкий для парсинга, имеет удобное и прямолинейное описание вершин и поверхностей объекта, и, в отличие от других форматов, позволяет напрямую задавать цвета отдельным вершинам или целым поверхностям. Остальные форматы требуют либо поддержки текстурирования поверхности, либо не поддерживают цвет вообще, либо поддерживают только цвет целой поверхности, не позволяя задать цвет отдельным вершинам для его последующей интерполяции.

3 Функциональные и нефункциональные требования к проекту

3.1 Функциональные требования

- Запуск и инициализация приложения:** при запуске приложение должно инициализировать необходимые компоненты: графический движок и пользовательский интерфейс.
- Рендеринг графических примитивов:** движок должен поддерживать рендеринг базовых графических примитивов (треугольников), для разбиения объектов в сцене на данные примитивы и последующей отрисовки.
- Поддержка произвольных объектов:** пользователь должен иметь возможность программно создать и отобразить на экране любой триангулируемый объект в пространстве.
- Работа с цветом:** движок должен поддерживать работу с цветами для реализации различных эффектов визуализации, таких как окрашивание целых треугольников или их отдельных вершин с последующей интерполяцией цвета на внутренние точки треугольника.
- Инкапсуляция внутренних методов:** внутренние функции рендера, математические вычисления и преобразования, отрисовка на экран должны быть скрыты от пользователя. Пользователь должен иметь возможность лишь создавать и перемещать объекты, менять угол и точку обзора.
- Загрузка трёхмерной сцены:** пользователь должен иметь возможность загрузить трехмерные модели сцены из поддерживаемых файловых форматов (OFF).
- Интерактивный просмотр сцены:** пользователь должен иметь возможность интерактивно взаимодействовать со сценой с помощью клавиатуры, включая вращение, масштабирование и перемещение камеры. Приложение должно предоставлять удобный и интуитивно понятный интерфейс, позволяющий легко взаимодействовать с трёхмерными сценами.
- Отображение результата рендеринга:** приложение должно оперативно отображать результаты рендеринга сцены на экране, чтобы внесённые пользователем изменения были видны ему в реальном времени.

3.2 Нефункциональные требования

1. **Язык программирования:** весь код должен быть написан на языке программирования C++ 20 в соответствии с требованиями стандарта.
2. **Codestyle и форматирование:** весь код должен соответствовать стилю LLVM styleguide. Для проверки соблюдения стиля должен использоваться файл .clang-format.
3. **IDE:** разработка должна вестись с помощью IDE Visual Studio Code.
4. **Использование библиотек и фреймворков:** в проекте используются две внешние библиотеки: glm для математических вычислений и sfml для реализации интерактивного пользовательского приложения.
5. **Управление версиями:** весь код проекта должен храниться в системе контроля версий Git. Все изменения в коде должны фиксироваться с использованием коммитов, содержащих информативные сообщения о внесенных изменениях. Для разработки новых функциональностей и исправления ошибок должны использоваться отдельные от главной ветки с последующими вливаниями в главную.
6. **Структура проекта и файлы CMakeLists:** В корне проекта и в ряде его поддиректорий должны присутствовать файлы CMakeLists.txt для сборки проекта с использованием CMake. Файлы CMakeLists.txt включает в себя правильные настройки компилятора и определение зависимостей для Cmake-сборки проекта и его частей.
7. **Юнит-тестирование:** для проверки корректной работы частей проекта в отдельной поддиректории должен быть реализован ряд юнит-тестов с использованием фреймворка googletest [1].
8. **Публикация кода:** код проекта должен быть опубликован на Github в публичном репозитории с инструкциями по загрузке и запуску. Это позволяет любому желающему ознакомиться с его содержанием, скачать, собрать и запустить интерактивное приложение или использовать публичные методы движка в собственном коде.

4 Теоретическая часть

4.1 Введение

В реальном мире мы наблюдаем и взаимодействуем с трёхмерными объектами. Однако в анимации, компьютерных играх и т.д. мы видим эти объекты на двумерном экране. Соответственно, для того, чтобы на экране трёхмерные объекты отображались корректно и правдоподобно, необходимо провести ряд преобразований из трёхмерной сцены в двумерную проекцию на экран. Именно этим и занимаются рендереры.

В общих чертах базовый процесс рендеринга выглядит так: сначала сложные трёхмерные объекты разбиваются на простые примитивы, с которыми удобнее работать и проводить любые преобразования. Затем все примитивы с помощью линейных преобразований переносятся в систему координат камеры — это необходимо, поскольку мы хотим иметь возможность наблюдать сцену из разных точек и под разными углами. Далее отбрасываются или обрезаются примитивы, которые не входят в поле обзора, заданное пирамидой, после чего проективным преобразованием пирамида зрения переводится в параллелепипед. Затем полученные примитивы переводятся в плоскость экрана путём отбрасывания z -координаты, и итеративно отрисовываются на экран путём сканлайна, причём при равенстве x и y координат, отрисован будет пиксель объекта, имеющий меньшую z -координату (глубину).

В последующих подразделах данного раздела эти пункты рассказаны более детально, а также подробно описана необходимая для них математика.

4.2 Геометрические примитивы

Отрисовывать целые сложные объекты неудобно и затратно, поэтому перед обработкой рендерером каждый объект делится на набор примитивов, с которыми уже удобно работать. Как правило, примитивами являются вершины, прямые линии (отрезки) и их комбинации, а также многоугольники. Так, например, в OpenGL [4] определено 10 различных видов примитивов (рис. 1). Однако в большинстве случаев достаточно лишь двух из них — вершины и треугольника. Таким образом, любой объект можно задать набором точек-вершин в глобальной системе координат и треугольников, состоящих из этих вершин.

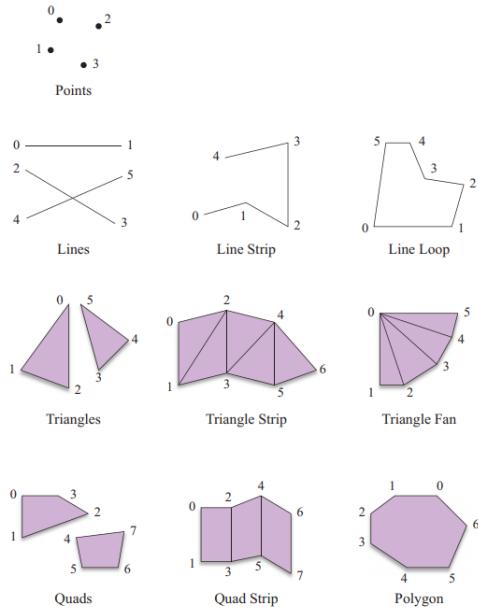


Рис. 1: Типы примитивов в OpenGL

Каждая вершина задаётся тремя координатами (x, y, z), а также может иметь дополнительные свойства, например, цвет или текстуру. Треугольник задаётся тройкой вершин, а свойства внутренних точек треугольника можно определить, проинтерполировав свойства его вершин (для этого можно использовать разные алгоритмы, например, барицентрические координаты или среднее арифметическое). Подробнее про интерполяцию рассказано в [4.4.3](#).

Во многих форматах 3D-моделей, объект задаётся набором вершин и поверхностей — многоугольников. В этом случае треугольники как примитивы тоже удобны: любую поверхность можно триангулировать и работать с ней уже как с набором треугольников.

Каждый объект существует независимо от других, с ним можно работать, не изменяя состояние других объектов. При этом любое изменение объекта (например, сдвиг или поворот) будет представлять собой соответствующее изменение набора примитивов, из которых состоит этот объект.

4.3 Преобразования пространства и камеры

4.3.1 Объект камеры

В реальном мире мы не наблюдаем все объекты сразу: во-первых, поле зрения имеет ограниченный размер, и мы не видим, например, объекты, находящиеся за нашей спиной; во-вторых, одни объекты могут загораживать другие и в этом случае мы видим только тот объект, что находится ближе к нам. Чтобы смоделировать эти явления программно, вводится объект камеры. Камера представляет собой точку в пространстве (точку, откуда смотрит наблюдатель) и матрицу поворота (поскольку, даже находясь в одной и той же точке, можно смотреть в разные стороны, и эта матрица как раз определяет, в каком направлении сейчас направлен взгляд). Введя объект камеры, мы сможем определить, какие объекты в данный момент попадают в поле зрения, а какие нет.

4.3.2 Однородные координаты

Все преобразования объектов наиболее рационально производить через преобразования их вершин, а все преобразования вершин — через представление их в виде вектора и умножения на матрицы преобразований. Однако, оказывается, что для некоторых необходимых преобразований недостаточно трёхмерных векторов и матриц: например, параллельный перенос вектора не выражается умножением на матрицу. Поэтому для удобства вводят четвёртую, так называемую *гомогенную* координату, и проводят соответствие векторов по следующему правилу:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}, w \neq 0$$

Таким образом, каждому трёхмерному вектору сопоставляется четырёхмерный с гомогенной четвёртой координатой, и каждому четырёхмерному вектору с ненулевой гомогенной координатой сопоставляется трёхмерный.

Такое расширение решает проблему с параллельным переносом — сдвиг вектора (x, y, z) на (x', y', z') выражается умножением на матрицу:

$$\begin{bmatrix} 1 & 0 & 0 & x' \\ 0 & 1 & 0 & y' \\ 0 & 0 & 1 & z' \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + x' \\ y + y' \\ z + z' \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x + x' \\ y + y' \\ z + z' \end{bmatrix}$$

Также теперь с помощью домножения на матрицу представляется возможным разделить все компоненты вектора на линейную функцию от его координат:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ ax + by + cz \end{bmatrix} \rightarrow \begin{bmatrix} \frac{x}{ax+by+cz} \\ \frac{y}{ax+by+cz} \\ \frac{z}{ax+by+cz} \\ 1 \end{bmatrix}$$

Как будет видно далее, добавления гомогенной четвёртой координаты будет достаточно для выражения всех необходимых преобразований в виде умножения матриц и/или умножения матрицы на вектор.

4.3.3 Сдвиг камеры

Как было замечено в предыдущем разделе, параллельный перенос вектора выражается одним умножением на матрицу. Таким образом, если мы хотим сдвинуть камеру из её текущего положения на вектор (x, y, z) , то достаточно домножить вектор её текущего положения на матрицу

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4.3.4 Поворот камеры

Матрица поворота камеры задаёт три базисных ортонормированных вектора, которые определяют положение камеры в пространстве (изначально матрица единичная и базис стандартный). Допустим, мы хотим повернуть камеру вокруг одной из осей на угол α . Тогда вектор, соответствующий оси, вокруг которой хотим повернуть, остаётся на месте, а другие два базисных вектора поворачиваются. Мы хотим повернуть любой вектор в плоскости, задаваемой базисными ортонормированными векторами \vec{e}_1, \vec{e}_2 , на угол α . Для этого достаточно повернуть сами базисные векторы на угол α . При повороте \vec{e}_1 перейдёт в $\cos \alpha \vec{e}_1 + \sin \alpha \vec{e}_2$ (буквально по определению тригонометрического круга), а \vec{e}_2 , соответственно, в перпендикулярный ему $\sin \alpha \vec{e}_1 - \cos \alpha \vec{e}_2$ (рис. 2). Таким образом, чтобы повернуть, камеру, например, вокруг оси Oz на угол α , достаточно умножить её текущую матрицу поворота на

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

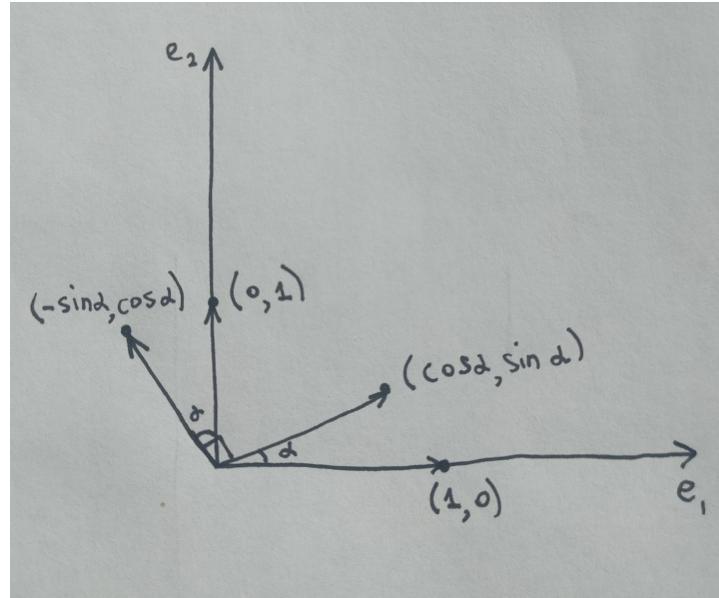


Рис. 2: Поворот базисных векторов в плоскости

Нетрудно заметить, что любая такая матрица поворота R ортогональна, то есть $R^{-1} = R^T$. В частности это значит, что если мы применим несколько матриц поворота $R_1 R_2 \dots R_k = B$, то, чтобы получить обратное к применённым поворотам преобразование, достаточно взять B^T .

Также отметим, что, как правило, при сдвиге мы хотим перемещать камеру не на абсолютный вектор сдвига, а относительно её текущего положения: влево-вправо, вверх-вниз, вперёд-назад. Как записать этот сдвиг с помощью формул? Допустим, мы хотим подвинуть камеру вправо на k . Тогда достаточно взять первый базисный вектор, умноженный на k , и прибавить его к сдвигу. Если хотим подвинуть камеру вниз на k , достаточно взять третий базисный вектор, умноженный на $-k$, и прибавить его к сдвигу. Аналогично со всеми остальными направлениями. В общем случае, если мы хотим подвинуть камеру на \vec{v} , учитывая текущий поворот камеры R , нужно применять сдвиг $R^T \vec{v}$.

4.3.5 Перенос объектов в пространство камеры

Перемещения и повороты камеры — это, по сути, перемещение центра координат и поворот базисных осей. Поэтому, чтобы представить координаты объектов в новом базисе, достаточно провести с ними преобразования, обратные преобразованиям камеры.

Так, если к камере применён сдвиг (x_c, y_c, z_c) , то к объектам надо применить сдвиг $(-x_c, -y_c, -z_c)$. Если к камере применена матрица поворота R , то к объектам надо применить матрицу поворота R^T . Таким образом, чтобы перевести объект в базис камеры, достаточно применить к каждой его вершине комбинацию двух матриц:

$$R^T \begin{bmatrix} 1 & 0 & 0 & -x_c \\ 0 & 1 & 0 & -y_c \\ 0 & 0 & 1 & -z_c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4.3.6 Пирамида зрения и её преобразование в параллелепипед

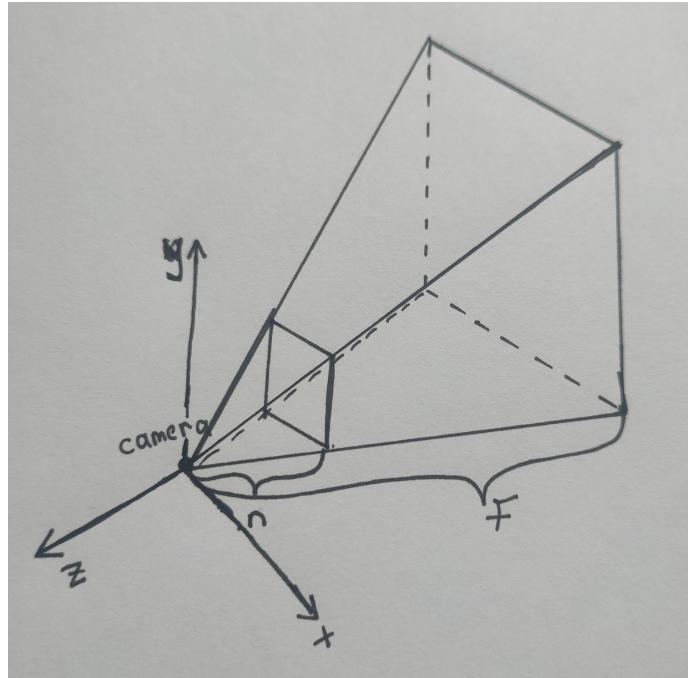


Рис. 3: Пирамида зрения

На рис. 3 изображена *пирамида зрения* — ограниченная часть пространства, в которую попадают все видимые для камеры объекты. Пирамида зрения ограничена шестью плоскостями. Две из них — ближняя, n и дальняя, f — определяются расстояниями n и f от камеры. Все объекты ближе n и дальше f для камеры невидимы. Остальные четыре плоскости соответствуют верхним, нижним, правым и левым рёбрам экрана и проходят через точку, в которой находится камера. Координатные оси в базисе камеры при этом направлены так, как показано на рис. 3.

Задавать область зрения в виде пирамиды необходимо, чтобы правильно реализовать эффект перспектизы: объекты, расположенные дальше от камеры, должны казаться меньше, а объекты, расположенные ближе к камере — больше. Однако пирамиду неудобно проецировать на экран, поэтому предварительно её преективным преобразованием переводят в параллелепипед. Преективное преобразование в нашем случае можно выразить умножением на следующую матрицу:

$$P = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Под действием матрицы P вектор v претерпит следующие преобразования:

$$v = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} nx \\ ny \\ (n+f)z - nf \\ z \end{bmatrix} \rightarrow \begin{bmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ n + f - \frac{nf}{z} \end{bmatrix}$$

Отсюда видно, что если до преобразования две точки имели координаты z_1, z_2 одного знака такие, что $z_1 < z_2$ (в следующем разделе будет ясно, что достаточно рассматривать только положительные z -координаты), то и после преобразования для новых координат будет верно $z'_1 < z'_2$. При этом ближняя и дальняя плоскости остаются инвариантными: $z = n$ переходит в $z = n$, а $z = f$ переходит в $z = f$.

Кроме того, координаты точки (x, y) переходят в $(\frac{nx}{z}, \frac{ny}{z})$. Если мы отбросим z координату, то из подобия треугольников (рис. 4) полученная точка будет являться корректной проекцией на ближнюю плоскость, в дальнейшем плоскость экрана.

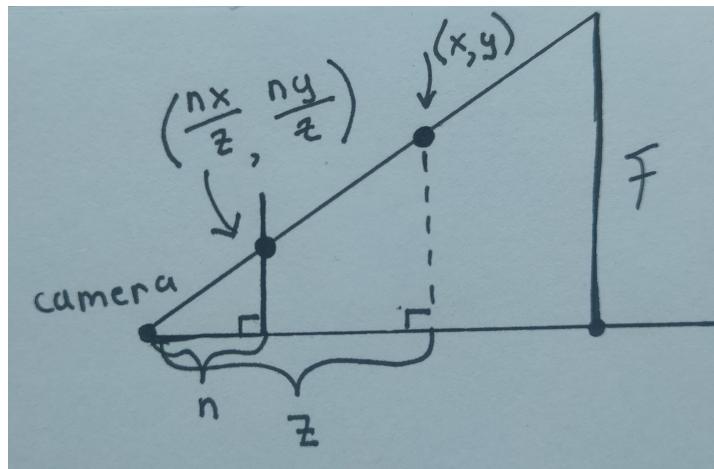


Рис. 4: Подобие треугольников при проективном преобразовании

4.3.7 Клиппинг треугольников

В предыдущем разделе при проективном преобразовании нам потребовалось деление на z . Но ведь деление не определено при $z = 0$, а если среди вершин присутствовали вершины с координатами z разного знака, то относительный порядок z -координат не сохранится. Однако, заметим, что ближняя плоскость находится на положительном расстоянии n от камеры. Поэтому точки с координатами $z < n$ камера не видит. А значит, перед проективным преобразованием можно просто убрать из рассмотрения вершины, которые камера не видит, и тогда оно точно будет корректным.

Тут есть нюанс: наши объекты состоят из треугольников, поэтому нам надо понимать, видны ли не отдельные вершины, а целые треугольники. Есть два простых случая: либо у всех вершин треугольника координата $z \geq n$, тогда он виден целиком; либо у всех вершин треугольника координата $z < n$, тогда он целиком не виден.

Если же в треугольнике у части вершин координата $z < n$, а у другой части координата $z \geq n$, то он пересекает ближнюю плоскость видимости и видимой остается та его часть, у которой z -координаты $\geq n$. Поскольку мы хотим всегда работать с треугольниками, эту новую видимую часть необходимо разбить на один или несколько треугольников. В зависимости от количества видимых вершин, в видимой части может получиться либо треугольник (1 видимая и две невидимые вершины), либо четырёхугольник (2 видимых и 1 невидимая) вершина, который можно разбить на 2 треугольника (рис. 5).

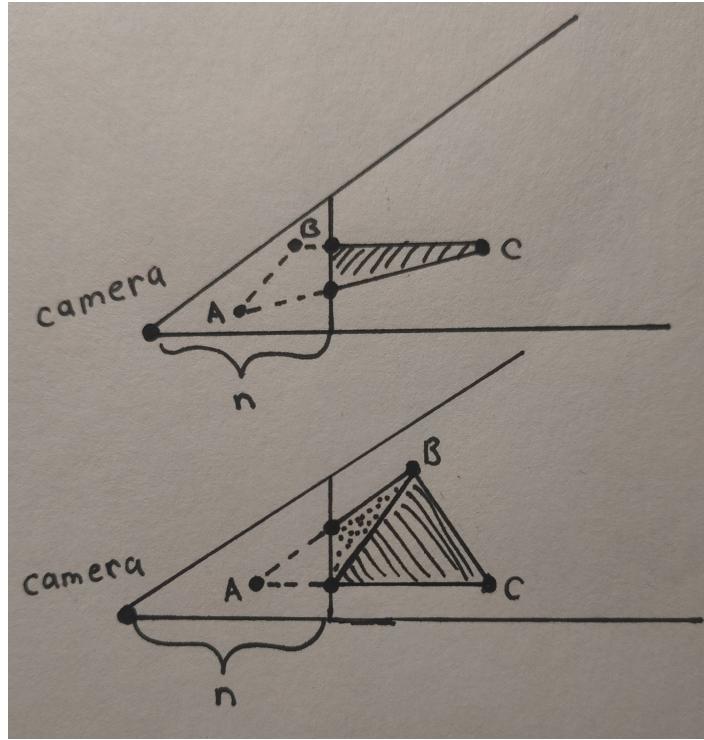


Рис. 5: Возможные варианты пересечения треугольника с ближней плоскостью

Для формирования новых треугольников осталось определить точки, в которых старый треугольник пересекается с ближней плоскостью. Это сводится к тому, чтобы научиться находить пересечение одного ребра треугольника с ближней плоскостью (при условии, что мы знаем, что оно его пересекает). Но ближняя плоскость задаётся как $z = n$, а значит, нужно найти точку на ребре с z -координатой, равной n . Если обозначить невидимый конец ребра за P_{back} , видимый за P_{front} , то точку пересечения $P_{intersect}$ можно найти по следующей формуле:

$$P_{intersect} = P_{back} + \frac{n - P_{back}.z}{P_{front}.z - P_{back}.z} \cdot \overrightarrow{P_{back}P_{front}}$$

Действительно, здесь мы просто откладываем от невидимого конца ребра вектор по направлению ребра, масштабированный так, чтобы у полученной точки оказалась координата $z = n$, как мы и хотели. Для определения цвета новой вершины нужно проинтерполировать цвета P_{back} и P_{front} (см. 4.4.3).

Описанный процесс по пересечению треугольников (или других примитивов) с плоскостью видимости принято называть *клиппингом*.

4.3.8 Область видимости

Для последующей работы с полученным параллелепипедом нам необходимо знать его положение в пространстве, а именно его крайнюю левую l , правую r , нижнюю b и верхнюю t точки. Крайнюю переднюю и заднюю точки мы уже знаем — это n и f , которые, как правило, подбираются и задаются вручную. Вообще говоря, l , r , t , b также можно задавать вручную, но более корректным будет вычислить их, используя горизонтальный угол обзора камеры α (также подбирается и задаётся вручную) и отошение высоты экрана к его ширине a .

Заметим, что l , r , b , t до преобразования пирамиды зрения в параллелепипед совпадают с крайней левой, правой, нижней и верхней наблюдаемой точкой задней плоскости соответственно. Рассмотрим плоскость, перпендикулярную направлению зрения камеры такую, что минимальная x -координата, наблюдаемая камерой $x = -1$, а максимальная $x = 1$. Каким будет расстояние e до такой плоскости? Поскольку мы знаем угол прямогоугольного треугольника $\frac{\alpha}{2}$ и длину противолежащего катета 1, то длина прилежащего катета $e = \frac{1}{\tan(\alpha/2)}$ (рис. 6). При этом, поскольку экран не обязательно квадратный, минимальная и максимальная видимая в данной плоскости y -координаты могут отличаться от ± 1 , а именно, они будут равны $\pm a$, при этом расстояние до данной плоскости всё ещё e (рис. 6). Это, в частности, значит, что вертикальный угол обзора камеры β

может отличаться от горизонтального, но знать его нам не обязательно, поскольку в дальнейших вычислениях он не понадобится.

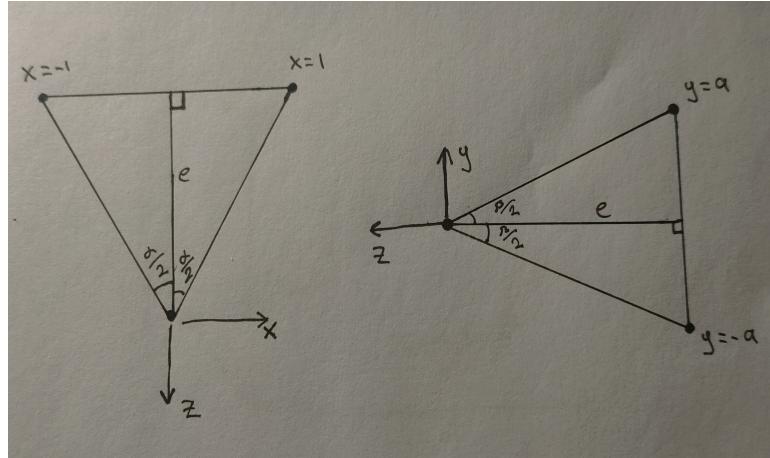


Рис. 6: Проекции области видимости камеры

Для дальней плоскости расстояние до неё равно f , при этом она параллельна рассмотренной. Тогда из подобия треугольников получаем $l = -\frac{n}{e}, r = \frac{n}{e}, b = -\frac{an}{e}, t = \frac{an}{e}$.

4.3.9 Сдвиг параллелепипеда и масштабирование в куб

Для удобства дальнейшей работы, после преобразования пирамиды зрения полученный параллелепипед сдвигается центром в начало координат и масштабируется в куб $(-1, -1, -1), (1, 1, 1)$ последовательным применением двух матриц:

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Тогда все видимые точки имеют координаты в диапазоне $[-1, 1]$.

4.4 Отрисовка на экран

4.4.1 Проекция на экран

Экран, на который проецируются точки, представляет собой прямоугольник с координатами углов $(0, 0)$ и (w, h) , где w — ширина экрана, h — высота экрана. При проекции точки из куба $(-1, -1, -1), (1, 1, 1)$ на экран z -координата отбрасывается, а x и y координаты масштабируются из отрезков $[-1, 1]$ в $[0, w]$ и $[0, h]$ соответственно. Таким образом, x переходит в $w \cdot \frac{x+1}{2}$, а y в $h \cdot \frac{y+1}{2}$, и, поскольку пиксели на экране имеют целочисленные координаты, то от новых x и y отбрасывается дробная часть.

4.4.2 Z-буфер

При проекции трёхмерных объектов на двумерный экран, в одну точку может спроектироваться несколько объектов с разными z -координатами. В этом случае нужно понять, точку какого из данных объектов отрисовывать на экран. В трёхмерном пространстве такая ситуация соответствует тому, что несколько объектов загораживают друг друга, и мы видим лишь ближайший из них. Значит, и на экран надо отрисовывать точку ближайшего из объектов, то есть точку с минимальной z -координатой. Этот метод называется *z-буфером*: для каждого пикселя экрана мы храним текущую минимальную z -координату (её также называют *глубиной*), изначально 1, и текущий цвет. Если в ходе отрисовки находится пиксель с теми же x, y координатами и меньшей глубиной, мы заменяем в буфере пиксель с большей глубиной на пиксель с меньшей.

4.4.3 Интерполяция цвета

Различные свойства, такие как, например, цвет, как правило, задаются только в вершинах треугольника. Однако, очевидно, что нам бы хотелось закрашивать не только точки-вершины, но и рёбра и внутренние части треугольников. Для этого нам надо по трём вершинам треугольника для внутренней точки уметь понимать, какой у неё будет цвет. Эту задачу можно свести к более простой: если мы умеем для отрезка с окрашенными концами понимать цвет внутренней точки, то для треугольника ABC и точки внутри него P мы можем провести прямую CP , понять, в какой точке Q она пересекает ребро AB , определить цвет Q , поскольку она лежит на отрезке AB , и затем цвет P , поскольку она лежит на отрезке CQ (рис. 7).

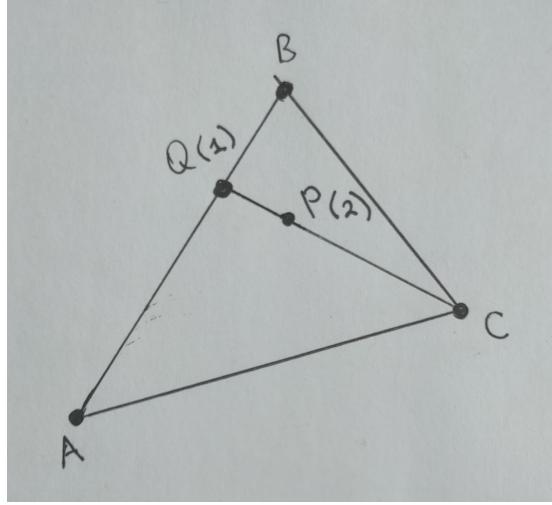


Рис. 7: При интерполяции сначала определяется цвет точки Q , затем цвет точки P

Таким образом, надо научиться решать следующую задачу: есть две вершины v, u , каждая со своим цветом, определить цвет некоторой внутренней точки отрезка w . Ясно, что цвет w должен быть пропорционален расстоянию w от v и u : чем ближе w находится к v , тем больше её цвет должен быть похож на цвет v ; чем ближе к u — тем больше на цвет u . Тогда цвет w можно задать просто пропорционально удалённости от каждого из концов отрезка (цвет задаётся в виде трёхмерного вектора RGB или четырёхмерного $RGBA$, поэтому с двумя цветами определены операции сложения, вычитания и умножения на число):

$$color_w = \left(1 - \frac{|vw|}{|uv|}\right) color_v + \left(1 - \frac{|uw|}{|uv|}\right) color_u = \left(1 - \frac{|vw|}{|uv|}\right) color_v + \frac{|vw|}{|uv|} color_u$$

Или, если обозначить за k коэффициент удалённости (его также называют *коэффициентом интерполяции*) w на отрезке uv от v :

$$color_w = (1 - k) \cdot color_v + k \cdot color_u$$

4.4.4 Отрисовка треугольника на экран с помощью сканирующей прямой

Рендерер обрабатывает каждый треугольник последовательно и независимо от других. Соответственно, для того, чтобы отрисовать все объекты, достаточно научиться отрисовывать один треугольник.

Для этого используется метод *сканирующей прямой*: треугольник отрисовывается последовательно по отрезкам. А именно, для одного треугольника во внешнем цикле рендерер перебирает все принадлежащие этому треугольнику целочисленные y -координаты в интервале $[0, h)$, а для фиксированной y -координаты — все принадлежащие треугольнику для данного y целочисленные x -координаты в интервале $[0, w)$.

Для удобства упорядочим вершины треугольника по возрастанию y -координаты: A вершина с минимальной y -координатой, B — со средней, C — с максимальной. Тогда понять диапазон y -координат для отрисовки легко — минимальный y равен $\max(0, A.y)$, максимальный y равен $\min(h - 1, C.y)$. А как при фиксированном $y = y_0$ понять диапазон координат по x ? Мысленно проведём прямую $y = y_0$ и посмотрим, в каких точках она пересекает наш треугольник. Одна из точек пересечения всегда будет на ребре AC ; вторая будет на ребре AB при $y_0 < B.y$ и на ребре BC при $y_0 \geq B.y$ (рис. 8). y -координаты концов отрезка $y_1 < y_2$ мы знаем,

y -координату внутренней точки y_0 тоже, так что можем получить коэффициент интерполяции $k = \frac{y_0 - y_1}{y_2 - y_1}$, и по нему получить x -координату вершины на ребре и её цвет.

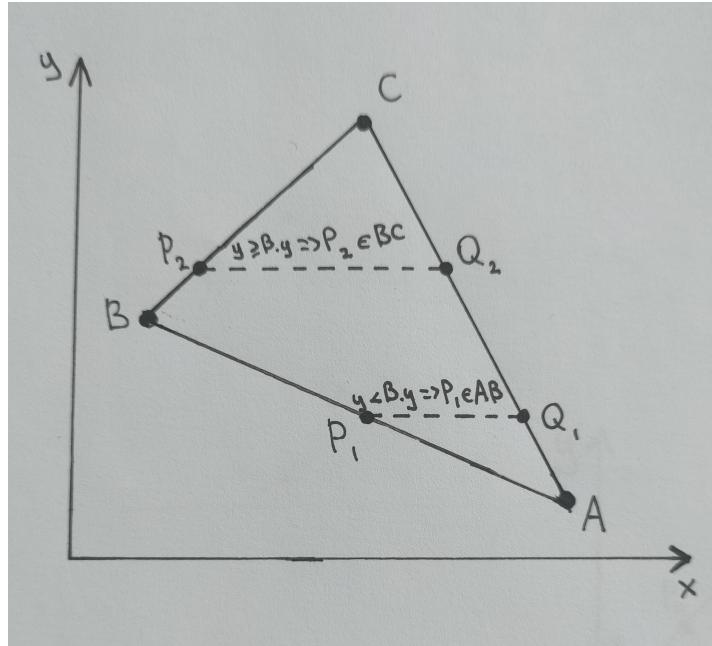


Рис. 8: Отрисовка отрезков треугольника с помощью сканлайна

Таким образом, мы научились получать вершины концов отрезка в треугольнике, параллельного оси Ox , который необходимо отрисовать. А для такого случая всё просто: если обозначить меньшую x -координату одной из полученных вершин за x_1 , большую за x_2 , то для отрисовки отрезка достаточно перебрать все x от $\max(0, x_1)$ до $\min(w - 1, x_2)$, получить их цвет по коэффициенту интерполяции $k = \frac{x - x_1}{x_2 - x_1}$ и обновить z -буфер, если глубина нового пикселя с координатами (x, y_0) оказалась меньше глубины лежащего в z -буфере.

4.5 Заключение

Подведём небольшие итоги данной главы и пройдёмся по ключевым этапам преобразования объектов и их отрисовки на экран:

- 3D-сцена состоит из объектов, каждый объект разбивается на набор примитивов — в нашем случае это вершины и треугольники. Чтобы применить какое-то преобразование к объекту, достаточно применить его ко всем примитивам объекта.
- Для удобства трёхмерные координаты объектов преобразовываются в четырёхмерные с так называемой гомогенной четвёртой координатой по правилу:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}, \quad \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}, w \neq 0$$

Далее работа всегда идёт с этими четырёхмерными координатами.

- Для управления углом и точкой обзора вводится объект камеры, задаваемый вектором — положением камеры в пространстве, и матрицей, задающей текущий поворот камеры. Камеру можно перемещать и поворачивать, домножая вектор положения камеры на соответствующую матрицу сдвига или матрицу поворота на соответствующую матрицу поворота. Для переноса координат вершины объекта в пространство камеры необходимо последовательно применить к ней две матрицы:

$$R^T \begin{bmatrix} 1 & 0 & 0 & -x_c \\ 0 & 1 & 0 & -y_c \\ 0 & 0 & 1 & -z_c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Здесь (x_c, y_c, z_c) координаты текущего положения камеры, R — матрица поворота камеры.

- Область видимости камеры задаётся в виде усечённой пирамиды, задаваемой 6 плоскостями, две из которых — ближняя и дальняя — перпендикулярны направлению зрения камеры, а расстояния до них равны n и f соответственно. Координаты крайних видимых точек на дальней плоскости l, r, b, t вычисляются через n, f , а также α — горизонтальный угол обзора камеры и a — отношение высоты к ширине экрана проекции.
- После переноса объектов в пространство камеры выполняется клиппинг — отбрасываются треугольники, у которых z -координаты всех вершин $< n$, остаются неизменными треугольники, у которых z -координаты $\geq n$, а треугольники, у которых у части вершин z -координата $< n$, а у части $\geq n$, обрезаются по передней плоскости, создавая один или два новых треугольника, которые уже являются полностью видимыми.
- После клиппинга пирамида зрения проективным преобразованием переводится в параллелепипед, после чего полученный параллелипед сдвигается своим центром в начало координат и масштабируется в куб $(-1, -1, -1), (1, 1, 1)$. Это задаётся последовательным применением трёх матриц:

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- При проекции на экран вершины её z -координата отбрасывается, x -координата переходит в $[w \cdot \frac{x+1}{2}]$, y -координата переходит в $[h \cdot \frac{y+1}{2}]$. При этом, если в одну точку на экране спроектировалось несколько вершин, из них будет отображена та, у которой наименьшая глубина (z -координата) — этот метод называется z -буфером.
- Рендерер отрисовывает каждый треугольник отдельно последовательно по отрезкам с помощью сканирующей прямой: сначала проходясь по диапазону видимых целочисленных y -координат треугольника, а для фиксированного y — по видимым целочисленным x -координатам для данного y , для каждого полученного пикселя обновляя при необходимости z -буфер. При этом для внутренней точки отрезка w её цвет можно получить, проинтерполировав цвет концов отрезка u, v :

$$color_w = (1 - k) \cdot color_v + k \cdot color_u,$$

где k — отношение расстояния от w до v к длине отрезка uv .

5 Имплементация движка и приложения

5.1 GLM

GLM [7] — это open-source C++ математическая библиотека для вычислений, необходимых для компьютерной графики. В движке из неё использованы часть элементов и операций. В частности, для работы с матрицами используется класс `glm::mat4` (в проекте Matrix4), поддерживающий умножение матриц, а также их поворот и сдвиг; для различных проверок на равенство используется функция `glm::equal`; а класс четырёхмерного вектора является наследником класса `glm::dvec4`.

5.2 SFML

SFML [11] — это open-source кроссплатформенная библиотека, которая позволяет создавать интерактивные оконные мультимедийные приложения. Она используется для создания оконного приложения, отрисовки пикселей на растеризованный экран и поддержки управления приложением с клавиатурой. Также для представления цвета в вершинах используется sf::Color.

5.3 Vector4

Vector4 — это класс четырёхмерного вектора с гомогенной координатой. Он является наследником класса glm::dvec4 (это нужно для удобства работы с матрицами), на нём переопределены операции сложения, вычитания и сравнения двух векторов, умножения и деления на число так, чтобы достигалось нужное соответствие трёхмерного вектора и четырёхмерного с гомогенной координатой. Кроме того, можно получить длину соответствующего трёхмерного вектора с помощью функции length() и нормализовать (привести к w -координате, равной 1) с помощью функции normalize().

Точка в пространстве отождествляется с вектором из нуля в эту точку, поэтому Vector4 используется также и для представления точек.

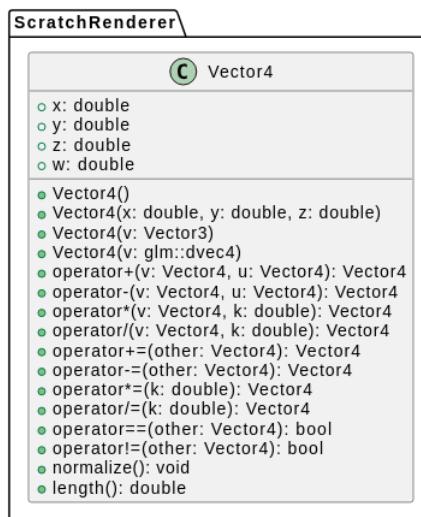


Рис. 9: Класс Vector4

5.4 Vertex

Для геометрических примитивов выделен отдельный namespace Primitives, куда входят два класса — Vertex и Triangle. Vertex — это класс примитива вершины, который хранит в себе её цвет и координаты, а также позволяет получить новую вершину, используя две вершины — концы отрезка и коэффициент интерполяции.

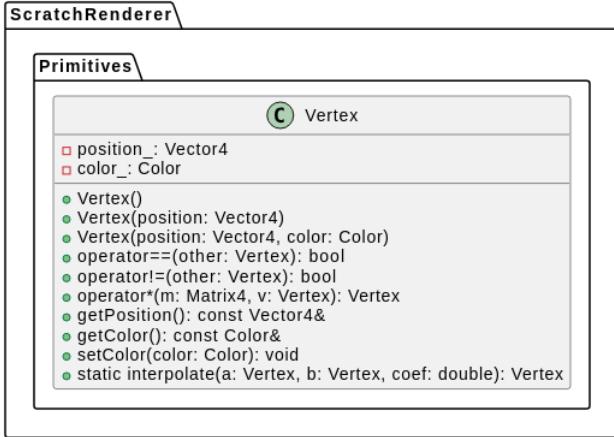


Рис. 10: Класс Vertex

5.5 Triangle

Triangle — класс примитива треугольника, хранящий в себе три вершины. Для удобства последующей отрисовки рендерером, при создании треугольника вершины всегда переупорядочиваются в порядке возрастания *y*-координаты. Имеется два метода для итерации по вершинам треугольника: один для итерации непосредственно по вершинам, а второй — по их координатам (для этого написан отдельный контейнер TrianglePositionsView с итератором ConstIterator). Также присутствует метод для получения нового треугольника по треугольнику и матрице преобразования (матрица применяется к координатам всех вершин старого треугольника).

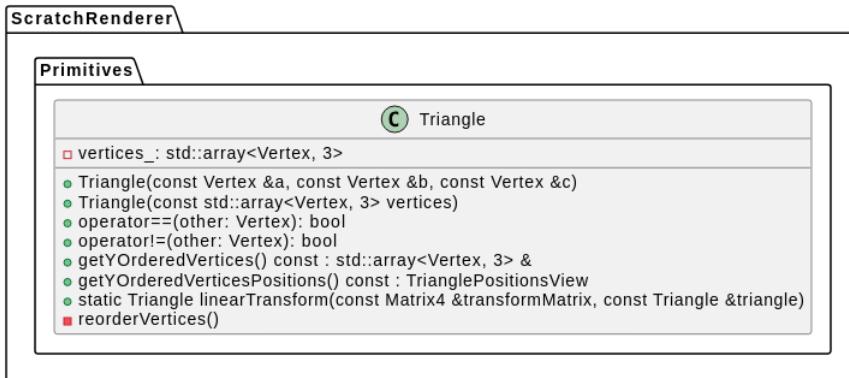


Рис. 11: Класс Triangle

5.6 Object

Object — класс трёхмерного объекта. Объект строится по набору треугольников. Программно можно вызывать методы для перемещения и поворота объекта, а также получить по объекту набор треугольников, из которых он состоит.

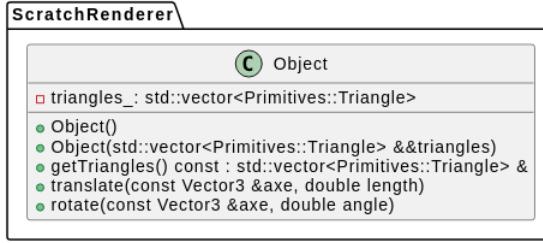


Рис. 12: Класс Object

5.7 World

World — класс мира, который представляет собой контейнер, хранящий в себе все объекты сцены. Позволяет добавить объект в сцену, а также проитерироваться по всем треугольникам всех объектов (так как рендерер всегда отрисовывает не целые объекты, а треугольники, то удобнее итерироваться сразу по ним, а не по объектам, и после этого по треугольникам каждого объекта).

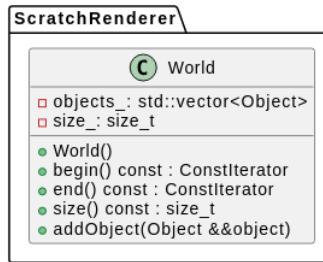


Рис. 13: Класс World

5.8 Camera

Camera — класс, представляющий абстракцию камеры. Он отвечает за текущее положение и поворот камеры, позволяет перемещать и вращать её, а также производит все операции по преобразованию объектов мира: перенос в пространство камеры, клиппинг, проективное преобразование усечённой пирамиды зрения в параллелепипед, сдвиг и масштабирование параллелепипеда. Для удобства сдвиг и поворот камеры хранятся сразу в виде матриц, которые нужно применять к вершинам объектов; а три матрицы проективного преобразования, сдвига и масштабирования параллелепипеда, поскольку они не меняются на протяжении всей работы приложения, создаются и перемножаются один раз при создании камеры, после чего к вершинам объектов применяется одна матрица (projectionMatrix_).

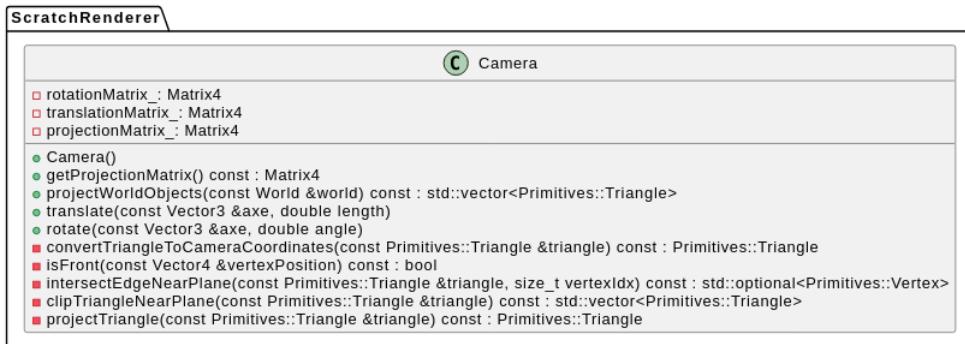


Рис. 14: Класс Camera

5.9 Screen

Screen — класс экрана, хранит ширину, высоту экрана и пиксели для отрисовки в виде массива sf::Vertex и выполняет преобразование вершин в пиксели экрана.

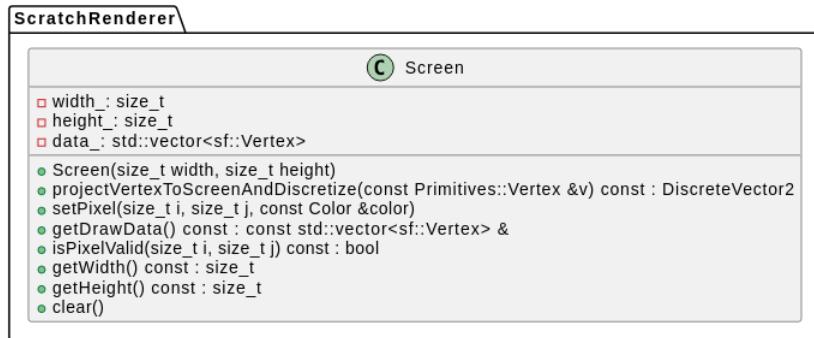


Рис. 15: Класс Screen

5.10 Loader

Loader — класс, который осуществляет парсинг файла .off-формата, и по нему создаёт и возвращает объект, который затем можно добавить в мир. Формат .off-файла [2]:

```
Line 1 (optional)
OFF
Line 2
vertex_count face_count edge_count
One line for each vertex:
x y z
for vertex 0, 1, ..., vertex_count-1
One line for each polygonal face:
n v1 v2 ... vn,
the number of vertices, and the vertex indices for each face.
```

В файле могут присутствовать дополнительные пустые строки и комментарии, которые начинаются с символа # и продолжаются до конца текущей строки.

Также .off-формат поддерживает (опционально) цвета: их можно задавать либо отдельным вершинам, либо целым поверхностям (что-то одно). В этом случае три (RGB) или четыре (RGBA) координаты цвета идут в конце строки после объявления координат вершины или поверхности соответственно. При этом координаты цвета могут быть заданы либо целочисленно в диапазоне от 0 до 255, либо дробным числом от 0 до 1 (в этом случае целочисленную координату можно получить, умножив на 255 и взяв целую часть). В случае цвета, заданного целой поверхности, загрузчик просто задаёт одинаковый цвет всем вершинам поверхности.

Поскольку поверхности могут состоять более, чем из трёх вершин, при обработке поверхности из n вершин загрузчик триангулирует её и добавляет в объект $n - 2$ треугольника.

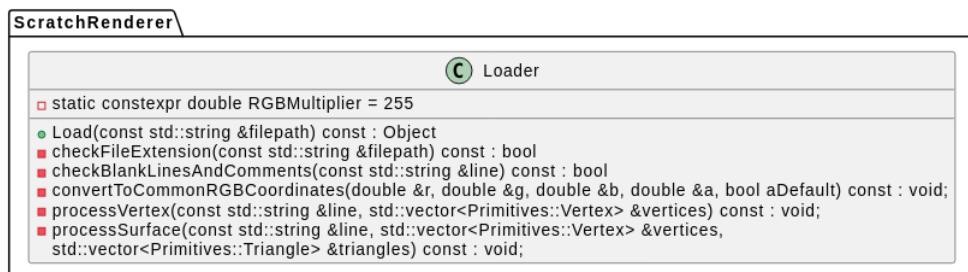


Рис. 16: Класс Loader

5.11 Storage2d

Storage2d — вспомогательный класс для работы с двумерной таблицей из некоторых объектов типа T , при этом реально храня внутри не двумерный, а одномерный вектор для оптимизации времени и памяти. Используется для реализации z -буфера.

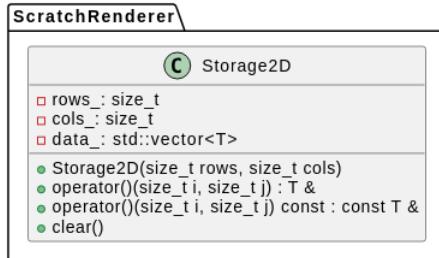


Рис. 17: Класс Storage2d

5.12 Renderer

Renderer — класс, который объединяет все предыдущие и отвечает за пайплайн преобразования трёхмерных объектов в двумерное изображение, а также реализует логику отрисовки. При создании рендерера, он инициализирует внутри себя объекты экрана и z -буфера. Логика работы с рендерером после инициализации следующая: он принимает объекты мира, камеры, а также окно приложения (sf::RenderWindow), куда производится отрисовка. Далее рендерер отдаёт камере мир и получает от неё набор преобразованных треугольников, итерируется по ним, и для каждого треугольника методом сканирующей прямой при необходимости обновляет z -буфер. После обработки всех треугольников полученные пиксели из z -буфера переносятся на экран и отрисовываются на окно, а сами z -буфер и экран очищаются для последующей отрисовки новых кадров.

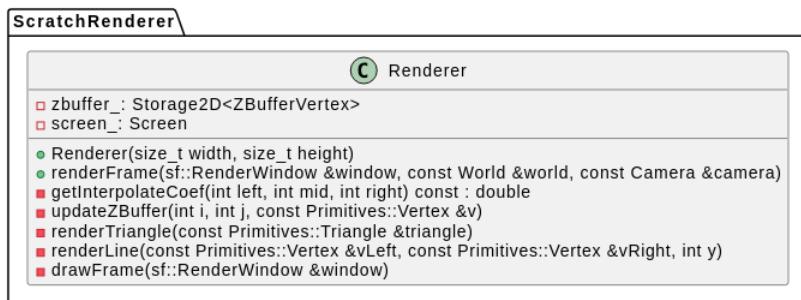


Рис. 18: Класс Renderer

5.13 Application

Application — класс приложения, реализованный с помощью библиотеки SFML. Он отвечает за создание окна приложения и его запуск, обработку нажатий на клавиатуру и обновление картинки. Логика работы с классом следующая: сначала добавляются все объекты сцены, после чего вызывается метод run(), который запускает бесконечный цикл до закрытия окна, в котором с помощью рендерера отрисовывает текущий кадр на экран, а также проверяет, прошли ли какие-то нажатия на клавиатуру, и если да, обновляет положение/поворот камеры.

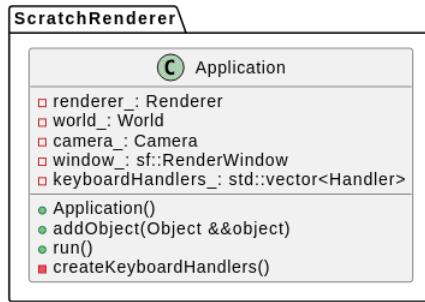


Рис. 19: Класс Application

5.14 Заголовочные файлы global_usings.h и configuraton.h

Заголовочные файлы global_usings.h и configuratotn.h не определяют новых классов. Файл configuration.h хранит ряд заданных вручную параметров, используемых в проекте, таких как высота и ширина окна приложения, название приложения, название поддерживаемого расширения для загрузчика, расстояния до ближней и дальней плоскости, горизонтальный угол обзора камеры, скорость перемещения и поворота камеры. Файл global_usings.h хранит общеспользуемые в проекте обозначения классов и константных выражений, такие как Matrix4, Vector3, Color, Epsilon и т.п.

5.15 Общая диаграмма классов с зависимостями

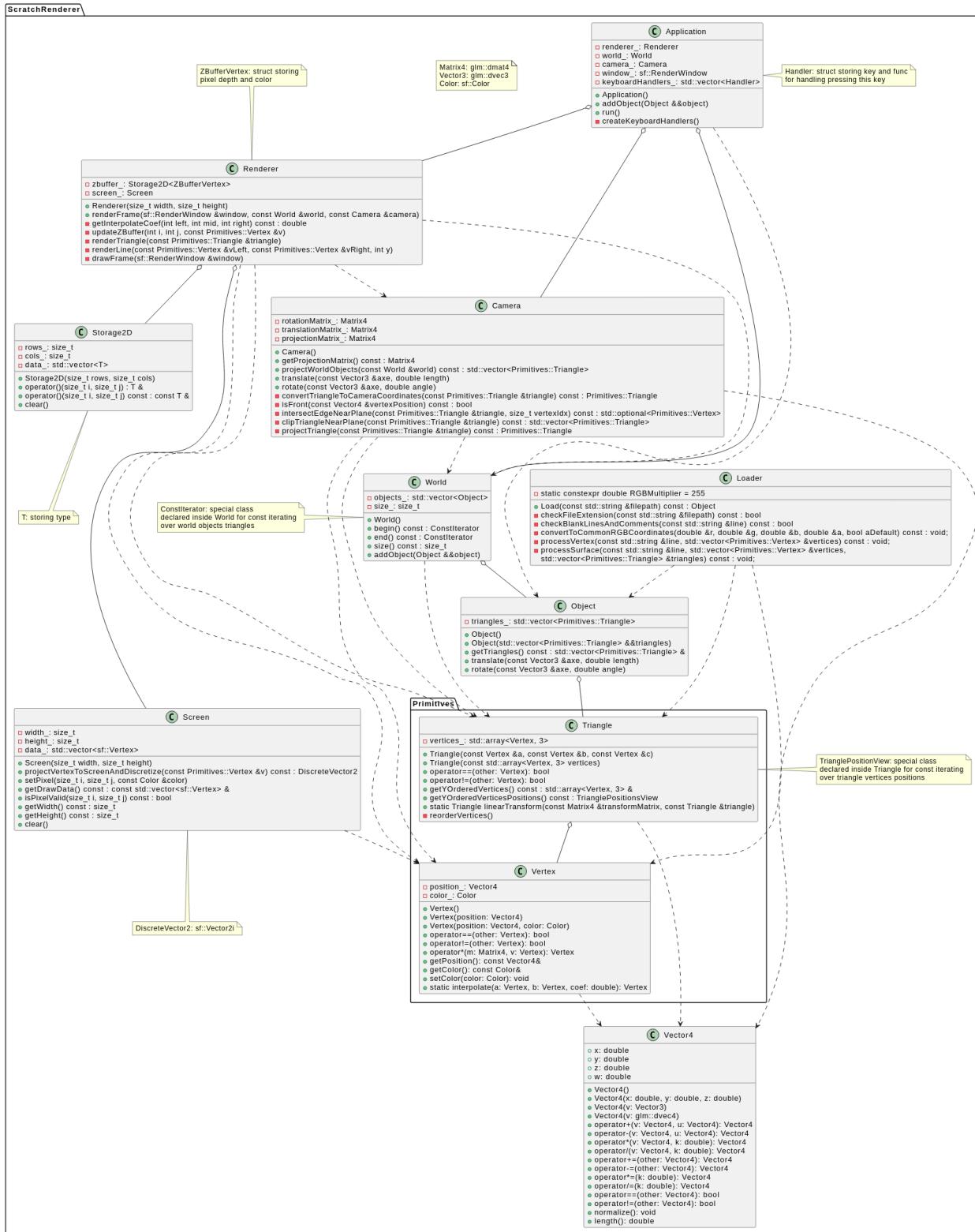


Рис. 20: Диаграмма классов. Обычными линиями обозначены классы, хранящиеся внутри другого как поля (отношение часть-целое), пунктирными — зависимые классы, один используется в функциях или методах другого, но не хранится

6 Тестирование

6.1 Unit-тестирование

Для проверки корректной работы отдельных модулей проекта были реализован и запущен ряд Unit-тестов с использованием фреймворка googletest [1], который позволяет упростить написание тестов и автоматизировать тестирование. Были реализованы тесты для следующих классов:

- **Vector4:** проверка корректности математических операций с четырёхмерными векторами с гомогенной координатой.
- **Primitives::Vertex:** проверка корректного создания вершины, умножения на матрицу, интерполяции.
- **Primitives::Triangle:** проверка корректного создания треугольника и получения его вершин, применения матрицы линейного преобразования.
- **Object:** проверка корректного создания, поворота и параллельного переноса объекта, получения треугольников объекта.
- **World:** проверка корректной итерации по треугольникам всех объектов в мире.
- **Storage2D:** проверка корректного создания двумерного контейнера, обращения к его элементам, очистки.
- **Screen:** проверка корректного проецирования вершины на экран, установки пикселей экрана, очистки экрана.
- **Camera:** проверка корректности преобразований треугольников мира: переноса в пространство камеры, клиппинга треугольников, проекции, масштабирования в куб.

Все запущенные тесты отрабатывают успешно, что говорит о верной реализации отдельных частей рендерера. Юнит-тесты также полезны при дальнейшей работе над проектом, например, при добавлении нового функционала, поскольку позволяют быстро выявить возможные баги в уже существующем коде и убедиться в том, что новые изменения не нарушают работу существующих компонентов.

6.2 Ручное тестирование

Для проверки работы рендерера как цельного проекта было использовано написанное тестовое приложение и опробовано несколько различных сцен. Одна из сцен была создана вручную, остальные были загружены из .off-файлов. Кроме того, на одну из сцен было добавлено сразу несколько объектов (кубов). С помощью ручного тестирования удалось убедиться в том, что рендерер, тестовое приложение и загрузчик .off-файлов работают, как должны: отрисовывается ожидаемая картинка (в том числе для достаточно сложных объектов с большим числом вершин и поверхностей), камера корректно поворачивается и двигается при нажатиях на клавиатуру, правильно отображаются объекты при выходе за границу экрана, верно интерполируется цвет, успешно загружаются объекты из .off-файлов.

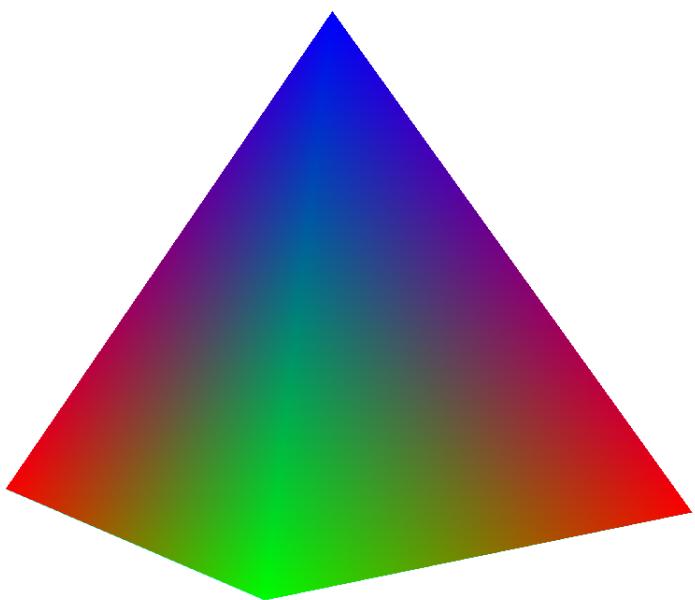


Рис. 21: Пирамида с гранями, залитыми градиентом

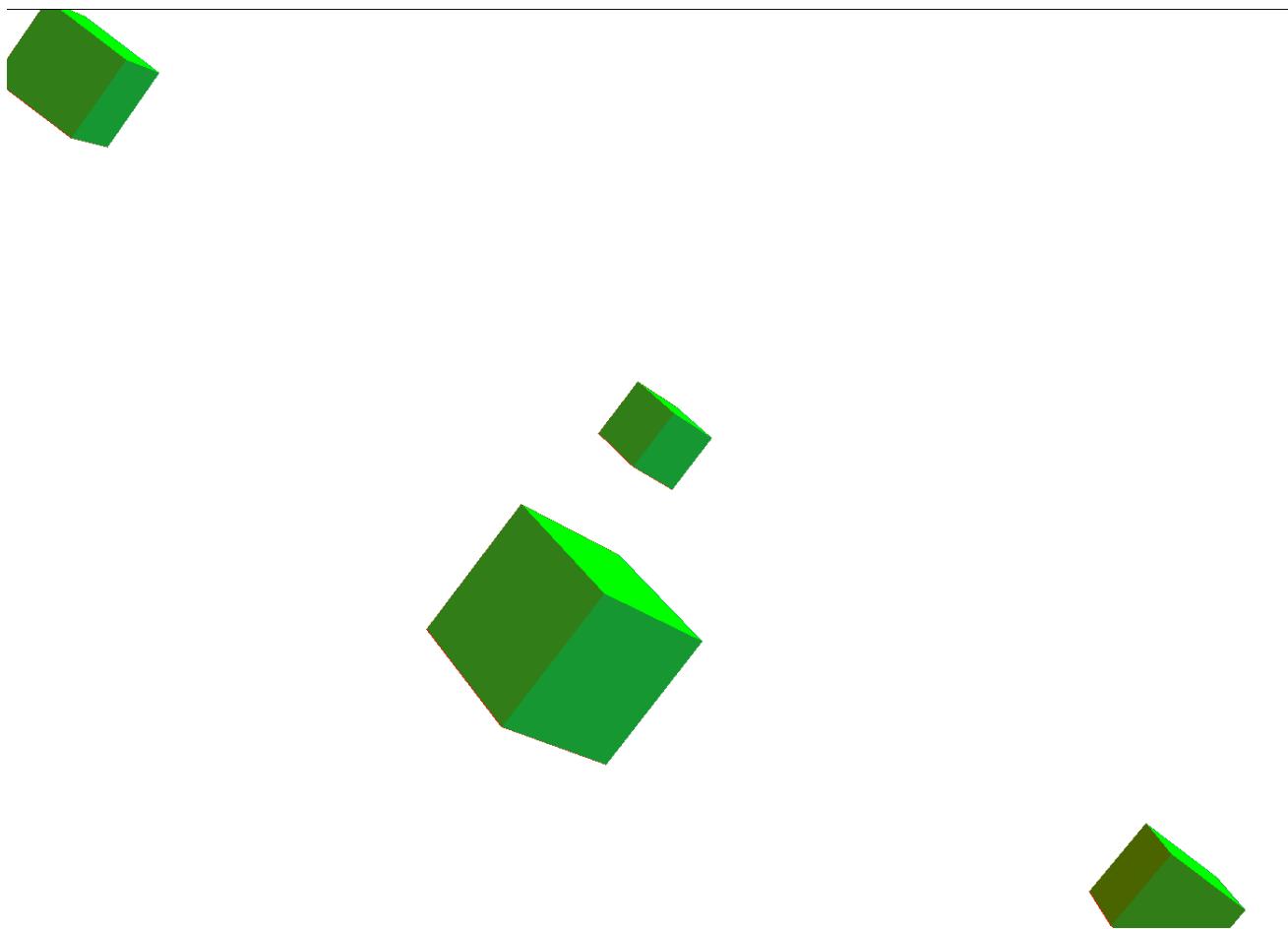


Рис. 22: Несколько кубов



Рис. 23: Модель, загруженная из .off-файла: гриб

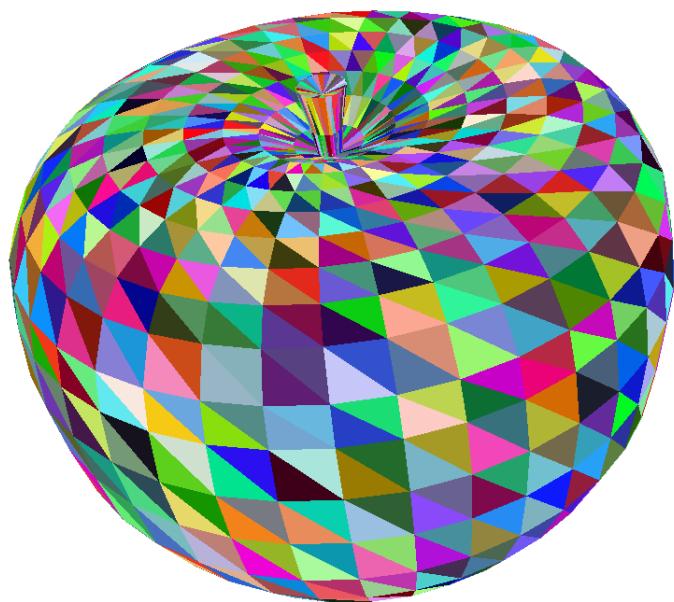


Рис. 24: Модель, загруженная из .off-файла: яблоко

7 Заключение

В рамках работы над проектом получены следующие результаты:

1. Создана библиотека для рендеринга 3D-сцен с открытым исходным кодом, выложенным на [Github](#).
2. Разработано тестовое приложение с возможностью управления камерой с клавиатуры для демонстрации работы рендерера.
3. Имплементирован загрузчик из файлов OFF-формата для удобного добавления 3D-объектов в сцену.
4. Библиотека протестирована как с использованием юнит-тестов, так и с использованием ручного тестирования (создания/загрузки из .off-файлов 3D-объектов и интерактивной работы с ними с помощью тестового приложения).

Несмотря на достигнутые результаты, есть также и потенциал для дальнейших улучшений и развития проекта. В частности, можно работать над оптимизацией алгоритмов рендеринга для повышения производительности, а также добавлением дополнительных эффектов и функциональностей для расширения возможностей рендерера, таких как текстурирование, освещение и др.

Список литературы

- [1] Goolgeletest c++ testing and mocking framework. 05.05.2024. URL: <https://github.com/google/googletest>.
- [2] John Burkardt. Off files. geomview object file format. 05.05.2024. URL: <https://people.sc.fsu.edu/~jburkardt/data/off/off.html>.
- [3] Marshall Burns. The stl format. standard data format for fabbers. 05.05.2024. URL: http://www.fabbers.com/tech/STL_Format.
- [4] Samuel R. Buss. *3D Computer Graphics, A Mathematical Introduction with OpenGL*. Cambridge University Press, University of California, San Diego, 2003.
- [5] Flávio Coutinho. Appendix b1. object files (.obj), advanced visualizer manual. 05.05.2024. URL: <https://fegemo.github.io/cefet-cg/attachments/obj-spec.pdf>.
- [6] Eigen api documentation. 05.05.2024. URL: <https://eigen.tuxfamily.org/dox/>.
- [7] Glm api documentation. 05.05.2024. URL: <https://glm.g-truc.net/0.9.9/api/index.html>.
- [8] Khronos Group. Opengl api documentation. 05.05.2024. URL: <https://www.opengl.org/Documentation/Specs.html>.
- [9] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics, Third Edition*. Course Technology PTR, 2012.
- [10] Qt documentation. 05.05.2024. URL: <https://doc.qt.io/>.
- [11] Sfml documentation. 05.05.2024. URL: <https://www.sfml-dev.org/documentation/2.6.1/>.
- [12] Greg Turk. The ply polygon file format. 05.05.2024. URL: <https://web.archive.org/web/20161204152348/http://www.dcs.ed.ac.uk/teaching/cs4/www/graphics/Web/ply.html>.