

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

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

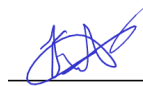
Отчет о программном проекте

на тему: 3D-renderer с нуля

(промежуточный, этап 2)

Выполнил:

Студент группы БПМИ191



Подпись

К.В. Амеличев

И.О.Фамилия

16.04.2021

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

1	Введение	3
1.1	Функциональные требования	3
1.2	Технические требования	4
2	Изучение аналогов	5
3	Содержательная часть	6
3.1	Основы 3d-графики	6
3.1.1	Объекты для отрисовки	6
3.1.2	Экран и камера	6
3.1.3	Общая последовательность шагов	6
3.1.3.1	Однородные координаты	7
3.1.3.2	Здоровенная матрица	8
3.2	Отрисовка двумерных объектов	9
3.2.1	Сортировка точек в треугольнике	9
3.2.2	Выделение полосы в треугольнике	10
3.2.3	Сложность	11
3.3	Библиотека: описание элементов	12
3.3.1	Авторское: Точки, матрицы, геометрия	12
3.3.2	Авторское: Объекты, мир	12
3.3.3	Авторское: Камера	12
3.3.4	Авторское: Экран	12
3.3.5	Авторское: Renderer	12
3.3.6	Авторское: Application	12
3.3.7	Стороннее: SFML	13
3.4	Библиотека: Pipeline	14
3.4.1	Описание.	14
3.4.2	Схема.	14
3.4.3	Псевдокод,	14
3.5	Тестовое приложение и текущие результаты.	17

Аннотация

В рамках данной работы я разрабатываю библиотеку для языка программирования C++, занимающуюся отрисовкой объектов в трехмерном пространстве. После написания библиотеки планируется сделать 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. Данная библиотека будет полезна тем, кто обрабатывает достаточно сложные объекты, чтобы не отрисовывать их вручную, но при этом недостаточно сложные, чтобы потребовалось большое количество настроек.

Задачами данной работы является исследование существующих технологий и принципов в сфере 3d-графики, разработка интерфейса, реализация функционала, создание тестового приложения, последующее документирование и оформление в едином стиле библиотеки с открытым исходным кодом, а также 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. [3] — математическая основа для библиотеки.
3. [2] — курс лекций по компьютерной графике

1.1 Функциональные требования

- Отрисовка 3d-объектов.
- База стандартных объектов для отрисовки, таких как куб или тетраэдр.
- Возможность создать любой триангулируемый объект в пространстве.
- Задание параметров объекта, масштабирование, цвет.
- Перемещение объектов.
- Поворот объектов произвольным образом.
- Возможность последовательной смены кадров, создание анимации
- Обработка взаимодействия пользователя и приложения — пользовательские повороты и перемещения объектов, в частности по взаимодействию с клавиатурой
- Возможность программно создавать и удалять объекты во время анимации.
- Скрытие интерфейса отрисовки от пользователя — он отвечает только за создание объектов и операции с ними
- Повороты и перемещение камеры
- Возможность работы в статическом режиме — создать кадр и сохранить его как файл с изображением.

Интерфейс для пользователя следующий: у приложения один главный класс Application, представителя которого надо создать. После чего Application создает окно с графикой пользователя. Пользователь может добавлять объекты Application'у, давать ему команды для трансформации содержимого (перемещения, поворотов), а также в необходимый момент перерисовывать кадр.

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
- Открытый исходный код
- Система поддержки версий Git
- Ошибки программиста отлавливаются assert-ами
- Системные ошибки обрабатываются как исключения и могут быть причиной аварийной остановки программы
- Google C++ styleguide

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

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 Содержательная часть

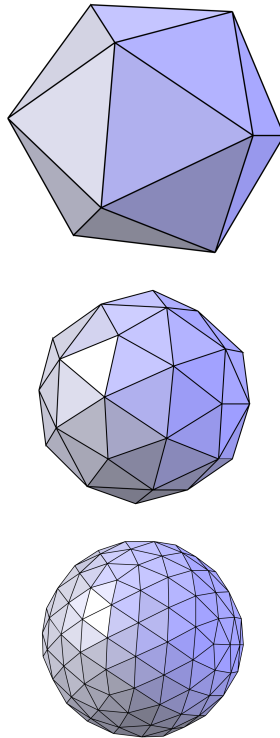
3.1 Основы 3d-графики

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, что тоже имеет смысл.

Каждый объект задается набором отрезков и треугольников в глобальной системе координат. Отрезки можно использовать для отрисовки ребер фигуры, а поверхности можно триангулировать. 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. При этом в рамках одной фигуры все отрезки и треугольники находятся в относительной системе координат, где имеют свои относительные координаты. Если с объектом не происходит механических манипуляций, то в относительной системе координат координаты отрезков и треугольников будут неизменными.



3.1.2 Экран и камера

3.1.3 Общая последовательность шагов

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 выбирать прямоугольник экрана, используется объект камеры. Этот объект задается своим положением в пространстве ($O_c = \begin{bmatrix} x_{camera} \\ y_{camera} \\ z_{camera} \end{bmatrix}$), а также системой из трех ортогональных

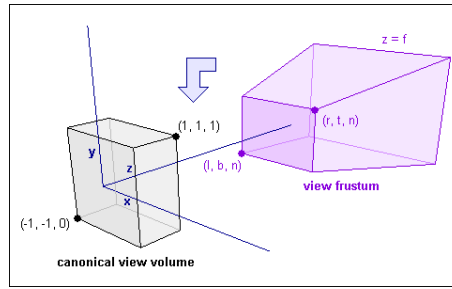
векторов, которые = задают направление, в котором смотрит камера, а также координатную плоскость, параллельную плоскости экрана. Назовем направление камеры вектором N_c .

По направлению N_c выбирается область видимости и прямоугольник экрана. Все вершины фигур проецируются на плоскость экрана через базисные векторы A_c, B_c . Каким образом проецируются? Для начала, вся система координат сдвигается на вектор $-N_c$. После этого берется матрица поворота для системы координат, которая приводит систему координат камеры к стандартной $oXYZ$. Иначе говоря, это такая матрица,

которая делает поворот мира, обратный к повороту камеры. $M \begin{bmatrix} A_c & B_c & N_c \end{bmatrix} = \begin{bmatrix} A & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & C \end{bmatrix}$.

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 параллелепипед. После этого проекция оказывается просто отбрасыванием координаты.

У этого подхода есть еще одно преимущество. 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.



Так что после поворота камеры происходит проективное преобразование, после чего искомым прямоугольник сначала масштабируется в куб $2 \times 2 \times 2$. У точки в кубе координаты (x, y) проецируются на экран, а координата z отвечает за глубину точки относительно экрана.

3.1.3.1 Однородные координаты

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

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

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}, w \neq 0 \rightarrow \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \\ 1 \end{pmatrix}$$

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

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

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

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

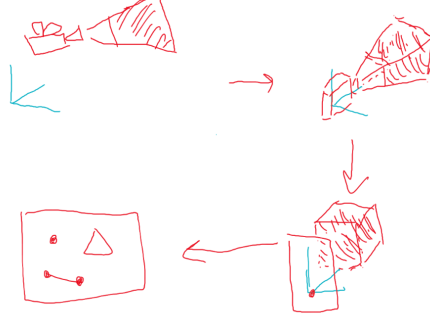
Утверждается, что такого дополнительного функционала хватает, чтобы сделать проективное преобразование пирамиды зрения в прямоугольный параллелепипед.

3.1.3.2 Здоровенная матрица

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. Проективное преобразование
4. Сдвиг центра видимого параллелепипеда в центр координат
5. Масштабирование в куб $2 \times 2 \times 2$
6. Преобразование для получения пикселей экрана

$$M = \begin{bmatrix} \frac{Screen_x}{2} & 0 & 0 & \frac{Screen_x-1}{2} \\ 0 & \frac{Screen_y}{2} & 0 & \frac{Screen_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & \frac{-l-r}{2} \\ 0 & 1 & 0 & \frac{-b-t}{2} \\ 0 & 0 & 1 & \frac{-n-f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & n & 0 \end{bmatrix} \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -Camera_x \\ 0 & 1 & 0 & -Camera_y \\ 0 & 0 & 1 & -Camera_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



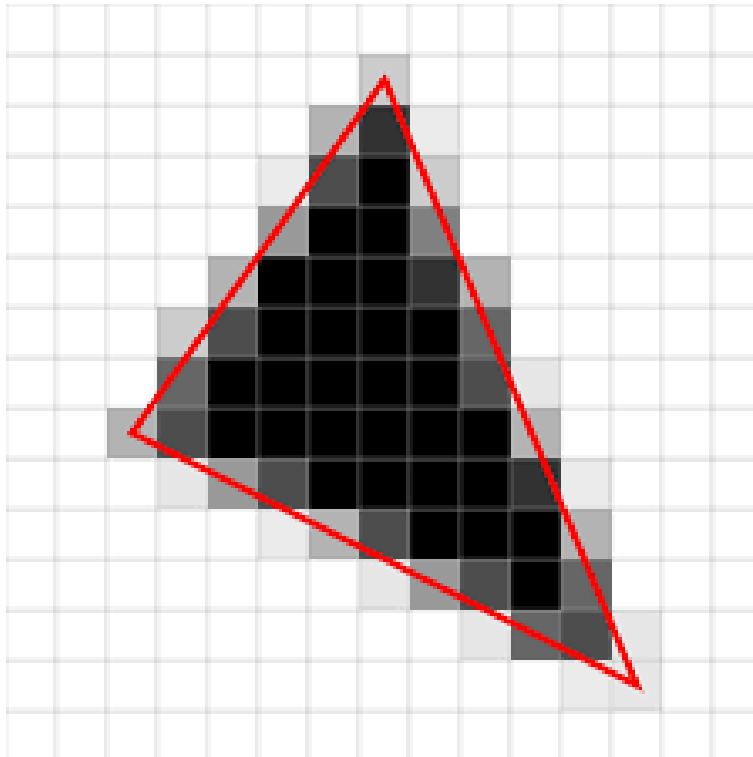
3.2 Отрисовка двумерных объектов

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.

Псевдокод отрисовки двумерного треугольника:

```
for (int x = triangle2d.get_left_x(); x <= triangle2d.get_right_x(); x++) {  
    // triangle logic to find the stripe  
    min_y = find_min_y(triangle2d, x);  
    max_y = find_max_y(triangle2d, x);  
  
    double min_z = ...; // get z_value for 2d-point (x, min_y)  
    double max_z = ...; // get z_value for 2d-point (x, max_y)  
    for (int y = ceil(min_y); y <= floor(max_y); y++) {  
        // linear scaling for z  
        z = (max_z - min_z) * (y - min_y) / (max_y - min_y);  
        screen->set_pixel(x, y, z);  
    }  
}
```

Надо теперь понять, как найти \min_y , \max_y . Поиск \min_z , \max_z будет не очень сложным — поскольку треугольник невырожденный, достаточно будет выделить в нем два базисных вектора, по ним выразить наш пиксель, затем с такими же коэффициентами взять базисные векторы треугольника в пространстве; полученную точку нужно будет преобразовать и найти ее z -value.



3.2.1 Сортировка точек в треугольнике

Чтобы в треугольнике выделить полосу, нужно рассмотреть несколько случаев. Для того, чтобы уменьшить число случаев, имеет смысл заранее отсортировать все точки треугольника против часовой стрелки. Для этого нужно найти первую точку как самую левую (среди самых левых - самую нижнюю), после чего сравнить векторное произведение двух векторов $(B - A) \times (C - A)$.

Псевдокод:

```

Triangle2d::Triangle2d(sf::Vector2f _a, sf::Vector2f _b, sf::Vector2f _c): a(_a), b(_b), c(_c) {
    for (int i = 0; i < 3; i++) {
        order_[i] = i;
    }
    if (a.x > b.x || (a.x == b.x && a.y > b.y)) {
        std::swap(order_[0], order_[1]);
        std::swap(a, b);
    }
    if (a.x > c.x || (a.x == c.x && a.y > c.y)) {
        std::swap(a, c);
        std::swap(order_[0], order_[2]);
    }
    if (cross(b - a, c - a) < 0) {
        std::swap(order_[1], order_[2]);
        std::swap(b, c);
    }
}

```

3.2.2 Выделение полосы в треугольнике

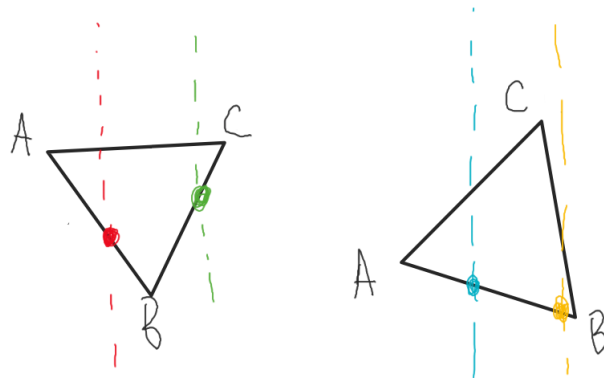
Чтобы для каждого слоя найти вертикальную полосу, я использую следующий алгоритм: точки треугольника заранее отсортированы против часовой стрелки, при этом первой в порядке сортировки идет самая левая, а среди самых левых — самая нижняя. Далее я реализую две симметричные функции для нахождения верхней 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.

Псевдокод поиска границы для полосы:

```

double find_min_y(triangle, x) {
    if (|triangle.b.x - x| < eps) {
        return triangle.b.y;
    }
    if (triangle.b.x > x) {
        v = triangle.a - triangle.b;
        if (|v.x| < eps) {
            return triangle.a.y;
        }
        k = (triangle.b.x - x) / v.x;
        v *= |k|;
        return (triangle.b + v).y;
    }
    else {
        v = triangle.c - triangle.b;
        if (|v.x| < eps) {
            return triangle.b.y;
        }
        k = (triangle.b.x - x) / v.x;
        v *= |k|;
        return (triangle.b + v).y;
    }
}

```



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 достаточно эффективный для нашей задачи.

3.3 Библиотека: описание элементов

3.3.1 Авторское: Точки, матрицы, геометрия

Матрицы `Matrix` нужны для работы с матричными вычислениями. Пользователю доступно создание матриц произвольного размера, операторы для сложения, вычитания и перемножения матриц. Также доступен поиск обратных матриц, выражение вектора через базис пространства, транспонирование.

Точки `Point4d` представляют собой набор из четырех координат или матрицу 4×1 . Это объект, который нужен для удобной обертки над несколькими переменными. Для него также доступны арифметические операции, а также отождествление с вектором и возможность нахождения длины. Также есть отождествление 4d-точек с трехмерными через единичную последнюю координату (однородные координаты).

Для работы с двумерной геометрией используется класс двумерного треугольника `Triangle2d`. Основным его смысл — давать нужную информацию о треугольнике — координаты точек, сохраненные в отсортированном порядке.

Все многомерные треугольники для отрисовки передаются как объекты класса `Triangle4d`, который выступает в роли контейнера.

3.3.2 Авторское: Объекты, мир

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`. Мир `World` является просто контейнером, в котором лежат все объекты. Объекты можно¹ поворачивать и двигать.

И объекты, и мир поддерживают возможность итерирования по своему содержимому: у объектов можно просмотреть все ребра и грани, а у мира можно посмотреть все объекты.

3.3.3 Авторское: Камера

Камера `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 ко всем точкам, которые надо отобразить. Также камеру можно поворачивать и двигать, что отражается на матрице преобразования.

3.3.4 Авторское: Экран

Экран `Screen` нужен для растрового представления экрана. Он представляет собой несколько табличек с данными по каждому пикселю экрана с возможностью доступа и изменения. А именно, позволяет поставить значение цвета пикселя и узнать текущее значение z -буфера в точке.

3.3.5 Авторское: `Renderer`

`Renderer` отвечает за логику отрисовки. Он принимает набор объектов `SurfaceObject`, после чего берет отрезки и треугольники в многомерном пространстве, с помощью камеры `Camera` переносит их в двумерное пространство, выполняет алгоритм растеризации и переносит весь кадр на экран `Screen`, после чего отрисовывает кадр с помощью библиотеки `SFML`.

3.3.6 Авторское: `Application`

Объект `Application` является основным объектом приложения для пользователя — именно оно осуществляет взаимодействие между всеми остальными объектами. Иначе говоря, `Application` берет все объекты `SurfaceObject` из контейнера `World`, после чего просит `Renderer` отрисовать эти объекты.

Также `Application` поддерживает интерактивный функционал, доступный в `SFML` — позволяет пользователю настроить listener-ы на различные системные события, которые умеет отлавливать `SFML` (например, нажатие на клавиши).

¹когда-нибудь будет можно, строго говоря

3.3.7 Стороннее: SFML

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

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

Используется для отрисовки массива точек (растеризованный экран), а также для отрисовки отрезков-ребер.

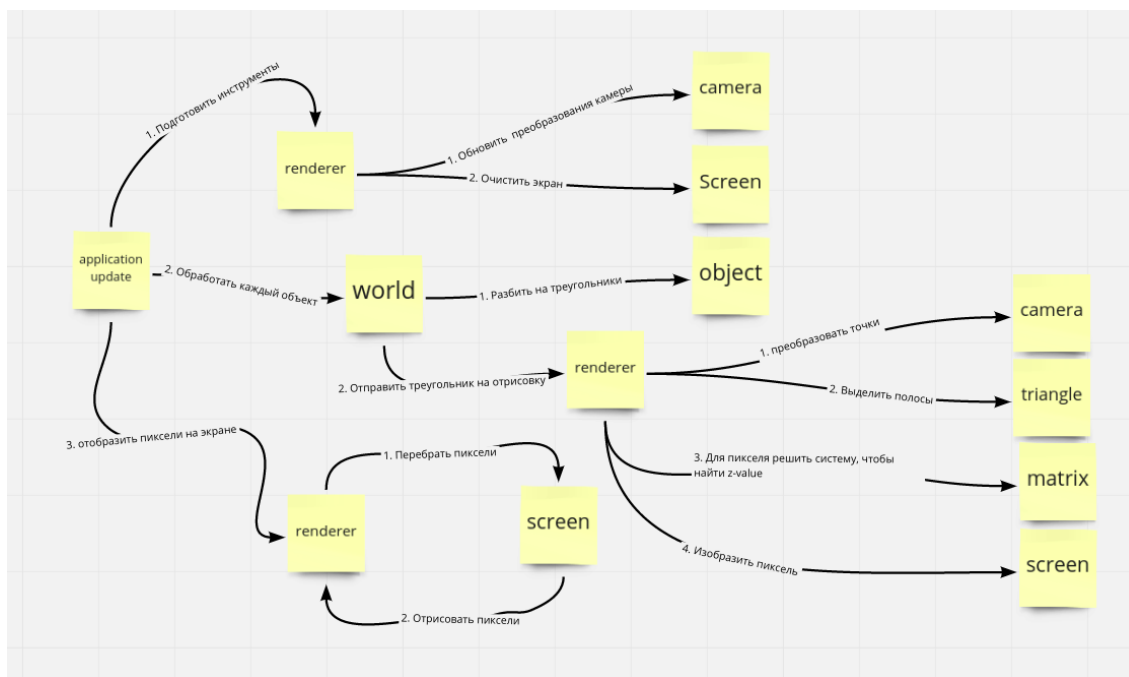
3.4 Библиотека: Pipeline

3.4.1 Описание.

В итоге основной Pipeline получился таким:

- Application.update() вызовет Renderer.prepare(), чтобы очистить экран Screen.clear() и обновить матрицу преобразования Camera.create_transform()
- Application.update() вложенными циклами переберет сначала все объекты в мире, а потом все треугольники в объекте.
- Эти треугольники отправятся на Renderer.draw() — сначала Camera.project_point() переведет их в координаты на экране, потом прогоняется алгоритм для двумерного треугольника: для каждой x-координаты находится полоса с помощью Triangle2d.find_max_y()/Triangle2d.find_min_y(). Чтобы узнать z-координату пикселя, надо узнать ее для двух крайних пикселей полосы; с помощью Matrix.solve_system() двумерный вектор выражается по базису треугольника, после чего нужная z-величина получается взятием этих же векторов в 3-мерном пространстве с полученными коэффициентами²
- Полученные пиксели отправляются в Screen.draw(), где вычисляется ближайший пиксель к плоскости экрана (минимальная z-value)
- Application.update() в самом конце делает Renderer.update(), который делает Screen.update(), который проходит по всем пикселям и вызывает Renderer.draw() для этого пикселя, в результате пиксель оказывается на экране.

3.4.2 Схема.



3.4.3 Псевдокод,

```
void Application::update() {
    renderer_>prepare();
    for (auto& object : *world_) {
        for (auto& triangle : object->triangles()) {
            renderer_>draw(triangle);
        }
    }
}
```

²На самом деле можно без решения системы, потому что там всегда в одном из трех базисов координата будет скаляром вдоль ровно одного базисного вектора. Но поскольку там надо еще понять, вдоль какого вектора, то я пока что просто систему решаю.

```

        triangles++;
    }
}
renderer_>update();
}

void Renderer::prepare() {
    window_.clear(sf::Color::White);
    screen_>clear();
    camera_>create_transform();
}

void Renderer::update() {
    screen_>update();
    window_.display();
}

void Renderer::draw(const Triangle4d& triangle4d) {
    Triangle2d triangle2d(camera_>project_point(triangle4d.a),
                          camera_>project_point(triangle4d.b),
                          camera_>project_point(triangle4d.c));

    sf::Vector2f left_point = triangle2d.get_left_point();
    sf::Vector2f right_point = triangle2d.get_right_point();

    Matrix<2, 2> basis = triangle2d.create_basis();
    for (int x = ceil(left_point.x); x <= floor(right_point.x); x++) {
        min_y = find_min_y(triangle2d, x);
        max_y = find_max_y(triangle2d, x);

        min_z = get_z(/* get z for (x, min_y) from basis and triangle2d and triangle4d */);
        max_z = get_z(/* get z for (x, max_y) from basis and triangle2d and triangle4d */);
        for (int y = ceil(min_y); y <= floor(max_y); y++) {
            z = (max_z - min_z) * (y - min_y) / (max_y - min_y);
            screen_>set_pixel(x, y, z, sf::Color::Black);
        }
    }
}

void Camera::create_transform() {
    transform_ = canonical2Screen * rect2Canonical * move2Center *
                * magic * transform_camera_.transpose() * move_camera;
}

Point4d Camera::transform_point(Point4d p) const {
    return transform_ * p;
}

sf::Vector2f Camera::project_point(Point4d p) const {
    Point4d transformed = transform_point(p);
    return {transformed.x / transformed.w,
            transformed.y / transformed.w};
}

```

```

double Camera::get_z_value(Point4d p) const {
    return transform_point(p).z / transform_point(p).w;
}

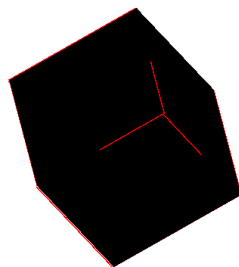
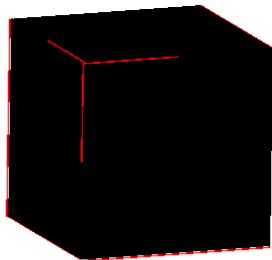
void Screen::set_pixel(int x, int y, double z, sf::Color color) {
    if (x < 0 || y < 0 || x >= get_screen_size() || y >= get_screen_size() ||
        z > get_max_z_value() || z < get_min_z_value()) {
        return;
    }
    if (z < z_value_(x, y)) {
        z_value_(x, y) = z;
        color_(x, y) = color;
    }
}

void Screen::update() {
    vector<sf::Vertex> data;
    for (int i = 0; i < get_screen_size(); i++) {
        for (int j = 0; j < get_screen_size(); j++) {
            if (z_value_(i, j) < get_max_z_value()) {
                data.push_back(sf::Vertex(sf::Vector2f(i, j), color_(i, j)));
            }
        }
    }
    renderer_>draw(data);
}

```


3.5 Тестовое приложение и текущие результаты.

Для теста на текущей стадии разработки используется приложение, в котором можно вращать и двигать кубик. Далее привожу две картинки того, как это у меня выглядит сейчас.



Список литературы

- [1] <https://lorensen.github.io/VTKExamples/site/VTKBook/00Preface/>. [VTKBook].
- [2] https://www.youtube.com/watch?v=4z6zRet_gkg&list=PLbCDZQXIq7uYaf263gr-zb0wZGoCL-T5G. [Urtech University lectures].
- [3] Eric Lengyel. *Mathematics for 3d game programming and computer graphics*. Course Technology PTR, 2012.