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

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте (итоговый)

на тему: 3D renderer с нуля

Выполнил:
Студент группы БПМИ192 _____ А.А.Смородинов
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

Задача проекта - изучить и реализовать алгоритмы отрисовки трёхмерных объектов на экране, в итоге должна быть написано с нуля приложение, в котором пользователь может взаимодействовать с трёхмерной сценой, задавать различные режимы отрисовки для объектов сцены, менять параметры освещения, источников света, камеры и экрана в реальном времени. При этом программа должна иметь минимальное количество зависимостей от сторонних библиотек.

1 Введение

Основная задача проекта - это изучение и реализация основных алгоритмов, использующихся в компьютерной графике, на которых основаны большинство программ 3d моделирования, 3d игр, 3d/VR симуляторов и других приложений, имеющих какое-либо отношение к трёхмерной графике. Помимо этого необходимо изложить теоретические основы этих алгоритмов, описать детали их реализации, написать сопроводительную документацию к коду и протестировать его.

1.1 Алгоритмы, которые необходимо изучить и реализовать

1. Построение матрицы перехода из глобальной системы координат в пространство камеры
2. Построение матриц проекции на экран (перспективная и ортогональная проекции)
3. Удаление фрагментов треугольников, лежащих вне пирамиды зрения (view frustrum)
4. Проверка точек на глубину с помощью z-буффера (при отрисовке объектов должны отрисовываться только ближайшие объекты к камере, но не те, которые находятся за ними)
5. Отрисовка отрезков на экране (алгоритм Брезенхэма)
6. Отрисовка треугольников на экране
7. Перспективно правильная интерполяция параметров объектов при перспективной проекции

1.2 Что сделано в проекте

На данный момент все перечисленные выше алгоритмы реализованы и также написано приложение, в котором пользователь может управлять камерой, перемещаясь по 3d сцене.
В сцену можно добавлять 3d модели, загружая их с диска в формате wavefront obj [4].
Поддерживаются только модели с треугольными гранями, в .obj файле может храниться только один объект.
Загрузка материалов из .mtl файла не реализована, но для модели можно вручную выбрать карту нормалей и диффузную карту (normal map и diffuse map).

1.3 Репозиторий

Репозиторий проекта: [1]

2 Описание функциональных и нефункциональных требований к программному проекту

2.1 Функциональные требования

1. Результат проекта - интерактивное приложение, доступное на windows/linux, в котором пользователь может
   - просматривать различные 3d сцены, управляя камерой с клавиатуры
   - настраивать параметры сцены
   - открывать и добавлять в сцену 3d объекты, сохранённые в различных форматах (на данный момент реализована загрузка простейших 3D моделей в формате wavefront obj [4])
2. В коде должны быть реализованы следующие классы:
   - Object
   - World
   - Screen
   - Camera
   - Renderer

2.2 Нефункциональные требования

1. Требования к надёжности. При отсутствии файлов данных на диске, или их некорректном формате (файлы изображений, 3d моделей, использующихся приложением) программа сообщает пользователю об ошибке и завершает работу.
2. Требования к техническим средствам пользователя.
   - Операционная система - windows 10 (в будущем возможно будет добавлена поддержка linux).
   - Место на диске - сама программа занимает несколько мегабайт. Также необходимо место для файлов данных (3d модели, текстуры, шрифты) - зависит от отрисовываемой сцены, но для небольших сцен также достаточно несколько мегабайт.
   - Количество ядер процессора - хотя бы одно.
   - Объём оперативной памяти - хотя бы 1 гигабайт (сама программа требует несколько сотен мегабайт).
3. Требования к коду
   - Каждый класс и функция в программе должны быть документированы и протестированы
   - Между классами должно быть минимальное количество зависимостей, каждый класс должен выполнять свою конкретную функцию
   - Разрешается использовать библиотеку для кроссплатформенной отрисовки буфера кадра на экране (например SFML - [5]), библиотеку для работы с матрицами (glm - [2]) и библиотеку для загрузки изображений в различных форматах (stb_image - [6])
4. Язык программирования - C++
5. Используемые библиотеки (статическая линковка)
   - glm 0.9.9 [2]
   - sfml 2.5.1 [5]
   - stb_image 2.26 [6]
6. Система поддержки версий - git
7. Линтер/форматтер - clang format, настройки основаны на google codestyle
8. Для сборки проекта используется IDE Microsoft Visual Studio 2019

3 Краткое описание работы алгоритмов, применяющихся в проекте

Далее будет представлено сжатое описание работы 3d рендерера. Более подробную информацию можно найти в книгах [9] и [10], если читатель захочет ознакомится более глубоко с тематикой проекта.

Формат и структура объектов
В самом простом случае можно считать, что каждый объект сцены - это трёхмерное тело, которое мы аппроксимируем многогранником, или их объединением.

Каждый многогранник, который мы отрисовываем, будем просто считать набором многоугольников, т.е. сделаем переход от объёмного объекта к его двумерной оболочке. Более того, все многоугольники можно триангулировать, поэтому в нашей модели можно считать, что объекты сцены - это просто наборы треугольников.

На практике обычно разделяют понятия объекта и его 3d модели. Объект - это не только некоторая модель, но также её позиция и ориентация в пространстве, а также другие дополнительные атрибуты. Такой подход позволяет хранить только один экземпляр модели, даже если в сцене участвует несколько объектов, использующих её. В этом случае говорят, что координаты вершин треугольников 3d модели лежат в локальной системе координат.

В проекте объекты именно так и реализованы: каждый объект содержит в себе 3D модель (набор треугольников с текстурными координатами), свою позицию (трёхмерный вектор), ориентацию (кватернион), коэффициенты масштабирования по осям x,y,z.

Преобразование систем координат
Для того, чтобы получить координаты треугольников в глобальной системе координат, нужно применить к координатам в локальной системе координат несколько преобразований.

Применив линейный оператор к вектору в $\mathbb{R}^3$ (то есть домножив его на матрицу 3×3) мы можем выполнить гомотетию (растяжение/сжатие) вектора относительно нуля, а также повернуть его в пространстве произвольным образом относительно нуля, или отразить его относительно плоскости, проходящей через ноль. На самом деле, с помощью применения линейного оператора мы можем выполнить произвольное линейное преобразование, а не только перечисленные ранее и их комбинации. Но тем не менее, мы можем заметить, что такое преобразование, как, например, поворот относительно произвольного ненулевого вектора v_0 не является линейным (в векторном пространстве, не в аффинном). Для того, чтобы выполнить такое преобразование, можно сначала вычесть из вектора v_0, затем повернуть его относительно нуля и в конце добавить к результату v_0, в итоге мы получим оператор $B(v) = A(v - v_0) + v_0 = Av + (E - A)v_0 = Av + v_1$.

Данный формат преобразований не очень удобен тем, что в случае линейных операторов их композиция считается как просто произведение матриц, а здесь композиция операторов считается менее тривиально: $B_1(v) = A_1v + t_1$, $B_2(v) = A_2v + t_2 \Rightarrow C(v) = B_2(B_1(v)) = (A_2A_1)v + A_2t_1 + t_2$.

Обычно для преобразования координат используют более элегантный способ:

1. По трёхмерному вектору $v = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ строится вектор $v' = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$
2. По оператору $B(v) = Av + t$ строится матрица $B' = \begin{bmatrix} A & t \\ 0 & 1 \end{bmatrix}$
3. Тогда $B'v' = \begin{pmatrix} Av + t \\ 1 \end{pmatrix} = \begin{pmatrix} B(v) \\ 1 \end{pmatrix}$

Таким образом, мы свели произвольное перобразования $\mathbb{R}^3$ вида $B(x) = Ax + t$ к линейному оператору на $\mathbb{R}^4$, где координата w вектора (четвёртая координата) равна 1.

Переход в пространство камеры

После того, как мы получили координаты вершин треугольников в глобальной системе координат, их необходимо перевести в систему координат камеры, то есть нужно так преобразовать R^3, чтобы фокус камеры камеры перешёл в начало координат, а вектор направления камеры перешёл в вектор $(0, 0, -1)^T$ (можно было выбрать и другой вектор, это просто вопрос соглашения, главное чтобы он был единичной длины и перпендикулярен вектору, который мы считаем направленным вверх).

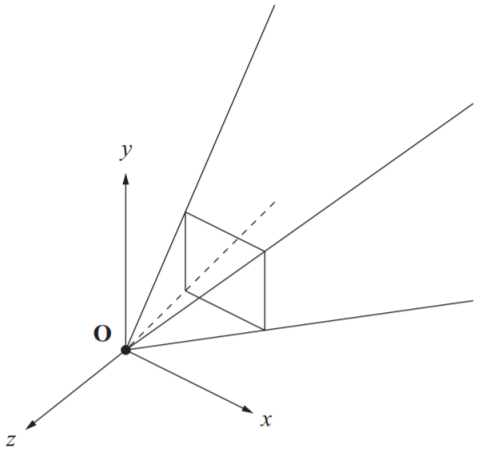


Figure 5.9. Camera space in OpenGL.

Рис. 1: Пространство камеры, картинка взята из [10]. Прямоугольник - это экран, на который будет проецироваться картинка, точка O - фокус камеры.

Здесь следует отдельно написать, что такое "вектор направления камеры". На самом деле, это просто вектор, который лежит на луче, соединяющем центр камеры с точкой (любой), которая будет проецироваться в центр экрана. Если мы нормализуем данный вектор, то, т.к. мы не хотим растяжения пространства в процессе перехода между системами координат, результирующий вектор также должен будет иметь длину 1.

Это преобразование является композицией поворота и параллельного переноса, поэтому его можно выразить в виде матрицы 4×4 как в прошлом абзаце. Коэффициенты этой матрицы можно посчитать явно, в библиотеке *glm* [2] эту матрицу можно получить с помощью функции $glm :: lookAt(eye, center, up)$, где eye - фокус камеры, $center$ - точка, в которую направлена камера ($center = eye + direction$), up - вектор, который мы считаем направленным вверх, в нашем случае это $(0, 1, 0)^T$.

Теперь, когда у нас есть координаты вершин треугольников в пространстве камеры, осталось лишь спроецировать их на плоскость экрана.

Проецирование на экран

Всего существует два наиболее распространённых вида проекций - ортографическая и перспективная. В случае ортографической проекции мы просто отбрасываем координату z и получаем в результате вектор $(x, y)^T$ в пространстве экрана. После этого, если мы хотим нормализовать координаты x, y, то есть преобразовать их из отрезков $[l, r], [b, t]$ в отрезок $[-1, 1]$, то это можно сделать по формулам $x' = \dfrac{2x - (l + r)}{r - l}$, $y' = \dfrac{2y - (b + t)}{t - b}$

Перспективная проекция

В случае перспективной проекции, для того, чтобы спроектировать точку
$v = (x_0, y_0, z_0)^T$ на двумерный экран, нужно найти пересечение луча $\{tv, t \geq 0\}$ с экраном, задающимся плоскостью $z = -n$ (напоминаю, что в нашей модели направление камеры - это $(0, 0, -1)^T$).

Также, для того чтобы ограничить возможные значения глубины z, вводится так называемая "дальняя плоскость", задающаяся уравнением $z = -f$. Все точки, находящиеся за этой плоскостью, то есть имеющие $z < -f$ не будут отрисовываться на экране. Ограничение диапазона возможных значений z позволит нормализовать z и эффективно хранить его с хорошей точностью, при этом расходуя небольшое количество памяти. Зачем вообще нужна координата z станет понятно дальше.

Формулы для вычисления перспективной проекции

Пусть $(x', y', z')^T \in \{(x, y, z)^T \mid z = -n\} \cap \{tv \mid t \geq 0\}$.

Тогда $x' = \frac{-n}{z_0}x_0$, $y' = \frac{-n}{z_0}y_0$, $z' = \frac{-n}{z_0}z_0 = -n$.

Координаты x', y' - это и будут спроецированные координаты точки v на экране. Единственное, что остаётся сделать с ними - это нормализовать, то есть перевести в диапазон $[-1, 1]$. Это нужно просто для удобства дальнейшей работы.

Если мы хотим преобразовать x из диапазона $[a, b]$ в $[-1, 1]$, то это делается с помощью простейшего преобразования: $x' = 2\frac{x-a}{b-a} - 1$

В итоге мы получаем следующие формулы для x', y', если предположить, что экран камеры - это прямоугольник $[l, r] \times [b, t] \times \{-n\} \subseteq \mathbb{R}^3$:

$x' = \frac{-2n}{r-l}(\frac{x_0}{z_0}) - \frac{r+l}{r-l}$

$y' = \frac{-2n}{t-b}(\frac{y_0}{z_0}) - \frac{t+b}{t-b}$

Помимо координат x', y' для отрисовки треугольника нам понадобится также координата z_0, которая необходима например для того, чтобы определять, какая точка должна отрисовываться на экране, в случае если точек с одинаковыми экранными координатами (полученными после преобразования x', y' в диапазон $[0, screen.width)$, $[0, screen.height)$ и округления их до целого числа) несколько.

Интерполяция координаты z (мотивация для последующих формул)

Здесь нам придётся забежать немного вперёд, и поговорить об интерполяции. После того, как мы получили координаты x', y' всех вершин треугольника в экранных координатах, мы уже имеем часть необходимых данных, необходимых для отрисовки треугольника на экране. Как было написано выше, нам также понадобится координата z_0 для каждой вершины треугольника.

Треугольник, который мы хотим отрисовать на экране, в результате будет преобразован просто в некоторый набор пикселей. Как уже было написано выше, чтобы понять, какие пиксели реально видимы для камеры, а какие - нет, необходимо знать глубину каждого пикселя треугольника. Но как определить эту глубину? Оказывается, что для того, чтобы найти координату z точки в треугольнике, достаточно знать её барицентрические координаты (об этом и будет следующий абзац) и координаты z вершин треугольника.

Барицентрические координаты

Определение 1. Барицентрическими координатами точки $M = (x, y, z)^T$ в треугольнике ABC, где $A = (x_0, y_0, z_0)^T, B = (x_1, y_1, z_1)^T, C = (x_2, y_2, z_2)^T$, называется такая тройка чисел (a, b, c), что $M = aA + bB + cC$, $a, b, c \in [0, 1]$, $a + b + c = 1$

Лемма 2. *(без доказательства)*
$M = (x, y, z)^T \in \triangle ABC$, $A = (x_0, y_0, z_0)^T, B = (x_1, y_1, z_1)^T, C = (x_2, y_2, z_2)^T$, *тогда* (a, b, c) - *бар. координаты точки* $M \Rightarrow z = az_0 + bz_1 + cz_2$.

Т.е. для того, чтобы вычислить координату z точки M нам достаточно знать её барицентрические координаты (по координатам пикселя внутри треугольника барицентрические координаты считаются несложно, здесь эта часть вычислений опущена) и координаты z_0, z_1, z_2 вершин треугольника.

Этот подход прекрасно работает в векторных пространствах, но, к сожалению, после проецирования метрика пространства меняется, поэтому z по данной формуле уже нельзя будет считать расстоянием (хотя всё ещё можно использовать для z буфера). В некоторых ситуациях полезно знать расстояние от камеры до объекта, поэтому напрямую это формулу применить нельзя, но есть её альтернатива, которая приводится здесь без доказательства:

$1/z = a/z_0 + b/z_1 + c/z_2$

Как мы видим, после взятия проекции нужно линейно интерполировать не z а $1/z$

Формулы для вычисления перспективной проекции (продолжение)

Как мы только что узнали, в процессе растеризации будет необходимо линейно интерполировать $1/z$, поэтому, мы хотим преобразовать z из диапазона $[-f, -n]$ в диапазон $[-1, 1](-n \to -1, -f \to 1)$ (то есть нормализовать) не линейно, а так, чтобы $z' = \frac{a}{z} + b$, это позволит интерполировать значения глубины линейно.

Заметим, что x', y', z' зависят от $x, y, 1/z$ следующим образом: $x' = ax/z + b$, $y' = cy/z + d$, $z' = e/z + g$
Пусть $v' = (v'_0, v'_1, v'_2, v'_3)^T = (-x'z, -y'z, -z'z, -z)^T$

Тогда $-x'z = -ax - bz$, $-y'z = -cy - dz$, $-z'z = -e - gz \Rightarrow v' = \begin{bmatrix} -a & 0 & -b & 0 \\ 0 & -c & -d & 0 \\ 0 & 0 & -g & -e \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} =$

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

Но тогда $x' = v'_0/v'_3, y' = v'_1/v'_3, z' = v'_2/v'_3$.

Резюме: для того, чтобы перевести точку из системы координат камеры в систему нормализованных экранных координат необходимо:

1. Дополнить вектор $(x, y, z)^T$ ещё одной координатой - единичкой. В итоге получаем вектор $v = (x, y, z, 1)^T$
2. Домножить вектор v слева на матрицу M: $v' = Mv$
3. Поделить первые три координаты вектора v' на четвёртую координату.
4. Первые три координаты вектора v' и будут искомыми нормализованными координатами x', y', z'.
5. При этом $v'_3 = -z$ нам понадобится позже. (z - координата z точки в пространстве камеры, v'_3 - последняя координата вектора v', нумерация с нуля)

После этого шага мы переводим x', y' в экранные координаты.

Растеризация треугольника

Теперь, когда у нас есть экранные координаты вершин треугольников, мы должны растеризировать треугольники, то есть отрисовать на экране пиксели, лежащие внутри них. В проекте алгоритм, растеризующий треугольник выглядит так:

1. Сначала находим квадрат, в котором лежит треугольник. Левый нижний угол квадрата - это просто $(\min(x_0, x_1, x_2), \min(y_0, y_1, y_2))$. Правый верхний - $(\max(x_0, x_1, x_2), \max(y_0, y_1, y_2))$.
   Также введём некоторые обозначения:
   - Вершины треугольника имеют координаты $(x_0, y_0, z_0), (x_1, y_1, z_2), (x_2, y_2, z_2)$, где x, y - в экранных координатах, а z - в нормализованных (экранные координаты отличаются от нормализованных тем, что экранные лежат в диапазоне $[0, screen.width - 1]$, или $[0, screen.height - 1]$, а нормализованные - $[-1, 1]$). Ось x направлена вправо, ось y - вверх.
   - Барицентрические координаты точки внутри треугольника будем обозначать как (α, β, γ).
   - Координаты z вершин треугольника в пространстве камеры (см. пункт 5 пред. параграфа) будем обозначать как (w_0, w_1, w_2).
2. После этого для всех пикселей внутри квадрата нужно проверить, лежат ли они внутри треугольника (это можно сделать например используя знаковое расстояние от пикселя до прямых, содержащих стороны треугольника).
3. Если пиксель лежит вне треугольника, то мы его пропускаем.
4. Вычисляем глубину пикселя z с помощью барицентрических координат точки, которой соответствует пиксель, и координат z_0, z_1, z_2 (по формуле, находящейся в конце параграфа про барицентр. координаты)
5. Сравниваем глубину пикселя с глубиной, записанной в z-буфере (это просто матрица минимальных значений z для каждого пикселя, где минимум берётся по пикселям всех треугольников, отрисованных в текущем кадре до настоящего момента), и если глубина больше чем в z-буфере (точка находится за какой-то другой), то тоже пропускаем данный пиксель.
6. Если мы хотим отрисовать не одноцветный треугольник (а например текстурированный), то все необходимые параметры для вычисления цвета пикселя тоже нужно интерполировать.

7. Перспективно-правильная интерполяция. (пример)

   - На данном этапе алгоритма у нас уже есть достаточно много информации о пикселе треугольника, который мы хотим отрисовать. Нам известны экранные координаты пикселя(x, y), а также расстояние от точки, которой этот пиксель соответствует, до камеры (z). Мы уже ответили на вопрос "где отрисовывать пиксель?", но пока ещё не знаем, "как его отрисовывать?", а именно, каким цветом.
   - Если считать, что все треугольники в мире являются одноцветными, то определить цвет пикселя очень легко, это просто константа для каждого треугольника.
   - Хотя такое предположение упрощает реализацию алгоритма растеризации, оно всё же достаточно сильно ограничивает возможности 3D рендерера.
   - Пусть, вместо этого, вместе с каждым треугольником мы будем хранить не только его положение в пространстве, но и так называемые "текстурные координаты вершин". Текстурными координатами вершины будет просто являться пара чисел $(u_i, v_i) \in [0, 1]^2$.
   - Тогда, зная текстурные координаты вершин треугольника, мы сможем "наложить" текстуру (2D изображение) на наш треугольник. Более формально, каждой точке внутри треугольника можно сопоставить точку внутри текстуры (а значит и её цвет), так чтобы вершине i треугольника соответствовала точка на текстуре с координатами (u_i, v_i).
   - За счёт данной техники, мы сможем отрисовывать не только одноцветные треугольники, но и произвольные цветные изображения, наложенные на них. Это позволит с использованием относительно небольшого числа треугольников создавать достаточно реалистично выглядящие и детализированные 3D модели.
   - Если немного подумать, то можно придумать достаточно простую формулу, по которой точки треугольника будут переходить в точки текстуры. Так как мы хотим построить взаимооднозначное соответствие между двумя треугольниками, то кажется довольно разумным просто взять за инвариант барицентрические координаты, так как они линейно меняются внутри треугольника, и по ним можно определять, где находится точка относительно вершин треугольника. В итоге формула бы выглядела так: $(u, v) = \alpha(u_0, v_0) + \beta(u_1, v_1) + \gamma(u_2, v_2)$.
   - К сожалению, такой подход не учитывает искажения перспективной проекции: объекты, находящиеся дальше от камеры на экране становятся меньше, но в треугольнике мы интерполируем текстурные координаты линейно. Например, пусть есть треугольник, одна из вершин которого уходит далеко в бесконечность (ну или просто достаточно далеко). Тогда, по нашей формуле, центр треугольника (пер. медиан) находился бы в центре спроект. треугольника, что показалось бы наблюдателю неестественным, так как этот центр на самом деле расположен очень далеко от нас (а значит должен быть близко к центру экрана, точке схода). К сожалению, мои ограниченные писательские способности плохо позволяют описать этот эффект, с этим гораздо лучше справляется картинка.

Рис. 2: Источник: [3].

   - Правильная интерполяция текстурных координат будет вычисляться по формуле $(u, v) = w \cdot (\alpha(u_0, v_0)/w_0 + \beta(u_1, v_1)/w_1 + \gamma(u_2, v_2)/w_2)$, данный факт приводится без доказательства.
   - Теперь, зная текстурные координаты точки (u, v), мы можем отрисовать на экране пиксель тем цветом, которому соответствует эта точка на текстуре.
   - В общем случае, цвет пикселя может определяться не только текстурой, но и различными параметрами освещения (например источниками света), параметрами материала, внешние свойства которого мы хотим передать, и другими вещами. За вычисление цвета отвечает шейдер - алгоритм (программа), который по входным данным (константам и параметрам, интерполирующимся внутри треугольника) определяет цвет пикселя.

8. Вычисленный цвет пикселя записывается в буфер цветов (матрица цветов пикселей).

9. Глубина z пикселя записывается в z-буфер.

В результате всех этих операций в итоге мы получим буфер цветов, который и будет выведен на экран.

4 Детали реализации приложения

4.1 Основные классы

`Application`
Основной класс приложения - это `Application`.
Он хранит в себе классы:

- `SFMLRenderer` - 3d рендерер
- `UserInterface` - отвечает за вывод на экран отладочной информации
- `eng::Scene` - сцена, которая хранит в себе 3d модели, источники света и камеру
- `sf::RenderWindow` - SFML окно, на которое отрисовывается 3d сцена и пользовательский интерфейс
- `eng::CameraControl` - класс, отвечающий за перемещение камеры пользователем с помощью ввода с клавиатуры

У класса `Application` есть функция `void Application::run()`, которая запускает основной цикл работы, в котором в каждом кадре:

- средствами `SFML` [5] обрабатывается ввод пользователя
- вызывается метод `size_t SFMLRenderer::render(eng::Scene& scene)` у поля `renderer_`, который отрисовывает сцену на экране и возвращает количество растеризованных треугольников.
- собирается и выводится на экран различная статистика о работе программы (например количество кадров в секунду) с помощью класса `UserInterface`

В итоге, для запуска приложения достаточно подключить в файле `main.cpp` header `"Application.h"` в функции `void main()` создать класс `Application` и вызвать у него функцию `void Application::run()`.

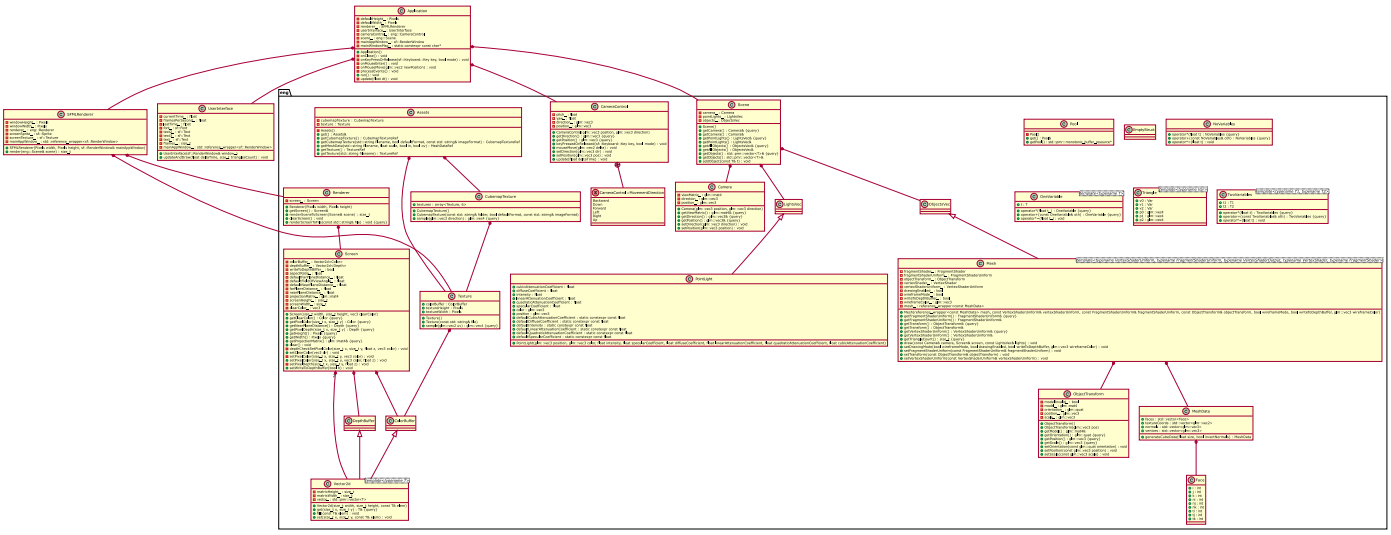
Как можно видеть, класс `Application` занимается специфическими для конкретного приложения задачами (обработка ввода пользователя, работа с `SFML` [5], создание сцены и т.д.), а собственно отрисовку 3D сцены он делегирует классу `SFMLRenderer`

`SFMLRenderer`
Класс, отвечающий за отрисовку сцены на SFML окне.
По сути это просто обёртка над классом `eng::Renderer`, умеющая отрисовывать буфер экрана `eng::Screen` на SFML окне.

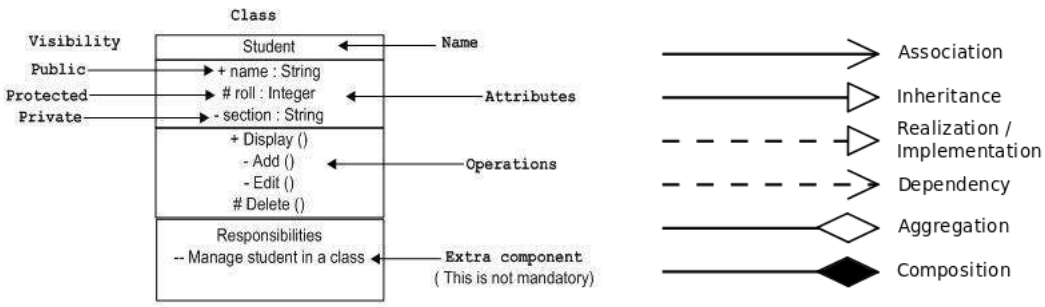
`eng::Renderer`
Класс, ничего не знающий о приложении, которое будет его использовать и конкретном графическом API, используемом для работы с оконной системой. Он умеет отрисовывать объекты сцены (то есть 3d модели, в программе это шаблонный класс `eng::Mesh`) на экране (`eng::Screen`).

`eng::Scene`
Класс, в котором хранятся объекты сцены (`eng::ObjectsVec`), источники света (`eng::LightsVec`), камера `eng::Camera`

4.2 Структурная диаграмма классов UML



4.2.1 Легенда и краткое описание элементов диаграммы



Картинки взяты отсюда [7] и отсюда [8]. В источнике [8] достаточно подробно описывается формат UML, если читатель хочет узнать больше про данный формат.

4.3 Диаграмма потока данных

4.3.1 Общая концепция

Все ресурсы, необходимые для работы программы (текстуры в форматах .png, .jpg, 3d модели в формате .obj, true type шрифт в формате .ttf), загружаются с жёсткого диска в оперативную память. Пользователь взаимодействует с программой с помощью мыши и клавиатуры, программа обрабатывает ввод пользователя и в соответствии с ним обновляет изображение на экране пользователя (на данный момент пользователь может переключать режимы отрисовки объектов, а также управлять камерой).

4.3.2 Поток данных в каждом кадре

В каждом кадре программы происходит примерно следующее:

1. Класс `Application` обрабатывает события, приходящие от SFML (ввод с клавиатуры, перемещение курсора).

2. Класс `Application` вызывает у класса `SFMLRenderer` метод `size_t render(eng::Scene& scene)` для отрисовки сцены (сцена как можно видеть передаётся по ссылке).

3. В результате `SFMLRenderer` вызывает у своего поля `eng::Renderer` метод `size_t renderSceneToScreen(Scene& scene)`, отрисовывающий сцену на экране (сцена снова передаётся по ссылке).

4. После этого он копирует экран `eng::Screen` попиксельно в текстуру в формате, поддерживаемом SFML, и отрисовывает её на окне SFML.

5. Класс `Application` вызывает у класса `UserInterface` метод `void updateAndDraw(Seconds deltaTime, size_t trianglesCount)` (никакие данные не копируются).

Как мы видим, данные в программе почти никогда никуда не копируются (кроме копирования экрана в SFML текстуру), а в большинстве случаев просто передаются по ссылкеь.

5 Тесты на производительность

На моём компьютере (процессор intel pentium 4415U, 2.3 GHz, ядра) программа в разрешении 1200x800 в среднем отрисовывает 1 кадр за 200 миллисекунд на сравнительно небольшой сцене с 7400 треугольниками.

Список литературы

[1] github репозиторий проекта. URL: https://github.com/asmorodinov/3d-renderer-from-scratch/tree/dev.

[2] glm. URL: https://github.com/g-truc/glm.

[3] image 3 source. URL: https://web.cs.ucdavis.edu/~amenta/s12/perspectiveCorrect.pdf.

[4] obj. URL: https://en.wikipedia.org/wiki/Wavefront_.obj_file.

[5] Sfml. URL: https://www.sfml-dev.org/.

[6] stb_image. URL: https://github.com/nothings/stb/blob/master/stb_image.h.

[7] uml image 1 source. URL: https://coderlessons.com/tutorials/akademicheskii/uchit-uml/uml-osnovnye-notatsii.

[8] uml image 2 source. URL: https://www.wikiwand.com/en/Class_diagram.

[9] Samuel R. Buss. *A Mathematical Introduction with OpenGL.* Cambridge University Press, 2003.

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