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

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте

на тему: 3D рендерер с нуля

(промежуточный, этап 1)

Выполнил:
Студент группы БПМИ213 _____ Д.С.Бонич
Подпись И.О.Фамилия

14.02.2023
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 14.02. 2023 _____ _____
Оценка (по 10-ти бальной шкале) Подпись

Москва 2023

Содержание

Аннотация

Главной целью данной работы является написание своего собственного 3D-рендерера – программы для отображения трехмерных объектов на плоский экран компьютера.

1 Введение

Рендеринг (англ. rendering – «визуализация») — это процесс получения изображения из модели, его строгого описания. Часто под рендерингом подразумевают 3D-рендеринг, где по описанию объектов из $\mathbb{R}^3$ строится плоское изображение, готовое к отображению на плоском экране компьютера. Программа осуществляющая ренедеринг называется рендерером, а результат её работы рендером. В данной работе мы рассмотрим только 3D-рендеринг и 3D-рендереры.В рендеринге можно выделить два основных типа: real-time и offline рендеринг. Real-time рендеринг подразумевает быстрый рендеринг изображения, достаточный для создания эффекта движения на экране. Этот тип используется в видеоиграх, где важно быстро рендерить постоянно меняющуюся модель мира. Offline рендеринг может требовать минуты, часы или даже дни для производства рендера, но как правило это оправдано высоким качеством финального изображения. Нас будет интересовать real-time рендеринг. При реализации рендеринга на вычислительных устройствах существует 2 принципиально разных подхода. Можно осуществлять рендеринг с помощью видеокарты, а можно использовать только компьютерный процессор. Первый способ является наиболее распространенным для real-time рендеринга. Причиной этому служат специфичные потребности алгоритмов осуществляющих рендеринг. Как правило требуется запускать небольшие программы для каждого пикселя. В компьютерной графике такие небольшие программы, которые можно выполнять параллельно, называют *шейдерами*. Так как видеокарта имеет большое количество слабых ядер, она является идеальным кандидатом для выполнения шейдеров и сильно выигрывает в производительности у процессора. Не смотря на это у второго подхода есть свои преимущества. Такой способ рендеринга называется *программным рендерингом* (software rendering) и может быть использован для применения специфичных алгоритмов визуализации, которые по каким-то причинам сложно или невозможно реализовать на существующих видеокартах. Это обусловлено тем, что многие видеокарты фиксируют некоторую часть алгоритмов входящих в рендеринг, не позволяя их поменять. Мы будем использовать программный рендеринг, так как хотим реализовать весь пайплайн рендеринга с нуля. Можно добавить еще разделение рендереров по основному принципу с помощью которого осуществляется рендеринг. Есть методы рассматривающие лучи света, а есть *растеризация*. Наш рендерер будет использовать растеризацию. Это значит что изначально мы не будем опиратся на свет и будем строить все сложные объекты сцены из кирпичиков — примитивов. Основным примитивом для нас будет являтся треугольник. С помощью треугольников можно получить или хорошо аппроксимировать любой объект в $\mathbb{R}^3$.

Проссумировав все выше сказанное, главной целью проекта является реализация собственного программного real-time 3D-рендерера, использующего подход растеризации в качестве основы.

Были реализованы следующие фичи:

- Растеризация треугольников с соблюдением top-left rule
- Screen-space clipping
- Frustrum clipping
- Z-buffering
- Face culling
- Линейная интерполяция значений внутри треугольника с учетом перспективы

- Наложение текстур на треугольники с учетом перспективы
- Набор функций для осуществления линейных преобразований, сдвигов, а также переходов между различными системами координат
- Загрузка и рендеринг моделей, сцен созданных в программах 3D моделлирования
- Точечный свет
- Phong shading

При написании использовалась серия видео от thebennybox, в ходе которых был реализован 3D-рендерер на языке Java. Также для реализации корректного наложения текстур ознакомился с серией статей от Chris Hecker[2]. Остальные более обширные источники указаны в списке литературы.

2 Требования к программе

Функциональные

Программа способна рендерить utah teapot с базовым освещением и разрешением 1920x1080. Пользователь программы имеет возможность исследовать заготовленную 3D сцену перемещаясь с помощью мыши и клавиатуры.

Нефункциональные

Вся программа целиком написана на C++. Графическая библиотека SFML используется только для вывода отрендереного изображения на экран. Также были использованы библиотеки с готовыми математическими классами, например с матрицами и векторами. Были использованы следующие библиотеки:

- SFML — для создания окна и вывода изображения на экран
- GLM — для классов матрицы и вектора, а также базовых операций с ними
- stb_image.h — для загрузки изображений разных форматов в оперативную память

Сборка проекта осуществляется с помощью CMake. Для форматирования кода используется ClangFormat, для линтинга и статического анализа Clang-Tidy. Кодстайл основан на Google C++ Style Guide. Используемая система контроля версий — git, удаленный репозиторий лежит на github.

3 Структура программы

Рендерер будет включать себя две основные части: графический пайплайн и инфраструктура вокруг него позволяющая взаимодействовать с пайплайном.

3.1 Обзор графического пайплайна

Для начала сосредоточимся на задаче рендеринга одного единственного треугольника. Для этого нам нужно познакомиться с различными полезными системами координат.

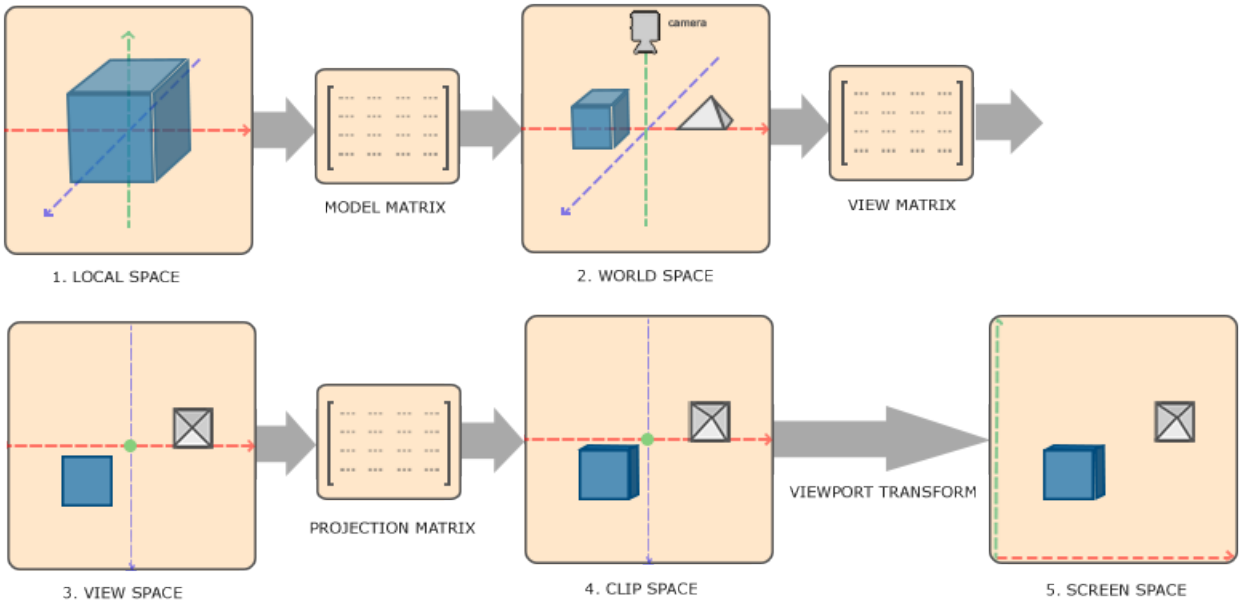


Схема взята отсюда

1. **Local space**. В этой системе координат мы задаем форму треугольника.
2. **World space**. Эта система координат говорит нам расположение треугольника в нашей сцене, как он повернут, а также его размер.
3. **View space**. Эта система координат говорит нам как расположен треугольник относительно наблюдателя, называемого камерой.
4. **Clip space**. В этой системе координат все «видимые» камерой вершины должны попасть в отрезок $[-1, 1]$.
5. **Screen space**. Здесь первые две координаты x, y считаются настоящими координатами пикселей на экране.

Как видно из схемы, мы можем переводить координаты точки из одной системы координат в другую, умножая соответствующую матрицу слева на вектор координат.

Мы будем задавать координаты точки с помощью вектора $\mathbb{R}^4$. Первые три координаты будут отвечать за координаты в $\mathbb{R}^3$ и обозначаться x, y и z соответственно, а последнюю компоненту будем обозначать как w. При задании координат треугольника в local space мы будем всегда считать $w = 1$. У этой координаты есть несколько полезных применений. Для начала, чтобы преобразовать координаты из local space в model space нам может потребоваться «сдвинуть» координаты точки на некоторый вектор. Но такое преобразование меняет начало координат, а значит не может быть линейным в $\mathbb{R}^3$. Работая же в $\mathbb{R}^4$ мы можем легко делать подобные аффинные преобразования:

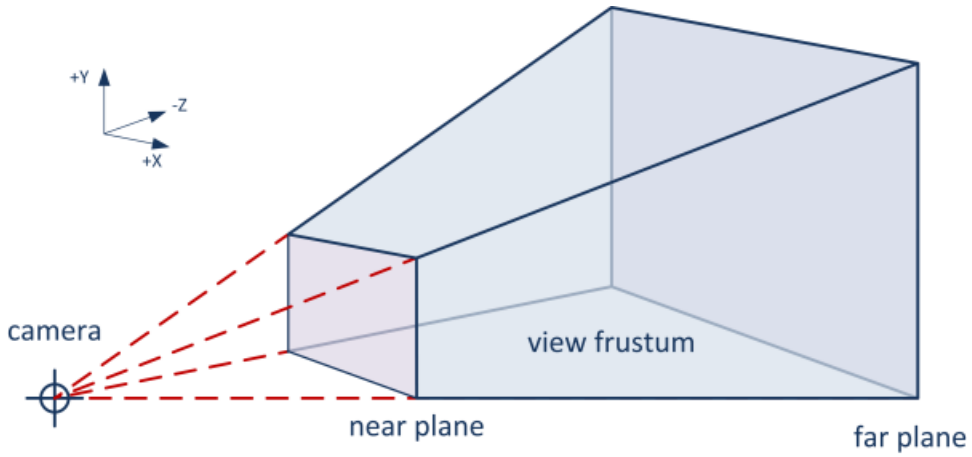
$$\begin{pmatrix} 1 & 0 & 0 & x_{add} \\ 0 & 1 & 0 & y_{add} \\ 0 & 0 & 1 & z_{add} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + x_{add} \\ y + y_{add} \\ z + z_{add} \\ 1 \end{pmatrix}$$

Тут мы можем видеть как вектор $(x, y, z)^T$ был сдвинут на вектор $(x_{add}, y_{add}, z_{add})^T$.

Помимо сдвигов, при преобразовании из local space в world space может также использоваться умножение всех координат кроме w на скаляр, а также поворот вектора в $\mathbb{R}^3$. Все эти операции являются линейными преобразованиями и могут быть выражены с помощью матричного умножения на вектор.

Чтобы перейти из world space к view space нам нужна информация о камере. Камера может задаваться её координатами в world space и вектором задающим направление в котором она смотрит. Из этой информации может быть построена view матрица.

Чтобы перейти от view space к clip space используется матрица проекции. Чтобы передать ощущение глубины объектов мы будем использовать перспективную проекцию.



Источник

Договариваемся, что камера смотрит в направлении $-z$, x идет вправо от камеры, а y вверх в рамках view space. Суть в том, что только точки попадающие в усеченный конус(view frustum) будут отображены. Все что находится за пределами view frustum считается расположеным вне обзора нашей камеры. Усеченный конус задается расстоянием до близкой плоскости(near plane), расстоянием до далекой плоскости(far plane), соотношением ширины near plane к высоте near plane, а также углом между левой и правой плоскостями, называемым FOV(field of view). Зная все это мы можем получить нашу матрицу проекции и перейти в clip space.

На самом деле чтобы перейти в clip space нам не достаточно просто умножить на матрицу проекции. Дело в том, что в этом преобразовании нам требуется поделить координаты, что не является аффинным преобразованием. Поэтому матрица проекции записывает число на которое нужно поделить все координаты в компоненту w. Действие деления всех координат вектора на w называется perspective division и должно происходить до перехода к screen space. Во время перехода к clip space view frustum отображается в куб с координатами от -1 до 1. Такие координаты называют normalized device coordinates(NDC).

На этой стадии происходит обрезка(clipping). Все вершины треугольника находящиеся за пределами куба удаляются, при этом добавляются новые вершины чтобы представить оставшуюся внутри куба часть треугольника. Такая остаточная фигура всегда выпукла, поэтому легко разбивается на треугольники.

Для преобразования из clip space в screen space достаточно знать ширину(hl) и высоту(vl) окна в пикселях. Тогда x и y координаты будут находится в отрезках $[0, hl-1]$ и $[0, vl-1]$, где целые координаты соответствуют центрам пикселей, ось x идет слева направо, ось y сверху вниз.

Для треугольника в screen space выполняется растеризация — процесс опредления пикселей на экране принадлежащих треугольнику. Используемый алгоритм растеризации следующий:

1. Отсортируем вершины по возрастанию их y координат и обозначим их в порядке сортировки как $y_{min}, y_{mid}, y_{max}$. Назовем стороны треугольника следующим образом: y_{min}-y_{max} – long edge, y_{min}-y_{mid} top edge, y_{mid}-y_{max} bottom edge.

2. Будем двигаться по одному пикселю вниз одновременно на long edge и top edge пока top edge не закончится. На каждом шаге пиксели между long edge и top edge на данной высоте y отмечаем как отображаемые.

3. Аналогично шагу 2, но теперь берем остаток long edge и bottom edge.

Также при растеризации выполняется линейная интерполяция некоторых значений лежащих в вершинах помимо координат. Сюда входят: базовый цвет, нормаль, координаты текстуры. Для того чтобы делать линейную интерполяцию во время растеризации, казалось бы достаточно вычислить градиент переменной которую мы хотим интерполировать. Но проблема в том что если мы делаем эту интерполяцию в screen space, то функция задающая значения в произвольных точках треугольника уже не линейная. И все это из-за perspective division. Но к счастью можно заметить, что зависимость интерполянта деленного на глубину линейна в screen space. Обратное к глубине так же линейно, так что интерполируя эти вещи мы можем восстановить значение исходного интерполянта в каждой точке.

Освещение обеспечивается точечными источниками света. Свет для каждого фрагмента треугольника является по сути вектором цвета умножаемым на реальный цвет фрагмента получаемый либо из его базового

цвета, либо из текстуры. Чтобы посчитать свет для фрагмента используются его координаты в world space, позиция камеры в world space, нормаль, а также информация об источнике света.

Чтобы более далекие от камеры треугольники не затирали пиксели более близких треугольников используется z-buffering. Это экранный буфер, в котором для каждого пикселя сохраняется z координата последнего пикселя треугольника который был туда записан. Тогда мы записываем цвет пикселя треугольника на экран только если z значение пикселя треугольника больше чем соответствующее значение в z-буфере.

Также можно опционально включить менее общий механизм, позволяющий не рендерить целые полигоны, если они не видны наблюдателю. Этот механизм называется face culling, но работает не для всех моделей. При использовании face culling мы считаем что у каждого полигона есть видимая и невидимая сторона. Чтобы определить какая из сторон какая мы используем порядок в котором перечислины вершины треугольника. Если они идут против часовой стрелки мы считаем, что такая стороно видима и невидима иначе.

3.2 Устройство инфраструктуры в программе

Главным классом в исходном коде является Application. Он отвечает за главный цикл отрисовки и обработки событий, а также хранит хранит внутри себя классы Window, Renderer, Camera используя композицию. Внутреннее представление сцены для отрисовки также лежит на нем.

Класс Window отвечает за создание окна, а также предоставляет интерфейс для обработки ввода компьютерной мыши и клавиатуры.

Класс Renderer отвечает за рендеринг моделей хранящихся в виде экземпляров класса Model. Для этого он использует класс PrimitiveRenderer способный рендерить только треугольники.

Класс Camera отвечает за хранение состояния камеры, а также предоставляет интерфейс для доступа к этим данным как для чтения, так и для изменения.

Список литературы

[1] Samuel R. Buss. *3D Computer Graphics: A Mathematical Introduction with OpenGL, 1st Edition.* Cambridge University Press, 2003.

[2] Cris Hecker. Perspective texture mapping.

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

[4] Боресков А.В. Шикин Е.В. *Компьютерная графика. Динамика, реалистические изображения.* ДИАЛОГ-МИФИ, Москва, 1995.

[5] Боресков А.В. Шикин Е.В. *Компьютерная графика. Полигональные модели.* ДИАЛОГ-МИФИ, Москва, 2001.