

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет информатики, математики и компьютерных наук

**Программа подготовки бакалавров по направлению
09.03.04 Программная инженерия**

Канделов Дамир Русланович

КУРСОВАЯ РАБОТА

Сравнение различных стратегий выборки в рендеринге трассировкой путей

Научный руководитель

Старший научный сотрудник

программной инженерии

факультета компьютерных наук

канд. физ. мат. наук

И.С. Бычков

Нижний Новгород, 2024

Содержание

1.	Введение	4
2.	Постановка задачи	6
3.	Исследование предметной области.....	7
3.1.	Что такое Path tracing?	7
3.2.	Отличие Path tracing от Ray tracing	8
3.3.	Алгоритм работы Path tracing	9
3.4.	Случайные числа в алгоритме Path tracing	13
3.5.	Влияние случайных чисел на качество изображений.	14
4.	Описание решения.....	15
4.1.	Реализация программы рендеринга изображений	15
4.1.1.	Написание базовой логики. Первые изображения.....	15
4.1.2.	Добавление новых функций в 3D движок.	16
4.2.	Использование случайных чисел в 3D движке	18
4.3.	Генерация случайных чисел	19
4.3.1.	Standard sampler.....	19
4.3.2.	Halton sampler	20
4.3.3.	Halton Sampler с использованием RandomDigit скремблинга.....	23
4.3.4.	Halton Sampler с использованием Owen скремблинга.....	25
4.3.5.	Blue Noise sampler	27
5.	Результаты. Сравнение алгоритмов.....	31
5.1.	Визуальное сравнение	31
5.1.1.	Сравнение методов выборки на сцене «Волк».....	31
5.1.2.	Сравнение методов выборки на сцене «Cornell Box».....	35

5.1.3. Результаты визуального сравнения	39
5.2. Сравнение с использованием метрик.....	40
5.2.1. Метрика MSE.....	40
5.2.2. Метрика FLIP	43
5.3. Результаты сравнений	44
6. Заключение.....	46
7. Библиографический список	48

1. Введение

В настоящее время компьютерная графика, а именно методы рендеринга изображений используются практически повсеместно. Рендеринг – это процесс генерации фотoreалистичных или нереалистичных изображений по заданным 2D или 3D моделям, с помощью компьютерных программ. Он используется для создания CGI-изображений (computer-generated imagery), в киноиндустрии для создания визуальных эффектов (VFX), в разработке компьютерных игр и симуляторов для отрисовки сцен, которые пользователь будет видеть на экране, а также в сферах дизайна и архитектуры.

Существует множество алгоритмов рендеринга для создания фотoreалистичных изображений, наиболее известными являются:

Rasterization (Растеризация) – это процесс перевода изображения, описанного в векторном формате, в растровое изображение. По сравнению с другими техниками рендеринга является одной из наиболее быстрых, из-за чего используется в большом количестве 3D движков, отрисовывающих изображение в реальном времени. Однако, минусом данной техники является возможность показать очень малое число оптических эффектов.

Ray tracing (Трассировка лучей) – это метод запуска «лучей» из камеры в каждый пиксель изображения, которые по мере своего движения (отражения и преломления) собирают аккумулируют цвета пересеченных объектов и, исходя из этих цветов, получают итоговые цвет пикселя. Ray tracing способен симулировать большое количество оптических эффектов, таких как отражения, преломления и дисперсию света, мягкие тени, глубину резкости, размытие в движении, каустику.

Path tracing (Трассировка путей) – этот метод похож на Ray tracing, однако он запускает большее число лучей через каждый пиксель, используя при этом методы Монте Карло, а также собирает информацию от всех источников света,

которые освещают данный пиксель. Засчет этого path tracing является наиболее близким к реальности по поведению глобального освещения. Если метод используется совместно с физически корректными материалами и объектами, то в результате получаются изображения, неотличимые от реальных фотографий.

Основными потребителями алгоритмов рендеринга являются кино- и гейм-дев индустрии. Последние тенденции в обоих индустриях – создание максимально реалистичной графики. Как можно заметить, алгоритм Path tracing подходит больше всего для решения этой задачи. Однако, основным его недостатком является большая вычислительная сложность. В данной курсовой работе будет проверена теория, о том, что от метода выбора случайных чисел во время работы алгоритма зависит качество, а также время получения изображений хорошего качества.

Таким образом, востребованность и актуальность темы данной работы обусловлена потребностями сферы разработки игр и сферы создания визуальных эффектов в кино, нацеленных на получение реалистичной графики в своих проектах за сравнительно малое количество времени.

2. Постановка задачи

В данной работе будет подробно рассмотрен алгоритм Path tracing. Для большего понимания его работы будет написан собственный 3D движок для рендеринга изображений и наглядного сравнения изменений в результатах при разных подходах.

На примере его работы мы разберем и сравним несколько наиболее популярных подходов к генерации случайных чисел.

Цель данного проекта: изучение алгоритмов генерации случайных чисел, а также выделение алгоритмов, с помощью которых можно будет улучшить время рендеринга изображений хорошего качества.

Для большего понимания устройства процесса рендеринга, предстоит дополнительно изучить графику в более общем понимании, а также написать собственный 3D движок, с помощью которого проводить сравнение написанных алгоритмов.

Задачами данного проекта являются:

1. Разобраться в алгоритме работы Path tracing.
2. Узнать, где в алгоритме используются случайные числа и почему метод их выбора влияет на качество результирующих изображений.
3. Написать собственную программу для рендеринга изображений с возможностью выбора используемого метода генерации случайных чисел.
4. Реализовать несколько подходов к генерации случайных чисел и имплементировать их в собственном проекте.
5. Срендерить изображения с использованием новых методов выбора случайных чисел.
6. Провести комплексный анализ полученных изображений и сделать выводы о результатах использования различных стратегий выборки в рендеринге трассировкой путей.

3. Исследование предметной области

3.1. Что такое Path tracing?

Path tracing – это метод рендеринга изображений по 3D сценам в компьютерной графике, который стремится воссоздать изображение максимально близкое к реальности, основываясь на физически верных формулах. Данный метод реализует все визуальные эффекты, которые возможно представить на изображении.

The rendering equation is

$$I(x, x') = g(x, x') \left[\epsilon(x, x') + \int_S \rho(x, x', x'') I(x', x'') dx'' \right]. \quad (1)$$

where:

- $I(x, x')$ is the related to the intensity of light passing from point x' to point x
- $g(x, x')$ is a “geometry” term
- $\epsilon(x, x')$ is related to the intensity of emitted light from x' to x
- $\rho(x, x' x'')$ is related to the intensity of light scattered from x'' to x by a patch of surface at x'

Рисунок 1. Оригинальное уравнение рендеринга, представленное в одноименной статье
«The Rendering Equation» Jim Kajiya, 1986

В 1986 году Jim Kajiya выпустил статью под названием «The Rendering Equation», которая объединяла компьютерную графику и физику. В ней было представлено уравнение, описывающее поведение света и позволяющее точно представить, как свет движется по 3D сцене. В этой же статье было и название нового метода рендеринга, который использовал данное уравнение: **Path tracing**.

Полученное уравнение рендеринга было довольно небольшим и понятным, однако решить его не просто из-за того, что сцены в компьютерной графике очень сложны и описываются миллионами или даже миллиардами треугольников в современном мире. Нельзя полностью решить это уравнение.

Jim Kajiya предложил решение этой проблемы: не требуется решать уравнение напрямую, его можно решить для каждого отдельного луча (например, с помощью алгоритма Ray tracing). Соответственно, получение фотorealистичных изображений возможно, если решить это уравнение для достаточного количества лучей.

$$L_0(p, \omega_0) = \int_{\Omega} f(p, \omega_0, \omega_i) L_i(p, \omega_i) \cos\theta_i d\omega_i$$

Рисунок 2. Другая форма уравнения рендеринга, на которой
более явно видны все составляющие

$L_0(p, \omega_0)$ – уходящие суммарное излучение света, отраженное в точке p в направлении ω_0

$f(p, \omega_0, \omega_i)$ – функция, которая определяет, количество излучения, которое отражается в точке p в направлении ω_0 от излучения, попадающего в точку p в направлении ω_i

$L_i(p, \omega_i)$ – световое излучение, падающее в точку p в направлении ω_i

$\cos\theta_i$ – закон косинусов Ламберта

Как было написано ранее, нельзя полностью решить это уравнение, поэтому для получения приближенного значения функции используются методы Монте-Карло. Это методы, для получения приближенного значения интеграла.

3.2. Отличие Path tracing от Ray tracing

Одним из наиболее частых вопросов при знакомстве с методами рендеринга является вопрос отличия Path tracing и Ray tracing. В алгоритме Path tracing используется Ray tracing, однако следует разделять эти два метода рендеринга.

Основное отличие Path tracing от Ray tracing в том, что в трассировке путей мы запускаем большее число лучей, а затем аппроксимируем итоговое значение цветовой характеристики, исходя из значений для запущенных лучей. В результате чего мы получаем более физически-корректное и реалистичное изображение.

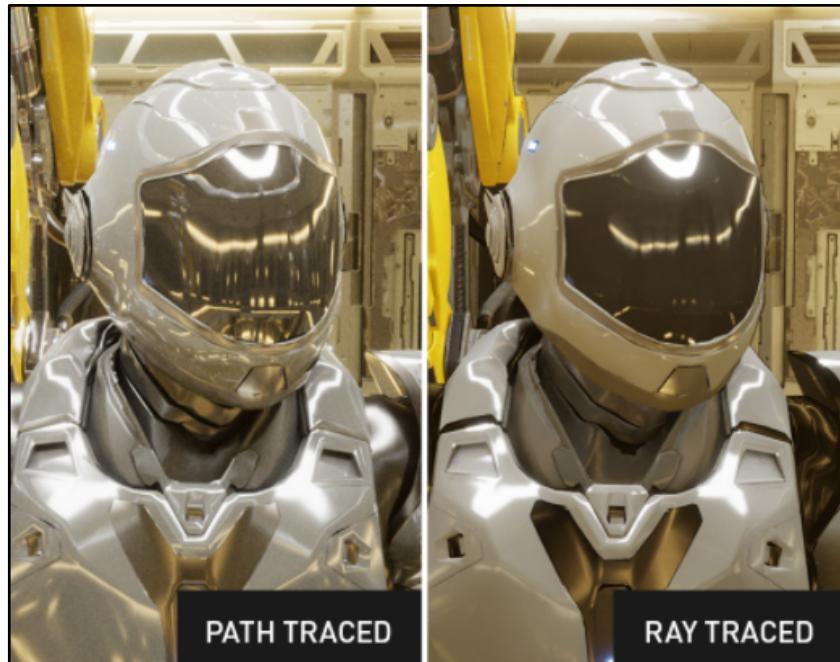


Рисунок 3. Сравнение Path tracing и Ray tracing [1]

Как мы видим, в данном примере явно заметно, что Path tracing симулирует большее количество света, что гораздо ближе к реальности: в отражениях мы должны хорошо видеть все источники света.

3.3. Алгоритм работы Path tracing

Разберемся, как работает path tracing. Как было сказано выше, алгоритм максимально приближен к тому, как это происходит в реальном мире. Но с одним исключением.

В реальном мире свет распространяется от источника света, отражается от объектов и попадает на сетчатку глаза. Однако реализация такого поведения имеет слишком большую вычислительную сложность, так как требуется

запустить лучи во все стороны от источника света (потому что свет распространяется во все стороны). Поэтому в компьютерной графике было принято решение развернуть эту последовательность действий: запускать лучи не из источников света, а из камеры (позиции, откуда пользователь смотрит на 3D сцену).

Далее, требуется понять, куда запускать лучи из камеры в пространстве.

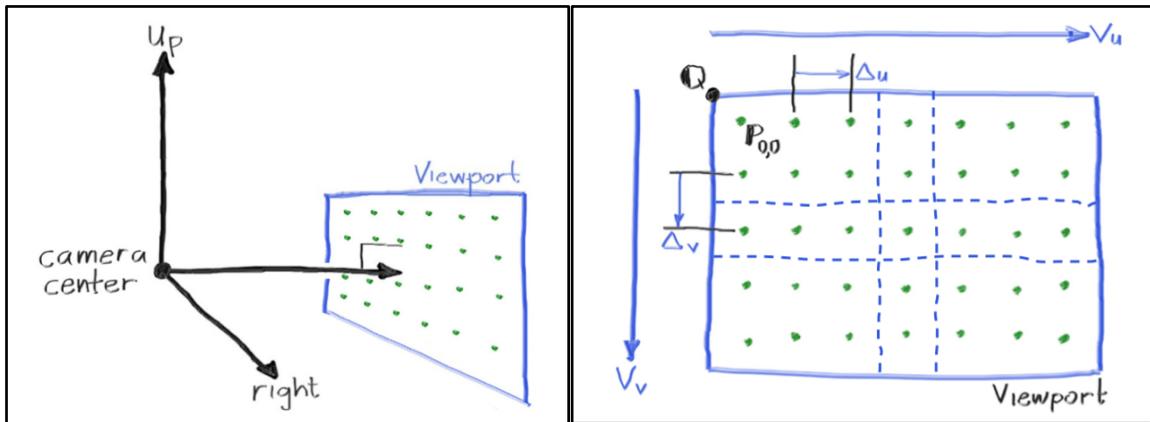


Рисунок 4. Генерация лучей из камеры [2]

Заранее задаются параметры камеры: размер изображения, количество пикселей и т.д. На этом этапе мы создаем Viewport – матрица пикселей изображения, расположенная между камерой и сценой. На ней мы явно выделяем пиксели. Чтобы получить цвет пикселя требуется запустить луч через этот пиксель.

Здесь мы и встречаемся с первым примером использования случайных чисел: если запускать лучи только через центры пикселей (зеленые точки на рисунке), то мы можем неправильно вывести цвет для данного пикселя. Самый простой пример такого явления – рендеринг краев объектов, например шара, когда мы запускаем один луч через центр, мы получаем только один цвет. Допустим луч попал в шар, то есть, грубо говоря, функция вернет цвет шара, а луч через следующий пиксель, уже не попадет в шар и вернет цвет фона. Получается грубая граница объекта (левая картинка, рисунок 5). Это отличается

от того, как мы видим это в жизни (правая картинка, рисунок 5). Для решения этой проблемы будем запускать не один луч через центр пикселя, а несколько через разные точки внутри пикселя. Чтобы получить эти точки нам нужно взять случайные координаты внутри пикселя. Затем для получения цвета нужно усреднить значение от получившихся цветов и записать в текущий пиксель.

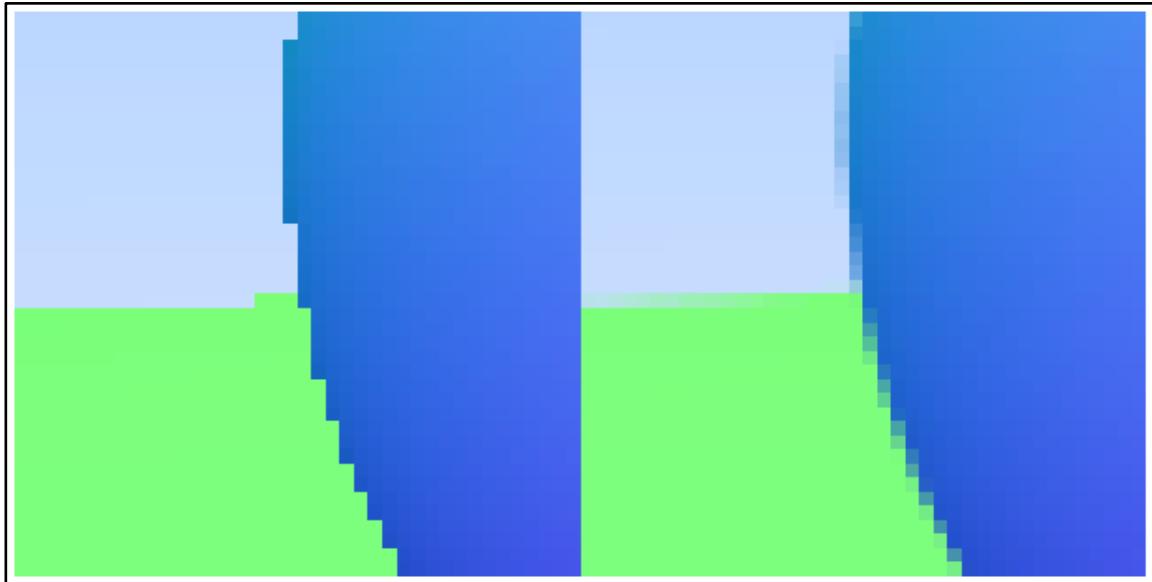


Рисунок 5. Рендеринг краев объекта [2]

(слева из камеры летит один луч через центры пикселей,
справа – несколько лучей, через случайные точки внутри пикселя)

После запуска луча из камеры, есть два варианта:

1. Он не достигнет никакого объекта на сцене (сцены бывают разными, необязательно они полностью окружены объектами);
2. Луч пересечет какой-то объект. Во втором варианте, в зависимости от свойств материала объекта, может произойти отражение или преломление, все это выполняется в точности, по законам физики.

Во время такого пересечения луч запоминает цвет объекта, а также освещенность от источников света в данной точке, и продолжает свое движение. Далее, может быть, либо достигнут предел количества отражений (его

устанавливают для уменьшения времени работы алгоритма), в этом случае в рекурсивную функцию возвращаем черный цвет; либо достигнут источник света.

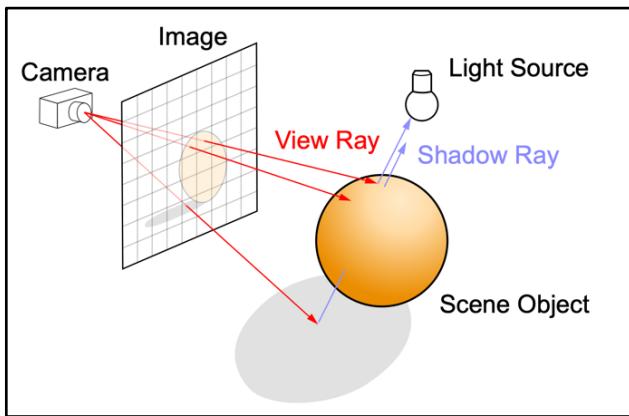


Рисунок 6. Пример лучей из камеры и получения цвета пикселя [3]

(View Ray – это лучи, которые мы запустили из камеры,
Shadow Ray – это лучи, направленные от точки пересечения к источнику света,
чтобы узнать освещенность в текущей точке)

Все полученные данные в ходе отражений и преломлений луча в дальнейшем используются в Уравнении рендеринга, и мы получаем итоговый цвет пикселя. Затем мы повторяем эту операцию, но уже с другими случайными числами: координатами точки в пикселе, через которую летит луч; направлением луча при отражении от материала и т.д.

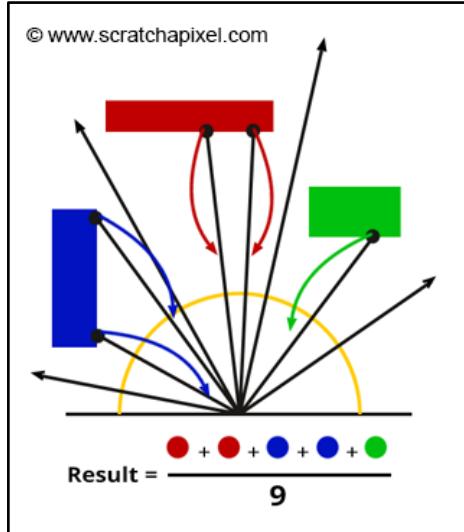


Рисунок 7. Получение цвета пикселя, исходя из нескольких запущенных лучей в разные стороны [4]

(берется среднее значение из цветов всех запущенных лучей)

В результате всех вышеописанных действий мы и получаем итоговый результат – срендеренное изображение. Теперь чуть больше углубимся в процесс использования случайных чисел.

3.4. Случайные числа в алгоритме Path tracing

При рендеринге изображений случайные числа используются на нескольких этапах, каждое использование случайного числа в отдельном случае необходимо брать отдельно. Будем называть различные случаи: dimension (дименшен / размерность).

В нашем алгоритме есть несколько дименшеноов:

- Выбор начальной точки для запуска луча через пиксель (2 разных дименшена – координата по Ox и по Oy) – дименшены 0, 1
- Выбор направления отраженного луча от материала, для которого угол падения не равен углу отражения (3 разных дименшена – координата по Ox, Oy и Oz) – дименшены 5, 6, 7

Также зачастую случайные числа используются еще в нескольких местах:

- Выбор источника света (один дименшен, отвечающий за индекс) – дименшен 2
- Выбор координаты на линейном / плоском источнике света (два дименшена: координаты по X и Y) - дименшены 3, 4
- Один дополнительный дименшена, отведенный под материалы, то есть вместе с 3 использующимися - 4 (в зависимости от сложности материалов в сцене, в моем проекте используются только три из них, для выбора направления луча) – дименшен 8

В реальных современных проектах, в зависимости от сложности проекта, количество дименшнов может сильно варьироваться. Для решения уравнения рендеринга в общем случае необходимо бесконечное число дименшнов. В нашем проекте мы ограничимся 32 дименшнами.

3.5. Влияние случайных чисел на качество изображений.

Есть теория, которая доказывается с использованием методов Монте-Карло том, что в зависимости от распределения случайных чисел могут получаться разные изображения. Чем лучше числа последовательно распределены, то есть никакие два числа не лежат рядом друг с другом, а также нет областей без чисел в нашем пространстве, тем лучше получится итоговое изображение.

В нашем случае это обосновано тем, что хорошо распределенные точки лучше покрывают плоскость, то есть, если говорить о, например начальных точках внутри пикселя, то у нас появляется квадрат размера 1x1, которые мы покрываем хорошо распределенными точками и, соответственно, после запуска лучей через эти точки мы будем иметь полное понимание о цвете этого пикселя.

Именно эту теорию мы будем пробовать применить и проверить на практике, а также нам предстоит найти подходящие алгоритмы генерации случайных хорошо распределенных чисел.

4. Описание решения

4.1. Реализация программы рендеринга изображений

В ходе работы над курсовой работой, для лучшего понимания алгоритмов рендеринга был написан собственный 3D движок для рендеринга изображений на языке C++ на основе алгоритма path tracing.

Это помогло понять на практике, как работает рендеринг изображений, какие техники применяются, где и как необходимо использовать хорошо распределенные случайные числа. Во время написания кода тестились и основополагающие, базовые подходы, существующие в графике, и более продвинутые, современные, которые используются в современных 3D движках.

4.1.1. Написание базовой логики. Первые изображения.

При написании собственного проекта требовалось понять, с чего начать и как дойти до результата, который бы позволил не только рендерить сцены на уровне использующихся сегодня 3D движков, но и позволял бы протестировать нашу основную гипотезу по ускорению рендеринга изображений за счет различных стратегий выбора случайных чисел.

За основу была взята серия книг Ray Tracing in One Weekend (P. Shirley, T. D Black, S. Hollasch). На первом этапе была реализована базовая логика:

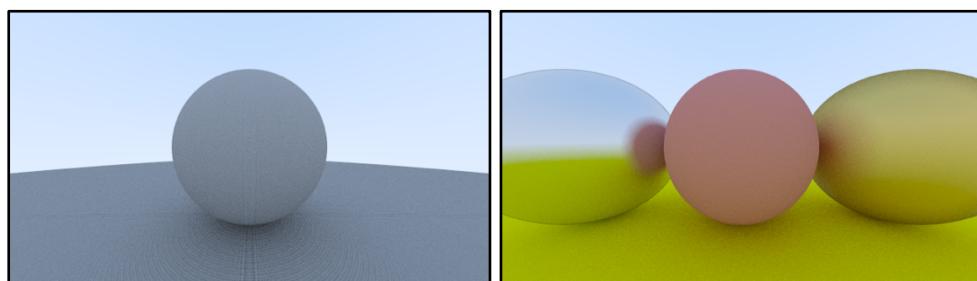


Рисунок 8,9. Первые срендеренные изображения

1. Работа лучей из камеры, аккумулирующих свет, из-за чего мы видим тени
2. Отражение с эффектом размытия – то, что характерно для металла

- отрисовка простейших геометрических форм (шар);
- создание стандартных материалов: металл, стекло, ламбертова поверхность (Lambertian surface);
- движение лучей из камеры, аккумулирующих цвет и свет от пересеченных объектов;
- пересечение лучей с объектами, преломление и отражение от различных материалов

важно сказать, что отражение происходит по-разному, в зависимости от отражающих свойств материалов – для некоторых действует закон угол падения равен углу отражения (например, стекло), однако в большинстве случаев, отраженный луч отклоняется от этого угла на некоторую случайную величину (как пример такого материала - метал)

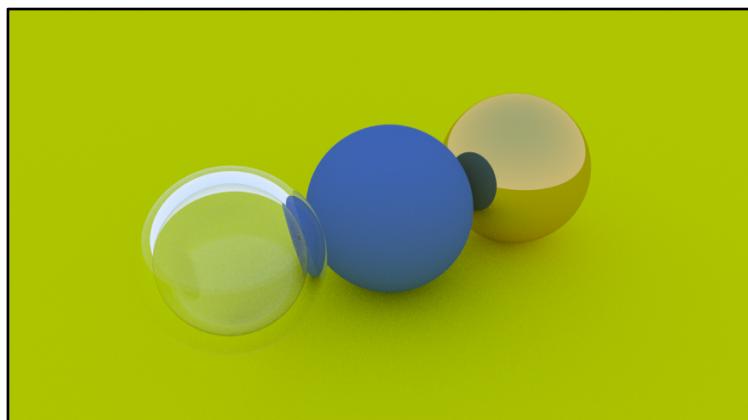


Рисунок 10. Стандартные материалы: стекло, Lambertian material, металл

4.1.2. Добавление новых функций в 3D движок.

На втором этапе стояли следующие задачи:

- добавить новые геометрические формы – треугольники, чтобы была возможность рендерить более сложные сцены;
- реализовать эффекты, которые используются в современных 3D движках: глубина резкости, размытие в движении;

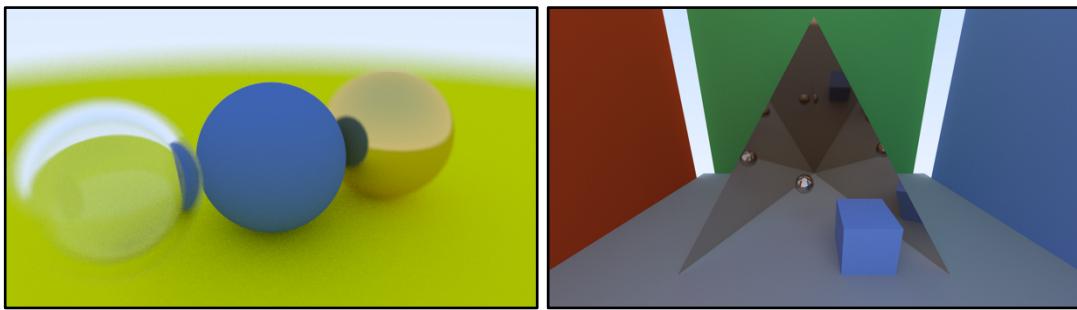


Рисунок 11, 12.

1. Эффект глубины резкости
2. Добавление в сцену треугольников

- добавить возможность загрузки сцен и материалов из файлов (в данном случае, использовались форматы .obj / .mtl с применением библиотеки tinyobjloader);
- добавить материалы со свечением, источники света;

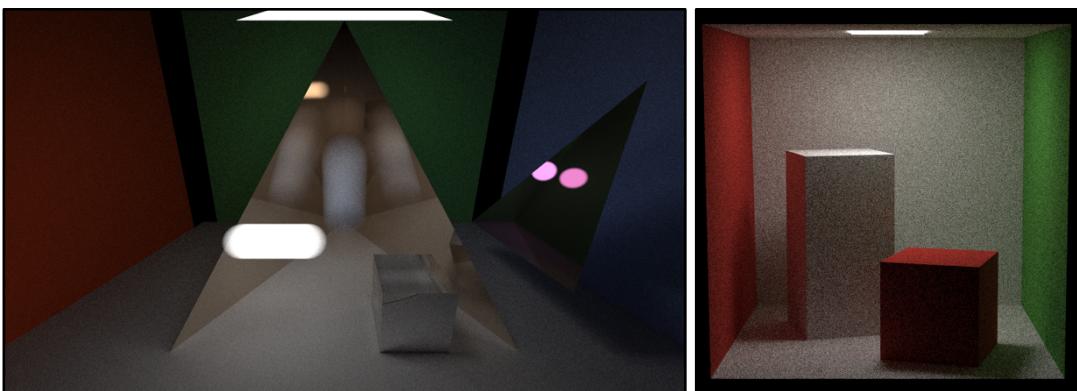


Рисунок 13, 14.

1. Добавление источников света, а также размытия в движении
 2. Сцена и материалы, загруженные из файла
- добавить поддержку и обработку нормалей у вершин треугольников (они используются для рендеринга закругленных форм);

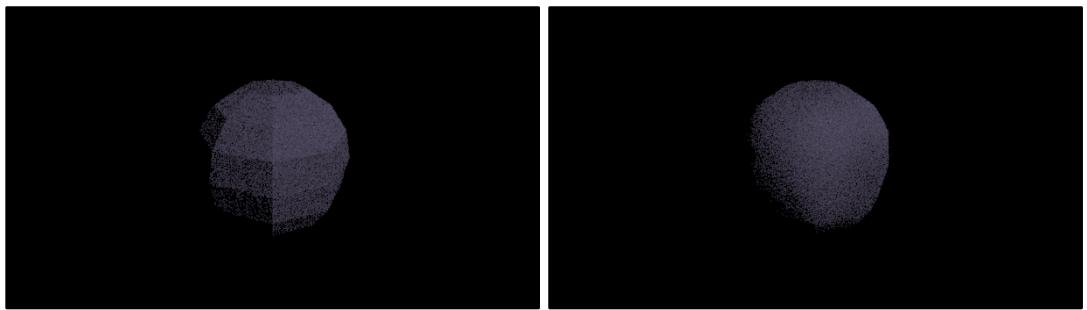


Рисунок 15, 16.

1. Шар из треугольников без нормалей в вершинах
2. Шар из тех же треугольников с поддержкой нормалей

- изменить использующуюся цветовую модель на более современную и более подходящую для нашей задачи модель Фонга (Phong model)

Эта модель гораздо более правильная с точки зрения физики, а так как мы хотим добиться изображений максимально приближенных к реальности, то использовать ее было просто необходимо. Также, в .mtl файлах материалы задаются именно в таком формате.

После того, как все вышеперечисленные пункты были сделаны, у нас появилась единственная функция, которая отвечает за всю генерацию случайных чисел в нашем движке.

4.2. Использование случайных чисел в 3D движке

Во время написания кода 3D движка, были явно замечены места, где используются случайные числа в нашем коде, они уже были подробно описаны ранее, поэтому здесь только их краткое описание:

- Начальная точка в пикселе, через которую пройдет луч, а также выбор направления, в котором полетит отраженный от объекта луч – дименшены 0 и 1;
- Материалы – выбор направления при случайному отражении от поверхности – дименшены 5, 6;

- Время запуска луча – требуется для создания эффекта размытия в движении, чтобы однозначно понимать, в каком месте находится объект в данный момент времени – дименшен 9.

4.3. Генерация случайных чисел

Среди различных подходов к генерации случайных чисел были выбраны: Halton sampler, Scrambled Halton sampler (с использованием RandomDigit и Owen скремблинга) и Blue Noise Sampler. Также для проведения сравнения, использовался встроенный генератор случайных чисел из C++, так как по умолчанию в проектах чаще всего используется именно он.

Как было написано выше: наша цель - сгенерировать хорошо распределенный рандомизированный набор точек, то есть набор точек, в котором почти отсутствуют точки, находящиеся очень близко друг к другу, и при этом нет регионов на координатной плоскости без точек.

4.3.1. Standard sampler

Базовый алгоритм, который используется по умолчанию в программах рендеринга – генерация случайных чисел, за счет встроенных в язык функций.

В качестве стандартного генератора в данном проекте использовался один из наиболее популярных генераторов в C++, работающий на основе линейного конгруэнтного метода. Он был открыт в 1969 году учеными P.A.W.Lewis, A.S.Goodman, J.M.Miller, а в 1988 был принят в качестве «минимального стандарта» Кейтом Миллером и Стивеном Парком.

```
float StandardRand() {
    static std::linear_congruential_engine<uint32_t, 16807, 0, 2147483647> engine;
    static std::uniform_real_distribution<float> distribution(0, 1);
    return distribution(engine);
}
```

Рисунок 17. Реализация Standard Sampler

4.3.2. Halton sampler

Первый алгоритм, который нам подходит - генератор случайных чисел Хальтона. Он генерирует последовательность точек с низким расхождением, которые последовательно хорошо распределены. Генерация таких точек основана на развороте битов числа (radical inverse) в системе счисления по основанию некоторого заранее заданного простого числа. Для разных дименшенов используются разные простые числа. В нашем случае для первых 32 дименшенов мы использовали первые 32 простых числа (2, 3, 5, 7, ... 131).

Подробнее о самом алгоритме: Radical inverse – это разворот битов числа, то есть, если у нас было число $a = d_N(a)d_{N-1}(a) \dots d_0(a)$, где d_i – i-ый разряд числа (цифра от нуля до максимальной цифры в некоторой системе счисления b), то после разворота битов у нас получится $0.d_1(a)d_2(a) \dots d_N(a)$.

Более формально:

$$a = \sum_{i=1}^N d_i(a)b^{i-1} \Rightarrow \text{RadicalInverse}(a) = \sum_{i=1}^N d_i(a)b^{-i}$$

Как можно заметить после переворота нашего числа мы получаем новое число, которое лежит в полуинтервале [0; 1).

Чтобы получить последовательность Хальтона с низким расхождением нам требуется посчитать RadicalInverse от последовательности натуральных чисел.

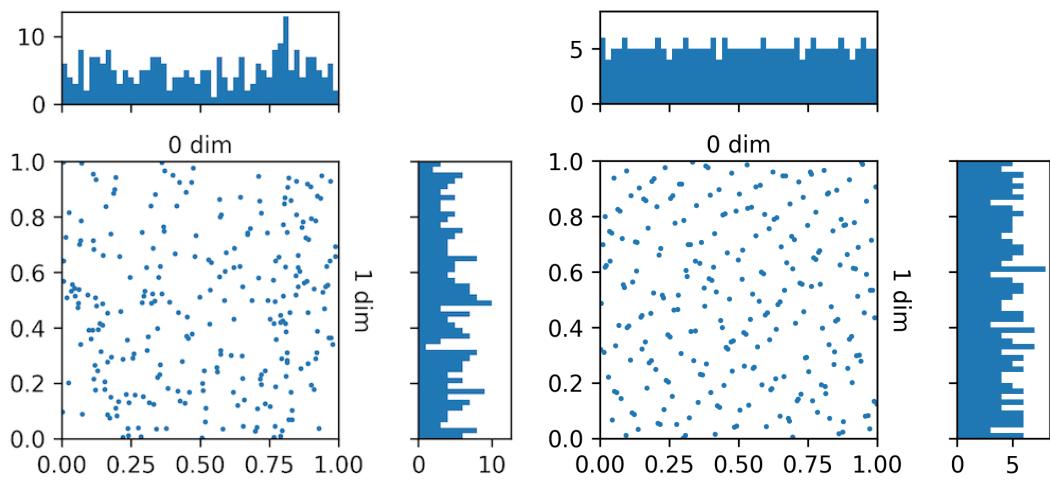


Рисунок 18. Слева – точки, сгенерированные стандартным генератором
Справа – Halton Sampler (нулевой и первый дименшен)

Построим графики распределения точек в первых двух дименшенах. Явно видно, что последовательность Хальтона более хорошо покрывает пространство.

То же самое показывают графики справа и сверху, которые демонстрируют распределение точек каждого отдельного дименшена. Однако тут заметны и минусы алгоритма Halton Sampler – точки кучкаются и образуют некоторый паттерн, что плохо для нашего движка, так как если соседние пиксели будут с одним паттерном, то этот паттерн станет заметен и на самом изображении.

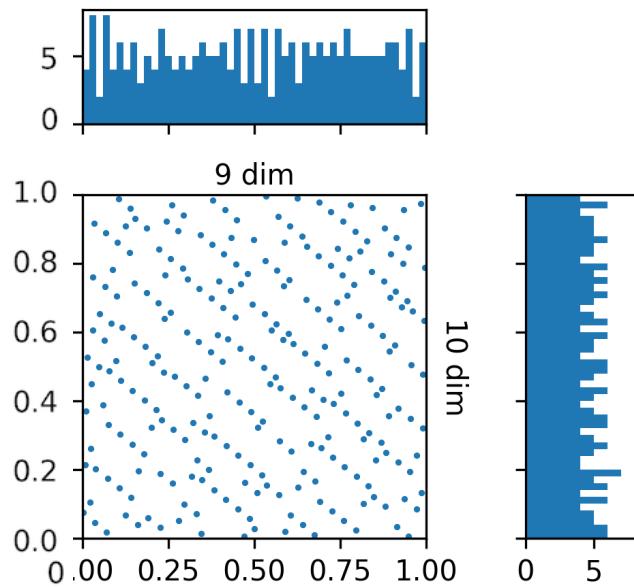


Рисунок 19. Halton Sampler для дименшеноов 9 и 10

Для следующих дименшенов ситуация становится еще хуже. В 9 и 10 дименшене мы получаем следующую ситуацию: точки становятся строго структуризованными и появляется явно различимый паттерн.

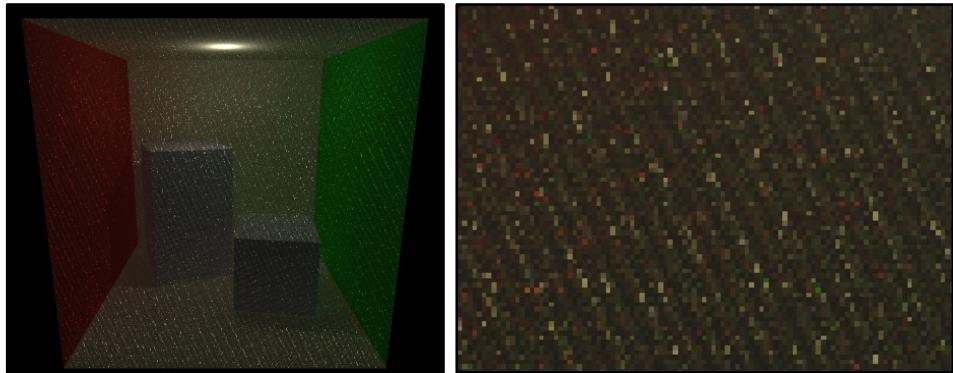


Рисунок 20, 21.

1. Сцена, срендеренная с использованием Halton Sampler
2. Более крупно видимый паттерн на изображении

Как было сказано ранее, если при выборе случайных чисел виден некоторый паттерн, то некоторый паттерн и структурированность будет явно видна и на результирующем изображении. В качестве примера – изображение, при рендеринге которого использовался Halton Sampler. Мы получаем пиксели примерно одного цвета, которые вместе образуют целые линии на изображении.

```
float HaltonRand(uint32_t value, const uint32_t base) {
    const float invBase = (float) 1 / (float) base;
    float result = 0;
    float invBaseN = invBase;

    while (value) {
        uint32_t next = value / base;
        uint32_t digit = value - next * base;
        result += invBaseN * digit;
        invBaseN *= invBase;
        value = next;
    }

    return std::max(std::min(result, 1.f - EPSILON), 0.f);
}
```

Рисунок 22. Реализация Halton Sampler

4.3.3. Halton Sampler с использованием RandomDigit скремблинга

Чтобы убрать появляющиеся паттерны зачастую используется скрембллинг, то есть перемешивание каким-то образом получающихся значений.

Первым и одним из наиболее простых и понятных является RandomDigit скрембллинг.

Для него заранее генерируется массив, размер которого равен основанию текущей системы счисления (b). В нем, для каждой цифры из b -ичной системы счисления взаимно-однозначно сопоставляется случайно выбранная цифра этой же системы счисления. Такие же массивы делаем для всех дименшенов.

То есть, если мы работаем в системе счисления по основанию 5, то массив может быть, например [3, 0, 4, 1, 2]. Соответственно, когда при развороте битов мы берем некоторую цифру, то мы берем не эту цифру, а ту, которая находится в нашем массиве по индексу нашей цифры. Если мы встретили цифру 3, то возьмем вместо нее 1 из нашего массива.

В результате, с помощью таких несложных операций (и по времени, и по дополнительной памяти), мы получаем следующие распределения:

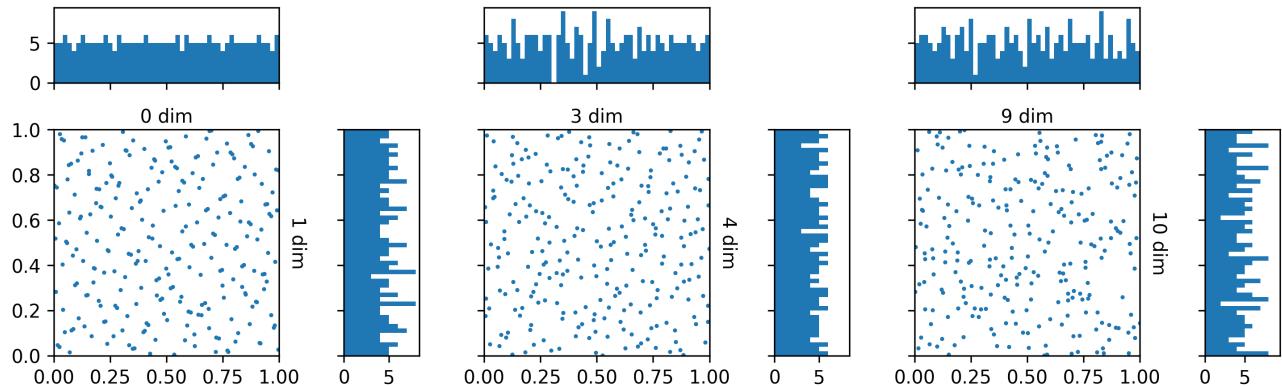


Рисунок 23. Распределение случайных чисел по дименшенам (0-1, 3-4, 9-10) с использованием RandomDigit скремблинга.

Как можно заметить, мы справились с задачей разбиения явно структурированных групп точек, однако все равно, в высоких дименшенах у нас не получается хорошее распределение – остаются области без точек.

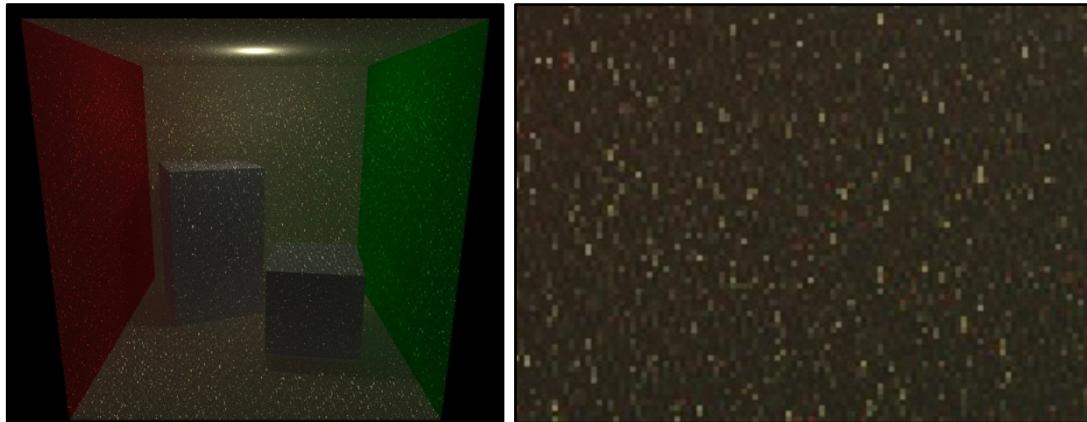


Рисунок 24, 25.

1. Сцена срендеренная с использованием Halton RandomDigit алгоритма
2. То же самое место, что и в изображении Halton Sampler



Рисунок 26. Оставшийся паттерн на срендеренном изображении

Если смотреть, на получившееся изображение, то можно заметить, что паттерн, который был в Halton Sampler мы на самом деле убрали и очень хорошо. Однако, появился другой, менее заметный.

```

float HaltonRandomDigitScrambling(uint32_t value,
                                    const uint32_t base,
                                    const std::vector<uint32_t> &permutation) {
    const float invBase = (float) 1 / (float) base;
    float result = 0;
    float invBaseN = invBase;
    while (value) {
        uint32_t next = value / base;
        uint32_t digit = (value - next * base);

        digit = permutation[digit];

        result += invBaseN * digit;
        invBaseN *= invBase;
        value = next;
    }
    return std::max(std::min(result, 1.f - EPSILON), 0.f);
}

```

Рисунок 27. Реализация Halton Sampler с использованием Random Digit скремблинга

4.3.4. Halton Sampler с использованием Owen скремблинга

Вторым более популярным и часто использующимся скремблингом является Owen скремблинг. Основная идея, в том, что при получении следующего числа мы не только совершаем какую-то операцию с текущим разрядом, но и смотрим и учитываем предыдущий результат. Существуют разные имплементации, в проекте использовалась одна из них (из блога Psychopath Renderer), немного переделанная.

В данном алгоритме при развороте бита мы смотрим не только на текущую цифру, но и на уже развернутые биты. То есть алгоритм при развороте i -го бита следующий:

1. Посчитаем вспомогательное значение, равное XOR от заранее заданного числа (некоторая 32-битная константа, разная для каждого дименшена) и уже развернутых битов (результата на предыдущей итерации).
2. Перемешаем биты получившегося в прошлом пункте числа, используя Laine-Karras hash. Он заключается в том, что мы перемешиваем биты

нашего числа с помощью операций XOR, умножения на нечетную константу, прибавления константы, а также XOR на наше число, умноженное на четную константу.

3. Чтобы получить новую цифру, мы обращаемся к массиву перестановок для нашей системы счисления (аналогичному тому, что используется в RandomDigit) по индексу равному сумме текущей цифры и получившемуся во втором пункте значения, все это берется по модулю от основания нашей системы счисления.

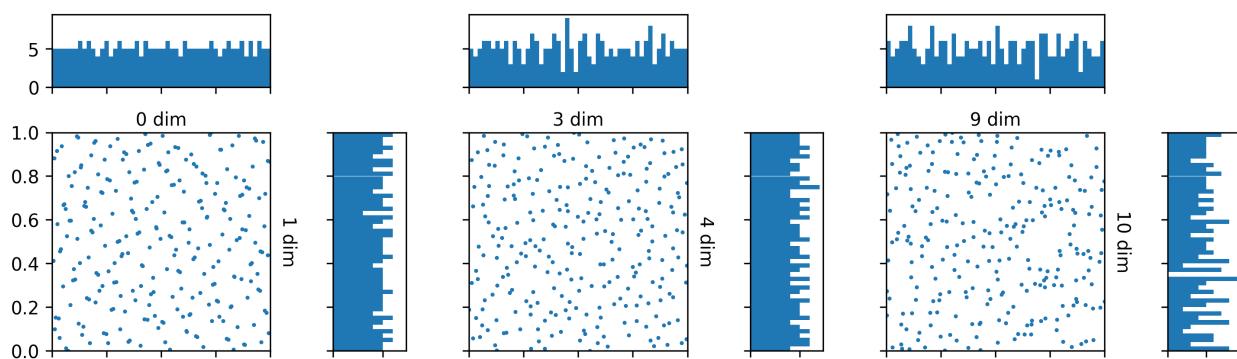


Рисунок 28. Распределение случайных чисел по дименшенам (0-1, 3-4, 9-10)
с использованием Owen скремблинга.

В результате мы получаем довольно хорошую картинку распределения, областей без точек у нас получилось меньше чем в RandomDigit скремблинге. А также у нас практически нет структурированных паттернов. Соответственно, появляется надежда на то, что и результаты у нас будут лучше.

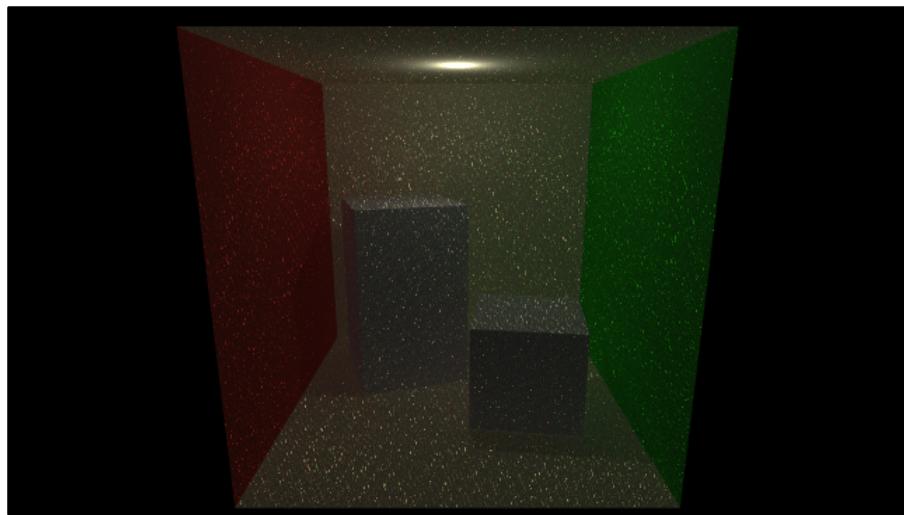


Рисунок 29. Изображение, срендеренное с использованием Owen скремблинга

На результирующем изображении видно, что мы пришли к довольно хорошему результату, однако появился другой видимый паттерн.

```
float HaltonOwenScrambling(uint32_t value,
                           const uint32_t base,
                           const std::vector<uint32_t> &permutation,
                           uint32_t hash) {
    const float invBase = (float) 1 / (float) base;
    uint32_t result = 0;
    float invBaseN = 1;
    while (value) {
        uint32_t next = value / base;
        uint32_t digit = value - next * base;

        uint32_t digit_hash = MixBits(hash ^ result);
        digit = permutation[(digit + digit_hash) % permutation.size()];

        result = result * base + digit;
        invBaseN *= invBase;
        value = next;
    }
    return std::max(std::min(result * invBaseN, 1.f - EPSILON), 0.f);
}
```

Рисунок 30. Реализация Halton Sampler с использованием Owen скремблинга

4.3.5. Blue Noise sampler

Заключительным алгоритмом выбора случайных чисел стал Blue Noise sampler. Основная идея алгоритма – заполнить текстуру хорошо

распределенными точками, сгенерировав некоторые начальные случайные точки и заполнив остальные точки, основываясь на том, что следующая точка должна находиться как можно дальше от всех существующих.

Blue Noise получил такое название, так как он содержит большее число высоких частот и, соответственно меньшее число низких частот. Это очень похоже на синий свет, который содержит более высокочастотный свет

В данном методе подход к генерации случайных чисел отличается от предыдущих: мы заранее генерируем некоторую текстуру с хорошим паттерном, а затем используем ее.

Генерация текстуры происходит следующим образом:

1. Выбирается размер текстуры, например, возьмем 128x128 пикселей. Генерируется нулевая матрица такого размера.
2. Создаем дополнительную переменную, отвечающую за индекс текущей точки в нашей текстуре. Изначально, она равна нулю.
3. Обычным алгоритмом выбора случайных чисел выбираем несколько начальных точек на нашей текстуре (количество можно выбирать разное, для размера нашей текстуры подойдет 16 точек).
4. Создаем дополнительную двумерную матрицу такого же размера и записываем в выбранные точки очень большие значения ($+\infty$), а также «окрашиваем» точки, которые лежат рядом с ними, по принципу: чем дальше от точки, тем меньше значение. В основной текстуре также записываем в те же точки значения индекса текущей точки, каждый раз прибавляя 1, после записи в текстуру.
5. Выбираем в нашей дополнительной матрице точку с наименьшим значением и записываем в текстуру значение индекса текущей точки.
6. Увеличиваем индекс текущей точки на 1.

7. В дополнительной матрице ставим $(+\infty)$ в выбранную точку, а также окрашиваем точки вокруг по принципу: чем дальше от точки, тем меньше значение.
 8. Повторяем шаги 5 – 7 пока не будут заполнены все точки в нашей текстуре.
 9. Делим каждое значение в нашей текстуре на размер текстуры \Rightarrow получаем значения в полуинтервале $[0;1)$ в нашей текстуре.
10. Текстура готова, пример получившегося результата:

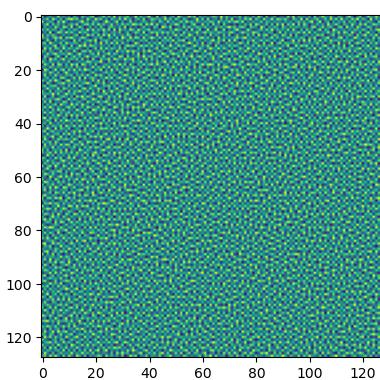


Рисунок 31. Blue Noise текстура

В результате работы Blue Noise у нас получается такая текстура, можно явно увидеть заметный и характерный для нашего алгоритма паттерн. В данном случае он играет положительную роль, так как, учеными доказано, что фоторецепторы в наших глазах лучше воспринимают и дорисовывают синий шум, что происходит из-за строения нашего глаза. Именно поэтому такой паттерн хорошо согласуется с нашим восприятием и, если мы увидим его на картинке, то визуально результаты будут лучше, чем у других алгоритмов.

Для того чтобы получить случайное значение из такой текстуры, необходимо обратиться к ней по индексу от индекса нашего текущего пикселя на экране, взятого по модулю от размера нашей текстуры. При таком выборе соседние пиксели будут буквально соседними пикселями в нашей текстуре, и мы явно увидим паттерн Blue Noise алгоритма в нашем изображении.



Рисунок 32, 33.

1. Изображение, сгенерированное с применением Blue Noise Sampler
2. Более крупно явно видимый паттерн, присутствующий на полу (добавлена яркость)

Нам удалось добиться переноса паттерна Blue Noise с текстуры в изображение и получить визуально более приятную глазу картинку.

```
float BlueNoiseRand(const SamplerState &currentState, uint32_t Dim) {
    size_t y = currentState.seed / IMAGE_WIDTH;      // В currentState.seed хранится
    size_t x = currentState.seed % IMAGE_WIDTH;      // линейный индекс пикселя
    size_t row = (y + (uint32_t) Dim * 100937
                  + currentState.sampleIdx * 1091
                  + currentState.depth * 133337) % blue_noise_texture.size();
    size_t column = (x + (uint32_t) Dim * 99523
                     + currentState.sampleIdx * 2399
                     + currentState.depth * 2549) % blue_noise_texture[0].size();
    return blue_noise_texture[row][column];
}
```

Рисунок 34. Реализация Blue Noise Sampler

В реализации явно видно, что при выборе текущего ряда и столбца мы берем x и y без умножения на константы => для соседних пикселей будут взяты соседние пиксели. sampleIdx, depth взяты с умножением на большое простое число для того, чтобы для разных сэмплов и разных глубин (количество отражений и преломлений текущего луча) не дублировались.

5. Результаты. Сравнение алгоритмов.

На основании всех предыдущих шагов и исследований, описанных в курсовой работе, мы получили изображения, сгенерированные с использованием различных методов выборки случайных чисел.

При работе с изображениями есть несколько вариантов сравнения изображений. Первым и самым очевидным является визуальное сравнение, затем идет сравнение по различным метрикам, будут использованы MSE и FLIP.

5.1. Визуальное сравнение

Во время визуального сравнения требуется просто посмотреть, какие изображения получились ближе к оригиналу, какие лучше или хуже, присутствуют ли паттерны. Если присутствуют паттерны, то как они влияют на качество изображения.

Сначала рассмотрим сцену «Волк». Стоит отметить, что эта сцена была сделана в программе Blender, после чего загружена в нашу программу при помощи сохраненных .obj / .mtl файлов. В ней присутствует большое количество полигонов – треугольников, из которых и был сделан волк. Как раз для таких сцен мы и пытаемся сделать процесс рендеринга быстрее, чем меньше лучей через пиксель (SPP – samples per pixel, сэмплов) мы запускаем, тем быстрее рендерится картинка.

В качестве второго примера мы рассмотрим ставшую уже стандартной для графики и рендеринга сцены – Cornell Box. Она тоже была загружена из Blender. В ней присутствуют и тени, и отражения света от стен, из-за чего цвет объектов отражается на рядом стоящих.

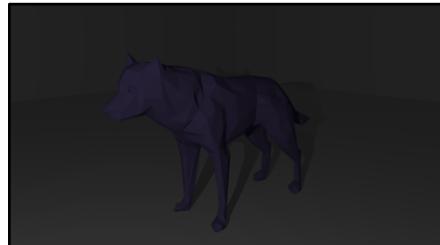
5.1.1. Сравнение методов выборки на сцене «Волк»

Сцена «Волк» при SPP = 1

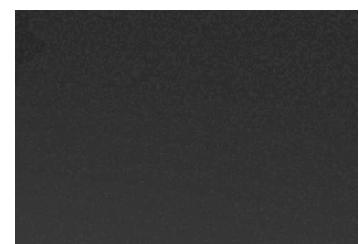
Рисунок 35. Изображение

Пол справа от волка укрупненно
(добавлена яркость)

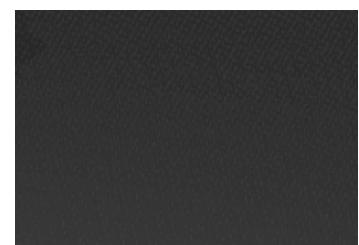
Оригинал



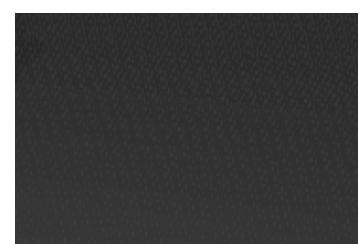
Standard
Sampler



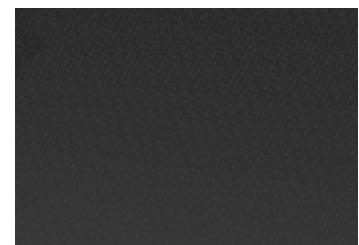
Halton
Sampler



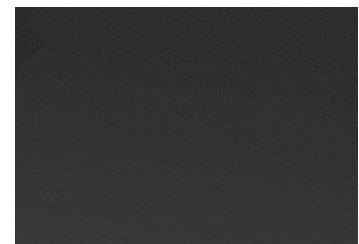
Halton
RandomDigit
Sampler



Halton Owen
Sampler



Blue Noise
Sampler



Как можно заметить, даже при 1 луче на пиксель, мы уже получаем интересные результаты.

Одновременно с этим, сразу же видно, что Halton Random Digit Sampler и Halton Owen Sampler показывают очень плохие результаты в этой сцене. Они разбивают паттерн Хальтона, однако образуют новый, то есть не справляются со своей задачей в данном примере. Мы еще увидим их результаты на другой сцене.

Стандартный сэмплер оставляет слишком много хорошо видимого шума, что понятно (1 луч на пиксель – это очень мало), сэмплер с использованием алгоритма Хальтона оставляет характерный для алгоритма Хальтона паттерн – пиксели образуют линии одного цвета на изображении, которые очень хорошо видны. А Blue Noise уже очень хорошо справляется с задачей, мы спокойно разглядываем паттерн Blue Noise текстуры, однако шум, который она образует на картинке гораздо меньше и менее заметен, чем при стандартном генераторе.

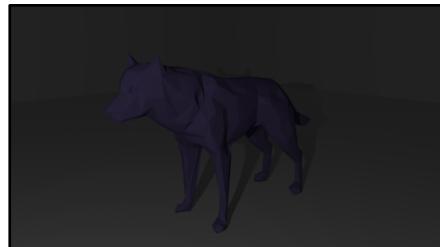
Рассмотрим, какие результаты мы получаем при увеличении количества лучей на пиксель.

Сцена «Волк» при SPP = 8

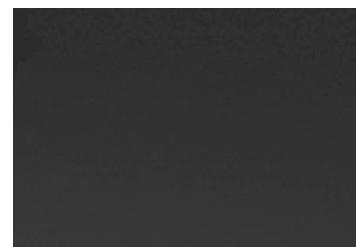
Рисунок 36. Изображение

Пол справа от волка
укрупненно (добавлена
яркость)

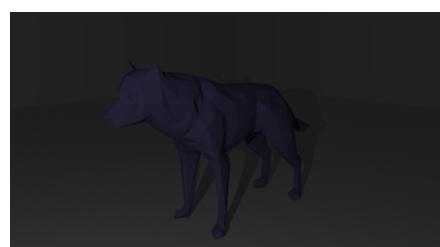
Оригинал



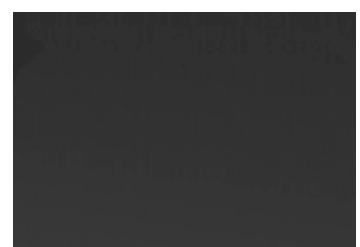
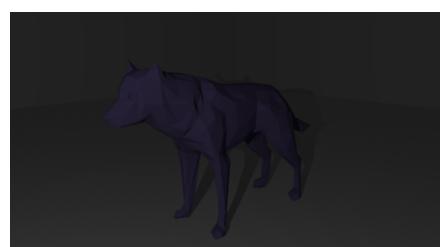
Standard
Sampler



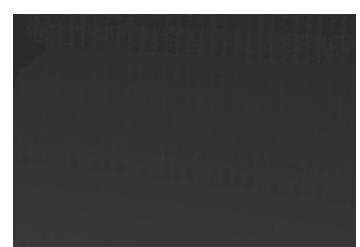
Halton
Sampler



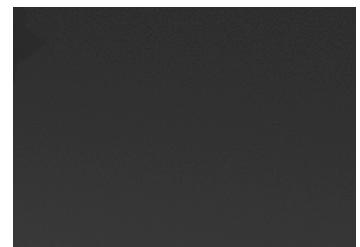
Halton
RandomDi
git Sampler



Halton
Owen
Sampler



Blue Noise
Sampler



При 8 лучах через каждый пиксель мы уже видим те результаты, которых хотели достичь. Картинки получаются визуально очень близкими к оригиналам у некоторых сэмплеров.

На картинке стандартного сэмплера все еще видно присутствующий шум, он стал гораздо меньше, по сравнению с картинкой с $SPP = 1$, но все равно, для достижения хорошего качества требуется гораздо больше лучей.

Halton Sampler также улучшил свое качество по сравнению с $SPP = 1$, но линии на изображении все еще остаются, а это явный минус и показатель, что сэмплов необходимо гораздо больше. RandomDigit и Owen скремблнги опять же не показывают хорошие результаты, однако можно сказать, что их изображения чуть лучше, чем у обычновенного Хальтона (паттерн стал чуть менее различим, несмотря на то, что виден хорошо).

А самым хорошим из всех алгоритмов стал Blue Noise Sampler. Изображение за 8 лучей через пиксель получилось очень хорошим и близким к оригиналам. Паттерн Blue Noise текстуры виден при приближении, а издалека практически незаметен. То есть еще несколько SPP , и мы получим идеальную картинку. Соответственно, можно сказать, что, оценивая чисто визуально мы за меньшее число сэмплов получили результат лучше, чем стандартный метод выбора и алгоритм Хальтона. То есть ускорили наш алгоритм рендеринга, за счет применения Blue Noise Sampler.

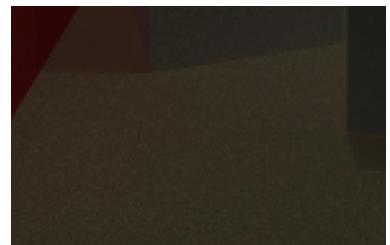
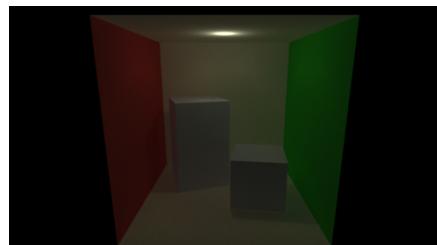
5.1.2. Сравнение методов выборки на сцене «Cornell Box»

Сцена «Cornell Box» при SPP = 8

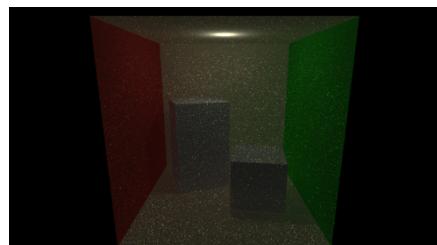
Рисунок 37. Изображение

Пол слева снизу

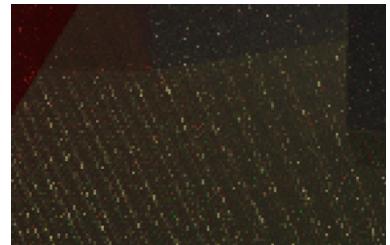
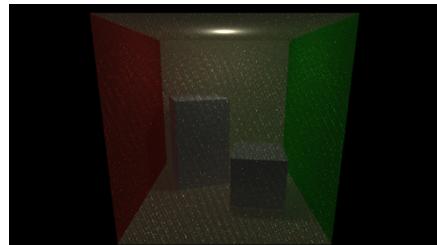
Оригинал



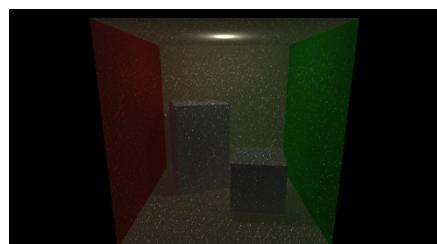
Standard
Sampler



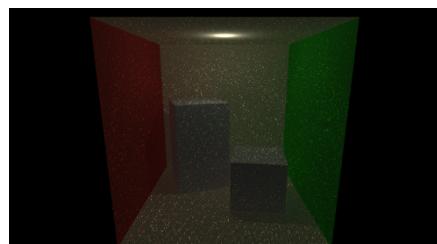
Halton
Sampler



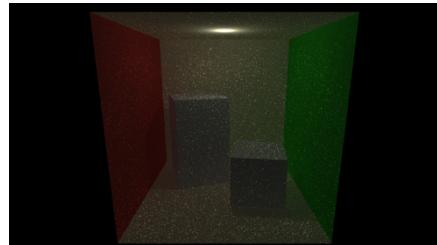
Halton
RandomDi
git Sampler



Halton
Owen
Sampler



Blue Noise
Sampler



Cornell Box не такая сложная сцена, как «Волк» по количеству полигонов, однако по количеству отражений она гораздо сложнее, так как объекты находятся внутри некоторой коробки. То есть лучи при попадании в эту коробку начинают отражаться от объектов, поэтому используется больше случайных чисел.

Как мы видим, Halton Sampler показывает характерный паттерн, из-за чего страдает качество. Однако, в этой сцене мы уже получаем очень хорошие результаты у алгоритмов скремблиングа.

Halton RandomDigit Sampler очень хорошо разбивает паттерн Хальтона, а так же по качеству не только не уступает Standard Sampler, но и чисто визуально превосходит его.

Halton Owen Sampler не так хорош, как RandomDigit, однако тоже справился с задачей убрать чистый паттерн Хальтона, но получил новый менее заметный, но заметный. Этот паттерн хоть и присутствует на картинке, однако он не уступает стандартному методу выбора.

Blue Noise Sampler, как и в сцене «Волк» показывает отличные результаты, не наблюдаются явные паттерны, а так же изображение получилось более приближенным к оригиналу, по сравнению с Standard sampler.

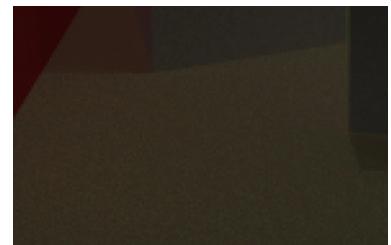
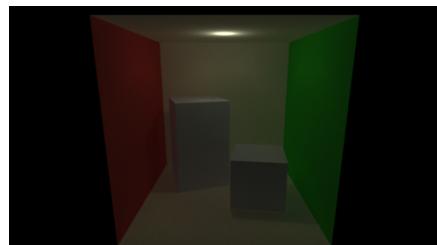
Посмотрим, что происходит на большом количестве SPP.

Сцена «Cornell Box» при SPP = 128

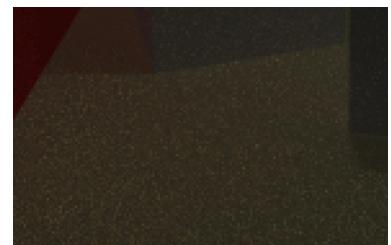
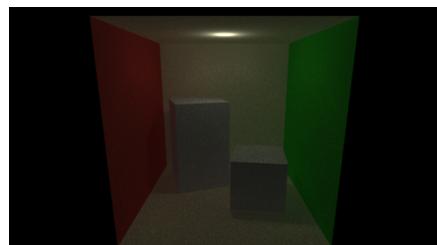
Рисунок 38. Изображение.

Пол слева снизу

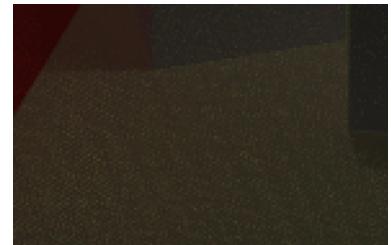
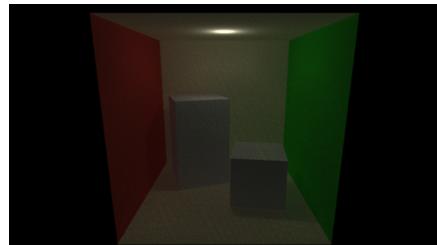
Оригинал



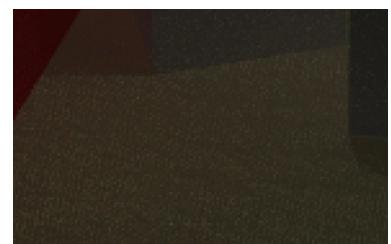
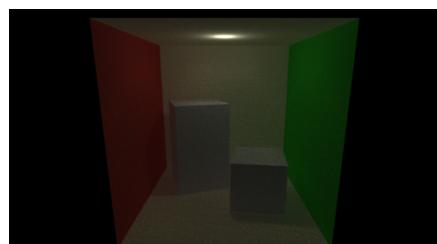
Standard
Sampler



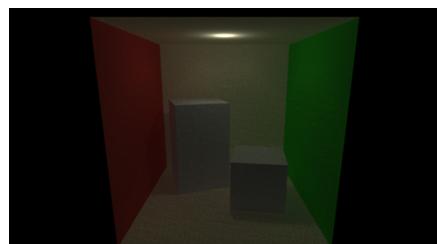
Halton
Sampler



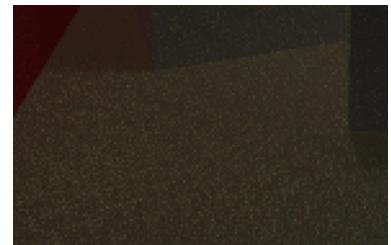
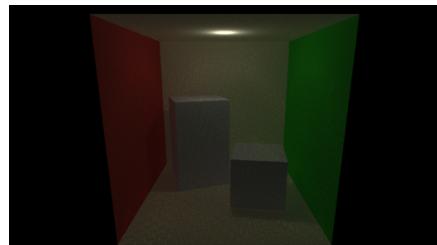
Halton
RandomDi
git Sampler



Halton
Owen
Sampler



Blue Noise
Sampler



Здесь мы видим, что Standard Sampler все равно оставил довольно много шумов на изображении, хотя само изображение за 128 сэмплов получилось уже довольно хорошим.

Halton Sampler практически растворил свой паттерн, но он все равно заметен, чисто визуально результаты чуть хуже, чем при стандартном выборе случайных чисел.

Halton RandomDigit Sampler прекрасно справился с паттерном Halton и получил очень хороший результат, изображение ближе к оригинал, чем при Standard Sampler.

Halton Owen Sampler - результаты схожи с RandomDigit алгоритмов, изображение практически отличное, никаких паттернов не наблюдается.

Blue Noise Sampler - мы видим паттерн Blue Noise текстуры, но несмотря на это, изображение чуть лучше, чем при Standard Sampler, и чуть хуже, чем при Halton RandomDigit и Owen Samplers.

Мы однозначно можем сказать, что для сцены Cornell Box улучшили время работы, за счет уменьшения количества сэмплов для получения хорошего результата в алгоритмах Halton RandomDigit и Halton Owen. Blue Noise также показал хорошие результаты и получил хорошее изображение за меньшее число сэмплов, по сравнению с Standard Sampler, однако проиграл алгоритмам скремблиинга последовательности Хальтона.

5.1.3. Результаты визуального сравнения

Как мы видим, мы на самом деле смогли получить более хорошие результаты за счет изменения метода выбора случайных чисел в нашем 3D движке.

Самые лучшие результаты показал алгоритм с использованием Blue Noise техники. Его изображения оказывались стабильно лучше, чем изображения с

использованием стандартного метода выбора случайных чисел. То есть мы получали лучшие изображения за меньшее число сэмплов \Rightarrow стабильно выигрывали во времени рендеринга изображений до хорошего результата.

Также хорошие результаты показали Halton RandomDigit Sampler и Halton Owen Sampler. Их результаты были либо на уровне Standard Sampler, либо превосходили его, однако были сцены, на которых их результаты были и чуть хуже, чем Standard Sampler. Соответственно, при использовании данных методов выбора случайных чисел стоит аккуратно смотреть, получаем ли мы прирост в качестве, или наоборот, теряем его. Однако, как можно заметить, в сценах, в которых мы получаем этот прирост, он очень значительный.

5.2. Сравнение с использованием метрик

Обратимся к математически строгим методам сравнения: метрикам MSE и FLIP. Эти две метрики смогут показать реальную ситуацию, что произошло с нашим изображением, получаем ли мы прирост в плане чистого сравнения изображений.

5.2.1. Метрика MSE

Это одна из самых популярных метрик, использующихся повсеместно для вычисления среднеквадратичной ошибки, а затем сравнения этих ошибок между собой.

В данном случае, если говорить про графику, то мы будем сравнивать между собой значения цветов пикселей. Каждый пиксель у нас задан четырьмя числами: red, green, blue, transparency – красный, зеленый, синий и прозрачность.

Чтобы посчитать MSE мы берем все эти значения, поэлементно вычитаем из значений оригинального изображения значения нашего изображения, затем возводим каждую получившуюся разность во вторую степень и суммируем. В результате получаем одно число – ошибку MSE для нашего изображения.

Требуется отметить, что эта метрика дает очень понятный результат, однако смотрит только на текущий пиксель. Несмотря на это, использовать данную метрику как одну из самых базовых вполне допустимо – мы честно увидим на сколько отличаются наши изображения попиксельно.

```
def MSE(original, ours):

    error = np.sum((original.astype("float") - ours.astype("float")) ** 2)
    error /= original.shape[0] * original.shape[1]

    return error
```

Рисунок 39. Реализация функции MSE для вычисления ошибки в изображениях (Python)

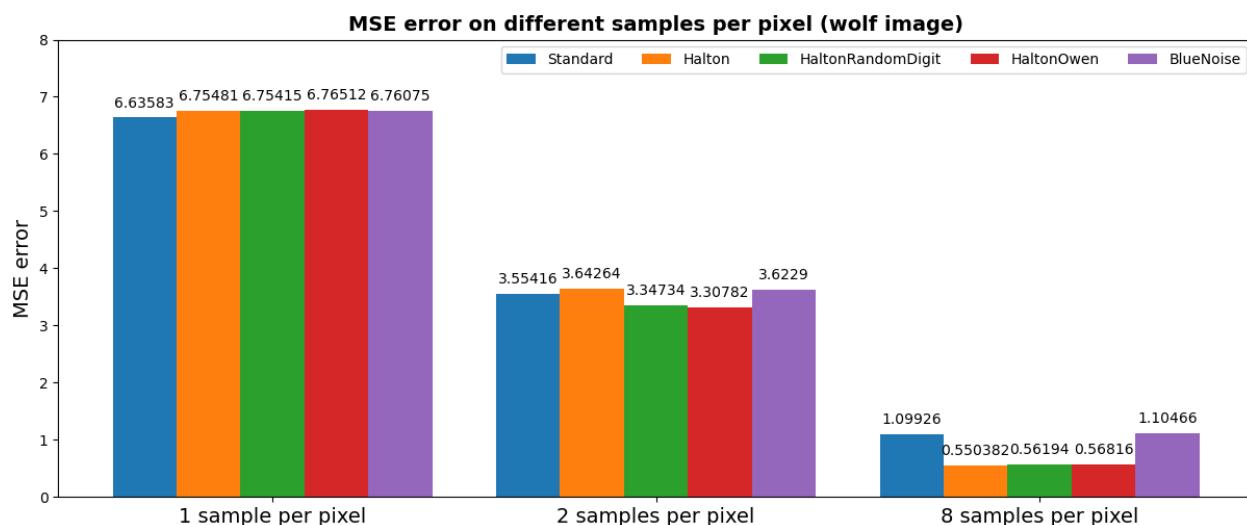


Рисунок 40. График сравнения MSE на сцене «Волк»

На данном графике представлена среднеквадратичная ошибка для различного количества лучей на пиксель, а также для различных методов выбора случайных чисел. Для сравнения использовались результаты для сцены «Волк».

Как мы видим, при $SPP = 1$ и $SPP = 2$ все алгоритмы показывают примерно одинаковые ошибки, а уже при $SPP = 8$ ошибка для алгоритмов Halton, Halton RandomDigit и Halton Owen оказывается в 2 раза меньше, чем аналогичная ошибка для Standard Sampler и Blue Noise sampler. Математически, это можно

интерпретировать как: изображения срендеренные с этими методами выбора будут гораздо ближе к оригинальному изображению. На самом деле, некоторое превосходство Halton Random Digit и Halton Owen появляется еще при SPP = 2, но там оно практически не заметно.

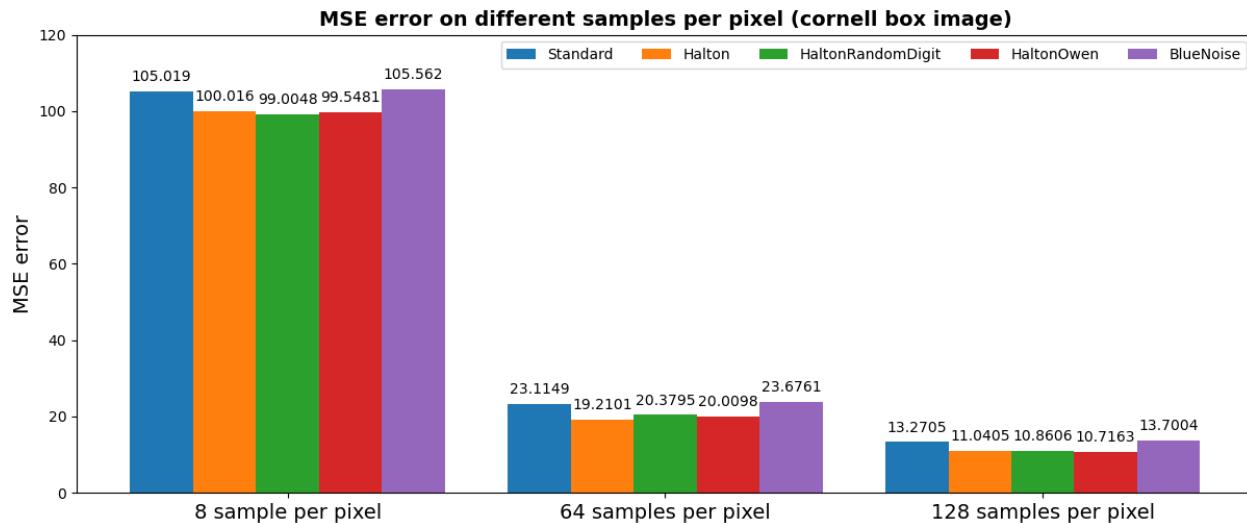


Рисунок 41. График сравнения MSE на сцене «Cornell Box»

Аналогичные результаты мы получаем и на другой сцене – Cornell Box. Тут мы взяли SPP = 8, 64, 128. Как мы видим, тут алгоритмы на основе последовательности Хальтона выигрывают уже более явно и стабильно. Значения ошибок для этих методов выбора случайных чисел заметно ниже, чем у Standard и Blue Noise Sampler.

В итоге, по значениям ошибки MSE можно сказать, что явно выделяются методы выборки Halton, Halton RandomDigit и Halton Owen Sampler. Их среднеквадратичные ошибки стабильно ниже, чем у стандартного метода выбора и Blue Noise Sampler.

5.2.2. Метрика FLIP

Метрика FLIP – одна из наиболее популярных в сфере графики именно для сравнения изображений. Она была представлена в статье 2020 года компанией NVIDIA.

Основное отличие от других алгоритмов сравнения заключается в том, что метрика FLIP оценивает различия между изображениями, не только по цветовым характеристикам, но и основываясь на принципах восприятия изображений человеком.

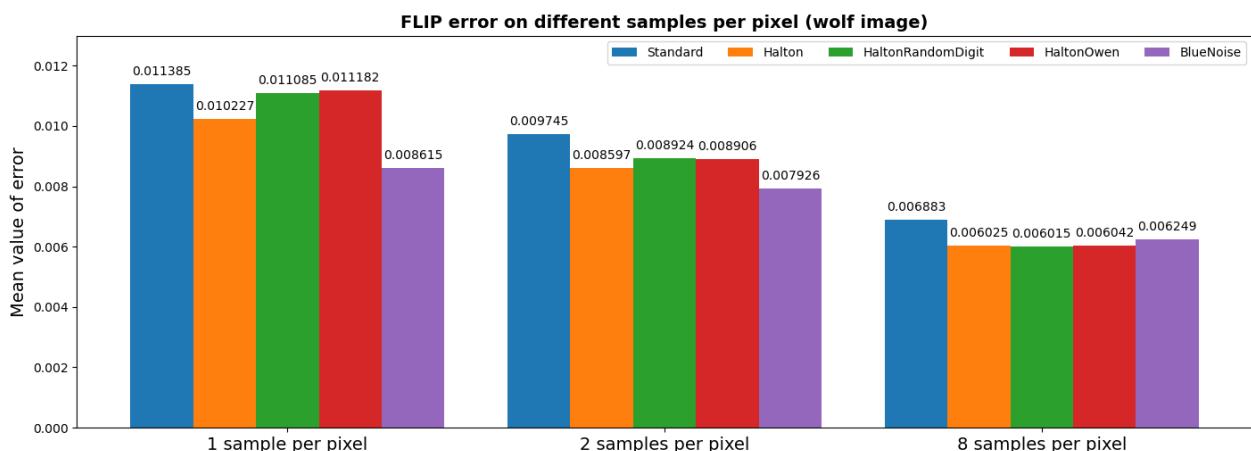


Рисунок 42. График сравнения FLIP на сцене «Волк»

На графике выше показано среднее значение ошибки, в зависимости от количества лучей, а также метода выбора случайных чисел.

Как можно заметить, графики довольно сильно отличаются от тех, что мы видели при MSE, и наблюдаемые тенденции также сильно отличаются.

Алгоритм Хальтона показывает стабильно более хорошие результаты, по сравнению со стандартным генератором случайных чисел.

Halton RandomDigit Sampler и Halton Owen Sampler показывают также неплохие результаты, они лучше стандартного, но хуже метода Хальтона. Несмотря на то, что эти два алгоритма должны были улучшить результаты Хальтона, они не показывают таковых результатов.

Blue Noise Sampler показывает опять же очень хорошие результаты, при маленьком количестве сэмплов, он оказывается самым лучшим среди алгоритмов, но при SPP = 8, его обгоняют методы основанные на последовательности Хальтона.

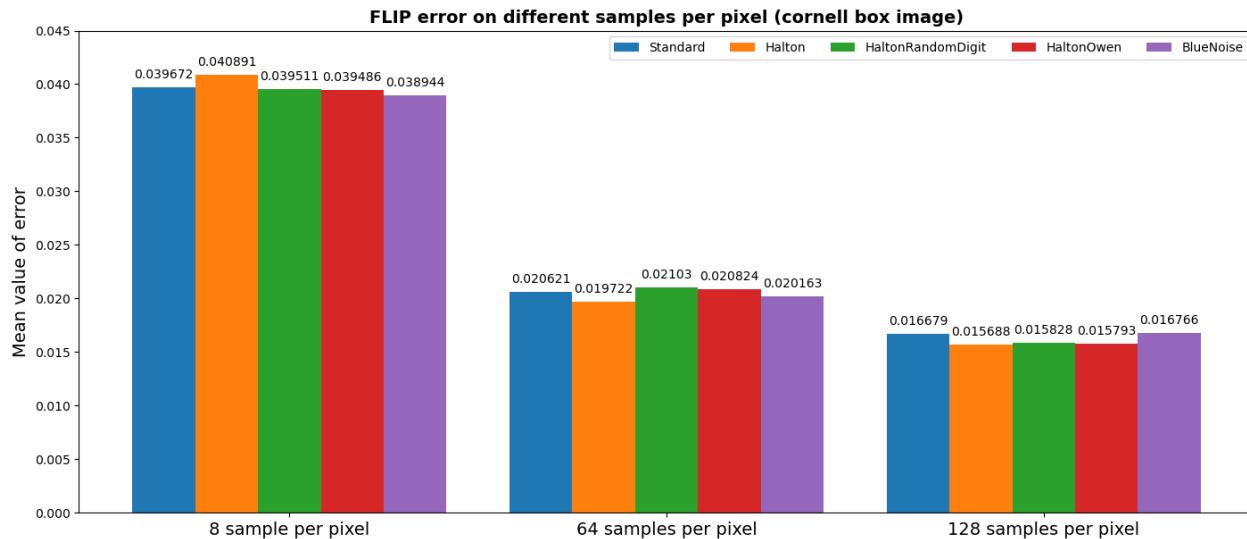


Рисунок 43. График сравнения FLIP на сцене «Cornell Box»

Данный график показывает аналогичную зависимость, но для сцены Cornell Box. При сравнении метрикой FLIP изображений, которые рендерили сцену Cornell Box – хорошо видимых тенденций практически нельзя найти, все ошибки примерно одинаковые. Но немного заметно, как практически всегда наши методы выбора случайных чисел оказываются лучше, чем при стандартном генераторе случайных чисел.

5.3. Результаты сравнений

Исходя из вышеперечисленных способов сравнения изображений и метрик: изображений, которые рендерились с разными методами генерации случайных чисел, можно однозначно сказать, что все исследования, которые мы проводили показывают хорошие результаты. Все наши алгоритмы оказываются в той или иной степени лучше, чем стандартный подход к генерации случайных чисел.

Самым полезным, хочется выделить метод выбора случайных чисел на основе Blue Noise. И визуально, и по метрикам, он практически всегда оказывался не только лучше стандартного алгоритма, но и лучше подходов, основанных на последовательности Хальтона.

Так же хорошими в некоторых случаях оказались Halton RandomDigit Sampler и Halton Owen Sampler, математические значения ошибок показывают, что эти два метода рендерят хорошие изображения. Однако в сценах, где происходит много отражений и преломлений, Halton RandomDigit также показывают прекрасные результаты, а Halton Owen часто оставляет заметные паттерны, поэтому Halton RandomDigit использовать оптимальнее.

Алгоритм Хальтона, может на первый взгляд показаться не таким хорошим, как остальные. Однако важно понимать, что Halton RandomDigit и Halton Owen – это лишь доработки последовательности Хальтона, поэтому Halton Sampler оказался тоже очень полезным для исследования в ходе курсовой работы. А также, на некоторых метриках он тоже показывал хорошие результаты, лучше, чем стандартный алгоритм генерации.

Самым главным результатом данного сравнения можно назвать то, что мы нашли и написали в коде методы выборки случайных чисел, которые рендерят лучшие изображения, по сравнению со стандартным методом генерации случайных чисел, за меньшее число лучей на пиксель, то есть использующие меньшее количество времени.

6. Заключение

Результатами данной курсовой работы стали:

- Подробное изучение существующих подходов к рендерингу изображений, как они устроены изнутри, а также методы их оптимизации и дальнейшего развития;
- Была написана собственная программа для рендеринга изображений на C++ с нуля, с возможностью дополнять ее новыми функциями, а также продолжать работать над ней, как над серьезным проектом;
- Было проведено исследование и поиск алгоритмов для генерации случайных чисел, которые могли бы поспособствовать уменьшению времени рендеринга изображений в хорошем качестве;
- Были выбраны и реализованы на языке C++ 4 метода генерации случайных чисел: Halton Sampler, Halton RandomDigit Sampler, Halton Owen Sampler, Blue Noise Sampler. Они были добавлены в программу рендеринга изображений и с их помощью были срендерены изображения, необходимые для сравнения;
- Было проведено сравнение и анализ написанных методов генерации случайных чисел с использованием современных метрик, использующихся в настоящих проектах по 3D графике;
- Были выделены алгоритмы, которые показывают наилучшие результаты за наименьшее время: Blue Noise Sampler, Halton RandomDigit Sampler.

Все материалы и результаты, которые были получены в ходе данной курсовой работы имеют большой практический смысл и потенциал. На примере простой программы для рендеринга было показано, что с помощью изменения подхода к генерации случайных чисел в алгоритме рендеринга могут быть получены хорошие результаты за меньшее время. Все результаты могут быть

полезны для всех, кто занимается рендерингом изображений и графикой, так как способны значительно ускорить рендеринг изображений хорошего качества.

А также знания и навыки, которые лично я получил в ходе написания данной курсовой работы оказались очень полезными, а их уровень близок к современному уровню развития графических приложений и алгоритмов рендеринга. Проект, который был написан в ходе данной курсовой работы, однозначно продолжит развиваться и расширяться.

Git-репозиторий курсовой работы и проекта доступен по ссылке:
<https://github.com/Kaparya/PathTracing>

7. Библиографический список

1. Brian Caulfield (2022) What Is Path Tracing?
2. Peter Shirley, Trevor D Black, Steve Hollasch (2018-2024) Ray Tracing in One Weekend
3. NVIDIA DEVELOPER (2018) Ray Tracing
4. Jean-Colas Prunier (2007-2024) Scratchapixel blog
5. Nick Evanson (2023) Path Tracing vs. Ray Tracing, Explained
6. James T. Kajiya (1986) THE RENDERING EQUATION
7. Peter Shirley, Trevor D Black, Steve Hollasch (2018-2024) Ray Tracing The Next Week
8. Peter Shirley, Trevor D Black, Steve Hollasch (2018-2024) Ray Tracing The Rest of Your Life
9. Matt Pharr, Wenzel Jakob, Greg Humphreys (2004-2023) Physically Based Rendering: From Theory To Implementation
10. Mark Jarzynski, Marc Olano, UMBC (2020) Hash Functions for GPU Rendering
11. Nathan Vegdahl (2022) Psychopath Renderer blog
12. The blog at the bottom of the sea (2017) Generating Blue Noise Sample Points With Mitchell's Best Candidate Algorithm
13. Bart Wronski (2021) Superfast void-and-cluster Blue Noise in Python (Numpy / Jax)
14. Per Christensen, Andrew Kensler, Charlie Kilpatrick (2018) Progressive Multi-Jittered Sample Sequences
15. Brent Burley (2020) Practical Hash-based Owen Scrambling
16. Iliyan Georgiev, Marcos Fajardo (2016) Blue-noise Dithered Sampling
17. Alan Wolfe, Nathan Morrical, Tomas Akenine-Moller, Ravi Ramamoorthi (2021) Rendering in Real Time with Spatiotemporal Blue Noise Textures