



Факультет программной инженерии и компьютерной техники

Теоретические основы компьютерной графики и
вычислительной оптики

Курсовая работа

Преподаватель: Жданов Дмитрий Дмитриевич

Выполнил: студент: Кульбако Артемий Юрьевич, Р4115

Санкт-Петербург
2022

ОГЛАВЛЕНИЕ

ОГЛАВЛЕНИЕ	2
ЗАДАНИЕ.....	3
ТЕОРЕТИЧЕСКИЕ АСПЕКТЫ	3
ВЫПОЛНЕНИЕ.....	6
ВЫВОД	10
СПИСОК ИСТОЧНИКОВ.....	12

ЗАДАНИЕ

Объектом исследования является процесс синтеза и визуализации фотореалистичных изображений с помощью компьютера. Выполнение курсовой работы (цель) состоит в разработке программы - трассировщика лучей (рендерера), результатом работы которой станет изображение (или набор изображений).

ТЕОРЕТИЧЕСКИЕ АСПЕКТЫ

Трассировка лучей - технология построения изображения трёхмерных сцен в компьютерных программах, при которой цвет конечного пикселя изображения находят путём моделирования пути луча от источника света до виртуальной камеры или наоборот (обратная трассировка). Первый метод даёт более реалистичные изображения, но требовательнее к ресурсам ввиду того что требует вычислять путь для значительно большего количества лучей (в теории их число может быть бесконечным). Обратная трассировка лучей в простейшем варианте требует вычислять путь для одного луча на пиксель. Подобная реализации возможна благодаря следующим геометрическим и физическим законам:

- Закон прямолинейного распространения света: можно представлять луч света как прямую из точки А в точку В.
- Закон обратимости светового луча: позволяет проводить трассировку от камеры к источнику света, используя те же физические правила и законы, что и при прямой трассировке.
- Закон независимого распространения света. Позволяет не заботиться о пересечении лучей друг с другом и выполнять моделирование в параллельном режиме.
- Закон отражения. Стоит учитывать, что отражения бывают двух видов: диффузные и зеркальные. Если поверхность гладкая, при падении на неё луча света, угол падения будет равен углу отражения. В случае, если на поверхности есть

неровности, и чем их больше, тем сильнее свет рассеется: на самом деле, закон отражения работает как обычно, просто этого не видно с позиции наблюдателя из-за расстояния. Когда этих неровностей немного, на поверхности появится блик.

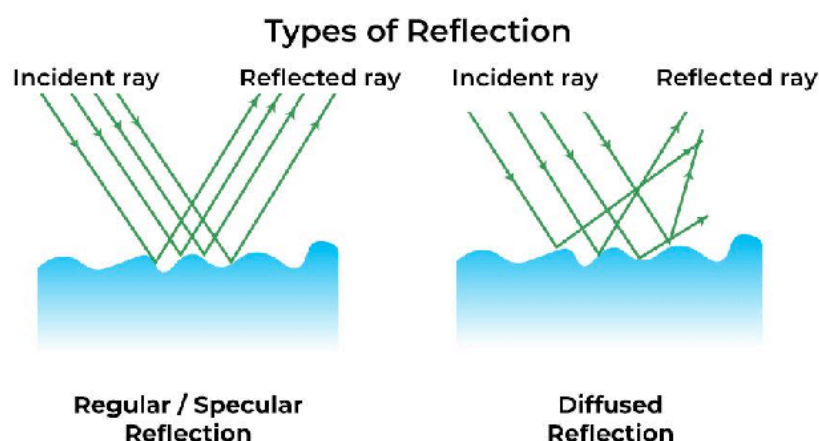


Рис.1 - виды отражений

При попадании лучей на поверхность они взаимодействуют с ней, что характеризуется в радиометрии освещённостью поверхности. Свет поглощается и отражается от поверхности в зависимости от характеристик взаимодействующих и могут отличаться для разных длин волн, в моей реализации трассировщика допущено небольшое упрощение: считается, что характеристики одинаковы для всех длин волн.

Для расчёты яркости, радиометрия этими понятиями:

- Сила излучения - концентрация энергии в заданном направлении.
- Освещённость - плотность потока, проходящего через поверхность определённой площади. Освещённость и сила излучения света связаны формулой: $E = \frac{I \cos \theta}{R^2}$, где $\cos \theta$ - косинус угла между падающим лучом и нормалью плоскости

падения; R - расстояние от источника света до освещаемой поверхности. Программная реализация:

```
let illumination = (self.point_light.intensity * cosθ) /  
(distance.powi(2));
```

- Яркость - поток, посылаемый в заданный телесный угол в заданном направлении с единичной площади источника. Для более точного расчёта яркости также необходимо учитывать яркость отражённого луча и ДФОС:

$$L = \frac{E * BRDF}{\pi} + L_{reflect} * K_s. E - \text{освещённость (Вт/м2), BRDF}$$

- двулучевая функция отражательной способности, $L_{reflect}$ - яркость отражённого луча, она накапливается при каждом отражении от поверхности (Вт/(ст*м2), K_s - коэффициент отражательной способности поверхности. Программная реализация:

```
ray.radiance = ((illumination * brdf) / std::f64::consts::PI) +  
reflect_ray.radiance * material.specular_reflection;
```

Таким образом, абстрактный алгоритм обратного трассировщика лучей можно описать так:

1. Задать сетку, определяющую количество выпущенных лучей
2. Пустить из камеры, расположенной за этой сеткой по лучу для каждой клетки
 1. Если нет пересечений луча с объектами на заданной длине, считаем, что либо объекты не пересекаются с этим лучём или расположены слишком далеко.
 2. Если пересечение есть, определяем, какого цвета она должна быть.
 1. Необходимо выпустить зеркальные лучи (их количество надо ограничить) или теньевые лучи, чтобы понять, освещается ли эта точка.
 1. Если не освещается, красим её черным (в реальном мире, существуют полутени, но для их реализации необходимо рассчитывать глобальную освещённость, что само по себе отдельная и сложная тема, поэтому здесь они не учитываются).
 2. Если освещается, вычисляем исходя из характеристик поверхности.
 1. Вычисляем освещённость.
 2. Вычисляем яркость.
 3. Если луч попал на зеркальную поверхность, пускаем луч от точки попадания.

1. Накапливаем характеристики луча, пока не достигнем максимальной разрешённой глубины.

При разработке алгоритма я опирался литературу из списка источников.

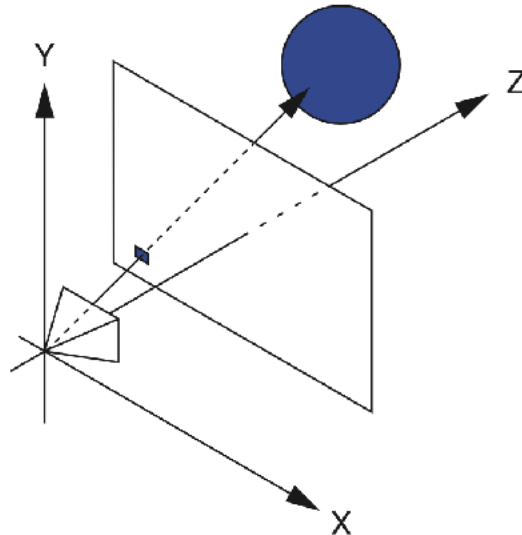


Рис.2 - модель сцены для обратного трассировщика

ВЫПОЛНЕНИЕ

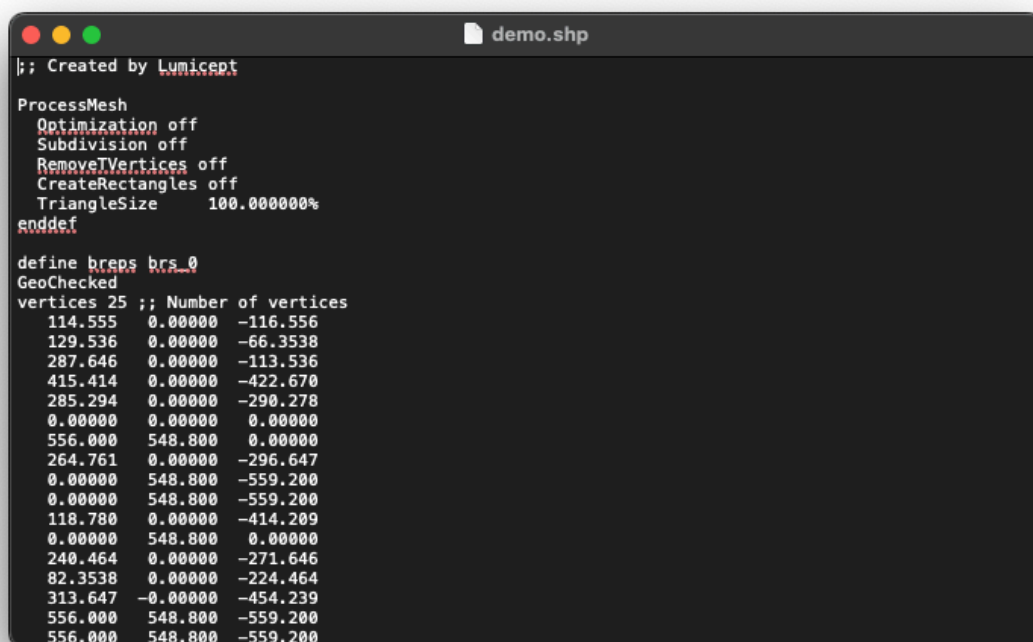
Любую разработку стоит начинать с выбора инструментов. Так как трассировка лучей (даже обратная) - очень ресурсоёмкий процесс, классическим языком программирования её является C++ ввиду компиляции исходного кода в код, наиболее оптимизированный для каждой конкретной среды исполнения, а контроль за памятью осуществляем сам программист, что позволяет проводить ещё больше оптимизаций, создавая программы, работающие максимально эффективно. Также для него существуют готовые библиотеки, созданные лучшими умами человечества, реализующие самые современные алгоритмы для графики (к примеру Intel Embree [\[1\]](#)).

Тем не менее, у C++ есть и ряд недостатков. В первую очередь, это сложный язык, при этом являясь достаточно свободным языком, что повышает сложность разработки программ и риск создания случайных трудноуловимых ошибок. Он тянет в себе огромную базу legacy решений, как в синтаксисе, так и в организации стандартных библиотек. Проще говоря, при написании программы на C++, много времени и сил тратится не на реализацию программы как таковой, а на использование языка. Я хотел использовать язык, который проще и понятнее C++, но при этом работает также (или почти также) эффективно. Таким языком является Rust [\[2\]](#). Он обладает всеми преимуществами C++, но за счёт более строго контроля за памятью, находит большинство таких ошибок на стадии компиляции, что значительно снижает риск появления проблем, библиотечные функции реализованы удобнее, синтаксис чище. Присутствует менеджер пакетов, позволяющий удобно подключать зависимости из общедоступных репозиториях.

В качестве формата файлов, воспринимаемых программой был выбран SHP [\[3\]](#). Изначально я планировал взять более современный OBJ [\[4\]](#), GLTF [\[5\]](#) и может даже USD [\[6\]](#), но остановился на SHP ввиду того, что его поддерживает Lumisect по умолчанию, а значит сделать сцену будет просто ибо к Lumisect уже успел привыкнуть. Формат текстовый, а потому для его чтения не нужны какие-либо дополнительные действия (десериализация, написание сложного конвертера и т.д.) в программе. Из минусов можно отметить то, что он хранит только геометрию без описание характеристик материала того или иного объекта/полигона, а потому пришлось создать в программе дополнительную сущность MATERIAL_LIBRARY для хранения некоторых базовых материалов.

Рис. 3 - Пример описания геометрии в SHP

Тестовая сцена представляет из себя созданную и экспортированную из Lumisect корнелльскую коробку - это



```
;; Created by Lumicept

ProcessMesh
  Optimization off
  Subdivision off
  RemoveTVertices off
  CreateRectangles off
  TriangleSize 100.000000%
enddef

define brp brs_0
GeoChecked
vertices 25 ;; Number of vertices
114.555 0.00000 -116.556
129.536 0.00000 -66.3538
287.646 0.00000 -113.536
415.414 0.00000 -422.670
285.294 0.00000 -290.278
0.00000 0.00000 0.00000
556.000 548.800 0.00000
264.761 0.00000 -296.647
0.00000 548.800 -559.200
0.00000 548.800 -559.200
118.780 0.00000 -414.209
0.00000 548.800 0.00000
240.464 0.00000 -271.646
82.3538 0.00000 -224.464
313.647 -0.00000 -454.239
556.000 548.800 -559.200
556.000 548.800 -559.200
```

стандартная сцена, которая используется для демонстрации результатов работы рендерера.

Было решено проводить моделирование в sRGB цветовой схеме, не спектральной. Да, такой вариант физически менее корректный, но более понятный, позволяет сосредоточиться на реализации самого трассировщика. К тому же, данные, рассчитанные в sRGB можно удобно записать в PNG/JPG файл, для просмотра данных в спектральном формате придётся пользоваться Lumicept, что не так удобно.

Интерфейс приложения - аргументы командной строки. Пользователь задаёт параметры рендеринга и получает картинку после работы программы в директории, указанной им же (либо там, где сохранена сцена, это поведение по умолчанию).

Таб. 1 - аргументы командной строки

short name	long name	Описание
-S	--scene_path	Путь до файла сцены в формате shp

-W	--width	Ширина изображения
-H	--height	Высота изображения
-I	--intensity	Интенсивность света
-A	--antialiased	Включить сглаживание (на данный момент не поддерживается)
-R	--render_path	Путь где будет сохранено изображение
	lx	Положение источника света по x
	ly	Положение источника света по y
	lz	Положение источника света по z
	cx	Положение камеры по x
	cy	Положение камеры по y
	cz	Положение камеры по z
	cf	Угол обзора камеры (FOV)

Для ускорения вычислений используется библиотека параллельных вычисления `rayon` [7]. С точки зрения кода это просто вызов специального итератора, лямбда-функция, переданная в качестве параметра будет выполнена параллельно. Благодаря особенности языка, концепцией владения, можно быть уверенным, что несколько потоков не создадут блокировок, гонок и других проблем, характерных для параллельных задач. Время расчётов благодаря параллельным вычислениям сократилось более чем в 5 раз (36сек → 7сек для 8-ядерного процессора). После формирования массива с информацией о цвете каждого пикселя, запись в файл осуществляется библиотекой `image` [8]. Также используется модуль `clap` [9], позволяющий удобно обрабатывать аргументы командной строки.

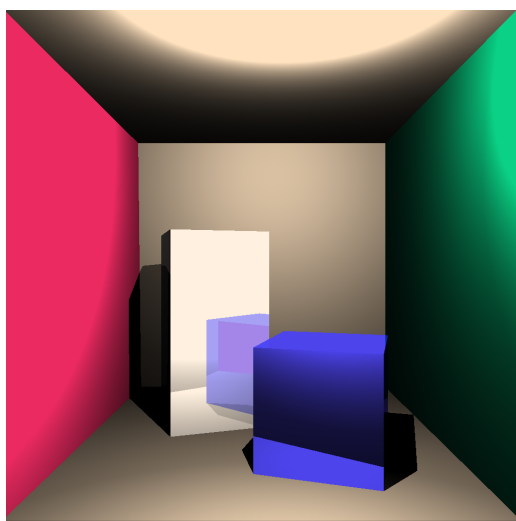
Архитектурно, разработанная программа крайне примитивна: это просто монолит без каких-либо необычных решений.

Таб. 2 - описание программных модулей

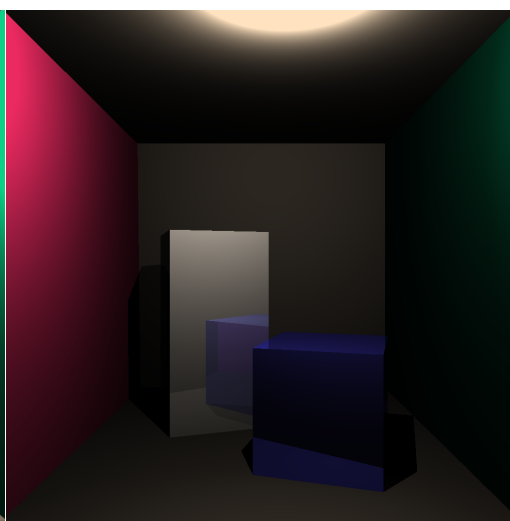
<code>vec3.rs</code>	Выполнение операций с геометрическими векторами.
----------------------	--

<code>utils.rs</code>	Различные вспомогательные функции.
<code>scene.rs</code>	Содержит информацию о сцене и функции рендеринга. Самый сложный файл.
<code>polygon.rs</code>	Инкапсулирует информацию о полигоне: его положение и материал.
<code>material.rs</code>	Характеристики материала. Также содержит статический массив с заранее заготовленными материалами.
<code>camera.rs</code>	Простая камера-обскура.
<code>main.rs</code>	Обработка аргументов командной строки и запуск рендеринга.
<code>lights/ray.rs</code>	Информация о конкретном луче света.
<code>lights/point_light.rs</code>	Точечный источник света.
<code>geom_loaders/shp_loader.rs</code>	Загрузчик сцен формата shp, реализует общий интерфейс <code>GeomLoader</code> , чтобы можно было легко добавить поддержку других форматов.

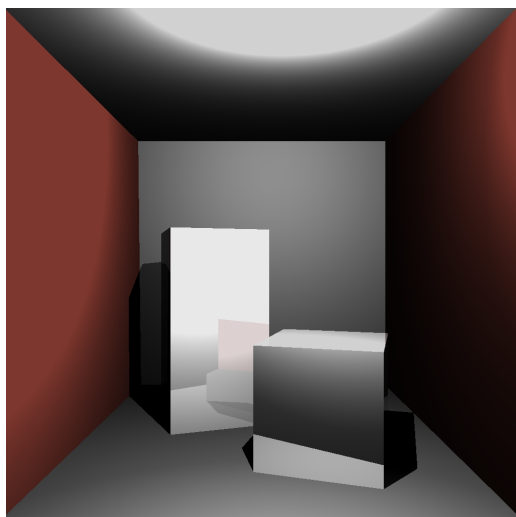
ВЫВОД



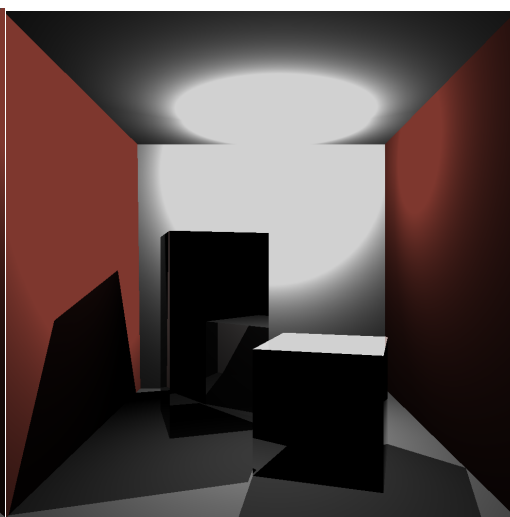
Пример отражений



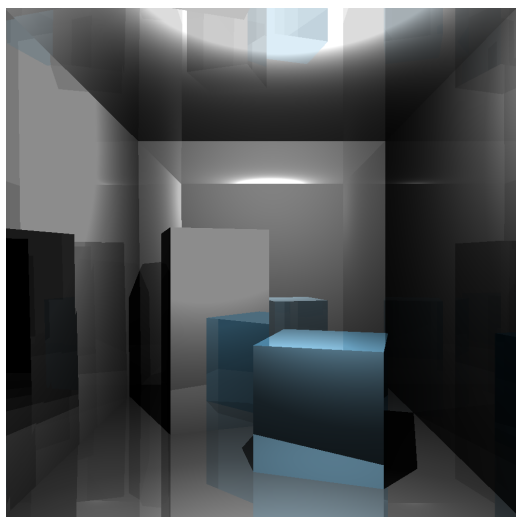
Слабый свет



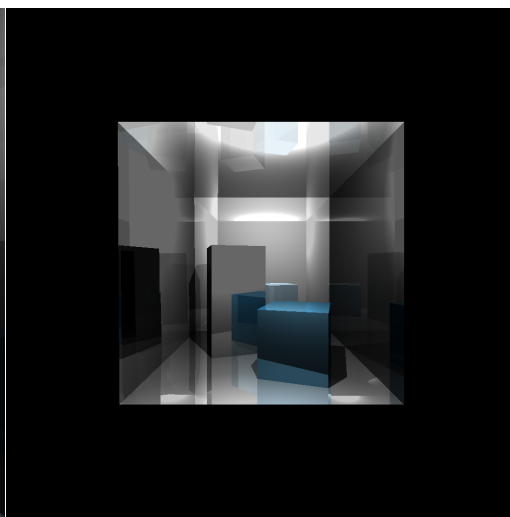
Другой цвет стен



Положение источника света



Всё зеркала



FOV увеличен (55° → 75°)

Результат хоть не идеален, но я им доволен, картинки генерируются, порой даже красивые. В процессе выполнения курсовой работы я систематизировал лекционные материалы и лабораторные работы, в следствие чего овладел навыком создания простого рендерера, который умеет красить сцену методом обратной трассировки лучей. Рендерер я планирую развивать по мере своих возможностей, как с точки зрения улучшения генерируемых им картинок, так и добавление поддержки новых форматов, параллельных вычислений на нескольких компьютерах и добавления глобального освещения, вероятно, в рамках соответствующего предмета. Исходный код доступен на Github:

<https://github.com/testpassword/Theoretical-foundations-of-computer-graphics-and-computational-optics/tree/master/course>



СПИСОК ИСТОЧНИКОВ

1. Сайт графической библиотеки Intel Embree [Электронный ресурс] – URL: <https://www.embree.org/> – Режим доступа: свободный. – Дата обращения: 03.11.2022.
2. Документация Rust на русском языке [Электронный ресурс] – URL: <https://www.rust-lang.org/ru/> – Режим доступа: свободный. – Дата обращения: 05.11.2022.
3. Описание формате Shapefile [Электронный ресурс] – URL: <https://ru.wikipedia.org/wiki/Shapefile> – Режим доступа: свободный. – Дата обращения: 05.11.2022.
4. Описание формате OBJ [Электронный ресурс] – URL: <https://ru.wikipedia.org/wiki/Obj> – Режим доступа: свободный. – Дата обращения: 05.11.2022.
5. Описание формате GLTF [Электронный ресурс] – URL: <https://www.khronos.org/gltf/> – Режим доступа: свободный. – Дата обращения: 05.11.2022.
6. Описание формате USD [Электронный ресурс] – URL: <https://graphics.pixar.com/usd/release/index.html> – Режим доступа: свободный. – Дата обращения: 05.11.2022.
7. Документация библиотечки для параллельных вычислений rayon [Электронный ресурс] – URL: <https://docs.rs/rayon/>

- [1.6.1/rayon/](#) – Режим доступа: свободный. - Дата обращения: 10.12.2022.
8. Документация библиотек для записи изображений [Электронный ресурс] – URL: <https://docs.rs/image/0.24.5/image/> – Режим доступа: свободный. - Дата обращения: 10.12.2022.
9. Документация библиотек для парсинга аргументов командной строки [Электронный ресурс] – URL: <https://docs.rs/clap/4.0.32/clap/> – Режим доступа: свободный. - Дата обращения: 14.12.2022.
10. Гамбетта Г. Компьютерная графика. Рейтрейсинг и растеризация. – Санкт-Петербург: Питер, 2022. – 224 с. – ISBN 978-5-4461-1911-0.
11. Мацуда К., Ли Р. WebGL: программирование трёхмерной графики. – Москва: ДМК Пресс, 2019. – 493 с. – ISBN 978-5-97060-146-4.