ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте

на тему: 3D renderer с нуля

Выполнил:
Студент группы БПМИ193 _____ И.Н.Притуляк
Подпись И.О.Фамилия

02.06.2021
Дата

Принял:
Руководитель проекта Дмитрий Витальевич Трушин
Имя, Отчество, Фамилия

доцент, к.ф.-м.н.
Должность, ученое звание

ФКН НИУ ВШЭ
Место работы (Компания или подразделение НИУ ВШЭ)

Дата проверки _____ 2021 _____ _____
Оценка (по 10-ти бальной шкале) Подпись

Москва 2021

Содержание

Аннотация

Цель данной работы — реализовать 3D-renderer с открытым исходным кодом. Реализация будет происходить с нуля, без использования низкоуровневых библиотек. Одной из основных целей также является изучить, как процесс рендеринга происходит снизу.

1 Введение

В приложениях, использующих 3D графику, часто стоит задача отображения объектов из виртуального 3D мира на 2D экран. Процесс преобразования трёхмерной модели в двумерную картинку, которая впоследствии отображается на экране монитора, называется 3D-рендерингом. Целью данного проекта является полностью с нуля реализовать программу, которая могла бы решать эту задачу.

В процессе реализации стоит несколько задач:

1. Изучение принципов работы 3D-рендерера посредством изучения литературы, посвящённой данной теме.
2. Краткое изложение необходимой теории.
3. Написание на языке C++ библиотеки, позволяющей проецировать сцену на экран монитора, и приложения, демонстрирующего её работу.
4. Описание архитектуры и дизайна реализации.
5. Тестирование написанного кода и приведение его результатов.
6. Написание сопроводительной документации, которая включает в себя инструкцию по работе с библиотекой и описание основных методов.

Работа состоит из нескольких частей. В первой части я рассказываю теорию, которая потребуется для реализации библиотеки, а также привожу алгоритмы, которые в ней используются. Дальше идёт архитектура библиотеки. Я рассказываю о том, какие классы она содержит и как эти классы взаимодействуют между собой. Также в этой главе более подробно задевается алгоритм работы с библиотекой. В следующей главе я поверхностно рассказываю реализацию основных методов, оцениваю их сложность. В этой же главе обсуждается работа с таймером, подсчёт FPS и отрисовка каркаса. В пятой главе прилагается инструкция по работе с библиотекой: что нужно настроить, и какие методы за это отвечают. В заключение работы я привожу результаты тестирования на двух сценах с разными параметрами — какой FPS вырабатывает рендерер и почему так происходит.

Большинство теоретического материала взято из книг [2] и [3], а также из интернета, а именно: процесс рендеринга [3, Chapter 1], однородные координаты [3, Chapter 4.4], барицентрические координаты [2, Chapter IV.1.3], пирамида зрения [3, Chapter 5.3], перемещение и вращение объектов [3, Chapter 4.3], видовое преобразование [1], перспективное преобразование [3, Chapter 5.5.1], растеризация [2, Chapter II.4]. Более подробно об этих вещах можно почитать по соответствующим сноскам. Часть картинок также взяты из этих источников.

Репозиторий проекта располагается по следующей ссылке: https://github.com/stabmind/3D-renderer.

1.1 Функциональные требования

Библиотека должна предоставлять пользователю следующие возможности:

- Отрисовка триангулированной 3D модели на экране монитора.
- Создание треугольника, который определяется тремя вершинами, заданными в любом порядке. Каждой вершине также соответствует какой-то цвет.
- Добавление треугольника или множества треугольников в мир.
- Задание параметров камеры, а также характеристик пирамиды зрения.
- Задание параметров окна, в котором будет отображаться проекция.

- Возможность повернуть камеру на любой угол относительно любого вектора, в том числе каждой оси.
- Возможность запустить интерактивный режим, в котором при помощи клавиатуры можно вращать и перемещать камеру, тем самым перемещаясь по миру.
- Возможность запустить динамический режим, в котором камера вращается по окружности вокруг центра.
- Возможность включить отображение количества кадров в секунду, которые генерирует рендерер.
- Возможность включить отрисовку каркаса, цвет которого также можно настроить.

Более подробное объяснение того, какие конкретно параметры нужно настроить и каковы функции описанных объектов, будет объяснено далее.

1.2 Нефункциональные требования

1. Требования к надёжности. Если метод требует некоторый формат входных данных, например неотрицательные значения параметров окна, и пользовательские данные ему не соответствуют, то программа возвращает соответствующее исключение.
2. Требования к программному обеспечению.
   - Язык программирования: C++ с поддержкой стандарта C++20.
   - Компилятор: g++ 10.3.1.
   - Используемые библиотеки: SFML [1] и Eigen [2] — для взаимодействия с пользователем (ввод с клавиатуры, вывод готового изображения на экран монитора) и работы с матрицами соответственно. Из первой библиотеки также используется таймер: для подсчёта FPS и при реализации движения камеры.
   - Система контроля версий: Git.
   - Система сборки: CMake версии 3.7.2 или более новой.
3. Требования к оформлению кода.
   - Style guide: Google [3].
   - Программа по форматированию в соответствии со style guide: Clang.
4. Системные требования. Для использования библиотеки достаточно, чтобы операционная система нормально функционировала.

[1] https://www.sfml-dev.org
[2] https://eigen.tuxfamily.org
[3] https://google.github.io/styleguide/cppguide.html

2 Теория

Прежде чем разобраться с архитектурой библиотеки, кратко изучим теорию, которая за ней лежит. Весь процесс делится на два этапа: настройка рендерера и обработка рендерером поставленной сцены, результатом которой будет изображение на экране монитора. Настройка рендерера будет рассмотрена в главе 5, она подразумевает постановку сцены, то есть задание триангулированной модели посредством такого примитива как треугольник, и задание камеры, при помощи которой будет происходить рендеринг модели на экран монитора. Сам рендеринг происходит в несколько этапов: модельное преобразование, видовое преобразование, клиппинг (*clipping*), перспективное преобразование и растеризация (рис. 1). В основе рендеринга лежит *z-буферизация* — способ отрисовки изображения в зависимости от расстояния до экрана, за которое отвечает z координата.

На протяжении всей работы нам потребуются следующие системы координат:

1. Локальная система координат — система координат, индивидуальная для каждого объекта
2. Глобальная система координат — система координат, в которой располагается сцена.
3. Система координат относительно камеры — система координат, в которой происходит перспективное преобразование.
4. Homogeneous clip space — система координат, которая получается перспективным преобразованием.

Все преобразования между этими системами координат происходят в пространстве $\mathbb{R}^4$ при помощи однородных координат посредством следующих матриц:

1. Model matrix — преобразование из локальной системы координат в глобальную.
2. View matrix — преобразование из глобальной системы координат в систему координат относительно камеры.
3. Projection matrix — преобразование из системы координат относительно камеры в homogeneous clip space.

Переходы между системами координат идут в описанном выше порядке. Однако прежде чем произвести перспективное преобразование происходит клиппинг — процедура, когда удаляется часть модели, которая не попадает на экран. После перспективного преобразования мы оказываемся в Homogeneous clip space, в этом пространстве все точки имеют нормализованные координаты, теперь их удобно проецировать на экран. Итоговые треугольники проходят процесс растеризации, а затем готовое изображение отображается на экране монитора.

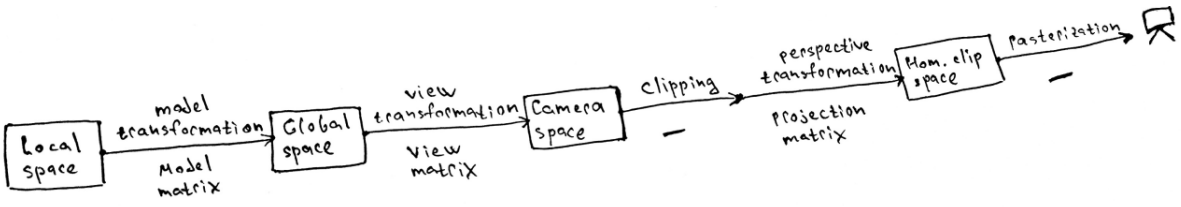

Рис. 1: Pipeline. Там, где происходят преобразования из одной системы координат в другую, сверху подписано преобразование, а снизу — матрица при помощи которой оно происходит.

2.1 Основные понятия

2.1.1 Однородные координаты

Перед рассмотрением процесса рендеринга удобно сначала обсудить, что такое однородные координаты и зачем они нужны. Однородные координаты удобны тем, что позволяют произвести поворот и смещение точки относительно какого-то вектора одним только домножением на соответствующую матрицу. При этом, если мы рассматриваем не точку, а вектор, то для него то же домножение будет давать только лишь поворот. Однородные координаты также удобны при перспективном преобразовании, обсуждение которого идёт дальше.

В $\mathbb{R}^3$ мы различаем точки и векторы, поэтому сопоставим им соответствующие однородные координаты:

1. Пусть $P = (P_x, P_y, P_z)$ — точка в $\mathbb{R}^3$. Тогда $\widetilde{P} = (P_x, P_y, P_z, 1)$ — её однородные координаты в $\mathbb{R}^4$.

2. Пусть $\vec{V} = (V_x, V_y, V_z)$ — вектор в $\mathbb{R}^3$. Тогда $\widetilde{V} = (V_x, V_y, V_z, 0)$ — его однородные координаты в $\mathbb{R}^4$.

В $\mathbb{R}^4$ мы различаем только вектора. Пусть $\widetilde{C} = (C_x, C_y, C_z, w)$ — вектор в $\mathbb{R}^4$. Если w равняется нулю, то $\widetilde{C}$ соответствует вектор в $\mathbb{R}^3$ равный $\vec{V} = (C_x, C_y, C_z)$. В противном случае ему соответствует точка равная $P = (\frac{C_x}{w}, \frac{C_y}{w}, \frac{C_z}{w})$. Отсюда также следует, что векторам $\widetilde{C}$ и $\lambda\widetilde{C}$ соответствует одна точка в $\mathbb{R}^3$, если только λ и w не равны нулю.

2.1.2 Барицентрические координаты

На этапе удаления частей треугольника, которые не попадают на экран, потребуются барицентрические координаты. Пусть x, y и z неколлинеарные вершины треугольника. Пусть также вершина u лежит внутри или на границе треугольника. Тогда её можно представить единственным образом в виде $u = \alpha x + \beta y + \gamma z$, где $\alpha + \beta + \gamma = 1$, и (α, β, γ) — её барицентрические координаты.

Пусть вершина u разбивает треугольник на три области: A, B и C (рис. 2). Тогда можно посчитать её барицентрические координаты следующим образом:

$$\alpha = \frac{A}{A + B + C}$$
$$\beta = \frac{B}{A + B + C}$$
$$\gamma = \frac{C}{A + B + C}$$

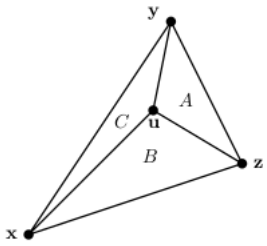

Рис. 2: Разбиение треугольника для вычисления барицентрических координат вершины u.

Зная барицентрические координаты, можно интерполировать функцию f на вершину u:

$$f(u) = \alpha f(x) + \beta f(y) + \gamma f(z)$$

2.1.3 Пирамида зрения

Обсудим, что представляет из себя камера и что такое пирамида зрения. В локальной системе координат камера представляет из себя точку в начале отсчёта и имеет направление, задаваемое некоторым вектором. На расстоянии n от этой точки вдоль направляющего вектора в перпендикулярной ему плоскости располагается прямоугольный экран, на который впоследствии будет производиться проекция модели. Параллельно экрану на бóльшем расстоянии f аналогично проходит ещё одна плоскость. Вместе с тем из центра координатной оси через стороны экрана область ограничивают ещё четыре плоскости. Часть модели, которая попала в эту область, будет проецироваться на экран. Конструкция же называется *пирамидой зрения* (рис. 3).

2.2 Перемещение и вращение объектов

Чтобы понять как перемещать и вращать объекты, достаточно научиться делать то же самое для точек. Как сместить точку на некоторый вектор очевидно — нужно просто прибавить к точке требуемый вектор смещения. С вращением же всё немного сложнее. Мы хотим повернуть точку v на угол θ относительно

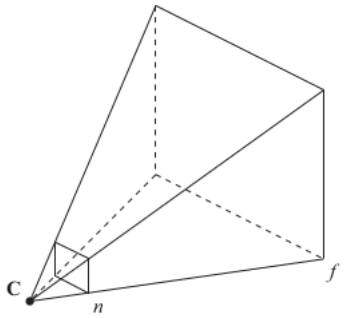

Рис. 3: Пирамида зрения.

единичного вектора $\vec{u}$ по часовой стрелке (рис. 4). Для этого достаточно сделать домножение на матрицу $R_u(\theta)$ следующего вида:

$$R_u(\theta) = \begin{pmatrix} \cos\theta + (1-\cos\theta)u_x^2 & (1-\cos\theta)u_x u_y - \sin\theta u_z & (1-\cos\theta)u_x u_z + \sin\theta u_y \\ (1-\cos\theta)u_x u_y + \sin\theta u_z & \cos\theta + (1-\cos\theta)u_y^2 & (1-\cos\theta)u_y u_z - \sin\theta u_x \\ (1-\cos\theta)u_x u_z - \sin\theta u_y & (1-\cos\theta)u_y u_z + \sin\theta u_x & \cos\theta + (1-\cos\theta)u_z^2 \end{pmatrix}$$

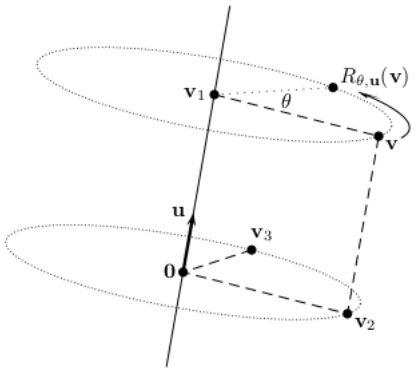

Рис. 4: Поворот точки v относительно единичного вектора $\vec{u}$ на угол θ по часовой стрелке.

2.3 Модельное преобразование

Чтобы перевести объект из локальной системы координат в глобальную, используется модельное преобразование. И хотя непосредственно в реализации оно не используется, его полезно рассмотреть для полноты изложения. Объекты хранятся в локальной системе координат и при помощи модельной матрицы каждый объект переходит в глобальную систему координат. Данное преобразование происходит при помощи смещения объекта на некоторый вектор. Если $\vec{h} = (h_x, h_y, h_z)$ — вектор смещения, то матрица преобразования M_{model} примет следующий вид:

$$M_{\text{model}} = \begin{pmatrix} 1 & 0 & 0 & h_x \\ 0 & 1 & 0 & h_y \\ 0 & 0 & 1 & h_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2.4 Видовое преобразование

Когда мы научились переводить объекты из локальной системы координат в глобальную, следующим преобразованием будет видовое. Данное преобразование переводит объекты из глобальной системы координат в систему координат относительно камеры. Пусть камера имеет направление $\vec{v_c}$, и задан некоторый неколлинеарный вектор $\vec{u}$. При помощи этих двух векторов зададим новую систему координат, базисы которой

образуют левую тройку: ось z направлена противоположно $\vec{v}_c$, ось x смотрит вправо, а y — вверх. Пусть векторы e'_1, e'_2 и e'_3 задают новые оси x, y и z соответственно. Тогда пример такой системы будет изображён на рисунке 5.

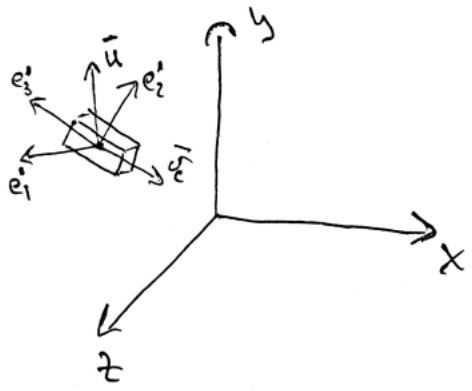

Рис. 5: Пример новой системы координат относительно камеры.
Векторы e'_1, e'_2 и e'_3 задают новые оси x, y и z соответственно.

Новый базис будет иметь следующий вид:

$$e'_1 = \frac{\vec{v}_c \times \vec{u}}{||\vec{v}_c \times \vec{u}||}$$
$$e'_2 = -\frac{\vec{v}_c \times e'_1}{||\vec{v}_c \times e'_1||}$$
$$e'_3 = -\frac{\vec{v}_c}{||\vec{v}_c||}$$

Пусть $\vec{h}$ — смещение камеры. По построению матрица $C = (e'_1, e'_2, e'_3)$ — ортогональная матрица перехода из стандартного базиса в новый. Тогда матрица видового преобразования M_{view} имеет вид

$$M_{\text{view}} = \left(\begin{array}{ccc|c} & C^T & & \begin{matrix}0\\0\\0\end{matrix} \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \cdot \left(\begin{array}{ccc|c} & E & & -\vec{h} \\ \hline 0 & 0 & 0 & 1 \end{array}\right)$$

2.5 Клиппинг

Прежде чем произвести перспективное преобразование, удалим части треугольника, которые не попадают на экран. Урезать треугольник будем сначала в координатах x и z, а затем в координатах y и z. В каждом случае стоит задача клиппинга двумерного треугольника трапецией. Для удобства, не умаляя общности, переименуем координаты и будем считать, что мы находимся в координатах x и y. Тогда применим следующий алгоритм:

1. Сначала сохраним вершины треугольника, которые лежат внутри трапеции, их z координата нам уже известна.

2. Затем сохраним вершины трапеции, которые лежат внутри треугольника. Чтобы интерполировать их z координату, посчитаем при помощи векторного произведения площади, необходимые для вычисления барицентрических координат.

3. Теперь попарно пересечём стороны треугольника и стороны трапеции. Точки пересечения также интерполируем, но в этот раз считая лишь отношение длин, и сохраним их.

4. В конце, когда у нас есть вершины, которые задают выпуклый многоугольник, мы его триангулируем. Для этого зафиксируем самую левую нижнюю вершину v, отсортируем относительно неё по полярному углу остальные. Затем проведём рёбра между вершиной v и всеми остальными, а также между соседними вершинами.

Может быть такое, что полученный для триангуляции многоугольник лежит в перпендикулярной для нас плоскости. В этом случае перед триангуляцией нужно найти пару координат, в которой это условие нарушается и далее работать в ней.

В приведённом выше алгоритме возникает подзадача проверки принадлежности вершины u некоторому многоугольнику. Она решается следующим образом:

1. Переберём все тройки смежных вершин многоугольника.
2. Для каждой конкретной тройки сопоставим названия a, b и c, где вершина b находится посередине.
3. Тогда в каждом случае должно выполняться условие, что $\langle b-a, c-a\rangle$ и $\langle b-a, u-a\rangle$ не имеют противоположные знаки, то есть лежат по одну сторону от ребра (a, b), включая само ребро.

Пример возможного результата клиппинга изображён на рисунке 6.

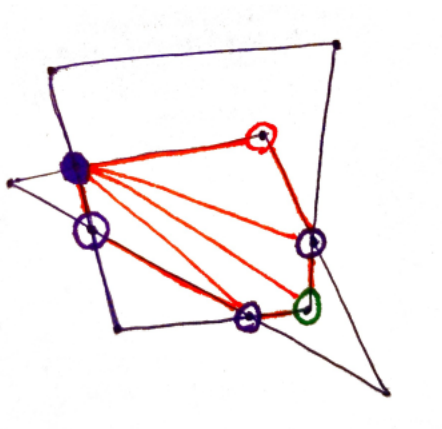

Рис. 6: Возможный результат клиппинга.
Красные вершины добавляются на первом этапе, зелёные — на втором, фиолетовые — на третьем.
На четвёртом этапе происходит триангуляция: выбирается фиолетовая закрашенная вершина и проводятся оранжевые рёбра.

2.6 Перспективное преобразование

Предпоследним этапом в процессе построения двумерной картинки является перспективное преобразование. Чтобы понять, как оно работает, рассмотрим систему координат, полученную на предыдущем этапе. Пусть левый нижний угол экрана имеет координаты $(l, b, -n)$, правый верхний — $(r, t, -n)$ (рис. 7).

Суть перспективного преобразования заключается в том, что мы преобразуем область, ограниченную пирамидой зрения, в квадрат $2 \times 2 \times 2$ с центром в начале координат (рис. 8). Полученные координаты интерпретируются следующим образом: x и y задают нормированное положение на экране, а z является глубиной, которая потом используется для отрисовки пикселей на растровом экране.

Матрицей перспективного преобразования является матрица M_{proj}, которая имеет следующий вид:

$$M_{\text{proj}} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

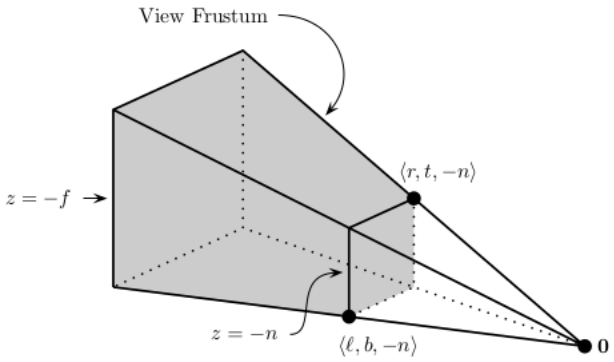

Рис. 7: Пространство камеры.

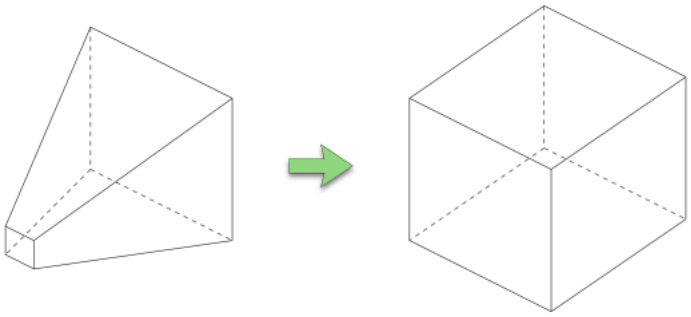

Рис. 8: Перспективное преобразование.

Отмечу одно из важных свойств, которое даёт перспективное преобразование, оно потребуется при интерполяции глубины треугольника — координата z теперь может быть интерполирована линейно, выражая при этом корректное расстояние от экрана монитора.

2.7 Растеризация

Последним этапом рендеринга является растеризация. Во время растеризации происходит переход из нормированной системы координат в окончательную картинку, которая затем будет изображена на экране. Но прежде чем растеризовать изображение, получим его координаты на экране монитора. Пусть экран имеет ширину w и высоту h. Пусть также точка $p = (x, y, z)$ принадлежит кубу, соответствующему нормированной системе координат. Тогда, применив линейное преобразование, получаем новые координаты следующим образом:

$$x' = \frac{x+1}{2}w \quad \text{и} \quad y' = \frac{y+1}{2}h$$

Округляя новые координаты вниз, получаем растровые координаты (i, j) такие, что $0 \le i < w$, $0 \le j < h$:

$$i = \min(\lfloor x' \rfloor, w-1) \quad \text{и} \quad j = \min(\lfloor y' \rfloor, h-1)$$

Теперь мы хотим научиться отрисовывать треугольники.

2.7.1 Отрисовка отрезка

Прежде чем отрисовывать весь треугольник, научимся отрисовывать отрезки. Пусть мы хотим провести отрезок между двумя точками с растровыми координатами (i_1, j_1) и (i_2, j_2). Не умаляя общности, мы считаем что $i_1 < i_2$. В силу симметрии можно считать что $j_1 < j_2$. Пусть также $\Delta i = i_2 - i_1$ и $\Delta j = j_2 - j_1$. Снова в силу симметрии считаем что $\frac{\Delta j}{\Delta i} \le 1$. Откуда вытекает следующий факт: для каждого i такого что $i_1 \le i \le i_2$ существует только один закрашенный пиксель с координатами (i, j) для некоторого j (отрезок AB на рис. 9).

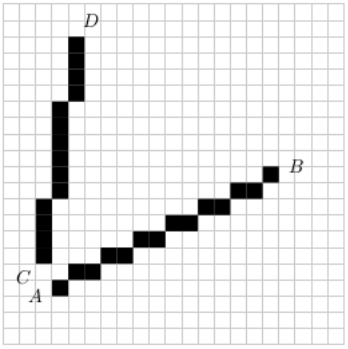

Рис. 9: Пример отрисовки прямых на растровом экране.

Пусть $\alpha = \frac{i-i_1}{i_2-i_1}$ и $j(i) = [(1-\alpha)j_1 + \alpha j_2]$. Тогда пиксели $(i, j(i))$, закрашенные для каждого i от i_1 до i_2, образуют требуемую прямую на растровом экране. Аналогичным образом делается интерполяция глубины, то есть $z(i) = [(1-\alpha)z_1 + \alpha z_2]$.

Итого получаем следующий алгоритм:

1. Проверяем условие $\left|\frac{\Delta j}{\Delta i}\right| \leq 1$. Если оно не выполняется, то меняем координаты x и y местами, отмечаем этот факт каким-нибудь флагом.
2. Проверяем условие $i_1 < i_2$. Если оно не выполняется, то меняем точки местами.
3. В цикле от i_1 до i_2 вычисляем $j(i)$ и $z(i)$ по формулам выше. Тем самым, учитывая состояние флага, мы получаем требуемые координаты и глубину пикселя на растровом экране.

2.7.2 Отрисовка треугольника

Теперь мы готовы отрисовать треугольник. Идея следующая: сначала отрисуем его стороны, а затем будем просто идти снизу вверх внутри ограничивающего его прямоугольника и отрисовывать точно таким же образом отрезки, концами которых будут точки пересечения соответствующих горизонтальных прямых с треугольником (рис. 10).

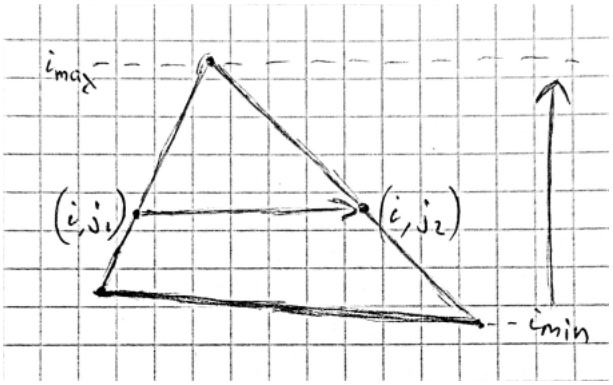

Рис. 10: Итерация отрисовки треугольника.

Для того, чтобы построить точный алгоритм, сначала определим границы цикла, отвечающего за проход строк, где идёт пересечение и отрисовка, а затем научимся определять сами точки пересечения. Пусть i_1, i_2 и i_3 — i координаты соответствующих вершин. Тогда границами внешнего цикла будут

$$i_{\min} = \min(i_1, i_2, i_3) \quad \text{и} \quad i_{\max} = \max(i_1, i_2, i_3)$$

Чтобы определить пересечение, во время отрисовки сторон запомним для каждого i точки с минимальным и максимальным j — они будут концами отрезка, который затем нужно отрисовать.

Итого получаем следующий алгоритм:

1. Отрисовываем стороны треугольника, попутно запоминая для каждой строки точки с минимальной и максимальной координатой j.
2. Идём по строкам от $i_{\min}$ до $i_{\max}$ и отрисовываем соответствующие им отрезки.

3 Архитектура

После того как мы разобрались с теорией, рассмотрим архитектуру библиотеки — из каких классов она состоит и как эти классы взаимодействуют между собой. Библиотека содержит следующие классы:

1. `Triangle`. Для обработки сцены рендерер требует, чтобы модель была триангулирована, поэтому каждая модель обязана быть представлена как композиция треугольников. Данный класс отвечает за то, чтобы представлять эти треугольники. Он предоставляет методы установки вершин и их цветов, получения вершин и их цветов, получения итераторов для обхода всех вершин.

2. `World`. Все треугольники хранятся в единой системе координат, называемой миром. Данный класс является их хранилищем. Он предоставляет методы добавления одного треугольника или нескольких треугольников сразу, а также методы получения итераторов для обхода всех треугольников в мире.

3. `Camera`. Каждый треугольник проецируется на экран при помощи камеры. Данный класс нужен, чтобы задать параметры камеры и соответствующей ей пирамиды зрения, которые нужны для составления матриц видового и перспективного преобразований. Он предоставляет методы установки данных параметров, методы поворота и смещения камеры, методы получения матриц описанных преобразований, методы получения параметров пирамиды зрения.

4. `Screen`. Проецирование модели происходит на экран монитора в отдельном окне. Данный класс используется, чтобы хранить готовое изображение, которое затем отображается на экране. Он предоставляет методы по установке размеров окна и параметров пикселей, получения размеров окна, цвета пикселя и его глубины, а также метод для очистки экрана.

5. `Renderer`. По имеющимся классам `World` и `Camera`, которые задают сцену, нужно получить итоговую картинку в классе `Screen`. Данный класс по заданным ему миру и камере производит проекцию модели, готовое изображение которой помещается в экран. Он имеет метод, который осуществляет процесс выше, а также методы активации режима отображения каркаса и установки его цвета.

6. `Application`. Чтобы связать все вышеописанные классы вместе, используется класс `Application`. Он предоставляет методы настройки мира, камеры, экрана и рендерера, метод активации отображения FPS. Также имеются методы запуска интерактивного и динамического режима.

Рассмотрим взаимодействие представленных классов. Основным из них является класс `Application`, именно при помощи него происходит настройка всех компонентов, а затем запуск требуемого режима. Этот класс содержит в себе классы `World`, `Camera`, `Screen` и `Renderer`. После их настройки вызывается один из методов: `RunInteractiveScene` или `RunRotatingScene`. Каждый из этих методов создает окно на экране монитора, в которое будет отображаться спроецированная модель, а затем запускает бесконечный цикл, который в первом случае при нажатии клавиш клавиатуры перемещает или поворачивает камеру, тем самым изменяя картинку в окне, а во втором — самостоятельно вращает камеру, тем самым создавая анимацию без участия пользователя. Для получения готового изображения на каждой итерации цикла вызывается метод `Render` класса `Renderer`, который принимает на вход классы `World`, `Camera` и `Screen`. Результат его работы сохраняется в последнем из них. Также отметим, что поскольку мир является хранилищем треугольников, то класс `World` для своей реализации непосредственно использует класс `Triangle`. На рисунке 11 изображено визуальное представление описанного выше взаимодействия.

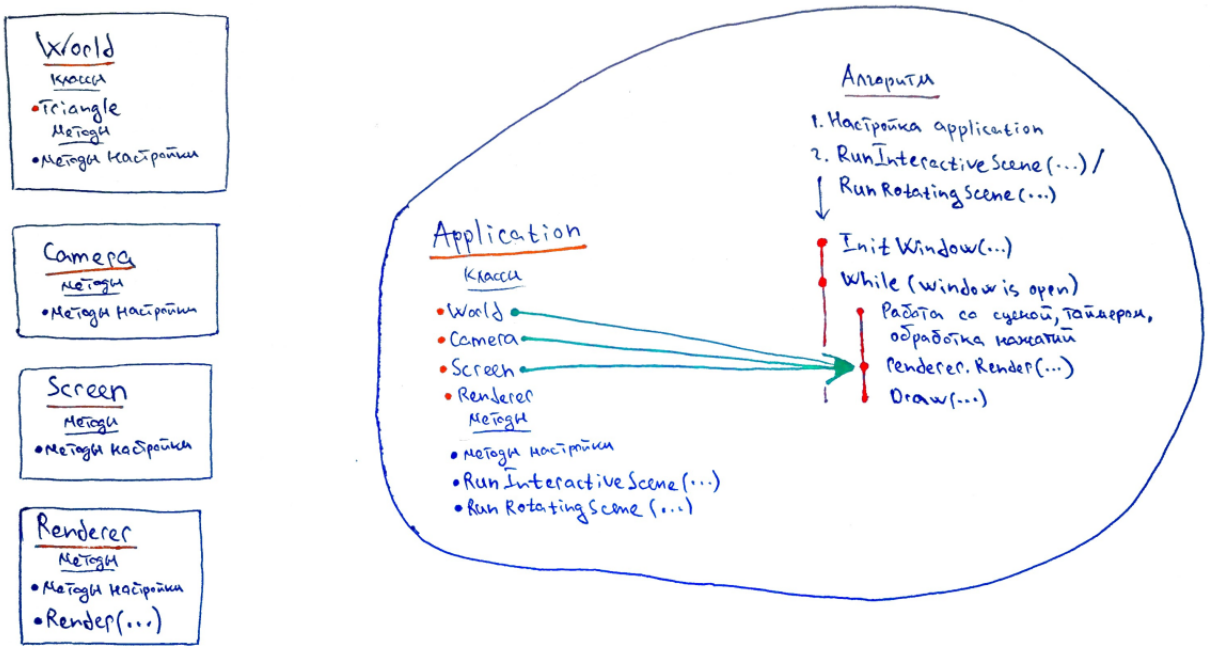

Рис. 11: Архитектура проекта. Точки оранжевого цвета отмечают классы, которые в явном виде хранит основной класс, а точки фиолетового цвета — методы, необходимые для функционирования всей системы.

4 Детали реализации и сложность алгоритмов

4.1 Метод Render

Алгоритм работы метода `Render` класса `Renderer` следующий:

1. Подготавливаем рендерер и очищаем экран. Это занимает $O(w \cdot h)$ времени.
2. Пробегаемся по треугольникам в мире, производим с каждым из них видовое преобразование, клиппинг, перспективное преобразование и растеризацию. В силу константности количества вершин у треугольника и трапеции данный процесс занимает $O(n \cdot w \cdot h)$ времени, где n — количество треугольников.

Итого получаем затраты $O(n \cdot w \cdot h)$ времени и $O(n + w \cdot h)$ памяти.

4.2 Методы RunInteractiveScene, RunRotatingScene и подсчёт FPS

План работы методов `RunInteractiveScene` и `RunRotatingScene` описан в главе 3. Здесь я уточню часть, где происходит работа со сценой, таймером и обработкой нажатий. В начале исполнения каждого из этих методов создаётся таймер, который замеряет время, уходящее на очередную итерацию цикла. Пусть очередная итерация цикла отработала за t секунд. Если активирован режим отображения FPS, то поверх сгенерированного изображения отображается округлённое количество кадров в секунду равное $\frac{1}{t}$. Если скорость движения камеры v, а скорость изменения угла u, то при необходимости движения или поворота, то есть при реакции на нажатие соответствующих клавиш, они будут произведены со скоростью $v \cdot t$ и $u \cdot t$ соответственно. Это обеспечивает одинаковую скорость движения при работе на компьютерах разной мощности.

Отдельно рассмотрим движение камеры при выполнении метода `RunRotatingScene`. На вход этот метод принимает длину радиуса r и скорость вращения v. Если поддерживать счётчик s, то на очередной итерации к нему прибавляется значение $v \cdot t$, а новые координаты камеры следующие: $x = r \sin(s)$, $y = r \cos(s)$, $z = 0$.

4.3 Отрисовка каркаса

В главе 2.7 я описал, как происходит процесс растеризации. Здесь я опишу, какие доработки вносятся, чтобы добавить отрисовку каркаса. Для этого нам потребуется ещё один двумерный массив, в котором мы будет отмечать, принадлежит ли очередной пиксель ребру треугольника. Теперь во время отрисовки рёбер вместо

их настоящего цвета будем ставить цвет каркаса, но запоминать настоящие параметры пикселя, а также отмечать принадлежность пикселя ребру. Во время же растеризации непосредственно треугольника остаётся лишь учитывать, является ли этот пиксель «свободным».

5 Инструкция по работе с библиотекой

В главе 3 описан процесс того, как работает библитека. Этого почти достаточно, чтобы уже начать ею пользоваться. Осталось понять как настраивать класс `Application`. Чтобы его настроить, необходимо задать параметры мира, камеры, экрана, рендерера и самого application. Делается это при помощи описанных ниже методов.

Методы для настройки мира:

- `void AddTriangle(const Triangle &triangle);`
  Добавляет треугольник в мир.

- `void AddTriangles(const World::TriangleVec &triangles);`
  Добавляет вектор треугольников в мир.

Методы выше требуют треугольник. Ниже приведён пример, в котором создаётся треугольник с вершинами $(1, 0, 0)$, $(0, 1, 0)$ и $(0, 0, 1)$. Им назначаются красный, зелёный и синий цвета соответственно.

```
1  Vertex v1({1, 0, 0}, {255, 0, 0});
2  Vertex v2({0, 1, 0}, {0, 255, 0});
3  Vertex v3({0, 0, 1}, {0, 0, 255});
4  Triangle triangle(v1, v2, v3);
```

Методы для настройки камеры:

- `void setCamera(double x1, double y1, double z1, double x2, double y2, double z2);`
  Задаёт координаты местоположения камеры (x_1, y_1, z_1) и её направление (x_2, y_2, z_2).

- `void setCameraPosition(double x, double y, double z);`
  Задает местоположение камеры (x, y, z).

- `void setCameraDirection(double x, double y, double z);`
  Задаёт направление камеры (x, y, z).

- `void setPivot(double x, double y, double z);`
  Устанавливает вектор $\vec{u}$ из главы 2.4 с направлением (x, y, z).

- `void setFrustum(double l, double r, double b, double t, double n, double f);`
  Устанавливает параметры пирамиды зрения из главы 2.1.3.

Методы для настройки экрана:

- `void setScreen(size_t w, size_t h);`
  Устанавливает ширину w и высоту h экрана в пикселях.

Методы для настройки рендерера:

- `void setWireframeVisible(bool is_visible);`
  Активирует отображение каркаса, если `is_vivisble` истинно, и деактивирует, если ложно. Изначально отображение каркаса отключено.

- `void setWireframeColor(const ColorType &color);`
  Устанавливает цвет с которым будет отображаться каркас. Изначально его цвет коричневый.

Методы для настройки application:

- `void setFPSVisible(bool is_visible);`
  Активирует отображение FPS, если `is_vivisble` истинно, и деактивирует, если ложно. Изначально отображение FPS отключено.

6 Тестирование

В своих тестах я использую процессор `Intel(R) Core(TM) i3-6100U CPU @ 2.30GHz`. Все результаты взяты путём запуска двух подготовленных сцен в интерактивном режиме с разными параметрами экрана. В тестировании используются модели куба и гор (рис. 12). Модель куба состоит из 12 треугольников, модель гор — из 100. Замеры FPS взяты вблизи и вдали, также зафиксировано минимальное значение, которое удалось получить. Результаты приведены в таблице 1.

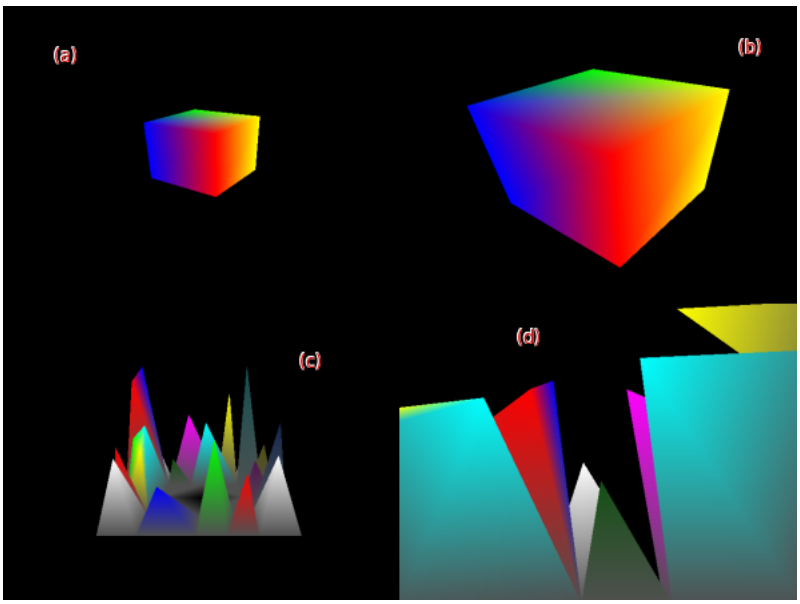

Рис. 12: Куб и горы.
На рисунках изображены куб и горы — вдали и вблизи соответственно.

	Куб			Горы		
	Вдали	Вблизи	Min	Вдали	Вблизи	Min
640 × 480	70	60	21	60	38	13
1280 × 720	28	21	11	25	14	7
1920 × 1080	10	8	5	10	6	3

Таблица 1: Результаты тестирования.

Как можно видеть, при маленьком разрешении разница FPS вдали практически не ощущается, а вблизи почти двукратная. При среднем разрешении наблюдается похожий результат, а при максимальном разница практически отсутствует. Во всех случаях при уменьшении расстояния количество FPS падает, так как увеличивается количество пикселей, которые покрывает итоговое изображение на экране монитора. Причём клиппинг хоть и влияет достаточно сильно на производительность, в данных примерах его эффект не так силён. В случае максимального разрешения мы получаем маленький FPS независимо от расстояния. Это связано с тем, что всё время уходит на инициализацию структур для работы с большим количеством пикселей. В данном случае эффект будет тем же, даже если запустить рендерер с пустой сценой.

Список литературы

[1] Getting started: Camera. URL: http://learnopengl.com/Getting-started/Camera.

[2] Samuel R. Buss. *3D Computer Graphics: A Mathematical Introduction with OpenGL*. Cambridge University Press, 2003.

[3] Eric Lengyel. *Mathematics for 3d game programming and computer graphics*. Course Technology PTR, 2012.