Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика, искусственный интеллект и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Визуализация воды с использованием вокселей»

Студент ИУ7-53Б (Группа) _____ (Подпись, дата) Виноградов И. А. (И. О. Фамилия)

Руководитель курсовой работы _____ (Подпись, дата) Куров А. В. (И. О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

Введение

Воксельная (англ. Voxel, Volumetric Pixel) графика используется для отрисовки трехмерных изображений.

Цель данной работы – реализовать программу для построения изображений трехмерных воксельных сцен в реальном времени, с возможностью визуализации воды.

Чтобы достигнуть поставленой цели, требуется решить следующие задачи:

— описать структуру трехмерной сцены, включая объекты, из которых состоит сцена и их материалы;

— выбрать алгоритмы компьютерной графики, позволяющие визуализировать трехмерную воксельную сцену в реальном времени;

— реализовать выбранные алгоритмы построения трехмерной сцены;

— исследовать возможности улучшения производительности программы и повышения сложности задаваемых сцен.

1 Аналитический раздел

В данном разделе будут рассмотрены возможножсти отрисовки трехмерных изображений в реальном времени, проанализированы особенности трехмерной воксельной графики; будет выбран метод решения поставленной задачи.

1.1 Выбор оборудования

Построение трехмерных изображений – вычислительно сложная, и в то же время, чрезвычайно параллельная задача [1]. Такие особенности стали причиной появляния специального оборудования для выполнения популярных задач компьютерной графики. Такие вычислительные устройства называются Графическими процессорами (англ. Graphics Processing Unit, GPU), хотя их применение не ограничивается вычислениями графических данных. Появление таких устройств вызвано ограничениями в дизайне центральных процессоров. Они используются для последовательного выполнения кода с большим числом условных переходов. Особенностью графических процессоров ялвяется наличие большого числа ядер, способных эффективно выполнять множество вычислений параллельно [2].

Несмотря на большие возможности параллельного выполнения вычислительных задач, графические процессоры имеют органиченные возможности условного выполнения программ, в частности, рекурсивных, малую скорость памяти, и возможности взаимодействия с ней.

Такие особенности породили целый ряд алгоритмов, оптимизированных для выполняения на графических процессорах, которые отличаются от аналогичных для процессоров общего назначения. В частности, графические процессоры могут иметь специальные аппаратные блоки для ускоренного выполнения алгоритмов трассировки лучей, и все из них имеют аппартное ускорение алгоритма z-буфера [3].

1.2 Выбор алгоритма отрисовки

При выборе алгоритма отрисовки должны быть учтены особенности поставленной задачи. Отрисовка будет выполняться в режиме реального времени. Этот факт предъявляет к алгоритму требование по скорости работы,

время отрисовки кадра не должно занимать больше 100 миллисекунд. Также для визуализации воды следует построить некоторые физические эффекты: отражение и преломление света. Выбираемый алгоритм отрисовки должен поддерживать визуализацию физических явлений.

Будут рассмотрены алгоритмы, эффекивная по времени реализация которых может быть получена при использовании графических ускорителей.

1.2.1 Алгоритм, использующий z-буфер

z-буфер является буфером глубины, используемым для хранения координат или глубины каждого видимого пикселя в пространстве изображения. При использовании глубина или значение z нового пикселя, который записывается в буфер кадра, сравнивается с глубиной этого пикселя, хранящейся в z-буфере. Если сравнение показывает, что новый пиксель находится перед пикселем, хранящимся в буфере кадра, то новый пиксель записывается в буфер кадра, а z-буфер обновляется новым значением z. В противном случае никаких действий не предпринимается. Концептуально алгоритм является поиском по x, y для нахождения наибольшего значения $z(x, y)$ [4].

Аппартные графические ускорители изначально были спроектированы для использования в мультимедийных приложениях, использующих алгоритм z-буффера. В силу этого почти все их конвееры заточены специально под эффективную по времени реализацию этого алгоритма с некоторыми расширениями.

Однако, алгоритм z-буффера плохо подходит для реализации физически-корректного рендеринга. В частности, симуляция отражения или преломления требует множественной отрисовки сцены из разных точек, что замедляет общий процесс отрисовки [5]. Дополнительно, получаемые при отрисовке результаты оказываются менее физически точными, чем при использовании алгоритма трассировки лучей. В силу этого, в последние годы традиционные конвееры отрисовки на графических ускорителях расширяются добавлением трассировки лучей для получения лучшего качества изображения.

1.2.2 Трассировка лучей

Почти все системы фотореалистичного рендеринга основаны на алгоритме трассировки лучей. Трассировка лучей на деле очень простой алгоритм;

он основан на отслеживании пути луча света через сцену, по мере его взаимодействия и отражения от объектов в окружающей среде. Несмотря на то, что существует множество способов реализовать алгоритм трассировки лучей, все такие системы должны включать в себя и симулировать следующие объекты и феномены.

1. Камеры: Модель камеры определяет, как и откуда просматривается сцена, включая то, как изображение сцены записывается на сенсоре. Многие системы рендеринга генерируют оптические лучи, начинающиеся в камере, которые затем прослеживаются в сцене.

2. Пересечение лучей и объектов: Нам необходимо точно определить, где заданный луч пересекает заданный геометрический объект. Кроме того, нам нужно определить некоторые свойства объекта в точке пересечения, такие как нормаль поверхности или его материал.

3. Источники света: Трассировщик лучей должен моделировать распределение света по всей сцене, включая не только местоположение самих источников света, но и способ, которым они распространяют свою энергию в пространстве.

4. Видимость: Чтобы знать, может ли заданный источник света передавать энергию в точку поверхности, нам нужно знать, есть ли непрерывный путь от точки до источника света. Можно построить луч от поверхности до источника света, найти ближайшее пересечение луча с объектом и сравнить расстояние пересечения с расстоянием до источника света.

5. Рассеяние на поверхности: Каждый объект должен предоставить описание своего вида, включая информацию о том, как свет взаимодействует с поверхностью объекта, а также о характере перераспределенного (или рассеянного) света. Модели рассеивания на поверхности обычно параметризуются таким образом, чтобы можно было смоделировать разнообразные внешние виды.

6. Непрямое распространение света: Поскольку свет может достигать поверхности после отражения от других поверхностей или прохождения через них, обычно необходимо проследить дополнительные лучи, исходящие из поверхности, чтобы полностью учесть этот эффект [6].

Характеристики алгоритма:

— поддержка физически-корректной отрисовки не требует больших модификаций алгоритма, дополнительных затрат памяти;

— физически-корректная отрисовка работает для большинства физических эффектов, исключая некоторые специфичные (к примеру, каустики);

— слабая аппаратная поддержка (только некоторые новые графические ускорители, и только некоторые графические API);

— большие вычислительные затраты.

1.3 Выбор алгоритма обхода воксельной сцены

Воксель (англ. Voxel, Volumetric pixel) – способ представления геометрии, альтернативный полигональному. Воксель представляет собой некоторое значение на регулярной решетке в трехмерном пространстве. Воксели часто испольуются при анализе медицинских и научных данных. Воксельная графика позволяет достигать большей детализации, чем возможно при использовании полигонов.

Главная проблема воксельной графики – большое число затрачиваемой памяти и медленный доступ к ней. Поэтому все воксельные алгоритмы фокусируются на ускорении доступа к индивидуальным вокселям, для использования в алгоритмах отрисовки. Рассмотрим и выберем алгоритм обхода воксельной сцены и структуру данных хранения вокселей.

1.3.1 Алгоритм быстрого обхода вокселей для трассировки лучей

Рассматриваемый алгоритм описан в классической статье задолго до массового распространения графических ускорителей и повсеместного использования трассировки лучей. В ней формализован и описан несложный алгоритм обхода воксельной трехмерной сетки с константным масштабом (см. рисунок 1).

Переход от одного вокселя к его соседу требует только двух сравнений чисел с плавающей запятой одного сложения чисел с плавающей запятой.

Кроме того, исключаются множественные пересечения лучей с объектами, находящимися в более чем одном вокселе [7].

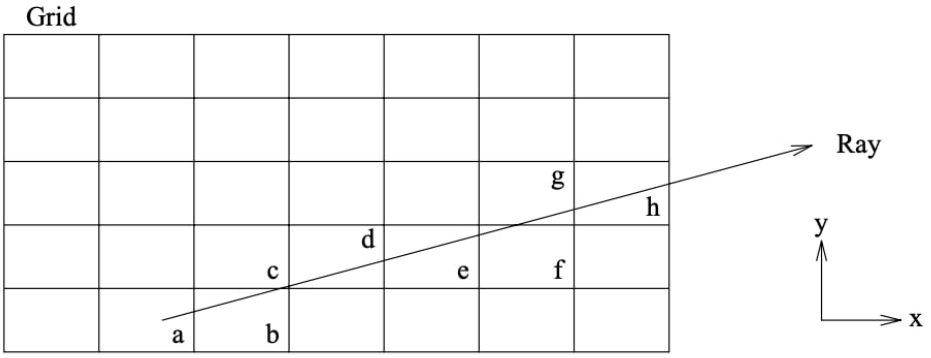

Рисунок 1 – Алгоритм быстрого обхода вокселей для трассировки лучей [7]

Авторами отмечается один существенный недостаток: высокие вычислительные затраты. Для решения этой пробремы они предлагают вводить оптимизационные структуры, к примеру, BVH (bounding volume hierarchy). Такие структуры должны разделять пространство мира и проверять луч на столкновение только с его частью, вместо всего пространства. Проблема такого подхода заключается в том, что BVH требует больших вычислительных затрат на построение, и само построение – NP-полная задача. Это делает рассматриваемый алгоритм малоприменимым во всех нетривиальных случаях.

1.3.2 k-d дерево

k-d дерево (сокращение от k-мерного дерева) - это структура данных для разделения пространства, предназначенная для организации точек в k-мерном пространстве. k-d деревья являются полезной структурой данных для нескольких приложений, таких как поиск с использованием многомерного ключа поиска (например, поиск по диапазону и поиск ближайшего соседа) и создание облаков точек. k-d деревья являются особым случаем деревьев бинарного разделения пространства. k-d дерево – это бинарное дерево, где каждая вершина – это точка в k-мерном пространстве. Каждая вершина, не являющаяся листом может быть представлена как неявная гиперплоскость, разделяящая пространство на две части. Точки в каждой из частей представлены соответсвующии поддеревьями (см. пример на рисунке 2).

k-d дерево стало популярным как основная оптимизационная структура в трассировщиках лучей. Были предложены алгоритмы оптимизации структуры

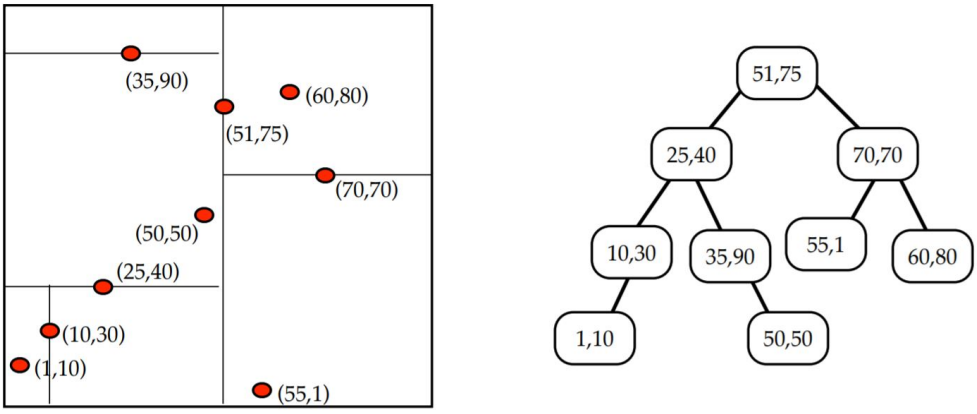

Рисунок 2 – Пример двумерного k-d дерева [8]

для графических ускорителей. Результаты замеров показывают, что скорость отрисовки сцены с использованием k-d дерева как минимум на порядок больше, чем скорость отрисовки в сетке с константным размером сетки.

k-d дерево можно использовать для отрисовки сцен с геометрией, хранимой в произвольном формате. Вариант k-d дерева, используемый при воксельной отрисовке – октодерево. Это k-d дерево в трехмерном пространстве, где у каждой родительской вершины 8 детей [9].

1.3.3 Разреженное воксельное октодерево

Проблема обычного октодерева – это то, что оно хранит данные для каждого из элементов пространства. Обычно сцены являются разреженными, что делает плотное хранение данных избыточным по памяти, и менее производительным. Поэтому чаще всего при отрисоввке воксельных сцен в качестве оптимизационной структуры данных применяется разреженное воксельное октодерево (см. рисунок 3).

Такая структура данных имеет как отличные временные характеристики, так и малые затраты памяти. Основная сложность заключается в организации данных в оперативной памяти, на диске, и в памяти графического ускорителя [11].

1.4 Выбор модели освещения

Модель освещения в случае трассировки лучей – это фундаментальная характеристика программы-отрисовщика. Это связано с тем, что трассировка лучей полагается на выбранную модель освещения при построении изображе-

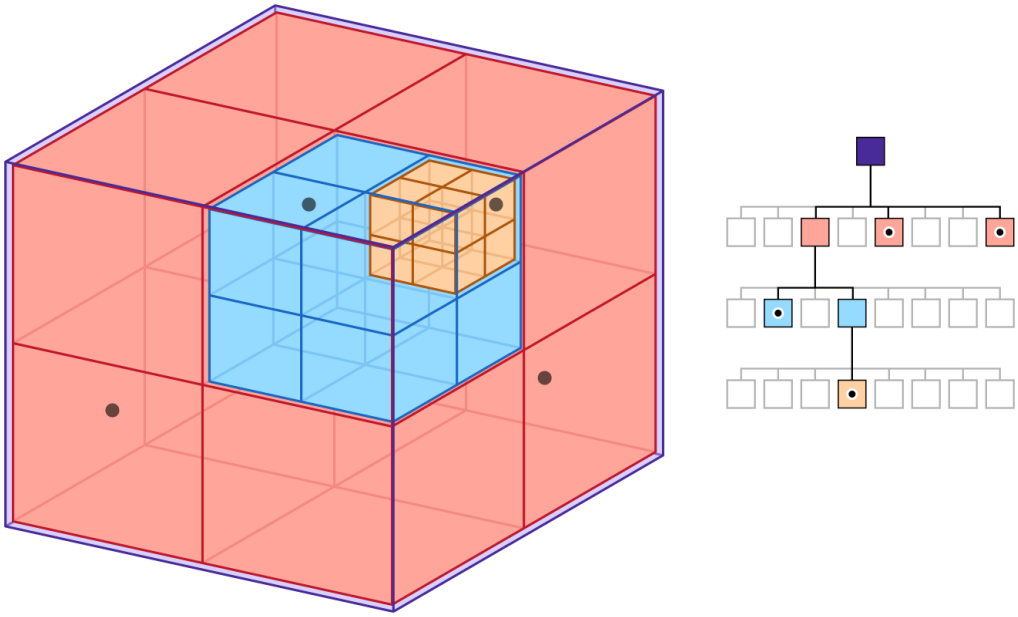

Рисунок 3 – Октодерево [10]

ния, и от выбранной модели зависят не только детали картинки, как в случае с z-буффером, но качества результата в целом.

1.4.1 Модель освещения Ламберта

Самая простая модель затенения основана на наблюдении, сделанном Ламбертом в 18 веке: количество энергии от источника света, которое падает на поверхность, зависит от угла поверхности к свету. Поверхность, направленная прямо на свет, получает максимальное освещение; поверхность, касательная к направлению света (или обращенная к свету), не получает освещение; и между ними освещение пропорционально косинусу угла θ между нормалью поверхности n и источником света l. Это приводит к модели освещения Ламберта:

$$L = k_d I max(0, n \cdot l), \tag{1}$$

где I – интенсивность источника света. Поскольку n и l – единичные векторы, мы можем использовать $n \cdot l$ как удобное сокращение для $cos(\theta)$. Вектор l вычисляется путем вычитания точки пересечения луча и поверхности из положения источника света [5].

1.4.2 Модель освещения Фонга

Модель освещения Ламберта не зависит от точки обзора: цвет поверхности не зависит от направления, с которого на нее смотреть. Многие реальные поверхности обладают определенной степенью блеска, создающего мерцание или зеркальные отражения, которые кажутся перемещающимися при изменении точки обзора. Ламбертово освещение не создает мерцания и создает очень матовый, меловидный вид, и множество моделей освещения добавляют зеркальную составляющую к ламбертовому освещению; ламбертова часть в таком случае является диффузной составляющей.

Очень простая и широко используемая модель для зеркальных мерцаний была предложена Фонгом, а позже обновлена Блинном до формы, которая наиболее распространена сегодня. Идея заключается в создании отражения, которое ярче всего, когда векторы v и l симметрично расположены относительно нормали поверхности, то есть, когда происходит зеркальное отражение; отражение затем постепенно уменьшается, по мере того, как векторы отдаляются от зеркальной конфигурации. Таким образом, формула модели освещения Фонга:

$$h = \frac{v + 1}{||v + 1||}, \tag{2}$$

$$L = k_d I max(0, n \cdot l) + k_s I max(0, n \cdot h)^2, \tag{3}$$

где h – бисскетриса угла между v и l; p – экспонента Фонга; k_s – коэффицент зеркальности [12] [5].

1.4.3 Физически-корректная модель

Модели Ламберта и Фонга просты, и поэтому их использование не требует больших вычислительных затрат. Но в то же время, они лишь пытаются построить упрощенную модель освещения пространства, в то время как может быть построена более общая симуляция физических явлений.

Подход, в котором для построение изображений используется симуляция физических явлений, а не ее приближение, называется физически-корректной моделью освещения. Реализация алгоритмов такого подхода построена на Методе Монте-Карло. Производится отрисовка сцены с множеством переменных

параметров, к примеру, направлению лучей, или распределению света по пространству, которые имеют заранее известные характеристики вероятностного распределения.

Задача построения изображения в физически-корректной модели – задача интеграции уравнения рендеринга [13]. Для интеграции используется статистический анализ распределения значений двулучевой функции отражательной способности [6].

Микрогранные модели

Многие подходы к моделированию отражения и пропускания поверхности, основанные на геометрической оптике, основаны на идее того, что шероховатые поверхности могут быть представлены в виде набора маленьких граней. Микрограни часто моделируются в виде поверхностей с высотой, где распределение ориентации граней описывается статистически (см рис. 4).

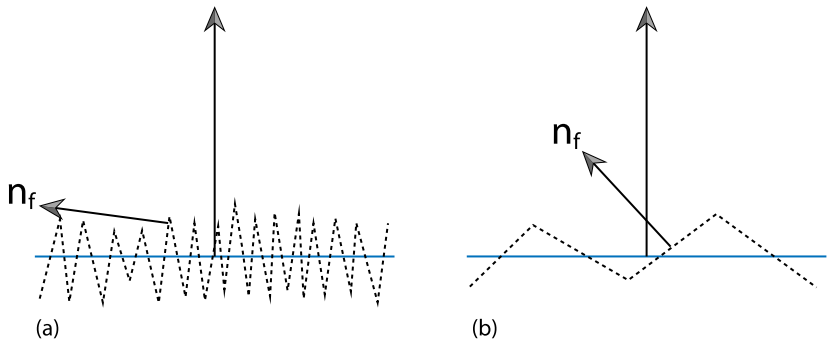

Рисунок 4 – Микрогранные модели [6]

Модели отражения, основанные на микрогранях, которые проявляют идеальное зеркальное отражение и пропускание, успешно используются для моделирования рассеяния света от различных глянцевых материалов, включая металлы, пластик и матовое стекло. Одной из важных характеристик поверхности микрограни является функция распределения, которая показывает дифференциальную площадь микрофасеток с нормалью поверхности. Две самые популярные модели микрограней – основанная на модели Бекерманна, и основанная на модели GGX. Их разница заключается в различных радиометрических свойствах, которые неважны для непрофессионального разработчика трассировщиков. Однако на практике, модель GGX показывает лучшие результаты при одинаковом количестве вычислений [6] [14].

Вывод

Было принято решение выполнять разработку для выполнения на графическом процессоре, поскольку это позволяет выполнять отрисовку в реальном времени. В качестве алгоритма отрисовки был выбран алгоритм трассировки лучей. В качестве алгоритма обхода сцены был выбран алгоритм быстрого обхода вокселей. В качестве модели освещения была выбрана физически-корректная модель с использованием микрограней с моделью GGX, поскольку такой вариант предоставляет лучший вариант получения реалистичных изображений [14].

2 Конструкторский раздел

В данном разделе будут описаны особенности написания программ для графических процессоров, будут разработаны алгоритмы, выбранные для решения поставленной задачи.

2.1 Особенности написания программ для графических процессоров

Написание программ для графических процессоров имеет следующие особенности:

— получение входных данных в виде последовательности графических примитивов;

— разделение программы на несколько этапов;

— использование специального языка программирования;

— отсутствие прямого доступа к памяти;

— высокая вычислительная стоимость условных переходов.

Входными данными являются треугольники. При отрисовке сцены используется два треугольника, определяющих прямоугольник видимости экрана. Программы, называемые шейдерами (англ. Shader), конвеера (англ. Pipeline) графического процессора деляется на несколько этапов. Двумя основными этапами, без которых отрисовка не может быть выполнена, это этапы выполнения вершинного (англ. Vertex) и фрагментного (англ. Fragment) шейдеров. Вершинный шейдер выполняется для каждой из вершин один раз, и его результаты линйеного интерполируются. Фрагментный шейдер выполняется для каждого пикселя экрана и выполняет подсчет цвета этого пикселя.

2.2 Разработка алгоритмов

2.2.1 Алгоритм трассировки лучей

На рисунке 5 приведена схема алгоритма работы программы. MAX_BOUNCE_COUNT – это целочисленная константа, обозначающая

максимальное число отскоков одного луча. Под цветом понимается трехмерный вектор в формате RGB в диапозоне $[0, 1)$. $rgb(0, 0, 0)$ – функция черного цвета.

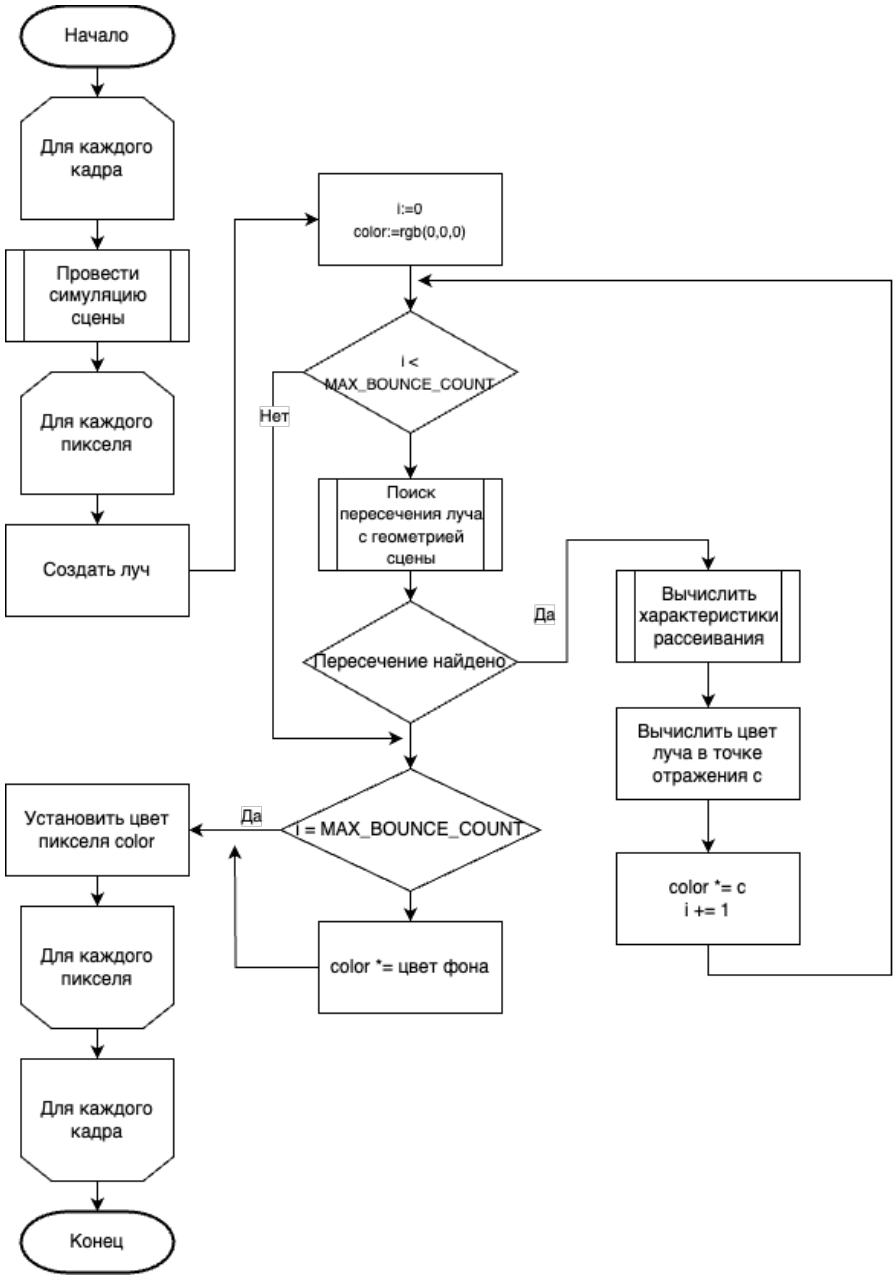

Рисунок 5 – Алгоритм отрисовки

Алгоритм трассировки лучей реализуется в фрагментном шейдере. Вычисление вектора направления луча производится в вершинном шейдере и линейно интерполируется между всеми пикселями. В силу этого в фрагметных шейдерах необходимо произвести нормализацию вектора направления луча перед его использованием.

Для передачи информации о камере используются две квадратных мат-

рицы размерности 4: матрица проекции камеры и матрица вида камеры. Матрица вида камеры содержит информацию о том, как точка из пространства сцены преобразуется в пространство камеры. Матрица проекции содержит информацию о том, как точка из пространства камеры преобразуется в пространство изображения. Разделения преобразования на две части позволяет уменьшить число вычислений в случаях, когда вид меняется, а проекция остается неизменной.

В листинге 1 приведен псевдокод вершинного шейдера, выполняющий вычисление координаты начала луча и вектора его направления.

Листинг 1 – Вычисление координаты начала луча и вектора его направления

```
vec4 t1 = inverse(projection_matrix) * vec4f(position, -1.0,
   1.0);
vec4 t2 = view_matrix * vec4f(t1.xyz, 0.0);
vec3 ray_direction = t2.xyz;
vec3 ray_origin = vec3(view_matrix[3][0], view_matrix[3][1],
   view_matrix[3][2]);
```

2.2.2 Алгоритм кеширования обратной репроекции

Временная обратная репроекция – это процесс отображения ранее сгенерированного кадра на текущий кадр. Это позволяет повторно использовать информацию или, в случае трассировки лучей, накапливать сэмплы для получения более четкого изображения даже при движении [15].

На рисунке 6 представлена схема алгоритма кэширования обратной репроекции.

Рисунок 6 – Алгоритм кеширования обратной репроекции

В листинге 2 приведен псевдокод процерования точки в пространстве текущего кадра на пространство другого кадра.

Листинг 2 – Проецирование одного кадра на другой

```
vec4 p = projection_matrix * inverse(previous_view_matrix) *
   vec4(position, 1.0);
p = p.xyz / p.w;
vec2 previous_uv = vec2((p.x / 2.0) + 0.5, (p.y / 2.0) + 0.5);
```

Вывод

В данном разделе были описаны особенности алгоритмов для выполнения на графических процессорах, алгоритмы и структуры данных, выбранные и разработанные для решения поставленной задачи.

3 Технологический раздел

В данном разделе будут рассмотрены особенности реализации программного обеспечения (ПО), разработанного в конструкторской части работы, и приведены примеры работы программы.

3.1 Требования к программному обеспечению

Программа должна предоставлять следующий функционал:

— получение изображения сцены с учетом материалов объектов сцены и освещения;

— интерактивное управление камерой;

— время отрисовки одного кадра не больше 100мс;

— симуляция распространения воксельных частиц воды.

Программа должна корректно и оперативно реагировать на все действия пользователя.

3.2 Средства реализации

В качестве графического API для реализации разработанного ПО был выбран WebGPU [16] – современный переносимый графический API. Он может быть использованным при разработке на любой из платформ, поддерживающих популярные API: Direct3D [17], Metal [18], Vulkan [19], OpenGLES [20], WebGL [21] и другие.

В качестве языка программирования для реализации ПО был выбран Rust – статически типизированный компилируемый язык программирования общего назначения [22]. Данный выбор обусловлен наличием возможностей компиляции на множественное число платформ, в том числе на WASM [23], наличием библиотек с поддержкой WebGPU, линейной алгебры.

Для реализации графического интерфейса была выбрана библиотека Dear ImGui [24]. Она реализует графический интерфейс в режиме IMGUI (англ. immediate mode GUI). Библиотека написана для использования с любым графическим API и представляет способ динамического программирования интерфейсов.

ПО реализовано для платформы WASM и нативных MacOS, Linux, Windows. Алгоритм отриасовки реализован в вершинном и фрагментном шейдерах на языке WGSL [25].

3.3 Реализации алгоритмов

3.3.1 Реализация алгогитма генерации случайных чисел

Для реализации алгоритма трассировки лучей требуется использовать генератор случайных чисел. В качестве алгоритма генератора случайных чисел был выбран xorshift32 [26]. Это сделано из-за возможности эффективного выполнения алгоритма на GPU. В листинге 1 представлен код, выполняющий генерацию случайных чисел.

Листинг 1 – Реализация генератора случайных чисел в шейдере

```
@group(0) @binding(1) var<uniform> random_seed: u32;
var<private> rng_state: u32;

fn xorshift32(state: u32) -> u32 {
    var x = state;
    x ^= x << 13u;
    x ^= x >> 17u;
    x ^= x << 5u;
    return x;
}

fn random_u32() -> u32 {
    let x = xorshift32(rng_state);
    rng_state = x;
    return x;
}

fn random_f32() -> f32 {
    let u = random_u32();
    return f32(u) * bitcast<f32>(0x2F800000u);
}
```

3.3.2 Реализация алгоритма быстрого обхода вокселей для трассировки лучей

В листинге 2 приведена реализация алгоритма быстрого обхода вокселей для трассировки лучей.

Листинг 2 – Реализация алгоритма быстрого обхода вокселей для трассировки лучей

```
fn voxel_traverse(ray: Ray) -> HitRecord {
    var record: HitRecord;
    let origin = ray.origin;
    let direction = normalize(ray.direction);
    let step = vec3f(sign(direction.x), sign(direction.y),
       sign(direction.z));
    var current_voxel = vec3i(floor(origin));
    let next_bound = vec3f(current_voxel + (step + vec3i(1)) /
       2);
    var t_max = (next_bound - origin) / direction;
    let t_delta = direction * step;
    for (var i: i32 = 0; i<MAXIMUM_TRAVERSAL_DISTANCE; i+=1) {
        if t_max.x < t_max.y && t_max.x < t_max.z {
            record.offset_id = current_voxel.x;
            record.t = t_max.x;
            record.normal = vec3f(-step.x, 0.0, 0.0);
            t_max.x += t_delta.x;
            current_voxel.x += step.x;
        } else if t_max.y < t_max.z {
            record.offset_id = current_voxel.y;
            record.t = t_max.y;
            record.normal = vec3f(0.0, -step.y, 0.0);
            t_max.y += t_delta.y;
            current_voxel.y += step.y;
        } else {
            record.offset_id = current_voxel.z;
            record.t = t_max.z;
            record.normal = vec3f(0.0, 0.0, -step.z);
            t_max.z += t_delta.z;
            current_voxel.z += step.z;
        }
        record.id = textureLoad(voxel_data, current_voxel, 0).r;
        if record.id != 0u {
            record.pos = ray_at(ray, record.t + 0.001);
            break;
        }
    }
    return record;
}
```

3.3.3 Реализация алгоритма трассировки лучей

В листинге 3 приведена реализация алгоритма трассировки лучей. Функция `scatter` выполяет расчет цвета и направления отражения луча.

Листинг 3 – Реализация алгоритма трассировки лучей

```
fn trace(ray_: Ray) -> TraceResult {
    var result: TraceResult;
    result.color = vec3f(1.0);
    var ray = ray_;
    let hrec = voxel_traverse(ray);
    if hrec.id == 0u {
        result.color = vec3f(0.5);
        return result;
    }
    let srec = scatter(ray, hrec);
    result.color *= srec.attenuation;
    ray.origin = hrec.pos;
    ray.direction = normalize(srec.direction);
    result.pos = hrec.pos;
    result.id = hrec.id;
    result.normal = hrec.normal;
    result.offset_id = hrec.offset_id;
    var i: i32 = 1;
    for (; i < MAX_BOUNCE_COUNT; i += 1) {
        let hrec = voxel_traverse(ray);
        if hrec.id == 0u {
            break;
        }
        let srec = scatter(ray, hrec);
        result.color *= srec.attenuation;
        ray.origin = hrec.pos;
        ray.direction = normalize(srec.direction);
    }
    return result;
}
```

3.3.4 Реализация алгоритма кеширования обратной репроекции

В листинге 4 приведена реализация алгоритма кеширования обратной репроекции.

Листинг 4 – Реализация алгоритма кеширования обратной репроекции

```
let point = projection_matrix * prev_view_matrix * vec4f(pos,
   1.0);
let p = point.xyz / point.w;
let puv1 = (p.xy + vec2f(1.0)) * 0.5;
let puv = vec2f(puv1.x, 1.0 - puv1.y);
let prev_normal = textureSample(prev_normal_tex,
   prev_tex_sampler, puv).rgb;
let prev_offset_id = textureSample(prev_offset_tex,
   prev_tex_sampler, puv).r;
let prev_mat_id = textureSample(prev_mat_tex, prev_tex_sampler,
   puv).r;
let prev_cache_tail = textureSample(prev_cache_tail_tex,
   prev_tex_sampler, puv).r;
let prev_color = textureSample(prev_color_tex, prev_tex_sampler,
   puv).rgb;
if result.material_id != 0.0 {
    if puv.x > 0.0 && puv.x < 1.0 &&
        puv.y > 0.0 && puv.y < 1.0 &&
        result.material_id == prev_mat_id &&
        distance(result.normal.xyz, prev_normal) < 0.1 &&
        result.offset_id == prev_offset_id
    {
        result.cache_tail = (1.0 - ALPHA) * prev_cache_tail;
        result.color = vec4f((ALPHA * result.color.xyz) + (1.0 -
           ALPHA) * prev_color, 1.0);
    } else {
        result.cache_tail = 1.0;
    }
}
```

Каждый кадр использует 4 буфера, содержащих следующую информацию о каждом пикселе:

— цвет,

— вектор нормали,

— координаты первого столкновения луча при трассировке,

— идентификатор материала.

Каждый кадр имеет доступ к буферам прошлого кадра, что позволяет выполнять обратную репроекцию.

Вывод

В данном разделе были описаны детали реализации разработанной программы. Также был рассмотрен процесс взаимодействия пользователя с программой. Были приведены примеры использования программы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Foster I.* Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. — 75 Arlington Street, Suite 300 Boston, MAUnited States : Addison-Wesley Longman Publishing Co., 01.1995.
2. *Fatahalian K.*, *Houston M.* A Closer Look at GPUs. — 2008.
3. Ray Tracing In Vulkan / D. Koch [и др.]. — 2020. — Дек.
4. *Rogers D. F.* Procedural Elements for Computer Graphics (2nd Ed.) — USA : McGraw-Hill, Inc., 1997. — ISBN 0070535485.
5. *Shirley P.*, *Marschner S.* Fundamentals of Computer Graphics. — 3rd. — USA : A. K. Peters, Ltd., 2009.
6. *Pharr M.*, *Jakob W.*, *Humphreys G.* Physically Based Rendering: From Theory to Implementation (3rd ed.) — 3rd. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 10.2016.
7. *Amanatides J.*, *Woo A.* A Fast Voxel Traversal Algorithm for Ray Tracing // Proceedings of EuroGraphics. — 1987. — Авг. — Т. 87.
8. *Stanford.* kd-Trees. — URL: `https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf`.
9. *Foley T.*, *Sugerman J.* KD-Tree Acceleration Structures for a GPU Raytracer. — 2005.
10. *Apple.* Octree [Электронный ресурс]. — URL: `https://developer.apple.com/documentation/gameplaykit/gkoctree`.
11. *Laine S.*, *Karras T.* Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation. — 2010.
12. *Phong B. T.* Illumination for Computer Generated Pictures. — 1975.
13. *Kajiya J. T.* The Rendering Equation // SIGGRAPH Comput. Graph. — New York, NY, USA, 1986. — Авг. — Т. 20, № 4. — С. 143—150. — ISSN 0097-8930. — DOI: `10.1145/15886.15902`. — URL: `https://doi.org/10.1145/15886.15902`.
14. Microfacet Models for Refraction through Rough Surfaces / B. Walter [и др.]. — 2007.

15. Accelerating Real-Time Shading with Reverse Reprojection Caching / D. Nehab [и др.]. — 2007. — Авг.

16. *W3C*. WebGPU working draft. — 07.2023.

17. *Nvidia*. DirectX12 [Электронный ресурс]. — URL: https://www.nvidia.com/en-gb/geforce/technologies/dx12/.

18. *Inc. A.* Metal [Электронный ресурс]. — 2023. — URL: https://developer.apple.com/metal/.

19. *Group K.* Vulkan® 1.3.264 - A Specification (with all registered Vulkan extensions) [Электронный ресурс]. — 2023. — URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/index.html.

20. *Group K.* OpenGL ES 3.2 [Электронный ресурс]. — 2023. — URL: https://www.khronos.org/opengles/.

21. *Group K.* WebGL [Электронный ресурс]. — 2023. — URL: https://www.khronos.org/webgl/.

22. *Klabnik S., Nichols C.* The Rust Programming Language. — USA : No Starch Press, 2018. — ISBN 1593278284.

23. *Group W. C.* WebAssembly Specification [Электронный ресурс]. — 2022. — URL: https://webassembly.github.io/spec/core/.

24. *Cornut O.* Dear ImGui [Электронный ресурс]. — URL: https://github.com/ocornut/imgui.

25. *W3C*. WebGPU Shading Language working draft. — 09.2023.

26. *Marsaglia G.* Xorshift RNGs // Journal of Statistical Software. — 2003. — Т. 8, № 14. — С. 1—6. — DOI: 10.18637/jss.v008.i14. — URL: https://www.jstatsoft.org/index.php/jss/article/view/v008i14.