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

(итоговый)

Выполнил:

Студент группы БПМИ191 _____ К.В. Амеличев
Подпись И.О.Фамилия

03.06.2021
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

В рамках данной работы я разрабатываю библиотеку для языка программирования C++, занимающуюся отрисовкой объектов в трехмерном пространстве. Также для наглядной демонстрации итогового результата сделано тестовое приложение на ее основе.

1 Введение

В рамках изучения тех или иных геометрических объектов возникает необходимость их графического отображения. Сложность в визуализации трехмерных объектов в том, что их нельзя изобразить на бумаге одним рисунком без потери информации. Настоящий трехмерный объект можно повертеть в руках и разглядеть со всех сторон, а вот рисунок на бумаге или экране может быть непонятен.

В рамках данной работы создается библиотека для языка программирования C++, которая визуализирует объекты в трехмерном пространстве, требуя от пользователя минимальных усилий. Важной особенностью этой работы является то, что работа с графикой построена буквально «с нуля» — из пререквизитов нужно только средство для отрисовки пикселей на экране.

Задачами данной работы является:

1. Исследование существующих технологий и принципов в сфере 3д-графики
2. Выбор архитектуры
3. Разработка интерфейса
4. Реализация функционала
5. Создание модульных тестов для проверки работоспособности библиотеки
6. Создание тестового приложения
7. Последующее документирование и оформление в едином стиле библиотеки с открытым исходным кодом
8. Описание теоретических знаний, которые были получены и применены в рамках исследования и разработки проекта.

Весь исходный код проекта находится в репозитории по следующей ссылке

Также используются следующие источники информации:

1. [1] — сопроводительная инструкция к большому пакету для работы с 3d-графикой, в которой рассказываются основные принципы создания подобных приложений.
2. [2] — курс лекций по компьютерной графике.
3. [3] — математическая основа для библиотеки.
4. [4] — документация библиотеки SFML.
5. [5] — документация к пакету Doxygen для документирования кода
6. [6] — Документация ПО Docker, которое используется для контейнеризации приложений
7. [7] — Документация пакета для создания юнит-тестов для C++-програм.
8. [8] — Описание стандарта языка C++, стандартной библиотеки
9. [9] — Инструкция по использованию Github Actions — средства для автоматизации тестирования кода

1.1 Функциональные требования

- Отрисовка 3д-объектов.
- База стандартных объектов для отрисовки, таких как куб или тетраедр.
- Возможность создать любой триангулируемый объект в пространстве.
- Задание параметров объекта, цвет.
- Возможность последовательной смены кадров, создание анимации.
- Повороты и перемещение камеры

- Обработка взаимодействия пользователя и приложения — пользовательские повороты и перемещения камеры, в частности по взаимодействию с клавиатурой
- Возможность программно создавать и удалять объекты во время анимации.
- Скрытие интерфейса отрисовки от пользователя — пользователю нужно только создавать объекты

Структура приложения следующая: для создания одного окна с графикой требуется создать одного представителя главного класса Application. После чего Application создает окно, в котором будут отрисовываться объекты. Пользователь может добавлять объекты (такие как куб, тетраедр, или произвольный объект, созданный на основе базового класса) каждому Application'у, давать ему команды для трансформации содержимого (перемещения, переворотов) с клавиатуры. В это время Application производит все изменения и производит смену кадров.

1.2 Технические требования

- Разработка на языке C++.
- Сборка проекта с помощью CMake.
- Работа с 2д-графикой с помощью библиотеки SFML.
- Отрисовка графики на CPU.
- Система поддержки версий Git.
- Открытый исходный код в репозитории на Github.
- Юнит-тестирование с помощью CXXTest.
- Автоматическое CI/CD тестирование с помощью Github Actions.
- Документирование с помощью Doxygen.
- Ошибки программиста отлавливаются assert-ами.
- В случае возникновения системных ошибок программа завершает выполнение с ошибкой.
- Google C++ styleguide.

Тестируется библиотека с помощью приложения, в котором создаются произвольные 3d-объекты, после чего они переносятся на экран в интерактивном режиме, с возможностью перемещения и поворота.

При этом следует заметить, что в ситуации, если пользователю не подходит готовая обертка приложения, отвечающая техническим требованиям, он может создать свой класс приложения, чтобы все сложные вычисления и проекции проводились на стороне библиотеки.

2 Изучение аналогов

Поскольку в основном 3d-графика в программировании используется для создания игр, то почти все библиотеки, которые мне удалось найти, имели перегруженный интерфейс под игры, как у Polycode, где надо создавать дополнительные xml-конфигурационные файлы.

Также есть такие популярные фреймворки, как OpenGL или Direct3d, но они настолько низкоуровневые, что при написании простого приложения почти все время уйдет на их изучение.

Получается, что у низкоуровневых библиотек нужно продумывать слишком много параметров, а у высокоуровневых библиотек сложная архитектура и протокол взаимодействия пользователя с библиотекой.

Разумеется, библиотека будет проигрывать в производительности аналогам, как минимум поскольку в ней не предусмотрена отрисовка через GPU. Но важной целью работы является именно создание архитектуры отрисовщика 3d-графики с нуля.

3 Теоретическая база

3.1 Основы 3d-графики

При работе над 3d-рендерером достаточно важной частью работы является понимание математики, происходящей внутри обработчика графики. Основная проблема заключается в том, что на одни и те же объекты можно смотреть с разных сторон, и в зависимости от этого получать разную картинку. Поскольку все случаи не разберешь, на помощь приходит линейная алгебра.

3.1.1 Объекты для отрисовки

Все объекты разделяются на два типа — отрезки и треугольники. На самом деле, достаточно только треугольников, но отрезки рисовать значительно проще, при этом ими можно отображать каркасы объектов, чего часто бывает достаточно.

Каждый объект задается набором точек-вершин в глобальной системе координат, а также наборами пар и троек индексов, которые определяют отрезки и треугольники. Отрезки используются для отрисовки ребер фигуры, а поверхности триангулируются. Объекты, созданные из отрезков и треугольников, не меняют своего расположения в глобальной системе координат при передвижении камеры, а также могут двигаться независимо друг от друга.

Например, если задать пирамиду OABCD с квадратом в основании, то у нее будет 8 ребер, 5 вершин, 5 граней и 6 треугольников в триангуляции.

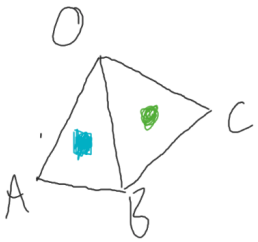

Рис. 1: Пирамида OABCD, вид сбоку

Рис. 2: Пирамида OABCD, вид снизу. Нижняя грань разбита на два треугольника

3.1.2 Экран и камера

Объекты надо как-то отображать на экране. Экран представляется произвольной плоскостью в пространстве, на который проецируются все точки. Поскольку экран имеет ограниченный размер, то и на плоскости выбирается ограниченный прямоугольник. Те точки, которые попадут на прямоугольник экрана, и будут отрисованы. Остальные находятся вне поля зрения.

Для того, чтобы правильно определить, какие объекты видно наблюдателю, используется объект камеры. Этот объект задается своим положением в пространстве ($O_c = \begin{bmatrix} x_{camera} \\ y_{camera} \\ z_{camera} \end{bmatrix}$), а также матрицей текущего пово-

рота камеры. Назовем ее M_c. Еще нам понадобятся габариты параллелипипеда, который видит наша камера — координаты ближней левой нижней и правой верхней дальней точки ((l, b, n) и (r, t, f) соответственно).

3.1.2.1 Шаг 0. Однородные координаты

Поскольку все операции с координатами имеет смысл выражать через векторы и матрицы, возникает проблема с тем, что не все афинные и проективные преобразования представляются произведением трехмерных матриц. Для этого вводят однородную систему координат, которая работает с четырехмерным пространством по следующему правилу:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \to \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}, w \neq 0 \to \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \end{pmatrix}$$

Такая система разрешает, во-первых, прибавить произвольное число к любой из координат трехмерного вектора:

$$\begin{bmatrix} 1 & 0 & 0 & a_x \\ 0 & 1 & 0 & a_y \\ 0 & 0 & 1 & a_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + a_x \\ y + a_y \\ z + a_z \\ 1 \end{bmatrix} \to \begin{bmatrix} x + a_x \\ y + a_y \\ z + a_z \end{bmatrix}$$

Также появляется возможноть разделить координаты на произвольную линейную функцию от координат:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ k_x & k_y & k_z & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ k_x x + k_y y + k_z z \end{bmatrix} \to \begin{bmatrix} \frac{x}{k_x x + k_y y + k_z z} \\ \frac{y}{k_x x + k_y y + k_z z} \\ \frac{z}{k_x x + k_y y + k_z z} \end{bmatrix}$$

Утверждается, что такого дополнительного функционала хватает, чтобы сделать необходимые нам преобразования камеры.

3.1.2.2 Шаг 1. Сдвиг камеры

Чтобы переместить камеру в центр координат, нужно из каждой координаты x, y, z вычесть ее текущее значение. Это простая единичная матрица с правым столбцом:

$$\begin{bmatrix} 1 & 0 & 0 & -x(O_c) \\ 0 & 1 & 0 & -y(O_c) \\ 0 & 0 & 1 & -z(O_c) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

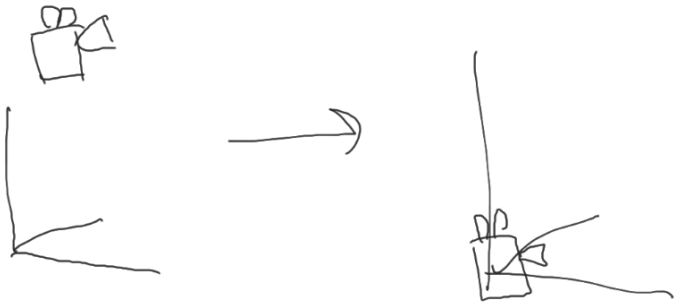

Рис. 3: Шаг 1. Камера оказывается сдвинута в центр координат.

3.1.2.3 Шаг 2. Поворот камеры

Для начала стоит понять, как происходит поворот. Если представить себе, что камера задает три ортонормированных вектора, соответствующие осям, то один из векторов остается на месте, а два другие перемещаются, причем в результате преобразований тройка остается ортонормированной. Тогда если мы хотим сделать поворот вокруг оси oX, вектор, соответствующий oX, остается на месте, в то время, когда оставшиеся два вектора поворачиваются. А именно, происходит преобразование вида $v \to \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, где φ — требуемый угол.

Почему это преобразование повернет вектор на угол φ. Мысленно возьмем проекцию на плоскость, в которой происходит поворот. Теперь разложим точку p на базисные векторы $\vec{e_1}, \vec{e_2}$. Можно заметить, что $A_\varphi e_1 = e_1 \cos\varphi + e_2 \sin\varphi, A\varphi e_2 = -e_1 \sin\varphi + e_2 \cos\varphi$.

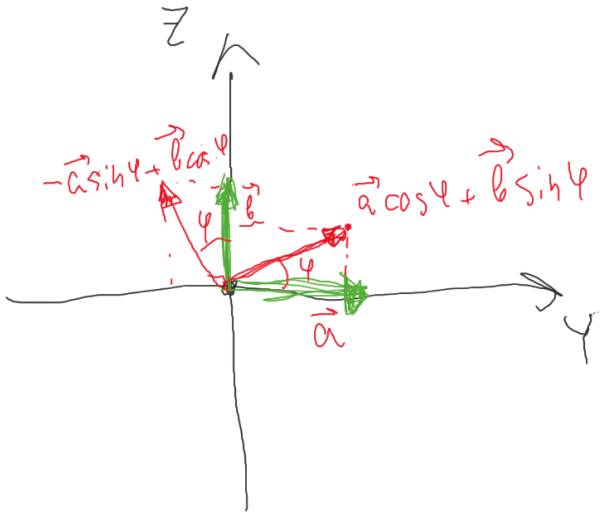

Рис. 4: Повороте базовых векторов в плоскости

Поскольку каждый происходящий поворот камеры является ортогональным преобразованием, то обратное преобразование — это просто транспонированная матрица всех поворотов. Поскольку мы хотим получить единичную матрицу, то нам надо просто транспонировать матрицу всех преобразований. А именно, если на

текущий момент базисные векторы пространства камеры после всех поворотов равны u, v, w, то надо сделать такое преобразование:

$$\begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

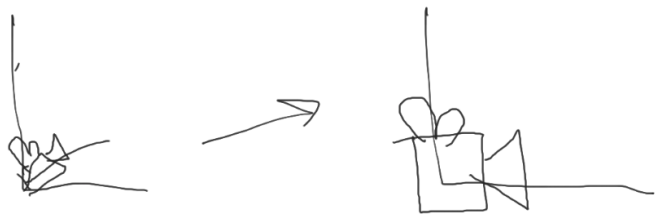

Рис. 5: Шаг 2. Теперь камера находится в центре координат, плоскость экрана — декартова система oXY, а «смотрит» камера в направлении oZ

3.1.2.4 Шаг 3. Проективное преобразование

Для того, чтобы добавить эффект перспективы, область видимости задается в виде усеченной четырехугольной пирамиды. Такой объект сложно спроецировать на экран, поэтому для начала его проективным преобразованием переводят в прямоугольный параллелипипед. После этого проекция оказывается просто отбрасыванием координаты.

У этого подхода есть еще одно приемущество. Отбрасываемая координата задает глубину точки относительно экрана. Поэтому, при прочих равных, надо будет в конкретном пикселе отрисовывать ту точку, которая имела меньшую глубину. Такой метод называется z-буфером.

Сначала покажу матрицу, а потом объясню, почему она делает то, что нам нужно.

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

Матрица проективного преобразвания переведет координаты $(x, y, z, w) \to (nx, ny, (n+f)z - nfw, z) = (\frac{nx}{z}, \frac{ny}{z}, n + f - \frac{nfw}{z})$. Можно заметить, что первые две координаты домножаются на $\frac{n}{z}$, а от преобразования третьей координаты нам будет важно только то, что оно сохранит знак неравенства. А именно, если раньше у одной точки положительная z-координата была меньше, чем у другой, то после преобразования будет больше. Это достигается, потому что от z тут зависит только знаменатель вычитаемого. В следующей части будет сказано, почему нам достаточно рассматривать только положительные точки. При этом $z = n$ переходит в $z = n$, а $z = f$ переходит в $z = f$ при $w = 1$.

А новые двумерные точки (назовем их x_0, y_0) оказываются верны (с точностью до масштаба) из подобия треугольников, образованных направлением, в котором смотрит камера, и вектором (x, y, z):

Если центр камеры лежит в точке O, плоскость экрана называется N, то $\frac{d(O,N)}{z} = \frac{d(O,(x_0,y_0,z_N))}{d(O,(x,y,z))}$. Координаты x_0, y_0 можно искать независимо в двух проекциях. Но в плоскости проекции для подобных треугольников можно посмотреть не только на катет и гипотенузу, но и на отношение двух катетов:

$$\frac{x}{z} = \frac{x_O}{z_N} \Rightarrow x_O = \frac{xz_N}{z}$$

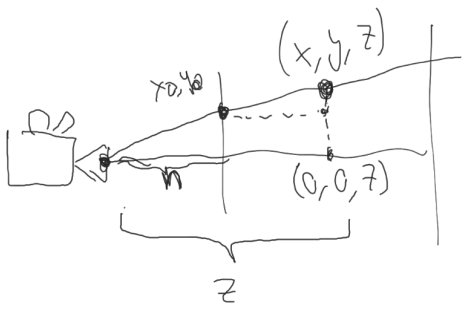

Рис. 6: Вот тут можно видеть подобие треугольников

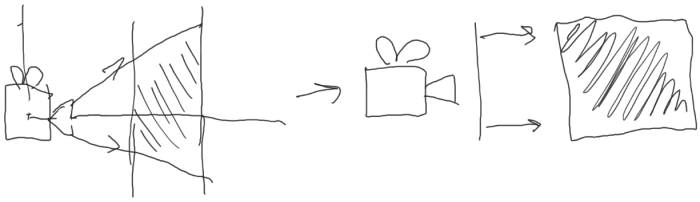

Рис. 7: Шаг 3. Теперь усеченная пирамида зрения превратилась в параллелипипед зрения

3.1.2.5 Шаг 2,5. Клиппинг

Можно заметить, что в предыдущем пункте мы делили на z. Строго говоря, деление происходило неявно, но, тем не менее, наша операция не обязана быть корректной, если z-координата была нулевой. А знак неравенства не сохранится, если z была отрицательной. Значит, надо отдельно обработать такие случаи.

Чему такой случай соответствует? Тому, что объект находится там, где мы точно его не видим, ведь мы видим только пирамиду обзора, а она находится на каком-то положительном расстоянии от камеры.

Для того, чтобы избавиться от такой проблемы, мы можем просто отфильтровать все объекты, которые оказываются слишком близко к камере. А именно, достаточно выбрать z_0 —координату меньше, чем n (потому что при z-координате равной n, проективное преобразование не сдвигает точку, поэтому она так и окажется на границе), после чего отсечь полупространство с z-координатами больше, чем z_0. Понятно, что если объект целиком попал по ту или иную сторону полупространства, то выбор бинарный — брать объект или не брать. А вот если треугольник пересекает эту плоскость, то задача немного сложнее.

Если у треугольника ABC одна точка (например, A) находится в отсекаемой зоне, то требуется найти пересечение отрезков AB и AC с искомой плоскостью (назовем получившиеся точки пересечения X и Y). Трапеция $BCYX$ должна быть отображена на экран, это можно сделать с помощью отображения треугольников BCY и BXY. Эти треугольники, в свою очередь, не конфликтуют с плоскостью отсечения, поэтому их можно дальше передавать в проективное преобразование и выводить в дальнейшем на экран.

Если у треугольника ABC две точки (например, AB) находятся в отсекаемой зоне, то требуется найти пересечение отрезков A и BC с искомой плоскостью (назовем получившиеся точки пересечения X и Y). Треугольник CYX должен быть отображен на экран. Он в свою очередь, не конфликтует с плоскостью отсечения, поэтому его можно дальше передавать в проективное преобразование и выводить в дальнейшем на экран.

Так что клиппинг надо проделать прямо перед проективным преобразованием, после чего перейти к следующим шагам.

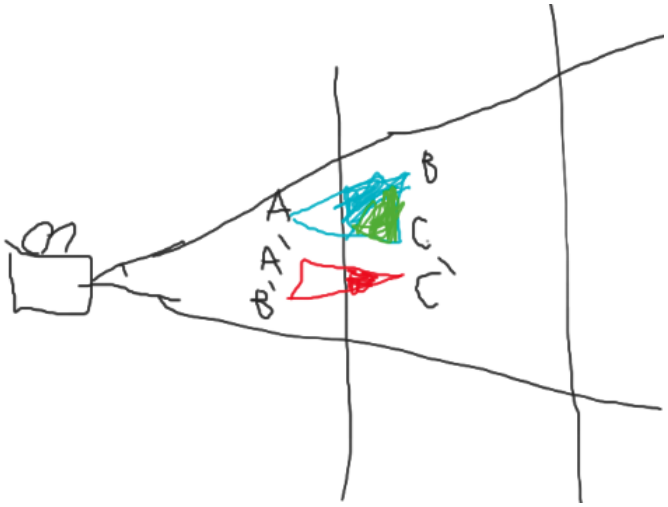

Рис. 8: Шаг 2.5. Клиппинг. Верхний треугольник разбился на два, от нижнего просто отсеклась часть.

3.1.2.6 Шаг 4. Перемещение параллелипипеда в центр координат

Чтобы было удобнее дальше работать, параллелипипед перемещается своим центром в центр координат.

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

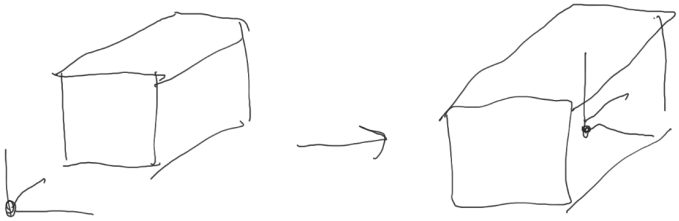

Рис. 9: Шаг 4. Теперь параллелипипед своим центром оказася в центре координат

3.1.2.7 Шаг 5. Масштабирование в куб $2 \times 2 \times 2$

Теперь объект масштабируется в куб $(-1,-1,-1),(1,1,1)$. Это нужно для того, чтобы в дальнейшем можно было работать с любым экраном.

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

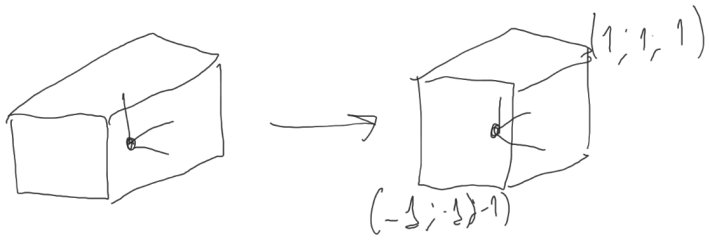

Рис. 10: Шаг 5. Масштабирование в куб

3.1.2.8 Шаг 6. Перенос в плоскость экрана

Куб надо отобразить на экране. Для этого мы масштабируем координаты (x, y), а также координата z отвечает за глубину объекта. Можно заметить, что в сдвиге присутствует $\frac{-1}{2}$. Этот сдвиг имеет смысл добавлять, потому что тогда точка перемещается не в угол пикселя, которому она соответствует, а в его центр.

$$\begin{bmatrix} \frac{Screen_x}{2} & 0 & 0 & \frac{Screen_x - 1}{2} \\ 0 & \frac{Screen_y}{2} & 0 & \frac{Screen_y - 1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

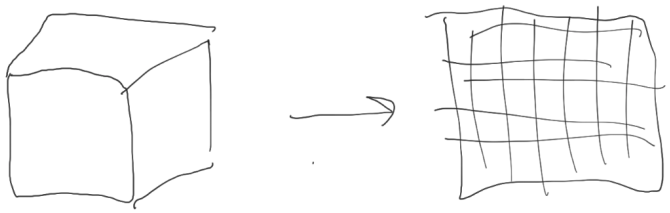

Рис. 11: Шаг 6. Объект перенесен на экран

3.1.2.9 Собираем вместе: общий процесс

В итоге получается, что камера должна сделать большое преобразование из нескольких матриц:

1. Сдвиг камеры в центр координат
2. Поворот всего мира для перехода в базис камеры
3. Клиппинг*
4. Проективное преобразование
5. Сдвиг центра видимого параллелепипеда в центр координат
6. Масштабирование в куб $2 \times 2 \times 2$
7. Прообразование для получения пикселей экрана

Получается, что если точка v попадала на экран, то мы могли провести все преобразования $v \to Mv$, где

$$M = \begin{bmatrix} \frac{Screen_x}{2} & 0 & 0 & \frac{Screen_x - 1}{2} \\ 0 & \frac{Screen_y}{2} & 0 & \frac{Screen_y - 1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & \frac{-l-r}{2} \\ 0 & 1 & 0 & \frac{-b-t}{2} \\ 0 & 0 & 1 & \frac{-n-f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot$$

$$\cdot \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -Camera_x \\ 0 & 1 & 0 & -Camera_y \\ 0 & 0 & 1 & -Camera_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Строго говоря, один объект отображается с помощью двух домножений на матрицу. А именно, сначала сделать перенос и поворот, потом сделать клиппинг, после чего сделать оставшиеся шаги.

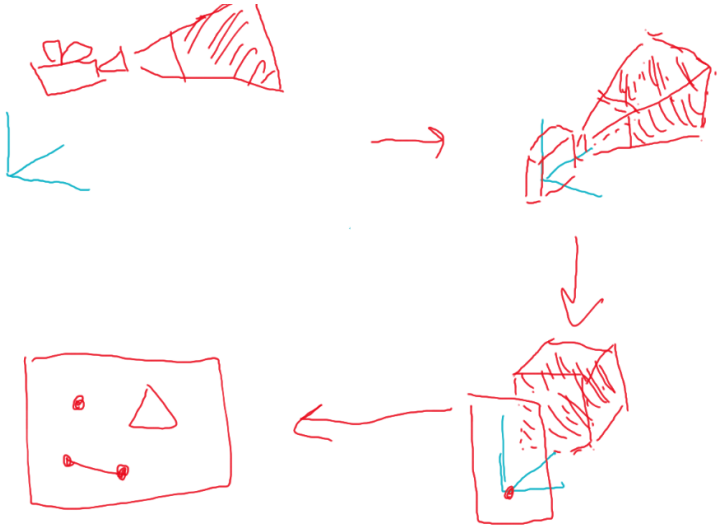

Рис. 12: Менее подробная и более наглядная схема.

3.1.3 Сложность

Для проекции одного треугольника надо сделать не более трех умножений на матрицу 4×4 (одно до клиппинга и одно или два после), а также $O(1)$ действий на клиппинг, причем клиппинг делает нетривиальные операции только для «сложных» объектов — тех, для которых надо вычислять координаты пересечения.

3.2 Отрисовка двумерных объектов

Дальше происходит отрисовка двумерных объектов на экране по пикселям и заданным вершинам. Для отрезка это достаточно несложная задача, а вот для заливки треугольника нужно придумывать эффективные способы. Автор использует так называемую «сканирующую прямую» — треугольник отрисовывается послойно, на каждом слое выделяется вертикальная полоса внутри треугольника, и заполняется только она. Вертикальные полосы выбираются только между самой левой x-координатой треугольника и правой x-координатой.

Таким образом, мы сначала перебираем x в отрезке $[min\_x, max\_x]$, затем y в отрезке $[min\_y(x), max\_y(x)]$, после чего z вычисляется как $\frac{(z(x,max\_y)-z(x,min\_y))\cdot(y-min\_y)}{max\_y-min\_y}$ (это просто линейное масштабирование вектора между крайними точками вертикальной полосы).

Чтобы посчитать $z(x, y)$, нужно выразить точку (x, y) в базисе из двух сторон треугольника, после чего получившиеся коэффициенты применить к исходному четырехмерному треугольнику:

$z(x, y) = z(\alpha \cdot \vec{AB_4} + \beta \cdot \vec{AC_4})$, где α, β находятся из уравнения $\begin{bmatrix}\vec{AB_2} & \vec{AC_2}\end{bmatrix}\begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \begin{bmatrix}x\\ y\end{bmatrix}$, а индексы 2 и 4 означают размерность треугольника, в котором берутся векторы.

Осталось понять, как найти min_y, max_y.

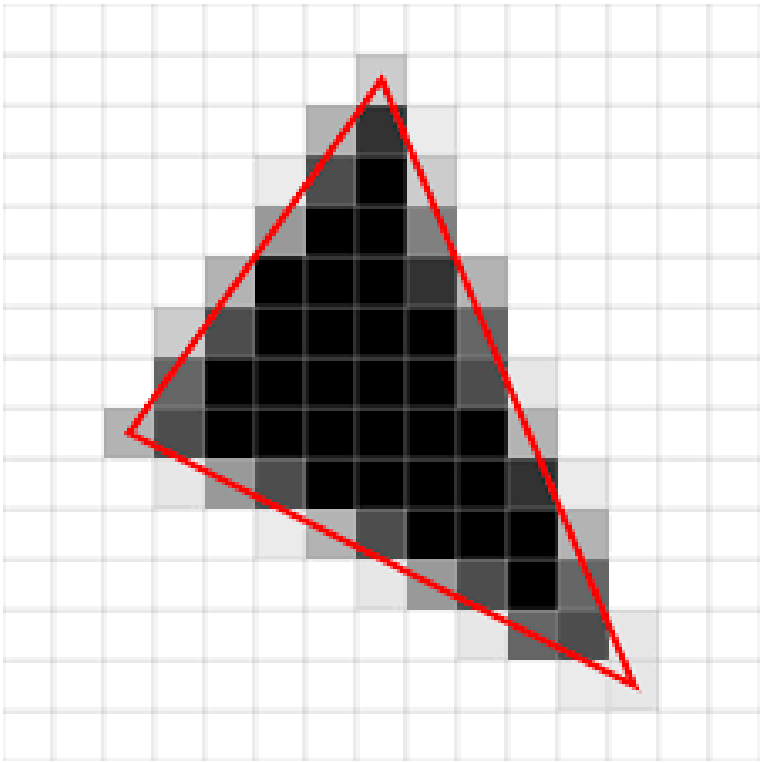

Рис. 13: Двумерный треугольник. Те клетки, которые пересекаются красной линией, имеют минимальную и максимальную y-координату в своей вертикальной полосе. Клетки между ними и надо закрасить.

3.2.1 Сортировка точек в треугольнике

Прежде, чем заполнять полосу в треугольнике, сделаем предобработку. Для того, чтобы уменьшить число случаев, имеет смысл заранее отсортировать все точки треугольника против часовой стрелки, начиная с самой левой (среди самых левых — самой нижней). Для этого нужно найти первую точку как самую левую (среди самых левых — самую нижнюю). Назовем эту точку A, а две оставшиеся B и C (их порядок пока что не определен). После чего достаточно вычислить векторное произведение двух векторов $(B - A) \times (C - A)$. Векторное произведение антисимметрично, поэтому мы можем поменять местами точки B и C так, что произведение будет неотрицательным. А это значит, что точка C будет лежать в левой полуплоскости относительно ориентированного отрезка AB, что и дает нам отсортированность против часовой стрелки.

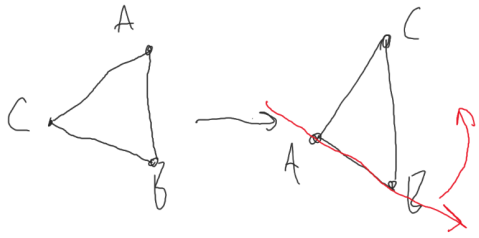

Рис. 14: Треугольник до и после переупорядочивания вершин

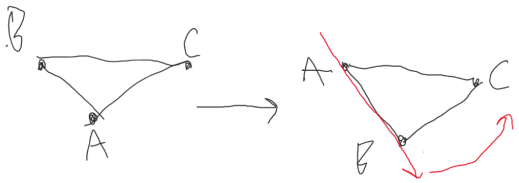

Рис. 15: Еще один треугольник до и после переупорядочивания вершин

3.2.2 Выделение полосы в треугольнике

На текущей стадии мы хотим для каждой вертикальной полосы в треугольнике найти ее границу. Имеющийся у нас инвариант — точки треугольника заранее отсортированы против часовой стрелки, при этом первой в порядке сортировки идет самая левая, а среди самых левых — самая нижняя.

Теперь для выделения полосы есть всего два случая — когда точка C левее точки B, и когда правее. В обоих случаях, зная x-координату полосы, нужные y-координаты вычисляются как точки на соответствующих отрезках.

Дальше я реализую две симметричные функции для нахождения верхней и нижней границы полосы. Рассмотрим поиск нижней границы. Я смотрю на отрезок AB. Если он пересекает полосу, то точка находится на нем, ее можно выразить из линейного уравнения. Если он не пересекает полосу, то точка находится на соседнем отрезке BC, линейное уравнение решается для него. Мы добились такого расположения точек за счет предварительной сортировки.

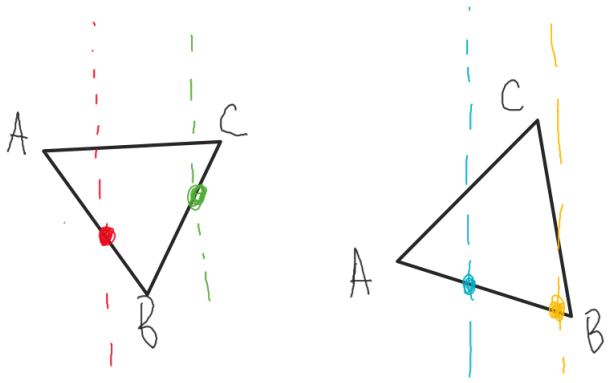

Рис. 16: Два случая расположения треугольников. Точки пересечений показывают минимальный y в полосе. Соответственно, в зависимости от случая, формулы получаются немного разные, а именно:

Для левой картинки	Для правой картинки
$min\_y(x) = \begin{cases} A.y + \frac{(x-A.x)\cdot(B.y-A.y)}{B.x-A.x} & x < B.x \\ C.y + (\frac{(x-C.x)\cdot(B.y-C.y)}{B.x-C.x} & x \geq B.x \end{cases}$	$max\_y(x) = \begin{cases} A.y + \frac{(x-A.x)\cdot(C.y-A.y)}{C.x-A.x} & x < C.x \\ B.y + \frac{(x-B.x)\cdot(C.y-B.y)}{C.x-B.x} & x \geq C.x \end{cases}$
$max\_y(x) = A.y + \frac{(x-A.x)\cdot(C.y-A.y)}{C.x-A.x}$	$min\_y(x) = A.y + \frac{(x-A.x)\cdot(B.y-A.y)}{B.x-A.x}$

3.2.3 Сложность

Если оценить сложность, то мы выделим необходимую область за $O(w)$, где w — ширина треугольника. Таким образом, самая неэффективная часть в отрисовке треугольника — отрисовка каждого пикселя за $O(S_\Delta)$. Но поскольку для любого растрового экрана это действие нам понадобится, можно считать, что алгоритм досточно эффективный для нашей задачи.

Таким образом, отрисовка всех объектов происходит за сумму площадей их треугольников.

4 Программная часть

4.1 Описание разделов

В результате работы над проектом получилось реализовать следующие части программного проекта:

1. Библиотека, реализующая функционал 3d-рендерера.
2. Приложение на основе библиотеки.
3. Документация.
4. Тесты.

4.2 Библиотека: описание элементов

4.2.1 SFML

SFML — это библиотека для отрисовки 2d-графики. У нее есть два основных приемущества:

1. Достаточно простая.
2. Поддерживает создание интерактивных приложений.

Используется для отрисовки массива точек (растеризованный экран).

4.2.2 SurfaceObject

SurfaceObject — это класс, который занимается хранением объектов в пространстве.

Структура объектов такая: есть объекты типа WireObject, есть объекты типа SurfaceObject, являющиеся наследником типа WireObject. WireObject хранит каркас фигуры (точки и ребра), а SurfaceObject расширяет WireObject данными о триангулированной поверхности объекта. Взаимодействие пользователей происходит с объектами типа SurfaceObject. Также у каждого объекта есть цвет, и к нему можно применить преобразование (а именно, ко всем его вершинам).

Чтобы иметь возможность итерироваться по данным SurfaceObject, предусмотрены методы begin и end для WireObject, благодаря которым можно перебрать все точки объекта. Также предусмотрен метод, возвращающий массив всех треугольников, относящихся к поверхности объекта.

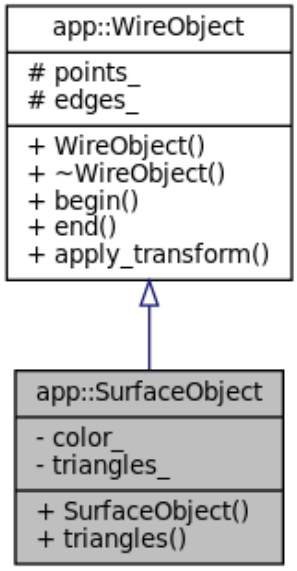

Рис. 17: Схема класса SurfaceObject и его зависимостей-владений

4.2.3 Camera

Camera — это класс, который «смотрит» на пространство с разных сторон.

Класс Camera отвечает за то, чтобы переводить объекты из системы координат мира в систему координат экрана. Для этого этот объект генерирует матрицу преобразования и применяет ее к точкам, которые надо отобразить. Камеру можно поворачивать и двигать, что отражается на матрице преобразования.

Для большего удобства есть четыре вида преобразования. Есть преобразование точки в координаты камеры (поворот + сдвиг). Есть преобразование из трехмерной точки в координатах камеры в трехмерную точку в пространстве (экран, z-value), а также два под-преобразования, которые выполняют предыдущее преобразование, но возвращают либо только координаты пикселя на экране, либо только его z-value.

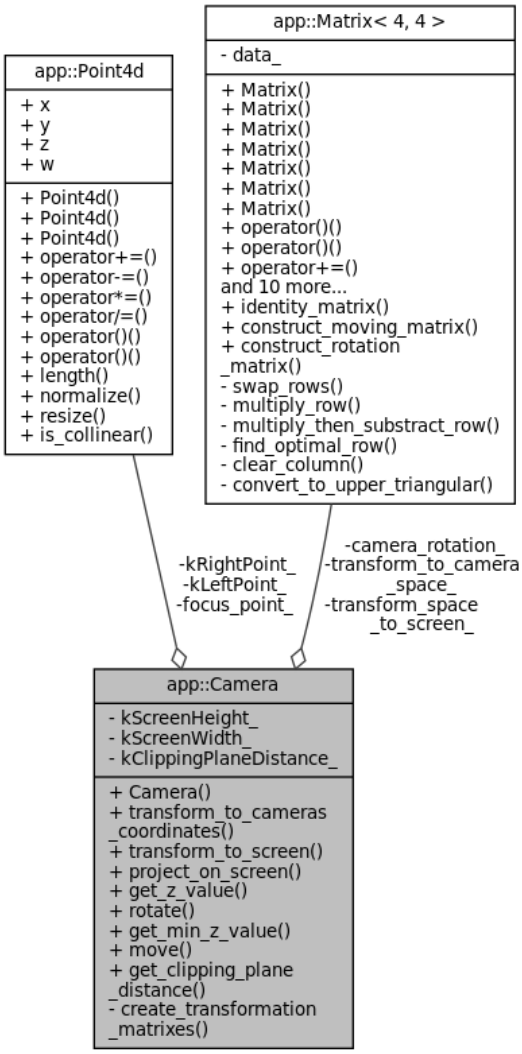

Рис. 18: Схема класса Camera и его зависимостей-владений

4.2.4 Renderer

Renderer — это класс, который берет на себя ответственность за отрисовку объектов.

Класс отвечает за логику отрисовки. Внутри себя он реализует все алгоритмы, связанные с отрисовкой 2d-графики, а также клиппинга. Для пользователя же есть всего два метода в интерфейсе — нарисовать треугольник (то есть добавить треугольник в кадр) и отрисовать кадр пользователю.

Ожидаемое использование Renderer'а цикличное — сначала в кадр добавляется много треугольников, после чего происходит отрисовка кадра, а затем процесс происходит снова.

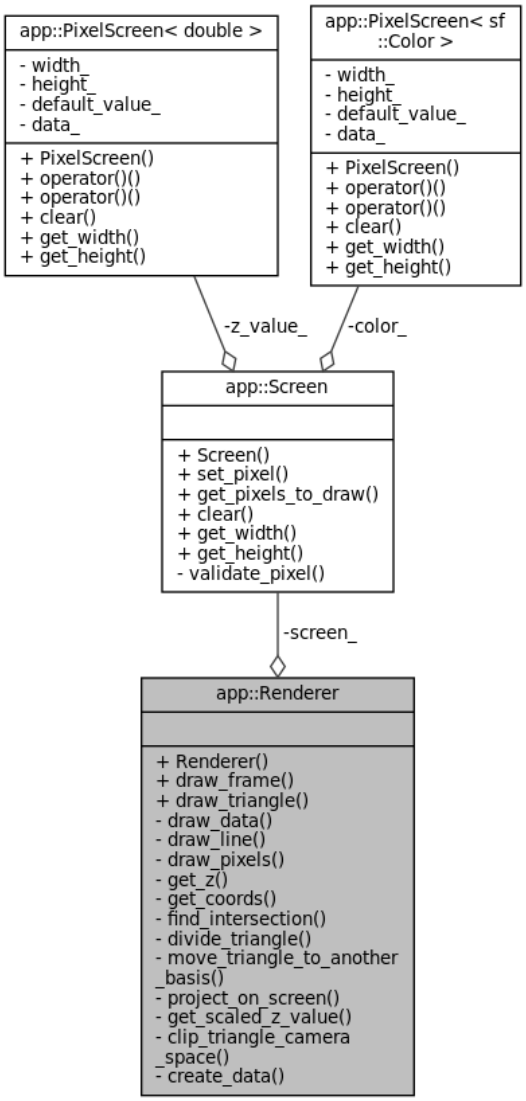

Рис. 19: Схема класса Renderer и его зависимостей-владений

4.2.5 Matrix

Matrix — это класс, котоый выполняет все матричные вычисления.

Пользователю доступно создание матриц произвольного размера, различные перегруженные операторы (сложение, вычитание, умножение и т.д.). Также доступен поиск обратной матрицы, решение системы линейных уравнений, транспонирование.

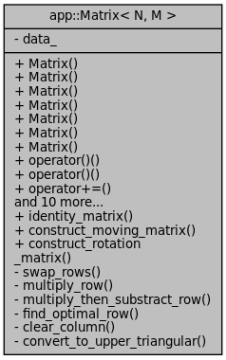

Рис. 20: Схема класса Matrix и его зависимостей-владений

4.2.6 Point

Point4d — это класс, который отвечает за соответствие между 3d- и 4d-точками.

Точки Point4d представляют собой набор из четырех координат. Это объект, который нужен для удобной обертки над несколькими переменными. Для него перегружены арифметические операции. Поскольку точка в пространстве соответствует вектору из нуля в себя, то объект точки отождествляется с этим вектором. В частности, есть возможность нахождения его длины, изменение этой длины и проверка на сонаправленность с другим вектором. Также доступно отождествление 4d-точек с трехмерными через единичную последнюю координату (однородные координаты), нормализация координат (приведение к такой трехмерной точке, у которой $w = 1$).

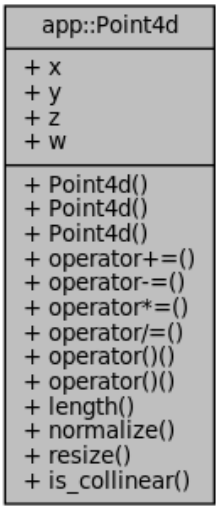

Рис. 21: Схема класса Point4d и его зависимостей-владений

4.2.7 Triangle

Triangle2d / Triangle4d — классы, который отвечает за работу с треугольниками в двумерном и четырехмерном пространстве.

Для работы с двумерной геометрией используется класс двумерного треугольника Triangle2d. Основной его смысл — давать нужную информацию о треугольнике — координаты точек, сохраненные в отсортированном порядке, и векторы, которые образуют треугольнк. Также треугольник дает y-координаты пересечения вертикальной полосы с собой. Также треугольник умеет собрать всю полезную информацию о себе в специальную структуру Renderer::DrawData, которая будет в дальнейшем использоваться для отрисовки.

Еще есть четырехмерный треугольник, который используется как удобная обертка для хранения трех точек и цвета. Из интересного — метод get_points возвращает точки треугольника в порядке, который был запрошен пользователем. Это нужно для того, чтобы порядок вершин был синхронизирован с 2d-версией этого треугольника.

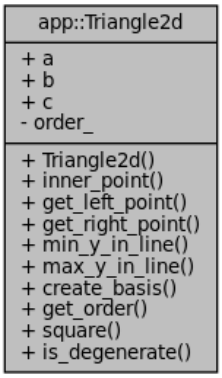

Рис. 22: Схема класса Triangle2d и его зависимостей-владений

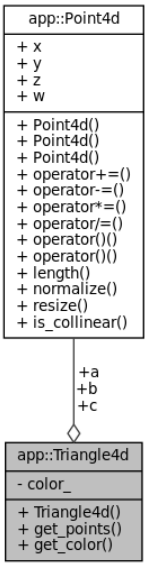

Рис. 23: Схема класса Triangle4d и его зависимостей-владений

4.2.8 Screen

Screen — класс, отвечающий за то, чтобы на экране показывались нужные пиксели.

Экран Screen нужен для растрового представления экрана. Он представляет собой два контейнера с данными по каждому пикселю экрана с возможностью доступа и изменения. Screen позволяет работать со значением цвета пикселя и текущим значением z-буфера в точке.

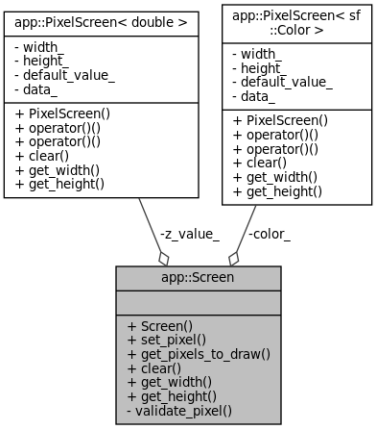

Рис. 24: Схема класса Screen и его зависимостей-владений

4.2.9 World

World — хранилище для объектов.

Мир World является просто контейнером, в котором лежат все объекты. Контейнер позволяет добавить и удалить элемент, а также проитерироваться по всем объектам в мире с помощью begin() и end() методов.

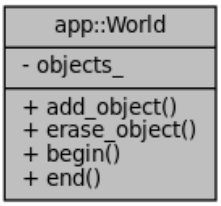

Рис. 25: Схема класса World и его зависимостей-владений

4.3 Тестовое приложение

Тестовое приложение реализовано через класс Application. Этот класс отвечает за создание окна приложения, обработку действий пользователя и обновление картинки.

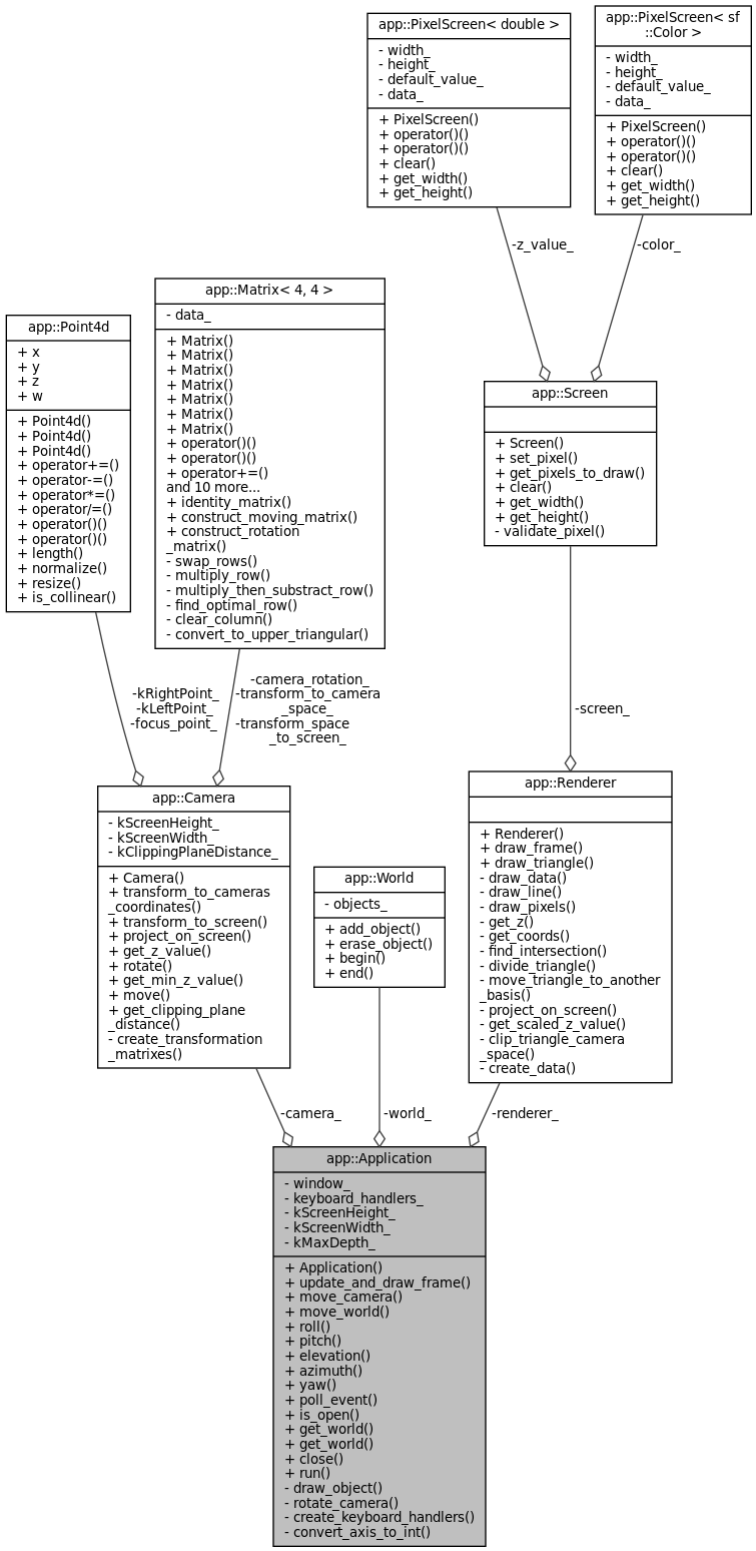

Рис. 26: Схема класса Application и его зависимостей-владений

4.4 Библиотека: Pipeline

4.4.1 Описание пайплайна.

В итоге Pipeline отрисовки получился таким:

- Application.update_and_draw_frame() переберет сначала все объекты в мире, а потом все треугольники в каждом из объектов. Эти объекты подаются на отрисовку в renderer.draw_triangle() вместе с объектом камеры.
- Renderer.draw_triangle() с помощью Camera и клиппинга делает проекции точек.
- Затем Renderer выполняет алгоритм растеризации треугольника и передает пиксели в Screen, который запоминает те пиксели, которые надо отобразить на экране (минимальная z-value).
- Application.update_and_draw_frame() в самом конце делает Renderer.draw_frame(), который берет списком все пиксели экрана для отображения и отрисовывает их с помощью SFML.

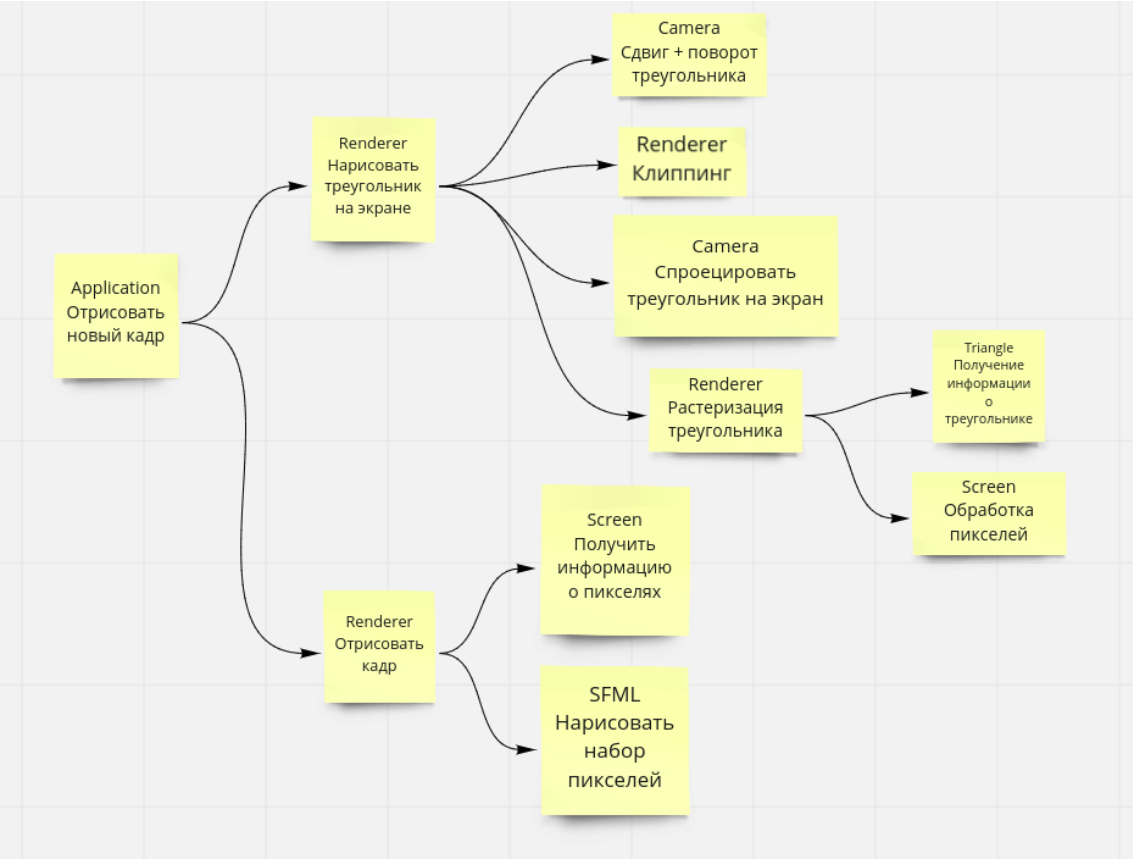

Рис. 27: Схема ключевых взаимодействий в пайплане.

4.5 Документация.

Для документации проекта использовался пакет Doxygen, который позволяет построить документацию прямо из программного кода. У этого подхода есть несколько приемуществ:

Во-первых, документация получается однородной и более красивой, чем если бы она оформлялась вручную: от программиста требуются только тексты с описаниями классов и методов.

Во-вторых, код естественным образом покрывается комментариями. Теперь при чтении исходного кода библиотеки гораздо понятнее, за что отвечают те или иные методы.

В-третьих, автоматическая обработка кода позволяет Doxygen самому задокументировать сигнатуры методов и классов, что избавляет программиста от лишней ручной работы.

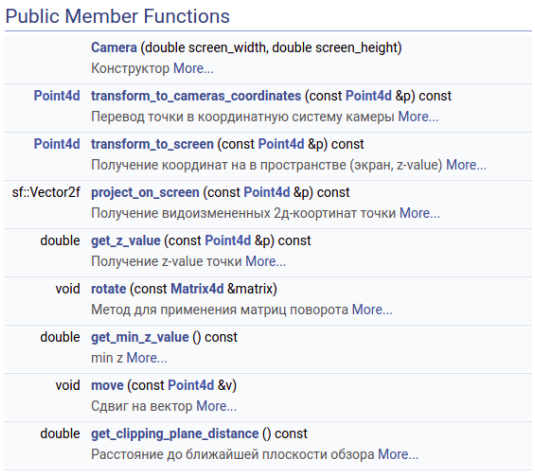

Рис. 28: Документация публичных методов класса Camera

4.6 Тестирование.

Тестирование проекта производилось двумя способами: ручным тестированием и автоматическим юнит-тестированием.

Тестирование с помощью тестового приложения проверяет в основном то, что картинка соответствует ожиданию, что никакие объекты не распадаются при перемещении, что при выходе за границы экрана объект корректно отображается видимая часть объекта.

Юнит-тесты проверяют то, что отдельные модули приложения работают корректно: матрицы правильно перемножаются, камера правильно проецирует точки, и так далее. Такие тесты были написаны на основе утилиты CxxTest, которая позволяет автоматизировать тестирование. Чтобы не перепроверять по многу раз, юнит-тестирование было вынесено в Github Actions — каждый новый коммит прогонялся на тестах автоматически. Для тестов использовался Docker-контейнер, который собирал библиотеку с нуля, за счет чего также тестировалась работоспособность проекта на Ubuntu.

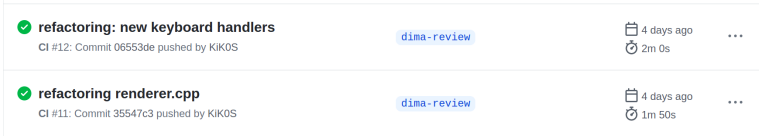

Рис. 29: Удачная сборка после коммитов

5 Результаты.

В рамках работы над проектом получены следующие результаты:

1. Создана библиотека для отрисовки 3d-объектов в пространстве с открытым исходным кодом.
2. На основе библиотеки разработано тестовое приложение.
3. Библиотека задокументирована.
4. Для библиотеки созданы автоматические тесты.
5. Теоретические результаты исследования оформлены и записаны.

Вот так выглядит объект в тестовом приложении:

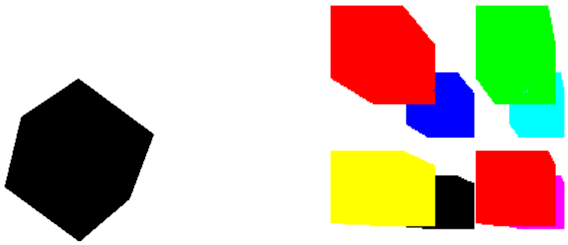

Рис. 30: Кубы в тестовом приложении

Также следует отметить, что скорость работы отдельных методов библиотеки была исследована профайлером в целях нахождения мест для оптимизации — производительность упирается в алгоритм растеризации, которому так или иначе нужно обрабатывать все пиксели экрана. Получается, что код имеет хорошие показатели по скорости в условиях работы на CPU.

```
index % time    self  children    called     name
                                               <spontaneous>
[1]     11.3    1.65    0.00                 app::Screen::set_pixel(int, int, double, sf::Color) [1]
-----------------------------------------------
                                               <spontaneous>
[2]      5.2    0.76    0.00                 app::PixelScreen<double>::operator()(unsigned long, unsigned long) [2]
-----------------------------------------------
                                               <spontaneous>
[3]      4.4    0.64    0.00                 app::Point4d app::operator*<4ul, 4ul>(app::Matrix<4ul, 4ul> const&, app::Point4d const&) [3]
-----------------------------------------------
                                               <spontaneous>
[4]      3.8    0.56    0.00                 app::Renderer::draw_line(int, double, double, app::Camera const&, app::Renderer::DrawData const&) [4]
-----------------------------------------------
                                               <spontaneous>
[5]      3.6    0.53    0.00                 app::Screen::validate_pixel(int, int) const [5]
-----------------------------------------------
                                               <spontaneous>
[6]      3.6    0.52    0.00                 app::Screen::get_pixels_to_draw() const [6]
-----------------------------------------------
```

Рис. 31: Результаты работы профайлера. Можно видеть, что основной ресурс процессора уходит на работу с пикселями на экране.

Список литературы

[1] https://raw.githubusercontent.com/lorensen/VTKExamples/master/src/VTKBookLaTeX/VTKTextBook.pdf. [VTKBook].

[2] https://www.youtube.com/playlist?list=PLbCDZQXIq7uYaf263gr-zb0wZGoCL-T5G. [Urtech University Computer Graphics lectures].

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

[4] https://www.sfml-dev.org/index.php. [Simple and Fast Multimedia Library Documentation]

[5] https://www.doxygen.nl/manual/docblocks.html [Doxygen documentation tool]

[6] https://docs.docker.com/ [Docker documentation]

[7] http://cxxtest.com/guide.html [Cxxtest guide]

[8] https://en.cppreference.com/w/ [CppReference]

[9] https://docs.github.com/en/actions [Github Actions guide]