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 БПМИ194 _____ А.В.Крупецков
Подпись 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 12.06 2021 _____ _____
Оценка (по 10-ти бальной шкале) Подпись

Москва 2021

Содержание

Аннотация

В этом проекте реализован простейший графический движок для отрисовки геометрических фигур с трехмерной сцены на двумерный пиксельный экран.

1 Введение

На сегодняшний день 3D-графика является

Основной задачей этого проекта является реализация библиотеки для отрисовки трехмерных объектов на плоском экране и приложения с интерактивным интерфейсом

В задачу входит:

1) Изучить теорию отображения 3d сцены на экран
2) Кратко изложить необходимую для понимания работы движка теорию в отчете
3) Написать интерактивное приложение по отображению 3d сцены на экран
4) Описать архитектуру проекта
5) Написать сопроводительную документацию

Необходимая теория для понимания работы графических движков и вывода изображений на экран изложена в книгах [2] и [1].

Ссылка на Github репозиторий - https://github.com/Sanches-wolf228/3D-renderer

2 Описание функциональных требований к проекту

Какие вещи программа умеет делать:
1) хранить 3d сцену
2) добавлять туда объекты
3) навигация камеры (вращение вниз-вверх, влево-вправо)
4) отрисовка сцены с помощью камеры на экран
5) отображение экрана пользователю
В программе должны быть реализованы следующие классы:

- Object
- World
- RusterScreen
- Camera
- Renderer
- Application

3 Описание нефункциональных требований к проекту

1) Язык программирования - C++17

2) Сторонние библиотеки: SFML для вывода плоского изображения GLM для матричных операций

3) Политика обработки исключений - программа может вызывать исключения, которые должна обработать вызывающая программа

4) Реакция на неправильный ввод - входные данные проверяются на корректность.

5) Оперативная память - 4 ГБ

6) Style guide - Clang Format

7) Для сборки нужно Visual Studio 2017

8) Система поддержки версий - git

4 Описание теории и методов

4.1 Общее описание алгоритма

В трехмерном пространстве в глобальной системе координат находятся объекты, представленные набором треугольников, которые мы хотим отрисовать.

В этом пространстве расположена камера, она является точкой в пространстве, через которую мы проецируем изображение на экран, также у камеры есть направление, в котором она смотрит, камера может вращаться в плоскости, параллельной OXY и по вертикали. Поэтому положение камеры в пространстве задается тремя параметрами: координатой v, углом поворота в горизонтальной плоскости $\varphi_1 \in [0; 2\pi]$ и углом поворота по вертикали $\varphi_2 \in [-\frac{\pi}{2}; \frac{\pi}{2}]$, как на рисунке 1

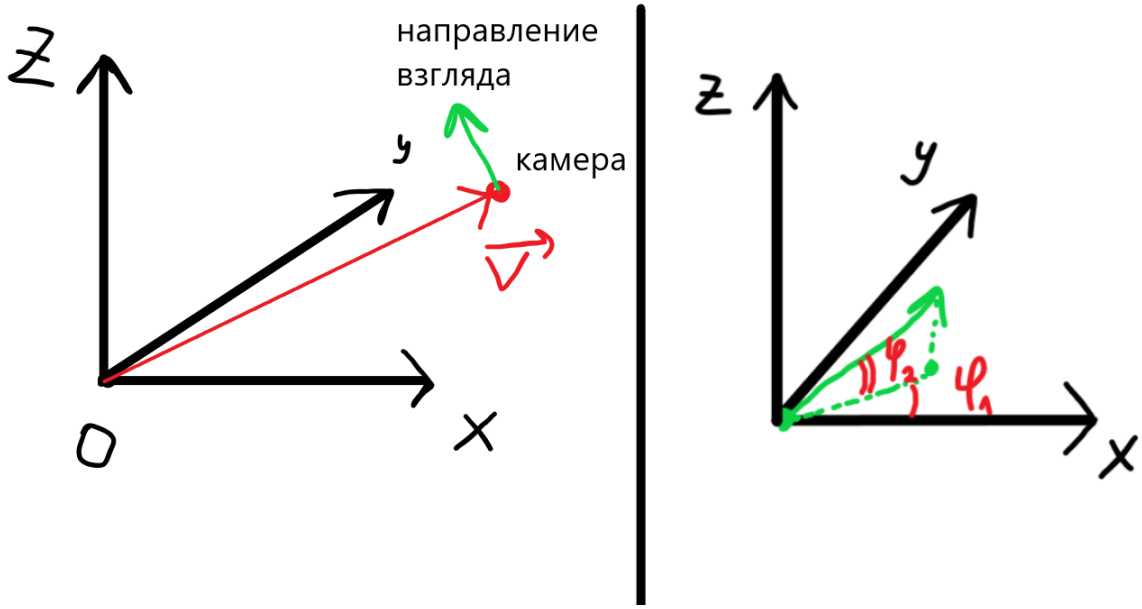

Рис. 1: Камера

Изображение фигур проецируется через камеру на экран. Объекты, которые мы видим, находятся внутри пирамиды зрения (см. рис 2) - это усеченная пирамида, кусок пространства, ограниченный передней и задней гранью. Все, что находится за её пределами мы не видим и все объекты обрезаются по ней, расстояние до передней грани (near plane) обозначим за n, а расстояние до задней грани (far plane) обозначим за f

Для того, чтобы спроецировать какую-то точку на экран, нужно провести отрезок из камеры в эту точку и взять точку пересечения экрана с этим отрезком. Но это можно посчитать проще - если выполнить проективное преобразование пространства, переводящее камеру на бесконечно удаленную прямую, после которого пирамида зрения перешла бы в куб, тогда проецирование на экран сводится к простому отбрасыванию глубины.

Для выполнения проективного преобразования нам понадобятся однородные координаты.

Точке $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ в трехмерном евклидовом пространстве соответствует координата $\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$, причем

$$\forall \omega \neq 0 \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x\omega \\ y\omega \\ z\omega \\ \omega \end{pmatrix}$$

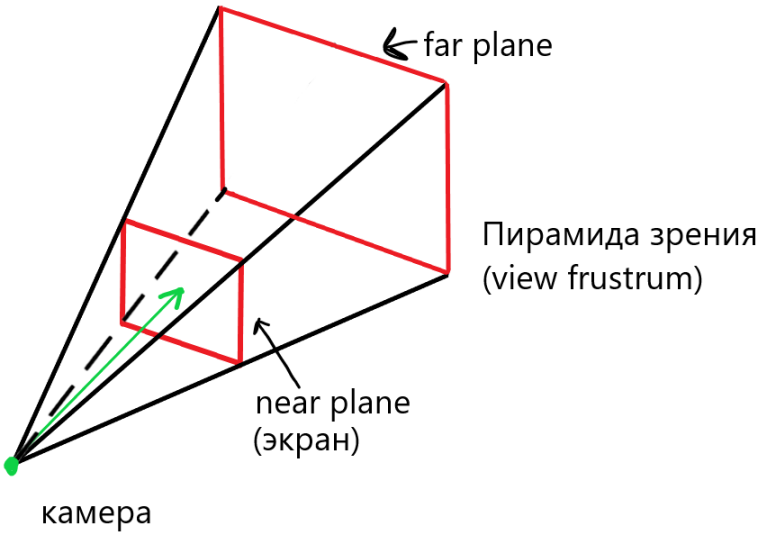

Рис. 2: Пирамида зрения

Так как разные объекты могут попадать в пирамиду зрения не целиком, а только частично, или не попадать вовсе, для каждого треугольника нужно совершить клиппинг - обрезку по границе пирамиды, получившаяся фигура после обрезки разбивается на треугольники и каждый треугольник отрисовывается отдельно. Пример клиппинга для двумерного случая - рисунок 3, здесь треугольник обрезается экраном и разбивается на 4 треугольника для отрисовки.

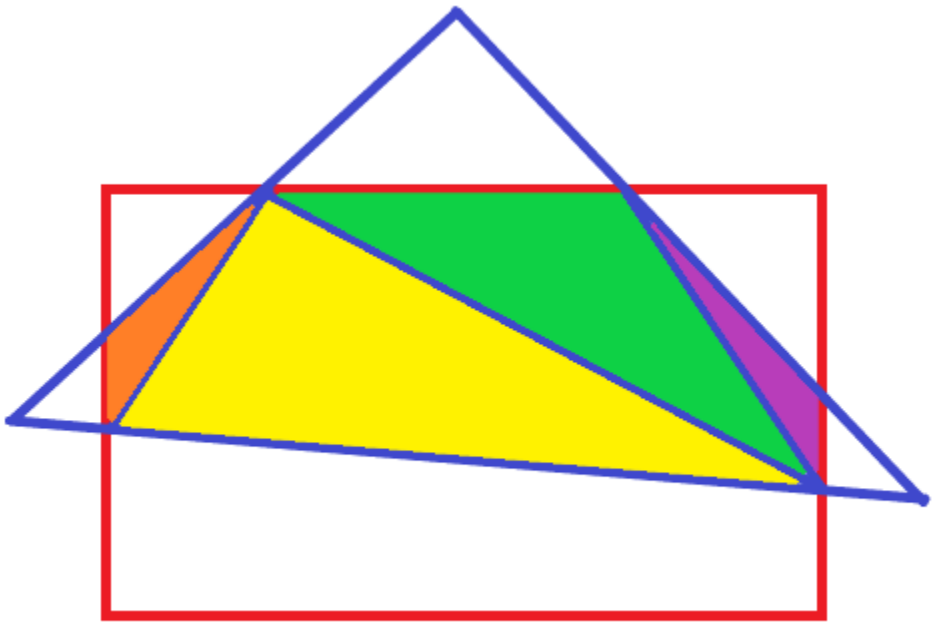

Рис. 3: Двумерный клиппинг

На данном этапе каждый треугольник, который мы хотим отрисовать, лежит внутри куба зрения, самое время заняться растеризацией - посчитать, какие пиксели экрана должны быть закрашены.

Заметим, что экранные координаты (которые соответствуют каждому отдельно взятому пикселю - см. рис 4) целые, так что координаты треугольника по осям X и Y округлим до ближайшего целого. Пример растеризованного треугольника - на рисунке 5. Здесь растеризуется треугольник с экранными координатами (2;1), (11;17), (17;5)

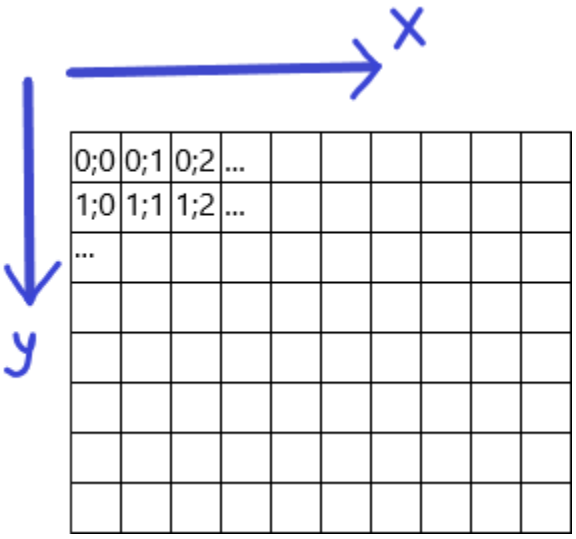

Рис. 4: Экранные координаты

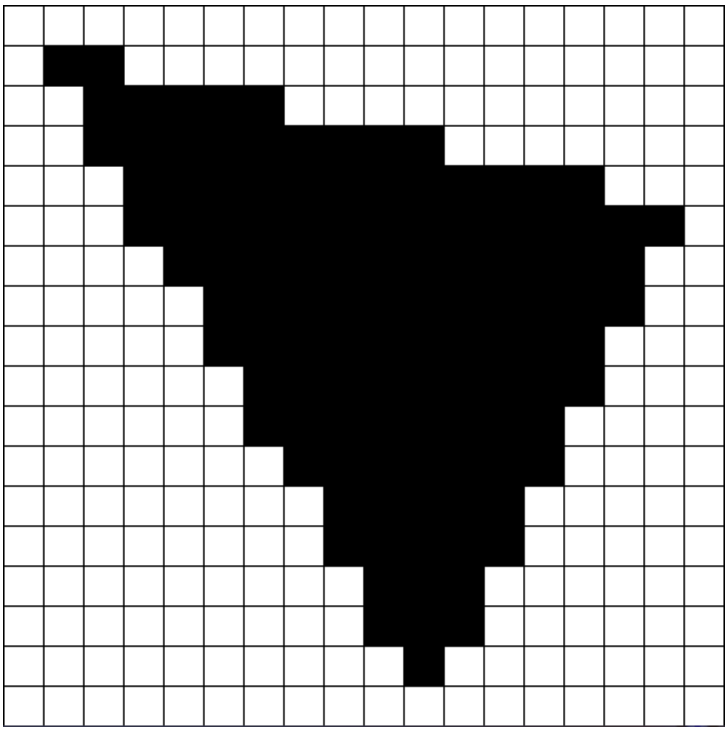

Рис. 5: Растеризация плоского треугольника

4.2 Перевод мира в систему координат, связанную с камерой

Вместо того, чтобы учитывать положение камеры и объектов в системе координат мира, мы будем считать положение объектов относительно камеры, которую мы хотим поместить в начало координат, и направить вдоль оси Z, то есть мы выполним движение пространства, такое, чтобы оно переводило координату камеры в начало координат, а вектор направления камеры (который задается двумя углами) переводило в положительное направление оси Z.

Посчитаем матрицу преобразования, помним, что мы используем однородные координаты, поэтому матрица имеет размеры 4 на 4.

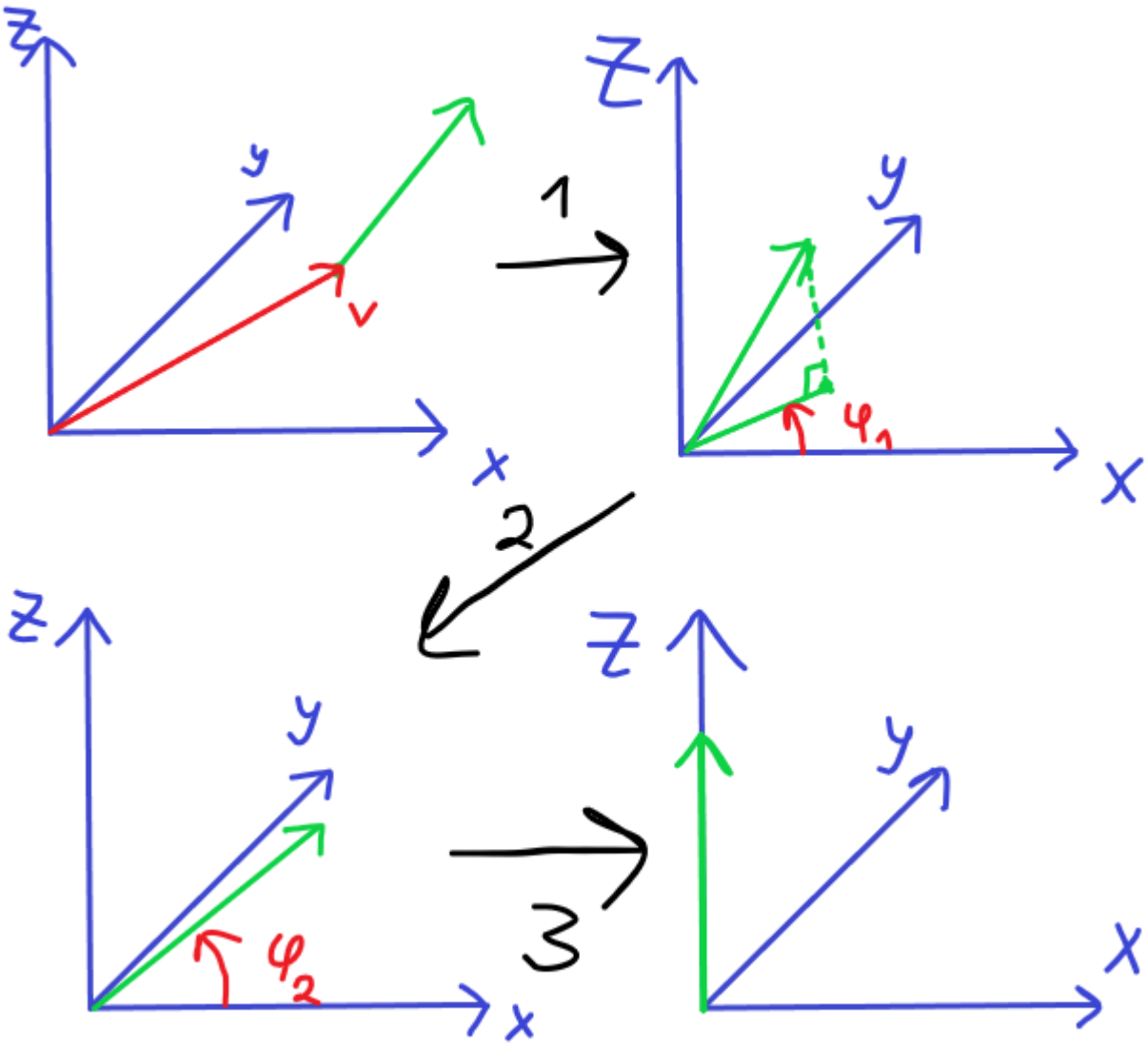

Рис. 6: Переход к системе координат камеры

1. **Сдвиг камеры в начало координат**

   Сначала мы выполняем параллельный перенос, прибавляющий к каждому вектору вектор -v, где $v = \begin{pmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{pmatrix}$ - координата камеры. Тогда камера перемещается в начало координат. Матрица для такой операции выглядит так:

$$\begin{pmatrix} 1 & 0 & 0 & -X_c \\ 0 & 1 & 0 & -Y_c \\ 0 & 0 & 1 & -Z_c \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x - X_c \\ y - Y_c \\ z - Z_c \\ 1 \end{pmatrix}$$

2. **Поворот в горизонтальной плоскости**

   Теперь нужно повернуть пространство вокруг оси Z на угол $-\varphi_1$, чтобы вектор направления камеры оказался на плоскости OXZ

   Матрица поворота на угол $-\varphi_1$ выглядит так:

$$\begin{pmatrix} \cos(-\varphi_1) & -\sin(-\varphi_1) & 0 & 0 \\ \sin(-\varphi_1) & \cos(-\varphi_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos(\varphi_1) & \sin(\varphi_1) & 0 & 0 \\ -\sin(\varphi_1) & \cos(\varphi_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3. **Вертикальный поворот**

   Осталось повернуть картинку вокруг оси Y на угол $\frac{\pi}{2} - \varphi_2$, чтобы вектор направления камеры оказался сонаправлен оси Z

   Матрица поворота на угол $\frac{\pi}{2} - \varphi_2$ выглядит так:

$$\begin{pmatrix} \cos(\frac{\pi}{2} - \varphi_2) & 0 & -\sin(\frac{\pi}{2} - \varphi_2) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\frac{\pi}{2} - \varphi_2) & 0 & \cos(\frac{\pi}{2} - \varphi_2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \sin(\varphi_2) & 0 & -\cos(\varphi_2) & 0 \\ 0 & 1 & 0 & 0 \\ \cos(\varphi_2) & 0 & \sin(\varphi_2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Итоговая матрица A_{camera}, переводящая мир в систему координат камеры

$$A_{camera} = \begin{pmatrix} \sin(\varphi_2) & 0 & -\cos(\varphi_2) & 0 \\ 0 & 1 & 0 & 0 \\ \cos(\varphi_2) & 0 & \sin(\varphi_2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} \cos(\varphi_1) & \sin(\varphi_1) & 0 & 0 \\ -\sin(\varphi_1) & \cos(\varphi_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & -X_c \\ 0 & 1 & 0 & -Y_c \\ 0 & 0 & 1 & -Z_c \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4.3 Проективное преобразование

После перехода к другой системе координат камера всегда находится в начале координат и смотрит в направлении оси Z. Теперь мы хотим выполнить проективное преобразование, которое нам позволит перейти к ортогональной проекции.

$$\begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = ?$$

На основании рисунка 7 разберемся, какие свойства от этого преобразования мы хотим получить

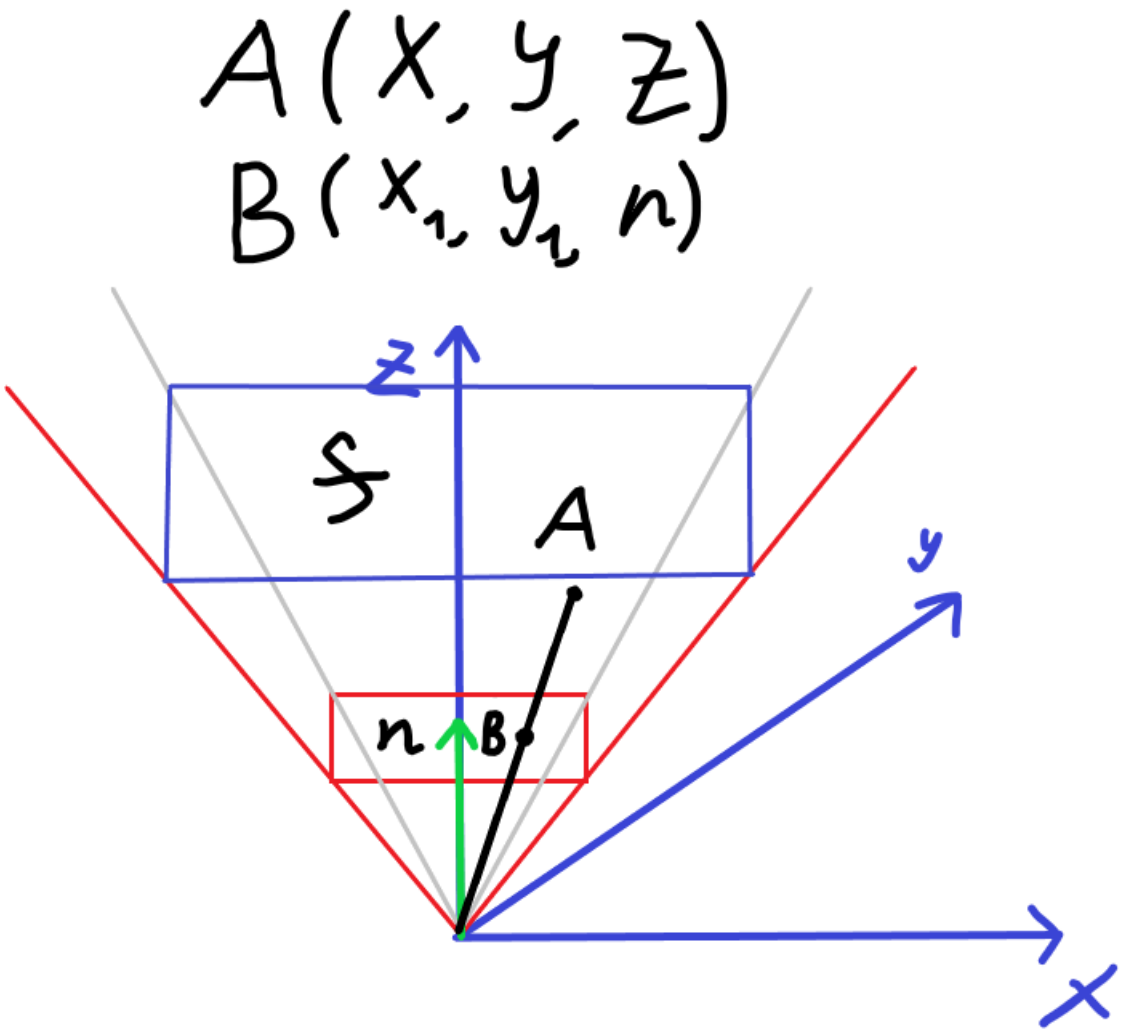

Рис. 7: Проективное преобразование

Ближняя грань находится на расстоянии n, а дальняя на расстоянии f, $f > n > 0$

Точка A лежит внутри пирамиды зрения (хотя, на самом деле принципиально, чтобы не в плоскости OXY) и имеет координаты $\{X; Y; Z\}$, а точка B лежит на пересечении прямой OA и ближней грани и имеет координаты $\{X_1; Y_1; n\}$

Мы хотим, чтобы точка A после преобразования проецировалось туда же, куда точка B, а точка B должна сохранить свое положение, так как она лежит на экране

Как можно заметить, точка B является образом точки A при гомотетии с центром в начале координат и коээфициентом $\frac{n}{Z}$, поэтому $B = \{X\frac{n}{Z}; Y\frac{n}{Z}; n\}$

Таким образом, можно записать такое условие:

$$\begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x\frac{n}{z} \\ y\frac{n}{z} \\ ? \\ 1 \end{pmatrix} = \begin{pmatrix} xn \\ yn \\ ? \\ z \end{pmatrix}$$

По таким условиям можно определить 3 из 4 строчек в матрице.

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ * & * & * & * \\ 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} xn \\ yn \\ ? \\ z \end{pmatrix}$$

Мы уже получили хорошее преобразование в том смысле, что оно правильно выполняет свою основную функцию - прямую, проходящую через начало координат, переводит в вертикальную. Так что осталось найти вид для третьей строчки. Рассмотрим плоскость вида $Z = const \in [n; f]$, она должна перейти в какую-то другую плоскость, которая тоже будет параллельна плоскости OXY (иначе на этой плоскости была бы точка с отрицательной z-координатой и мы бы ни при каком угле обзора не смогли увидеть её прообраз, что, очевидно, не так), из этого следует следующий факт - **координата Z_1 не зависит от x и y**

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} xn \\ yn \\ ? \\ z \end{pmatrix}$$

Тогда первые два числа в третьей строчке равны нулю, а на остальные 2 числа мы составим уравнение и решим его.

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix} = \begin{pmatrix} xn \\ yn \\ n^2 \\ n \end{pmatrix} = \begin{pmatrix} xn \\ yn \\ an+b \\ n \end{pmatrix}; \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} x \\ y \\ f \\ 1 \end{pmatrix} = \begin{pmatrix} xn \\ yn \\ f^2 \\ f \end{pmatrix} = \begin{pmatrix} xn \\ yn \\ af+b \\ f \end{pmatrix}$$

$$\begin{cases} an + b = n^2 \\ af + b = f^2 \end{cases}$$

$$a(n-f) = n^2 - f^2 => a = n + f$$
$$b = n^2 - an = n^2 - n^2 - nf = -nf$$

Итоговая матрица, полученная по камере:

$$B_{camera} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

4.4 Клиппинг

Обрезать треугольники мы будем на двух этапах:

- после перевода в систему координат камеры, но до проективного преобразования, в этот момент все треугольники можно обрезать по высоте, это нужно сделать, чтобы точки, близкие к плоскости OXY ничего нам не портили - так как точки в однородных координатах, нам приходится делить на z, а это плохо, когда z близко к нулю

- После проективного преобразования, но до растеризации - чтобы все треугольники, которые нам нужно было нарисовать, целиком помещались на экране.

Именно эти этапы были выбраны, так как в обоих случаях задача клиппинга по сути двумерная (на первом z-координаты константы, на втором они вообще не берутся в расчет)

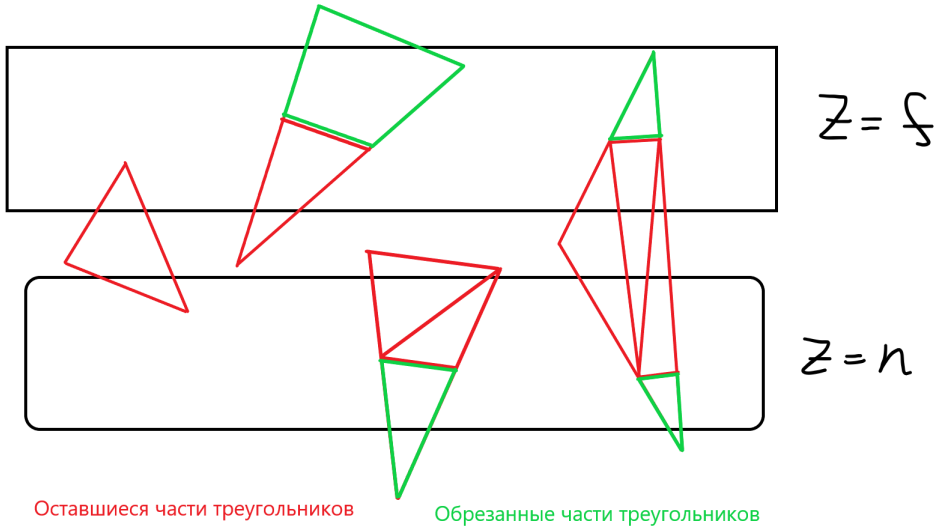

Рис. 8: Обрезка треугольников по высоте

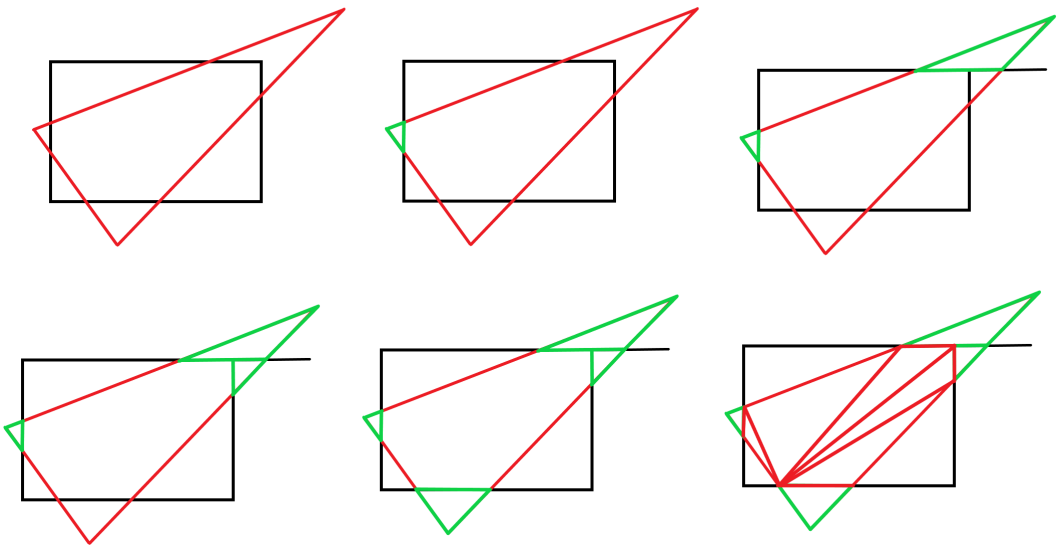

Рис. 9: Обрезка треугольников экраном

На рисунке 8 показаны возможные случаи разрезания треугольника, если он попал в область видимости по глубине.

На рисунке 9 показан пример разрезания треугольника при пересечении с экраном.

Алгоритм выглядит так - мы по очереди рассматриваем стороны экрана (или плоскости) и проверяем, пересекает ли каждый отрезок эту сторону, если пересекает - добавляем точки пересечения в массив. Когда получили все точки пересечения, обходим массив из точек, и из каждых трех соседних составляем треугольник, который отправится на следующий этап, то есть мы сделали обрезали треугольник, получили несколько треугольников поменьше, которые отправляем дальше на отрисовку (рис. 10)

Рис. 10: Что происходит после клиппинга

4.5 Растеризация треугольников с z-буфером

После обрезки треугольника по границам экрана мы сделаем несколько действий, которые упростят нам жизнь:

- У треугольника, отданного на растеризацию запомним цвет и запишем его в экранных координатах (целые координаты x и y, и глубина в виде числа с плавающим знаком), так мы обрежем лишнюю информацию.
- Отсортируем вершины по координате x экрана по возрастанию

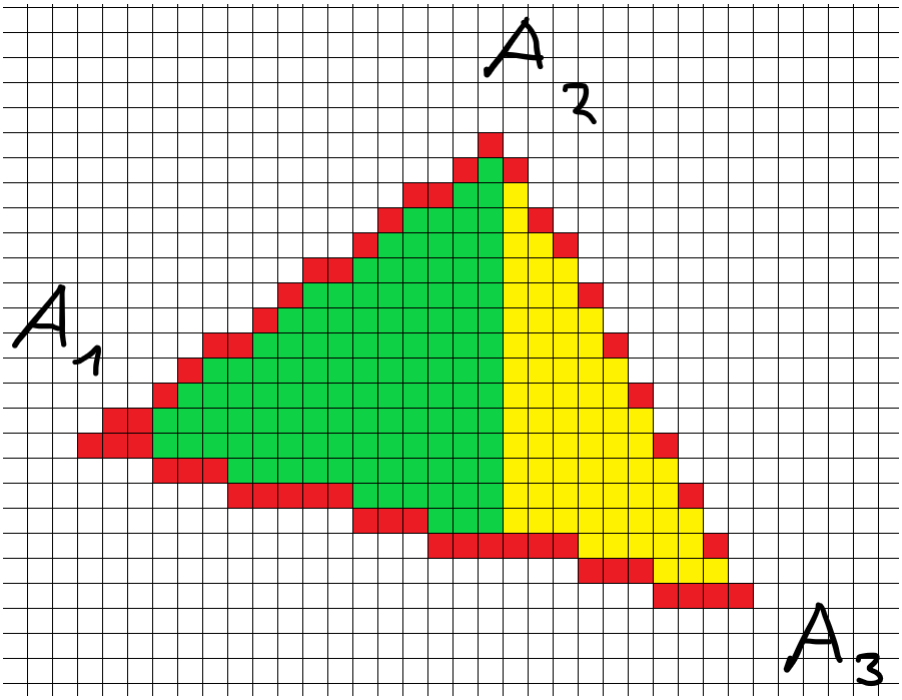

Рис. 11: Пример растеризации треугольника

На примере рисунка 11 покажем, как реализована растеризация с заполнением буфера глубины

$$A1.x \leq A2.x \leq A3.x$$

Мы проходим все иксы сначала от A1.x до A2.x, а потом от A2.x до A3.x и для каждого икса определяем, какие пиксели лежат внутри вертикальной полосы.

Мы определяем целые числа y_{up} и y_{down}, как показано на рисунке 12. Для каждого отрезка мы отрисовываем пиксели, начиная с y_{up}, если другой отрезок выше, или начиная с y_{down}, если он ниже соответственно.

Посчитаем коэффициенты для y в явном виде.

$\alpha = \frac{y_2 - y_1}{x_2 - x_1}$

$\beta = \frac{y_1 x_2 - y_2 x_1}{x_2 - x_1}$

Теперь как мы выбираем глубину каждого пикселя - сначала мы считаем глубину пикселей на границах нашего вертикального отрезка (см. рис. 13).

Здесь мы использовали формулу из геометрии масс, потому что масса на отрезке, так же, как и глубина, меняется линейно. Также мы использовали приближение, что точка лежит близко к отрезку (это так по нашему способу построения, расстояние по оси Y от точки до отрезка не больше $\frac{1}{2}$)

Далее у нас есть вертикальный отрезок, на концах которого задана глубина. Мы считаем глубину каждого промежуточного пикселя, используя эту же самую формулу, как на рисунке 13

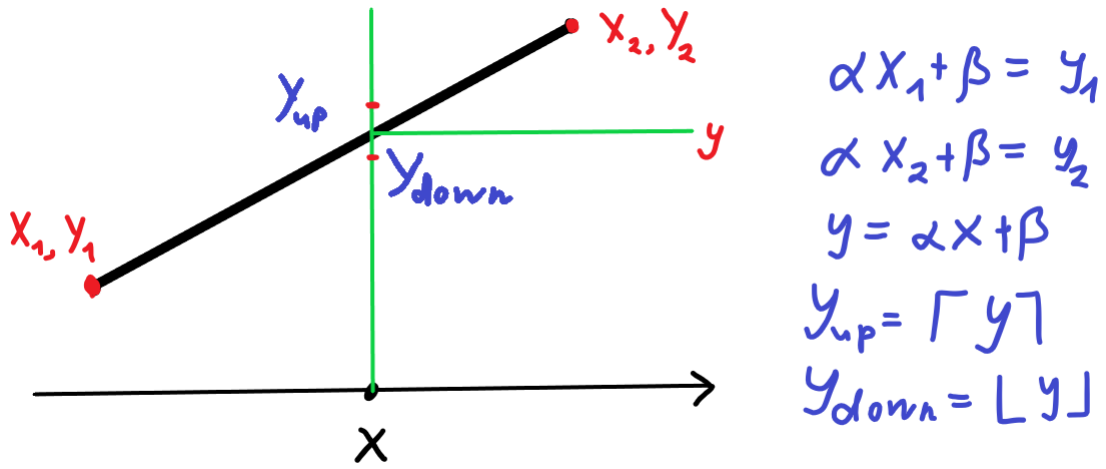

Рис. 12: Вертикальное сканирование

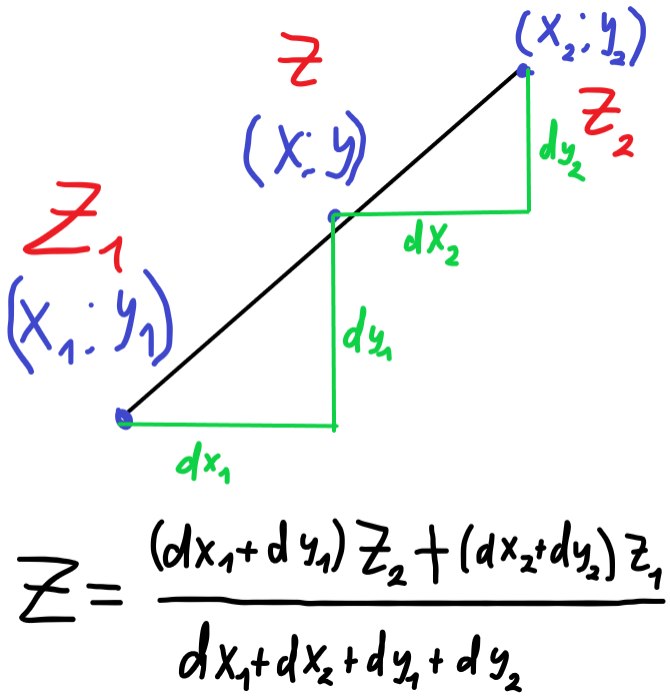

Рис. 13: Подсчет глубины

5 Архитектура проекта

- Screen

  Содержит информацию о пикселях - цвет и глубину

- figures.h

  Заголовочный файл, содержащий элементарные структуры, использующиеся в разных частях программы, в том числе треугольники

- Object

  Класс объекта на 3D сцене

- World

Класс, содержащий список объектов

- Camera

  Камера нужна для выбора точки и окна, через которое мы будем смотреть на мир

- Renderer

  Целиком берет на себя процесс отрисовки

- Application

  Класс, управляющий всеми компонентами программы

- Test

  Файлы с заранее написанными тестами для разных компонент программы.

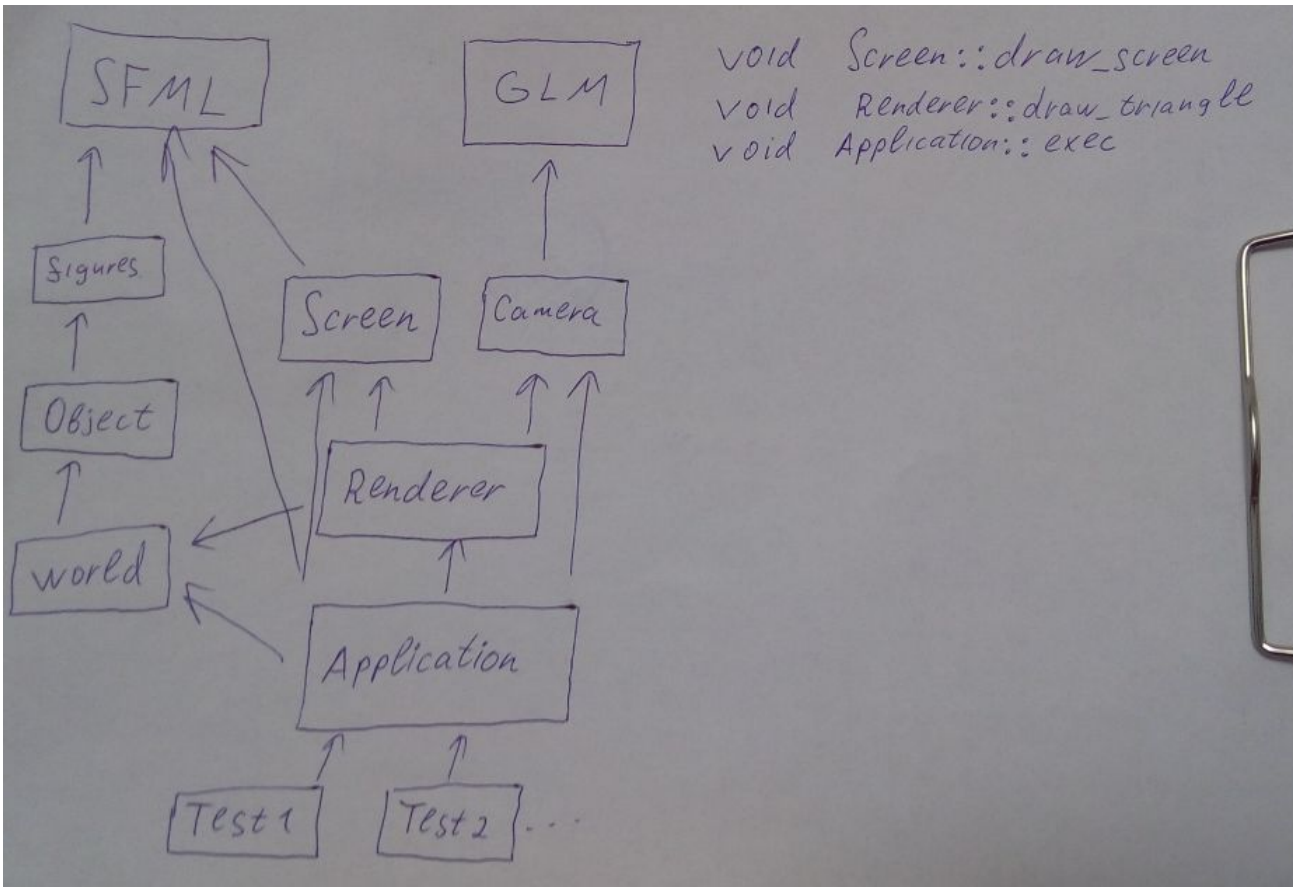

Рис. 14: Схема классов проекта

На структурной схеме классов проекта изображены зависимости, кто про кого знает 14

Заголовочный файл figures.h содержит структуры
struct Point { double coordinates[3] };
struct RasterPoint { int x; int y; double z };
struct RasterTriangle { RasterPoint vertices[3] };
struct Triangle { Point vertices[3]; sf::Color color };

class Object
владеет массивом треугольников, есть методы move() и rotate() для управления всеми треугольниками сразу

class World
владеет массивом из объектов содержит метод add(Object), добавляющий объект на сцену

class Screen (RasterScreen)
std::vector<sf::RectangleShape> pixels;
std::vector<double> z_buffer;
Содержит массив пикселей (которые являются прямоугольниками из библиотеки SFML) и z-буфер, имеет метод
draw_screen(sf::RenderWindow &window);
который отрисовывает в окне картинку

class Camera
Знает GLM, как средство работы с матрицами.
Внутри этого класса есть функции, вычисляющие матрицы, нужные для проективных преобразований и смены базиса

class Renderer
имеет метод
void draw_triangle(Triangle &source, RasterScreen &screen, const Camera &cam);
По камере, треугольнику и экрану отрисовывает на экране треугольник

class Application
имеет метод exec(), запускающий программу и работающий в цикле обработчика событий
Хранит мир, экран, камеру, рендерер, графическое окно SFML

6 Результаты работы программы

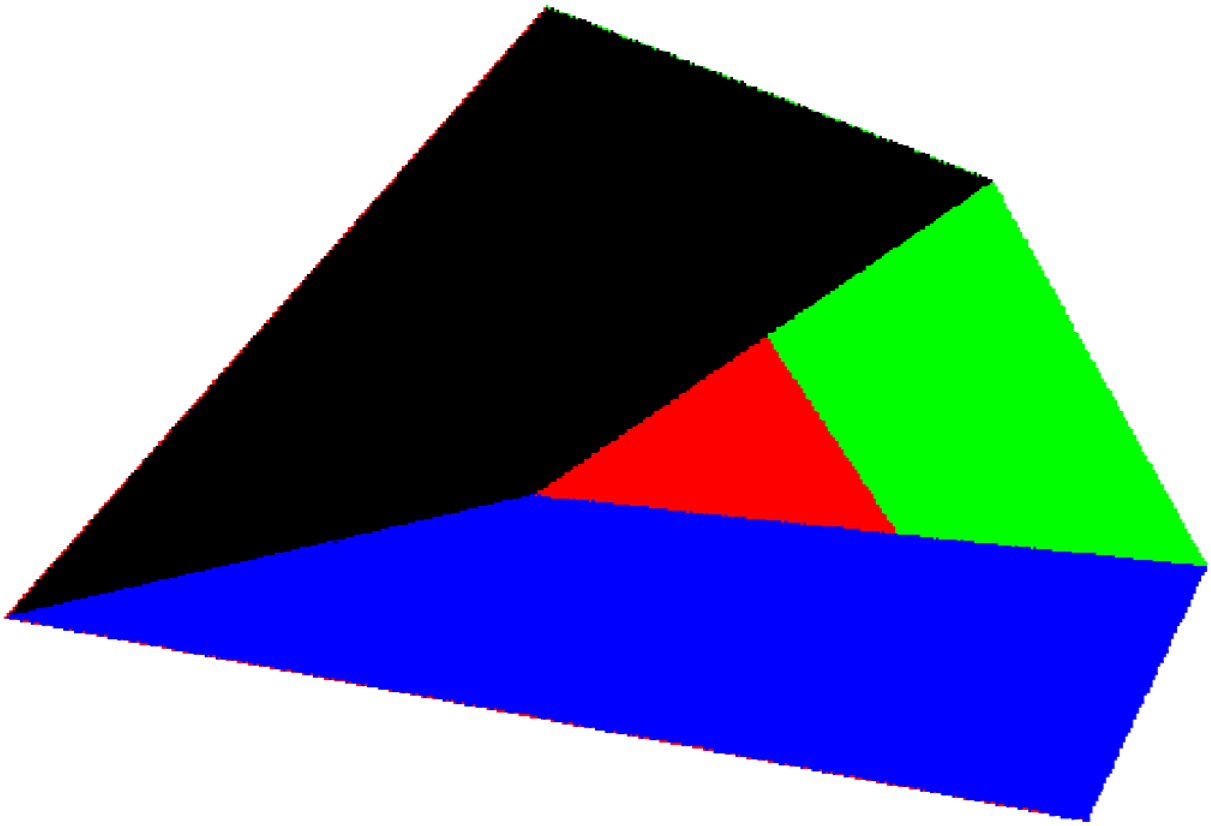

Рис. 15: Тетраэдр

На рисунке 15 можно видеть результат отрисовки тетраэдра.

Мы видим внешнюю поверхность черной и синей грани, и через срез видим внутреннюю поверхность зеленой и красной грани

Список литературы

[1] Samuel R. Buss. *3-D Computer Graphics.* Cambridge University Press, New York, 2003.

[2] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics, third edition.* Cengage Learning PTR, 2011.