

# Содержание

<b>Введение</b>	<b>5</b>
<b>1 Аналитический раздел</b>	<b>7</b>
1.1 Графический конвейер . . . . .	7
1.2 Описание объектов сцены . . . . .	7
1.3 Обоснование выбора формы задания трехмерных моделей .	8
1.3.1 Задания поверхностных моделей . . . . .	9
1.3.2 Вывод . . . . .	10
1.4 Выбор алгоритма удаления невидимых ребер и поверхностей	10
1.4.1 Алгоритм, использующий Z-буфер . . . . .	11
1.4.2 Алгоритм обратной трассировки лучей . . . . .	12
1.4.3 Алгоритм Робертса . . . . .	13
1.4.4 Алгоритм художника . . . . .	14
1.4.5 Алгоритм Варнока . . . . .	15
1.4.6 Вывод . . . . .	16
1.5 Анализ и выбор модели освещения . . . . .	17
1.5.1 Модель Ламберта . . . . .	17
1.5.2 Модель Фонга . . . . .	18
1.5.3 Вывод . . . . .	19
1.6 Проблема неравенства граней по площади . . . . .	19
<b>2 Конструкторский раздел</b>	<b>20</b>
2.1 Общий алгоритм решения поставленной задачи . . . . .	20
2.2 Шаги графического конвейера . . . . .	20
2.3 Шаг отбрасывания невидимых объектов . . . . .	22
2.3.1 Асинхронный анализ виртуальной геометрии . . . . .	24
2.3.2 Создание или обновление виртуальных объектов . . .	26
2.4 Алгоритм z-буфера . . . . .	26
2.5 Отрисовка объектов . . . . .	29
2.6 Выбор используемых типов и структур данных . . . . .	31
<b>3 Технологический раздел</b>	<b>34</b>
3.1 Требования к программе . . . . .	34

3.2	Выбор языка программирования и среды разработки . . . . .	34
3.3	Структура программы . . . . .	36
3.4	Интерфейс . . . . .	38
3.5	Результаты работы программного обеспечения . . . . .	40
<b>4</b>	<b>Экспериментальный раздел</b>	<b>46</b>
4.1	Технические характеристики . . . . .	46
4.2	Постановка эксперимента 1 . . . . .	46
4.2.1	Цель эксперимента . . . . .	46
4.3	Постановка эксперимента 1 . . . . .	47
4.3.1	Цель эксперимента . . . . .	47
<b>Заключение</b>		<b>49</b>
<b>Список литературы</b>		<b>50</b>

# Введение

В современном мире компьютерная графика используется достаточно широко. Типичная область ее применения – это кинематография и компьютерные игры.

На сегодняшний день большое внимание уделяется алгоритмам получения реалистичного изображения. Такие алгоритмы являются одними из самых затратных по времени, потому что они должны учитывать множество физических явлений, таких как преломление, отражение, рассеивание света. Для повышения реалистичности изображения также учитывается дифракция, вторичное, троичное отражение света, поглощение.

Можно заметить, что чем более качественным является изображение на выходе алгоритма, тем больше времени и памяти используется для его синтеза. Это и становится проблемой при создании динамической сцены, так как на каждом временном интервале необходимо производить расчеты заново.

Целью курсовой работы является изучение работы графического конвейера, способов обработки данных на GPGPU, применимости чрезвычайно параллельных алгоритмов к компьютерной графике, а также программная эмуляция работы графического конвейера на видеокарте. Предполагается, что большая часть вычислений будет проводиться на графическом ускорителе. Некоторые шаги, например шаг куллинга, можно вынести на процессор.

Чтобы достигнуть поставленной цели, требуется решить следующие задачи:

- описать структуру графического конвейера;
- выбрать способ представления объектов сцены;
- проанализировать существующие алгоритмы построения изображения и обосновать выбор тех из них, которые в наибольшей степени подходят для решения поставленной задачи;
- реализовать выбранные алгоритмы;
- разработать программное обеспечение для отображения сцены;

- провести анализ производительности работы программы в зависимости от конфигурации сцены и доступности ресурсов.

# 1 Аналитический раздел

В данном разделе представлено описание работы графического конвейера, объектов сцены, а также обоснован выбор алгоритмов, которые будут использованы для ее визуализации.

## 1.1 Графический конвейер

Графический конвейер – концептуальная модель, описывающая шаги, выполняемые графической системой при отображении модели на экране. Этапы графического конвейера выполняются чрезвычайно параллельно на графическом ускорителе. Современные средства разработки предлагают разработчику API (OpenGL, Vulkan, Metal, DirectX) для обращения к конвейеру.

Тем не менее, отдельные части конвейера скрыты слоями абстракции, а некоторые все же реализованы аппаратно (растеризация треугольников, z-буферинг).

В курсовой работе программно реализованы некоторые шаги работы конвейера.

## 1.2 Описание объектов сцены

Сцена состоит из нескольких источников света, и нескольких трехмерных объектов. Визуальные эффекты (туман, воду) предлагаются, по желанию, и в целях демонстрации, добавить на этапе отрисовки с помощью программируемых шейдеров.

Источник света представляет собой материальную точку, испускающую лучи света во все стороны (если источник расположен в бесконечности, то он имеет направление).

## 1.3 Обоснование выбора формы задания трехмерных моделей

Отображением формы и размеров объектов являются модели. Обычно используются три формы задания моделей.

### 1. Каркасная (проволочная) модель.

Одна из простейших форм задания модели, так как мы храним информацию только о вершинах и ребрах нашего объекта. Недостаток данной модели состоит в том, что она не всегда точно передает представление о форме объекта.

### 2. Поверхностная модель.

Поверхностная модель объекта — это оболочка объекта, пустая внутри. Такая информационная модель содержит данные только о внешних геометрических параметрах объекта. Такой тип модели часто используется в компьютерной графике. При этом могут использоваться различные типы поверхностей, ограничивающих объект, такие как полигональные модели, поверхности второго порядка и др.

### 3. Объемная (твёрдотельная) модель.

При твёрдотельном моделировании учитывается еще материал, из которого изготовлен объект. Т.е. у нас есть информация о том, с какой стороны поверхности расположен материал. Это делается с помощью указания направления внутренней нормали.

При решении данной задачи подойдут будут использоваться поверхностная модель. Этот выбор обусловлен тем, что каркасные модели могут привести к неправильному восприятию формы объекта, а реализация объемной модели потребует большего количества ресурсов на воспроизведение деталей, не влияющих на качество решения задачи в ее заданной формулировке.

### 1.3.1 Задания поверхностных моделей

На следующем шаге необходимо определиться со способом задания поверхностной модели.

#### 1. Аналитический способ.

Этот способ задания модели характеризуется описанием модели объекта, которое доступно в неявной форме, то есть для получения визуальных характеристик необходимо дополнительно вычислять некоторую функцию, которая зависит от параметра.

#### 2. Полигональная сетка.

Данный способ характеризуется совокупностью вершин, граней и ребер, которые определяют форму многогранного объекта в трехмерной компьютерной графике.

При этом существует несколько способов хранения информации о сетке.

#### 1. Список граней.

Объект – это множество граней и множество вершин. В каждую грань входят как минимум 3 вершины;

#### 2. «Крылатое» представление.

Каждая точка ребра указывает на две вершины, две грани и четыре ребра, которые её касаются.

#### 3. Полурёберные сетки.

То же «крылатое» представление, но информация обхода хранится для половины грани.

#### 4. Таблица углов.

Это таблица, хранящая вершины. Обход заданной таблицы неявно задаёт полигоны. Такое представление более компактно и более производительно для нахождения полигонов, но, в связи с тем, что вершины присутствуют в описании нескольких углов, операции по их изменению медленны.

## 5. Вершинное представление.

Хранятся лишь вершины, которые указывают на другие вершины. Простота представления даёт возможность проводить над сеткой множество операций.

### 1.3.2 Вывод

Стоит отметить, что одним из решающих факторов в выборе способа задания модели в данном проекте является скорость выполнения преобразований над объектами сцены. Поэтому при реализации программного продукта в данной работе наиболее удобным представлением является модель, заданная полигональной сеткой – это поможет избежать проблем при описании сложных моделей. При этом способ хранения полигональной сетки – это список граней, так как он содержит явное описание граней, что поможет при реализации алгоритма удаления невидимых рёбер и поверхностей. Также этот способ позволит эффективно преобразовывать модели, так как структура будет включать в себя список вершин.

## 1.4 Выбор алгоритма удаления невидимых ребер и поверхностей

Перед выбором алгоритма удаления невидимых ребер необходимо выделить несколько свойств, которыми должен обладать выбранный алгоритм, чтобы обеспечить оптимальную работу и реалистичное изображение, а именно:

- алгоритм должен использовать как можно меньше памяти;
- алгоритм должен быть параллельным, пошаговые алгоритмы недопустимы;
- алгоритм должен иметь высокую реалистичность изображения.

### 1.4.1 Алгоритм, использующий Z-буфер

Суть данного алгоритма – это использование двух буферов: буфера кадра, в котором хранятся атрибуты каждого пикселя, и Z-буфера, в котором хранится информация о координате  $Z$  для каждого пикселя.

Первоначально в Z-буфере находятся минимально возможные значения  $Z$ , а в буфере кадра располагаются пиксели, описывающие фон. Каждый многоугольник преобразуется в растровую форму и записывается в буфер кадра.

В процессе подсчета глубины нового пикселя, он сравнивается с тем значением, которое уже лежит в Z-буфере. Если новый пиксель расположен ближе к наблюдателю, чем предыдущий, то он заносится в буфер кадра и происходит корректировка Z-буфера [1].

Для решения задачи вычисления глубины  $Z$  каждый многоугольник описывается уравнением  $ax + by + cz + d = 0$ . При  $c = 0$  многоугольник для наблюдателя вырождается в линию.

Для некоторой сканирующей строки  $y=\text{const}$ , поэтому имеется возможность рекуррентно высчитывать  $z'$  для каждого  $x' = x + dx$ :  $z' - z = -\frac{ax'+d}{c} + \frac{ax+d}{c} = \frac{a(x-x')}{c}$ .

Получим  $z' = z - \frac{a}{c}$ , так как  $x - x' = dx = 1$ .

При этом стоит отметить, что для невыпуклых многогранников предварительно потребуется удалить не лицевые грани.

#### Преимущества

- простота реализации;
- алгоритм легко распараллелить;
- оценка трудоемкости линейна.

#### Недостатки

- сложная реализация прозрачности;

Данный алгоритм идеально подходит для решения поставленной задачи, так как он просто распараллеливается.

## 1.4.2 Алгоритм обратной трассировки лучей

Алгоритмы трассировки лучей на сегодняшний день считаются наиболее мощными при создании реалистичных изображений.

Изображение формируется из-за того, что свет попадает в камеру. Выпустим из источников света множество лучей (первичные лучи). Часть этих лучей “улетит” в свободное пространство, а часть попадет на объекты. На них лучи могут преломляться и отражаться. При этом часть энергии луча поглотится. Преломленные и отраженные лучи образуют новое поколение лучей. Далее эти лучи опять же преломятся, отразятся и образуют новое поколение лучей. В конечном итоге часть лучей попадет в камеру и сформирует изображение. Это описывает работу прямой трассировки лучей.

Метод обратной трассировки лучей позволяет значительно сократить перебор световых лучей. В этом методе отслеживаются лучи не от источников, а из камеры. Таким образом, трассируется определенное число лучей, равное разрешению картинки [2].

### Преимущества

- высокая реалистичность синтезируемого изображения;
- работа с поверхностями в математической форме;
- вычислительная сложность слабо зависит от сложности сцены.
- просто распараллелить;

### Недостатки

- производительность.

Данный алгоритм не отвечает главному требованию – скорости работы, но при некоторой адаптации скорость работы алгоритма можно повысить. Также алгоритм, в чистом виде, не используется в реальном графическом конвейере.

### 1.4.3 Алгоритм Робертса

Данный алгоритм работает в объектном пространстве, решая задачу только с выпуклыми телами.

Алгоритм выполняется в 3 этапа.

#### Этап подготовки исходных данных

На данном этапе должна быть задана информация о телах. Для каждого тела сцены должна быть сформирована матрица тела  $V$ . Размерность матрицы -  $4 * n$ , где  $n$  – количество граней тела.

Каждый столбец матрицы представляет собой четыре коэффициента уравнения плоскости  $ax + by + cz + d = 0$ , проходящей через очередную грань.

Таким образом, матрица тела будет представлена в следующем виде

$$V = \begin{pmatrix} a_1 & a_2 & \dots & a_n \\ b_1 & b_2 & \dots & b_n \\ c_1 & c_2 & \ddots & c_n \\ d_1 & d_2 & \dots & d_n \end{pmatrix} \quad (1.1)$$

Матрица тела должна быть сформирована корректно, то есть любая точка, расположенная внутри тела, должна располагаться по положительную сторону от каждой грани тела. В случае, если для очередной грани условие не выполняется, соответствующий столбец матрицы надо умножить на  $-1$ .

#### Этап удаления рёбер, экранируемых самим телом

На данном этапе рассматривается вектор взгляда  $E = \{0, 0, -1, 0\}$ . Для определения невидимых граней достаточно умножить вектор  $E$  на матрицу тела  $V$ . Отрицательные компоненты полученного вектора будут соответствовать невидимым граням.

#### Этап удаления невидимых рёбер, экранируемых другими телами сцены

На данном этапе для определения невидимых точек ребра требуется построить луч, соединяющий точку наблюдения с точкой на ребре. Точка будет невидимой, если луч на своём пути встречает в качестве преграды

рассматриваемое тело [3].

### **Преимущества**

- работа в объектном пространстве;
- высокая точность вычисления.

### **Недостатки**

- рост сложности алгоритма – квадрат числа объектов;
- тела сцены должны быть выпуклыми (усложнение алгоритма, так как нужна будет проверка на выпуклость);
- удаляет только линии, удалить поверхности не получится;
- плохо распараллеливается;
- сложность реализации.

Данный алгоритм не подходит для решения поставленной задачи из-за высокой сложности реализации как самого алгоритма, так и его модификаций, отсюда низкая производительность.

### **1.4.4 Алгоритм художника**

Данный алгоритм работает аналогично тому, как художник рисует картину – то есть сначала рисуются дальние объекты, а затем более близкие. Наиболее распространенная реализация алгоритма – сортировка по глубине, которая заключается в том, что произвольное множество граней сортируется по ближнему расстоянию от наблюдателя, а затем отсортированные грани выводятся на экран в порядке от самой дальней до самой ближней. Данный метод работает лучше для построения сцен, в которых отсутствуют пересекающиеся грани [4].

### **Преимущества**

- требование меньшей памяти, чем, например, алгоритм Z-буфера.

## **Недостатки**

- недостаточно высокая реалистичность изображения;
- сложность реализации при пересечения граней на сцене;
- недостаточное распараллеливание - отрисовка объектов строго последовательная.

Данный алгоритм не отвечает главному требованию – параллельности. Также алгоритм художника отрисовывает все грани (в том числе и невидимые), на что тратится большая часть времени.

### **1.4.5 Алгоритм Варнока**

Алгоритм Варнока [5] является одним из примеров алгоритма, основанного на разбиении картинной плоскости на части, для каждой из которых исходная задача может быть решена достаточно просто.

Поскольку алгоритм Варнока нацелен на обработку картинки, он работает в пространстве изображения. В пространстве изображения рассматривается окно и решается вопрос о том, пусто ли оно, или его содержимое достаточно просто для визуализации. Если это не так, то окно разбивается на фрагменты до тех пор, пока содержимое фрагмента не станет достаточно простым для визуализации или его размер не достигнет требуемого предела разрешения.

Сравнивая область с проекциями всех граней, можно выделить случаи, когда изображение, получающееся в рассматриваемой области, определяется сразу:

- проекция ни одной грани не попадает в область;
- проекция только одной грани содержится в области или пересекает область, то в этом случае проекции грани разбивают всю область на две части, одна из которых соответствует этой проекции;
- существует грань, проекция которой полностью накрывает данную область, и эта грань расположена к картинной плоскости ближе, чем

все остальные грани, проекции которых пересекают данную область, то в данном случае область соответствует этой грани.

Если ни один из рассмотренных трех случаев не имеет места, то снова разбиваем область на четыре равные части и проверяем выполнение этих условий для каждой из частей. Те части, для которых таким образом не удалось установить видимость, разбиваем снова и т. д.

## Преимущества

- меньшие затраты по времени в случае области, содержащий мало информации.

## Недостатки

- алгоритм работает только в пространстве изображений;
- большие затраты по времени в случае области с высоким информационным содержимым;
- алгоритм нельзя распараллелить простым способом.

Данный алгоритм не отвечает требованию параллельности, а также возможны большие затраты по времени работы.

### 1.4.6 Вывод

Для удаления невидимых линий выбран алгоритм обратной трассировки лучей. Данный алгоритм позволит добиться максимальной реалистичности и даст возможность смоделировать распространение света в пространстве, учитывая законы геометрической оптики. Данный алгоритм можно модернизировать, добавив в него обработку новых световых явлений. Также этот алгоритм позволяет строить качественные тени с учетом большого числа источников. Стоит отметить тот факт, что алгоритм трассировки лучей не требователен к памяти, в отличие, например, от алгоритма Z-буфера.

## 1.5 Анализ и выбор модели освещения

Физические модели материалов стараются аппроксимировать свойства некоторого реального материала. Такие модели учитывают особенности поверхности материала или же поведение частиц материала.

Эмпирические модели материалов устроены иначе, чем физически обоснованные. Данные модели подразумевают некий набор параметров, которые не имеют физической интерпретации, но которые позволяют с помощью подбора получить нужный вид модели.

В данной работе следует делать выбор из эмпирических моделей, а конкретно из модели Ламберта и модели Фонга.

### 1.5.1 Модель Ламберта

Модель Ламберта [6] моделирует идеальное диффузное освещение, то есть свет при попадании на поверхность рассеивается равномерно во все стороны. При такой модели освещения учитывается только ориентация поверхности ( $N$ ) и направление источника света ( $L$ ). Иллюстрация данной модели представлена на рисунке 1.1.

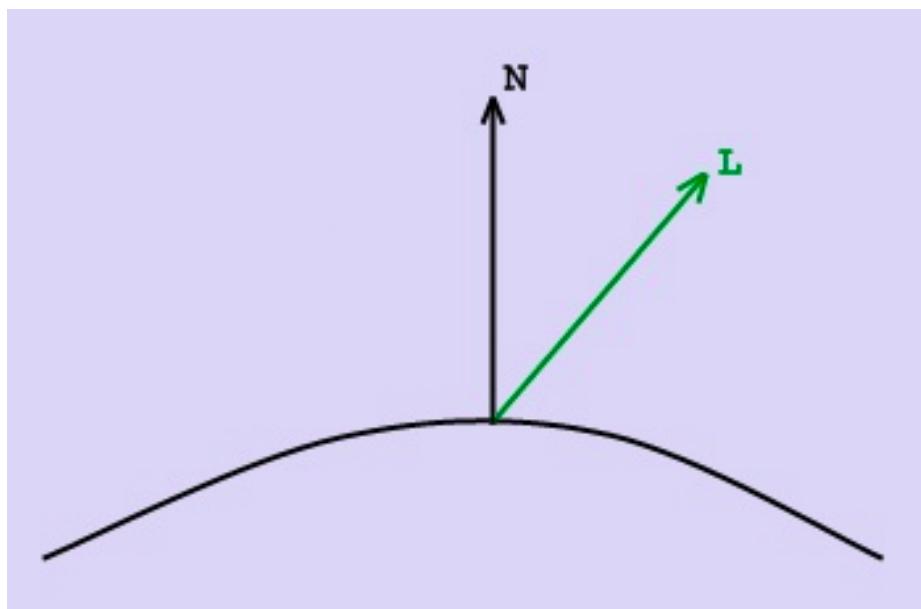


Рисунок 1.1 – Направленность источника света

Эта модель является одной из самых простых моделей освещения и

очень часто используется в комбинации с другими моделями. Она может быть очень удобна для анализа свойств других моделей, за счет того, что ее легко выделить из любой модели и анализировать оставшиеся составляющие.

### 1.5.2 Модель Фонга

Это классическая модель освещения. Модель представляет собой комбинацию диффузной и зеркальной составляющих. Работает модель таким образом, что кроме равномерного освещения на материале могут появляться блики. Местонахождение блика на объекте определяется из закона равенства углов падения и отражения. Чем ближе наблюдатель к углам отражения, тем выше яркость соответствующей точки [6].

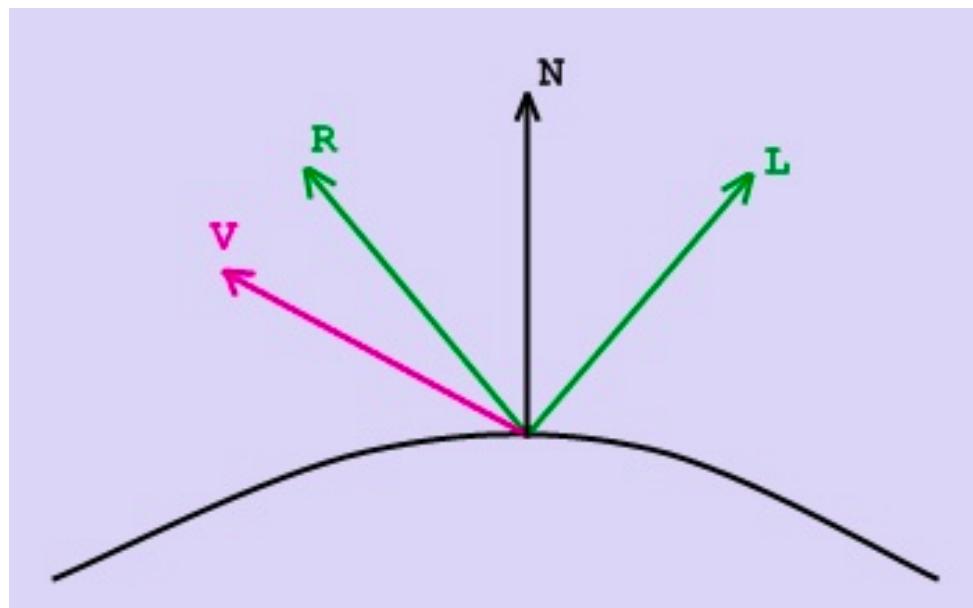


Рисунок 1.2 – Направленность источника света

Падающий и отраженный лучи лежат в одной плоскости с нормалью к отражающей поверхности в точке падения (рисунок 1.2). Нормаль делит угол между лучами на две равные части.  $L$  – направление источника света,  $R$  – направление отраженного луча,  $V$  – направление на наблюдателя.

### **1.5.3 Вывод**

Для освещения была выбрана модель Фонга, потому что планируется реализовать программную поддержку шейдеров. Во время работы шейдера мы уже получаем барицентрические координаты пикселя на этапе растеризации, эти координаты можно легко использовать интерполяции нормали и освещения Фонга, так как они уже посчитаны.

## **1.6 Проблема неравенства граней по площади**

В решении последующей задачи фигурирует проблема неравенства по площади граней. Разные грани модели содержат, после проецирования, разное количество пикселов. Модель параллельности SIMD, используемая в графическом ускорителе не может независимо обрабатывать грани разной площади. Графический ускоритель шлет инструкции группе ядер, и если ядро видеокарты обрабатывает грань с большой площадью, другие ядра не могут продолжить обрабатывать потоки, пока то не закончит. Эта проблема получила название "Warp Divergence" - дивергенция варпов (в англоязычной литературе). С ней связано решение использовать виртуальную геометрию (введена позже).

## **Вывод**

В данном разделе был проведен анализ алгоритмов удаления невидимых линий и модели освещения, которые возможно использовать для решения поставленных задач. В качестве ключевого алгоритма, который также можно оптимизировать, выбран алгоритм обратной трассировки лучей, который будет реализован в рамках данного курсового проекта.

## **2 Конструкторский раздел**

В данном разделе будут рассмотрены требования к программе и алгоритмы визуализации сцены.

### **2.1 Общий алгоритм решения поставленной задачи**

1. загрузить объекты сцены;
2. отобразить объекты сцены;
3. позволить пользователю изменять параметры и показать результат в реальном времени.

### **2.2 Шаги графического конвейера**

Рассмотрим работу графического конвейера.

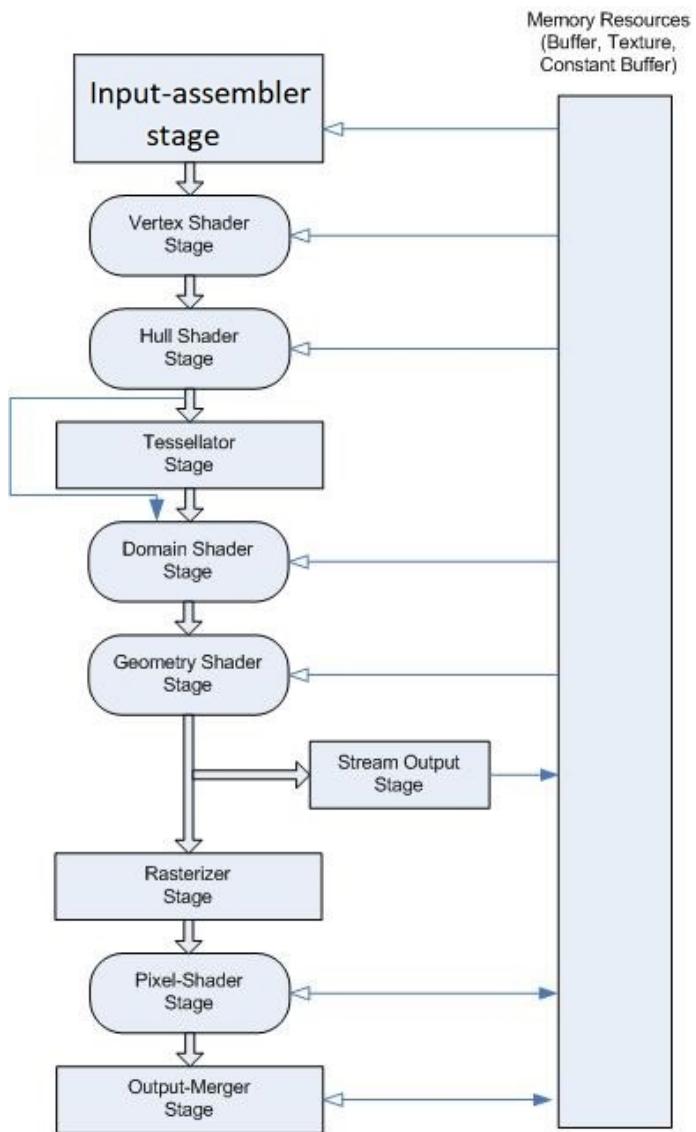


Рисунок 2.1 – Работа графического конвейера

Для простоты реализации, и учитывая факт, что большинство из этих шагов недоступны разработчикам, было принято решение ограничить их число.

1. шаг отбрасывания невидимых объектов (работает на ограничивающих сферах на процессоре);
2. шаг тесселяции;
3. шаг формирования z-буфера;
4. шейдер вершин;
5. шаг отбрасывания целиком невидимых поверхностей;

## 6. шаг растеризации и шейдер пикселей.

Шаги шейдера вершин, отбрасывания целиком невидимых поверхностей, а также растеризации объединены в один. Это связано с производительностью реализации. Рассмотрим шаги по порядку.

### 2.3 Шаг отбрасывания невидимых объектов

Отбрасывание полностью невидимых объектов происходит на процессоре. Для этого используется пирамида видимости, представленная на рисунке 2.2. Это обусловлено использованием перспективной проекции. Пирамида является усеченной и состоит из 6 граней. Эти грани можно аналитически описать с помощью уравнений плоскости. Создается оболочка, содержащая трехмерный объект. Для определения видимости объекта решается задача нахождения расположения оболочки относительно граней пирамиды видимости. Виды оболочек показаны на рисунке 2.3.

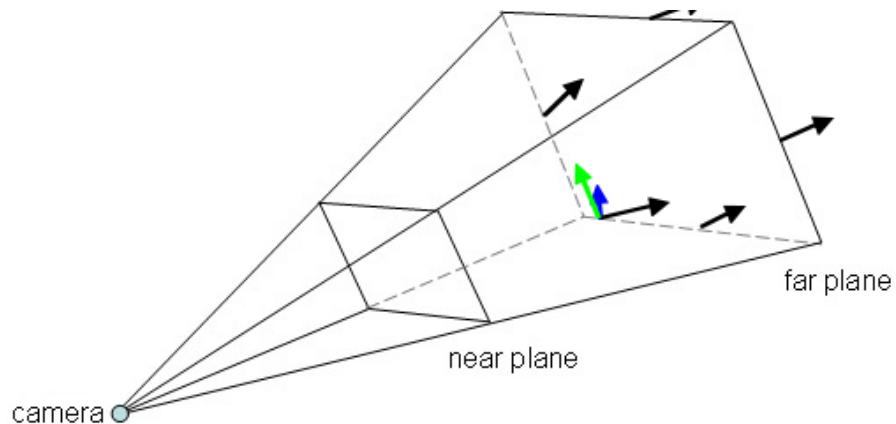


Рисунок 2.2 – Пирамида видимости

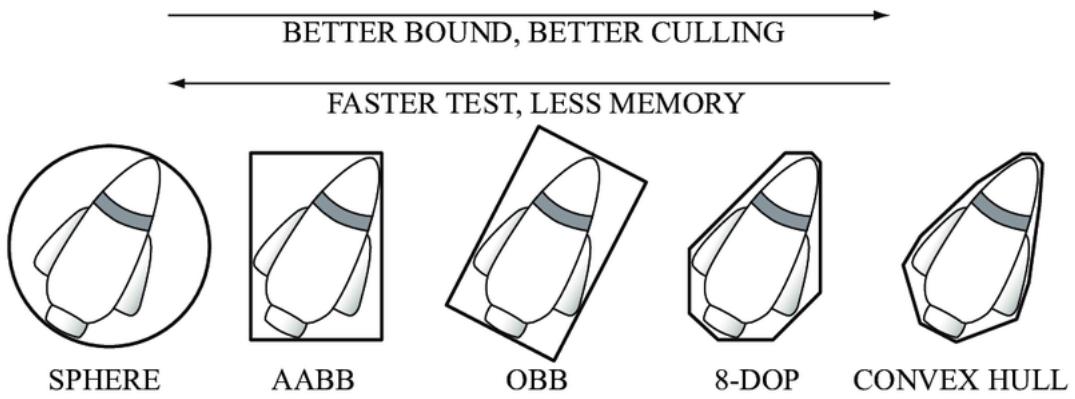


Рисунок 2.3 – Виды оболочек, левее - используют меньше ресурсов, правее - предоставляют большую точность

Для быстродействия и простоты имплементации была выбрана сферическая оболочка. В данном случае возможны три варианта расположения сферы относительно плоскости, показанные на рисунке 2.4. Для того чтобы объект считался полностью невидимым, его сфера должна находиться по невидимую сторону от любой из граней усеченной пирамиды. Иначе, объект считается видимым, или частично видимым, и происходит вызов его отрисовки.

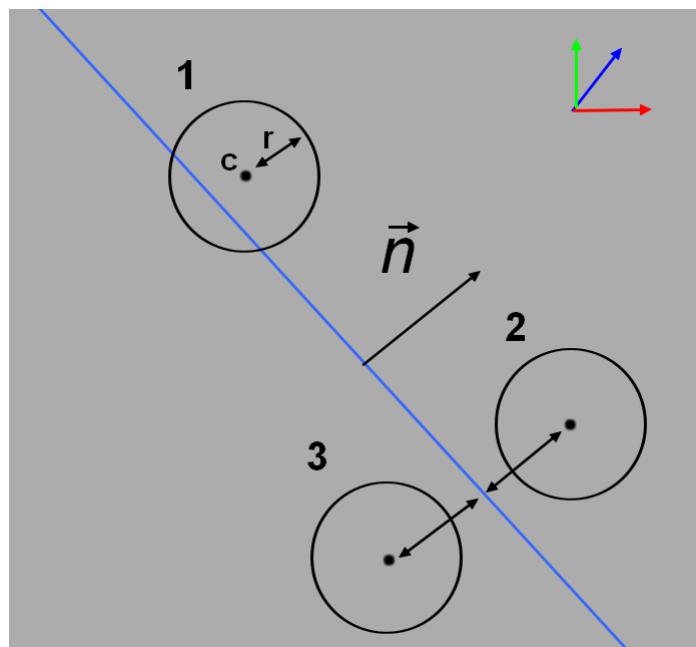


Рисунок 2.4 – Варианты расположения сферы относительно плоскости

### 2.3.1 Асинхронный анализ виртуальной геометрии

Данный шаг, в совокупности с последующим, аналогичен тесселляции в реальной графическом конвейере. Виртуальной назовем геометрию, которой не существует в статическом описании модели на файле. Виртуальная геометрия генерируется процедурно, во время выполнения программы.

В этом шаге считается площадь каждой грани отрисовываемой модели (прошёдшей фильтрацию по области видимости). Границы, площадь которых превышает порог, рекурсивно разбиваются на подграницы виртуальной геометрии в следующем шаге. При этом, на данном шаге вычисляется количество виртуальных граней, чтобы заранее выделить необходимый объём видеопамяти при построении геометрии. Таким образом, увеличивается количество отрисовываемых треугольников. Теоретически, уменьшение количества тоже возможно, но выходит за рамки работы.

Используется несколько анализаторов моделей, каждый из которых работает асинхронно. Параллелизм идет по граням отдельной модели. Количество анализаторов задано статически, и превышает количество виртуальных моделей. Таким образом все виртуальные модели анализируют.

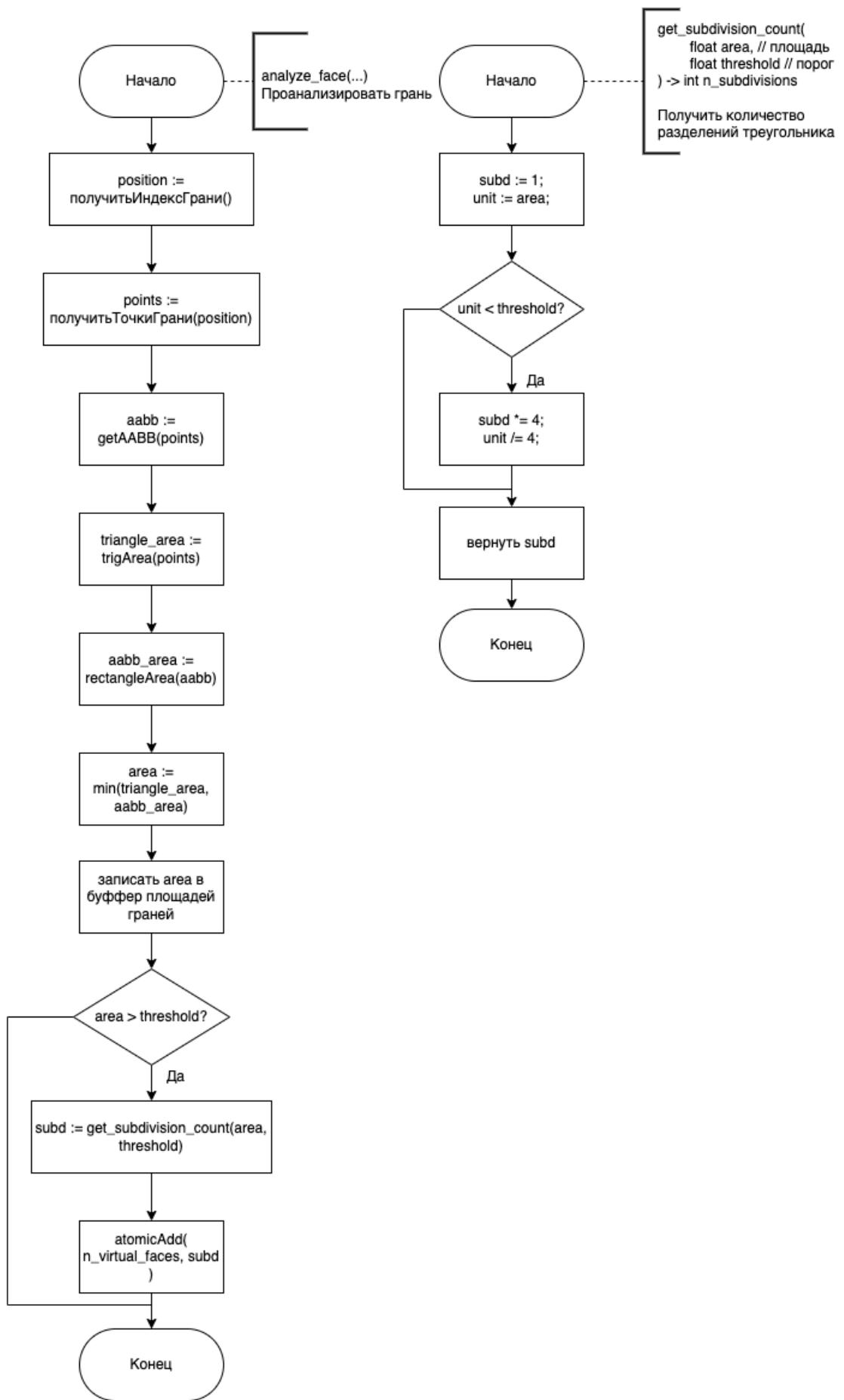


Рисунок 2.5 – Разработка алгоритма анализа геометрии

## 2.3.2 Создание или обновление виртуальных объектов

Выделяются буферы для вершин, граней и дополнительных векторов нормалей, текстур. Новые треугольники копируются в отдельную модель, называемую виртуальной. Логически, это отдельная модель, которая инъектируется в множество отрисовываемых моделей на этапе отрисовки. Границы, заменяемые новой моделью, помечаются как неактивные, и не обрабатываются.

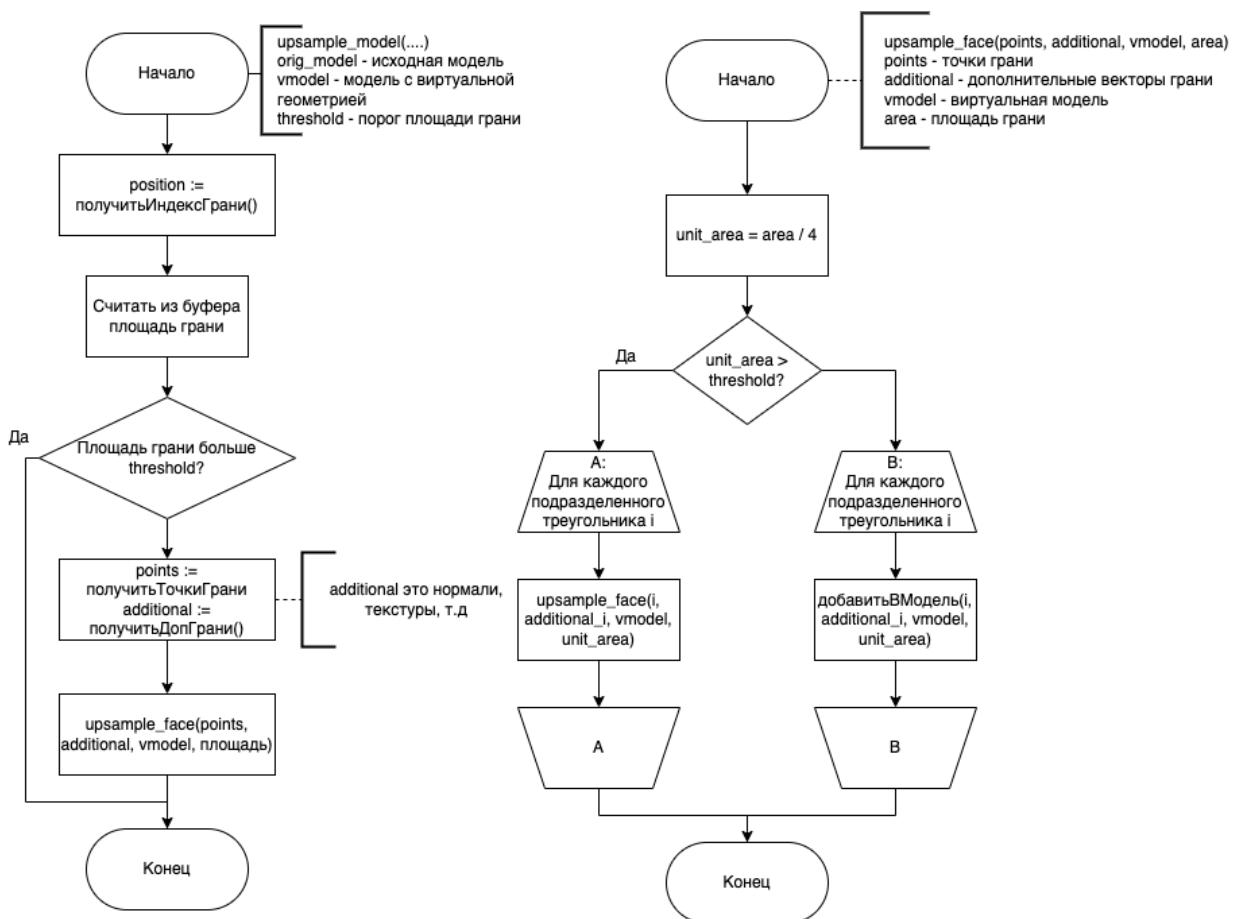


Рисунок 2.6 – Разработка алгоритма создания виртуальной геометрии

## 2.4 Алгоритм z-буфера

Алгоритм z-буфера был выбран из за его чрезвычайной параллельности и, как следствие, быстродействия. Реальный графический конвейер использует

зует именно этот алгоритм.

Для демонстрации проблем, возникших в реализации курсового проекта, можно рассмотреть подход к разработке алгоритма Z-буфера.

Можно реализовать его тремя способами:

- параллельность ведется по граням моделей. При этом приходится использовать операцию AtomicCAS, блокирующую шину памяти.
- параллельность ведется по пикселям на экране. При этом в цикле рассматривается множество граней модели.
- модификация первого подхода, но несколько моделей отрисовываются в отдельные буфера, которые потом параллельно объединяются.

Были проведены тесты, в которых последний подход оказался самым быстродействующим. Отчасти это связано с отсутствием конфликтов в банках памяти, при выполнении неделимой операции atomicCAS к разным буферам. С подобными проблемами прошлось сталкиваться во всех аспектах реализации.

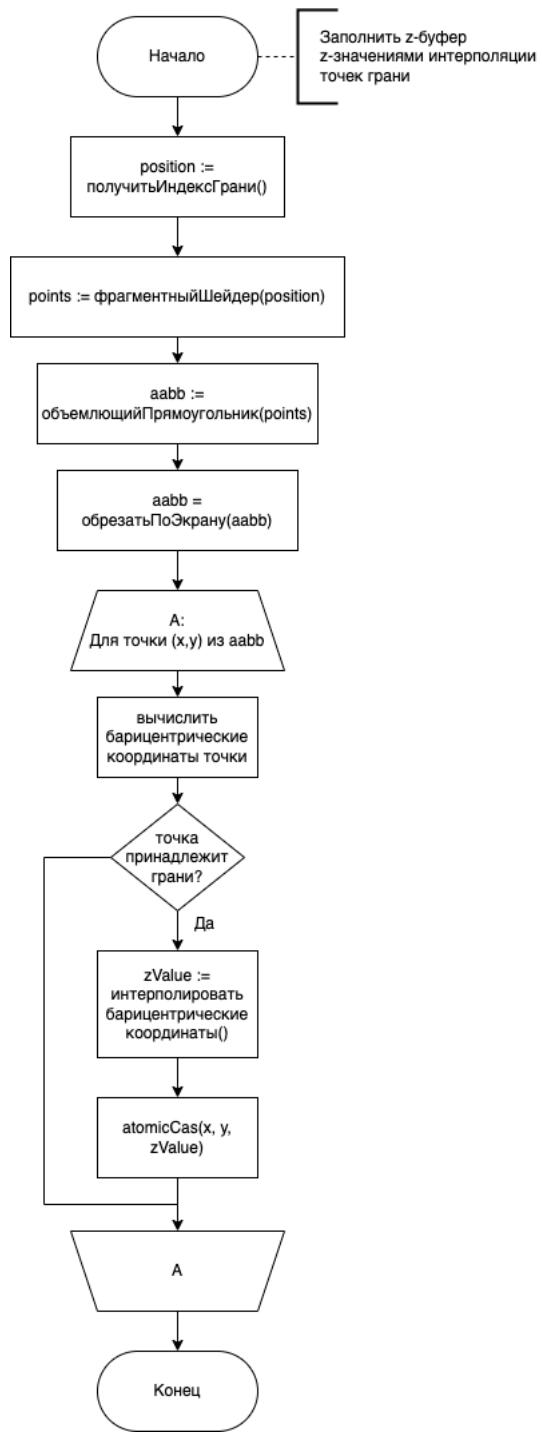


Рисунок 2.7 – Разработка алгоритма z-буфера

Как было оговорено выше, модели на сцене равномерно распределяются между z-буферами. Всего буферов  $2^n$  штук. В реализации  $n=32$ .

После обработки каждой, происходит слияние z-буферов в один, используя алгоритм параллельной редукции. Идея его работы показана на рисунке 2.8. Параллелизм происходит по пикселям буфера. Два буфера на каждом шаге обрабатываются независимо (асинхронно) от других.

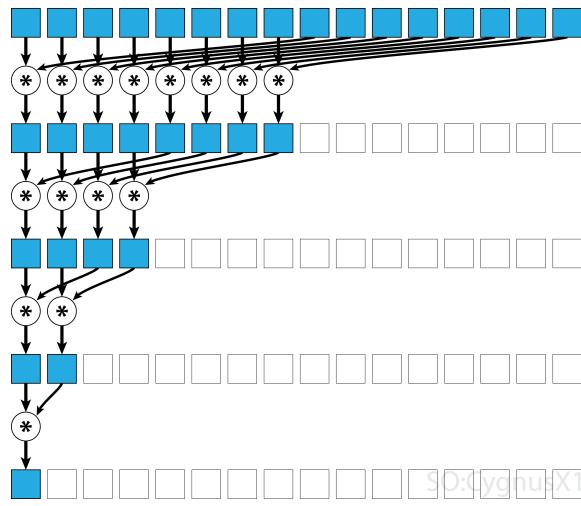


Рисунок 2.8 – Идея работы алгоритма параллельной редукции

## 2.5 Отрисовка объектов

После заполнения z-буфера, происходит отрисовка объектов, используя алгоритм растеризации.

Каждая грань отрисовывается в отдельном потоке. При этом используется информация z-буфера. Если интерполированное значение проецированного пикселя равно значению z-буфера, такрй пиксел выводится на растр.

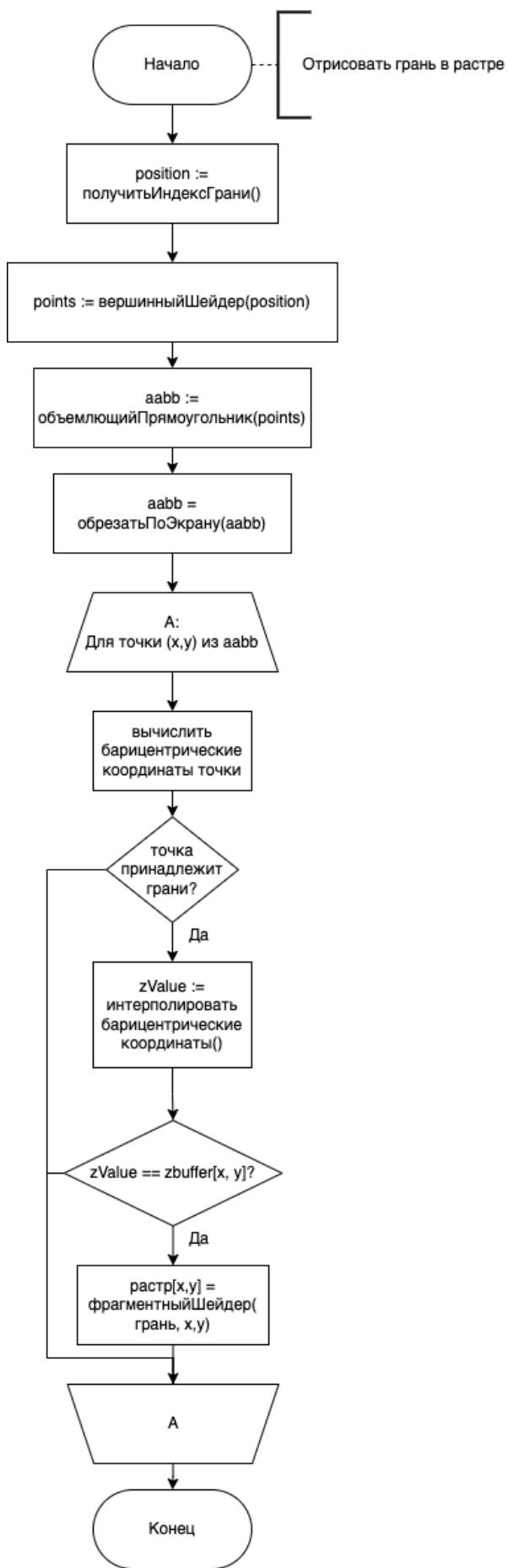


Рисунок 2.9 – Разработка алгоритма растеризации

## 2.6 Выбор используемых типов и структур данных

Для разрабатываемого ПО необходимо реализовать следующие типы и структуры данных.

1. Сцена - список объектов.

```
class Scene {  
private:  
    std::vector<SceneObject> models{}; // Модели  
    std::shared_ptr<Camera> camera{}; // Камера  
    glm::vec3 light_dir{0, 0, 1}; // Направление освещения  
    int id_counter = 0; // id последнего зарегистрированного объекта  
    int scene_id = 0; // версионный вектор лампорта сцены  
    int time = 0; // время сцены  
    bool sorted = false; // отсортированы ли объекты по времени  
  
    // callback, вызывается каждый тик  
    std::function <void(Scene &)> on_update = [] (Scene &) {};
```

2. Объекты сцены

```
struct ModelRef {  
    TextureRef texture{};  
    glm::vec3 *vertices{}; // вершины  
    glm::vec3 *normals{};  
    glm::vec2 *textures{};  
    // индекс текстуры для грани  
    glm::ivec3 *textures_for_face{};  
    glm::ivec3 *faces{}; // грани  
    // кол-во вершин, граней, текстур  
    int n_vertices = 0;  
    int n_faces = 0;
```

```

        int max_texture_index = 0;
        int id = 0;
        // объемлющая сфера для куллинга
        Sphere *bounding_volume{};

    RegisteredShaders shader = RegisteredShaders::Default;

    bool is_virtual = false; // является ли модель виртуальной
};

```

### 3. Текстура

```

struct TextureRef
{
    int x = 0; // ширина
    int y = 0; // высота
    int n = 0; // кол-во каналов
    uchar3 *data{}; // данные текстуры
    [[nodiscard]] __device__ uchar3 get_uv(float u, float v)
};

```

### 4. Шейдер.

```

struct BaseShader
{
    glm::mat4 projection;
    glm::mat4 view;
    glm::mat4 model_matrix;

    ModelRef &model;
    const DrawCallBaseArgs &base_args;

    glm::vec3 pts[3]{};
    glm::vec3 normals[3]{};
    glm::vec2 textures[3]{};
    glm::vec3 light_dir{};

```

```
glm::vec2 screen_size{};  
int position = 0;  
  
float4 vertex(int iface, int nthvert, bool load_tex);  
  
bool fragment(glm::vec3 bar,  
              uint &output_color,  
              float z_value);  
};
```

5. Математические абстракции:

- (a) вектор;
- (b) сфера;
- (c) луч;
- (d) плоскость;

## Вывод

В данном разделе были подробно рассмотрены алгоритмы, которые будут реализованы, приведены схемы алгоритмов для решения поставленной задачи и описаны используемые структуры.

# **3 Технологический раздел**

В этом разделе будет обоснован выбор языка программирования и среды разработки, рассмотрена диаграмма основных классов и разобран интерфейс, предлагаемый пользователю.

## **3.1 Требования к программе**

Программа должна предоставлять следующие возможности:

- визуальное отображение сцены;
- перемещение объектов;
- выбор сцены;
- профилирование времени отрисовки кадра;
- настройку виртуальной геометрии;
- настройку куллинга;
- настройку сцены и объектов сцены;

## **3.2 Выбор языка программирования и среды разработки**

Существует множество языков, а также сред программирования, многие из которых обладают достаточно высокой эффективностью, удобством и простотой в использовании.

Для разработки данной программы были выбраны языки C++ и Nvidia Cuda. Данный выбор обусловлен следующими факторами:

- Cuda позволяет написать функции, которые будут выполняться на графическом ускорителе. Это один из немногих языков общего назначения, имеющих эту возможность (к примеру, GLSL предназначен только для написания шейдеров);

- Cuda - надмножество языка C++;
- объектные файлы C++ и Cuda компонуются;
- C++ позволяет реализовать некоторые идиомы программирования, которые полезны в данном проекте для производительности, например: статический полиморфизм, использование структур(POD), низкоуровневое взаимодействие с памятью (cudaMalloc, cudaMemcpy);
- трехмерные объекты, также как и математические абстракции, естественным образом представляются в виде структур, что позволяет легко и эффективно организовывать их взаимодействие, при этом сохраняется читаемый и легко изменяемый код;
- стандартная библиотека Thrust, представляющая инструментарий STL в виде c++, оптимизированный для работы на графическом ускорителе, используя чрезвычайную параллельность, сортирующие сети, и упаковку потоков (stream compaction);
- у меня есть опыт использования обоих языков.

В качестве среды разработки была CLion. Некоторые факторы по которым была выбрана данная среда.

1. включает весь основной функционал: параллельная сборка, отладчик, поддержка точек останова, сборки и т.д;
2. разработчики имеют возможность расширить любой функционал, включая компиляцию, отладку;
3. поддерживает оба языка программирования;
4. данная среда бесплатна для студентов;

В конечном итоге стек используемых инструментов принимает вид:

1. C++ / Cuda;
2. gcc + nvcc;
3. cmake;
4. OpenGL для вывода растра на экран;

### 3.3 Структура программы

Так при написании программы используется язык C++, он имеет сразу несколько парадигм. Так как упор программы делается на lockless многопоточность, особое внимание удалено функциональной парадигме. Из за плохой нативной поддержки, и производительности объектно ориентированного кода, он в меньшей мере был использован на графическом ускорителе. Если полиморфизм и реализован, то только статический.

Написано большое количество вспомогательных структур, но они не запускаются на видеокарте и не обладают параллельностью. Их использование не релевантно тематике работы.

Условно классы в программе можно разделить на несколько групп по выполняемым функциям.

Рассмотрим работу вызова отрисовки:

```
void DrawCaller::draw(DrawCallArgs args_unculled, Image &image)
{
    // interface
    interface->log_fps();
    interface->log_before_culling(args_unculled.models.size());

    DrawCallArgs args;

    if (interface->is_culling_enabled()) {
        args = culler->cull(args_unculled,
                             *args_unculled.base.camera_ptr);
    } else {
        args = args_unculled;
    }
    interface->log_after_culling(args.models.size());

    if (interface->is_virtual_geometry_enabled()) {
        virtual_geometry_manager->
            populate_virtual_models(args, image,
```

```

        args_unculled) ;
    }

    // dispatch
    image_resetter->async_reset(image);
    size_t streams_to_use = std::min(args.models.size(),
        (size_t)n_streams);
    for (size_t i = 0; i < streams_to_use; i++)
    {
        zfillers[i]->resize(image);
        zfillers[i]->async_reset();
    }

    for (size_t i = 0; i < args.models.size(); i++)
        zfillers[i % zfillers.size()]->async_zbuf(args, i);

    // sync
    for (size_t i = 0; i < streams_to_use; i++)
        zfillers[i]->await();
    image_resetter->await();

    auto zbuffers = get_z_buffers();
    // parallel dispatch and sync
    parallel_reduce_merge(z_mergers, zbuffers,
        (int)streams_to_use);
    auto final_zbuffer = get_final_z_buffer();

    // dispatch
    for (size_t i = 0; i < args.models.size(); i++)
        rasterizers[i % rasterizers.size()]->async_rasterize(
            args,
            i, image, final_zbuffer);

    // sync

```

```

        for (size_t i = 0; i < streams_to_use; i++)
            rasterizers[i]->await();

        interface->draw_widget();
    }
}

```

## 3.4 Интерфейс

На рисунке 3.1 представлен интерфейс программы.

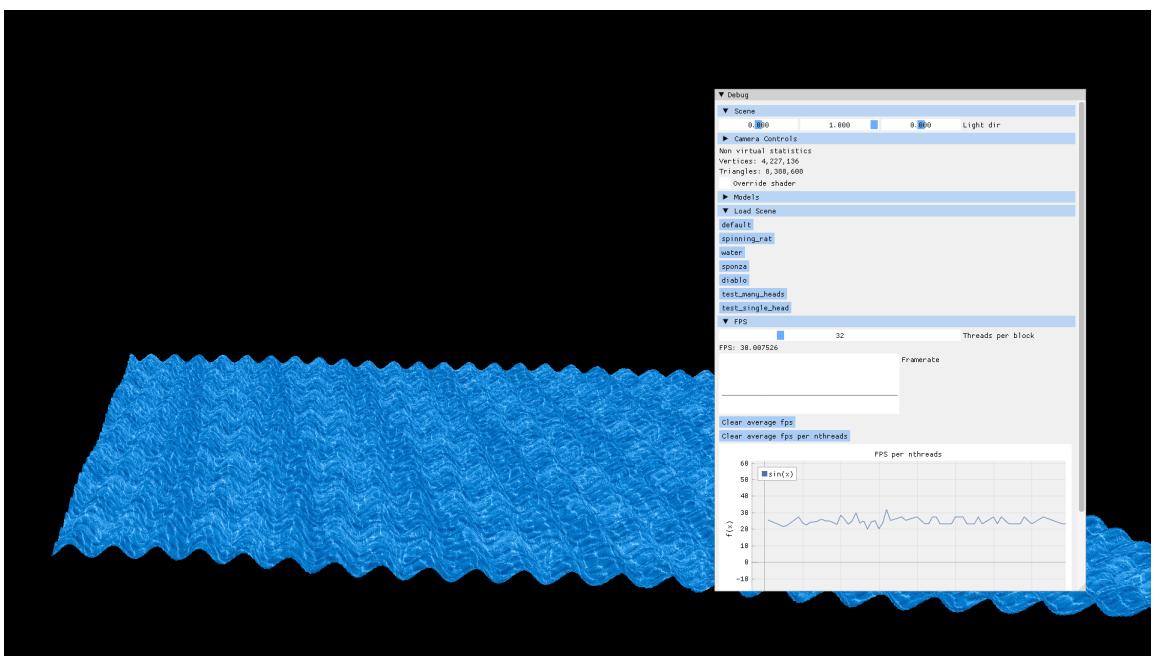


Рисунок 3.1 – Идея работы алгоритма параллельной редукции

Интерфейс состоит из секций:

- секция "Scene"
  - слайдер "Light Dir" – контролирует направление источника освещения
  - секция "Camera Controls"
    - \* слайдер "Camera XY" – XY координаты камеры;
    - \* слайдер "Camera Z" – Z координаты камеры;
    - \* слайдер "Look dir yaw" – угол камеры по горизонтали;

- \* слайдер "Look dir pitch" – угол камеры по вертикали;
  - \* слайдер "FOV" – параметр поля зрения камеры
  - \* слайдер "zFar" – крайняя координата z пирамиды видимости для обрезки моделей;
- заголовок "Non virtual statistics" – статистика невиртуальных моделей - количество отрисовываемых граней и вершин;
- чекбокс "Override Shader" и выбор Shader - выбрать глобальный шейдер для использования всеми объектами сцены.
- секция "Models" – содержит список объектов
  - слайдер "Position" – позиция модели на сцене;
  - выбор "Shader" – шейдер, который использует модель;
- секция "Load Scene" – содержит список предопределенных сцен для загрузки;
- секция "FPS" – содержит информацию о производительности (кадрах в секунду)
  - слайдер "Threads per Block" – слайдер с выбором кол-ва потоков, которые будут использовать Cuda kernel;
  - график "FPS" – отображает кол-во кадров в секунду;
  - кнопки "Clean X" – очищают график "FPS per nthreads";
  - график "FPS per nthreads" – по оси X - кол-во кадров, по Y - кол-во фпс (среднее);
- секция "Culling" – содержит настройки удаления полностью невидимых объектов (куллинга)
  - чекбокс "Enable Culling" – включает удаление невидимых объектов;
  - заголовок "Called to draw models: n" – n - кол-во объектов поступивших на отрисовку;
  - заголовок "Drawn after culling: n" – n - кол-во отрисованных объектов;

- график ”Culling Percentage” – по оси X - время, по оси Y - процент отброшенных объектов;
- секция ”Virtual Geometry” – виртуальная геометрия
  - слайдер ”Threshold” – площадь грани в пикселях, после которого она будет дробиться на подграниц;

## 3.5 Результаты работы программного обеспечения

На рисунке 3.2 приведен результат отрисовки модели головы.



Рисунок 3.2 – Модель головы

На рисунке показана та же голова, вблизи. Видна проблема разных площадей граней, видимые артефакты на шее. Это связано с тем, что их площадь слишком большая, и алгоритм растеризации попросту останавливается, не дорисовывая эти грани. Количество кадров в секунду (FPS) равно 30.

На рисунке 3.4 представлен вид геометрии объекта. Видны треугольники больших площадей, с теми же артефактами.

На рисунке 3.5 включена виртуальная геометрия. Число треугольников увеличилось, артефакты пропали.

Рисунок 3.6 показывает ту же модель, при ее обычном шейдере, без артефактов, с обновленной виртуальной геометрией. При этом количество кадров возрастает до 60. Можно менять положение камеры и при этом наблюдать изменение виртуальной геометрии в реальном времени.

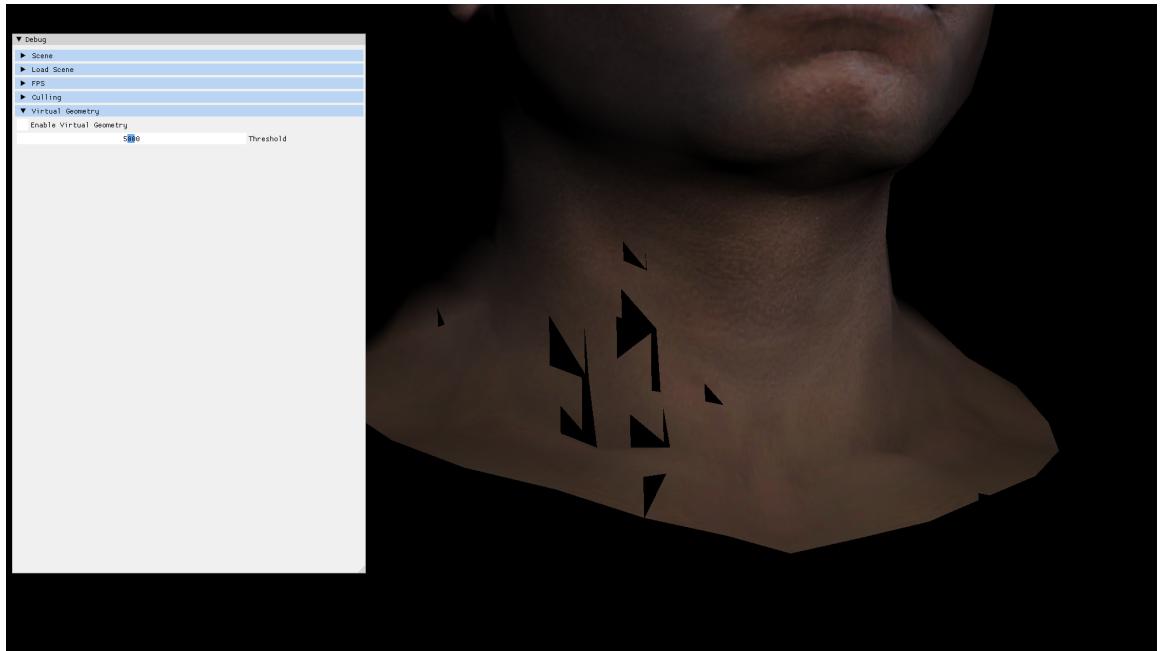


Рисунок 3.3 – Модель головы вблизи, видны артефакты из-за разности площадей грани

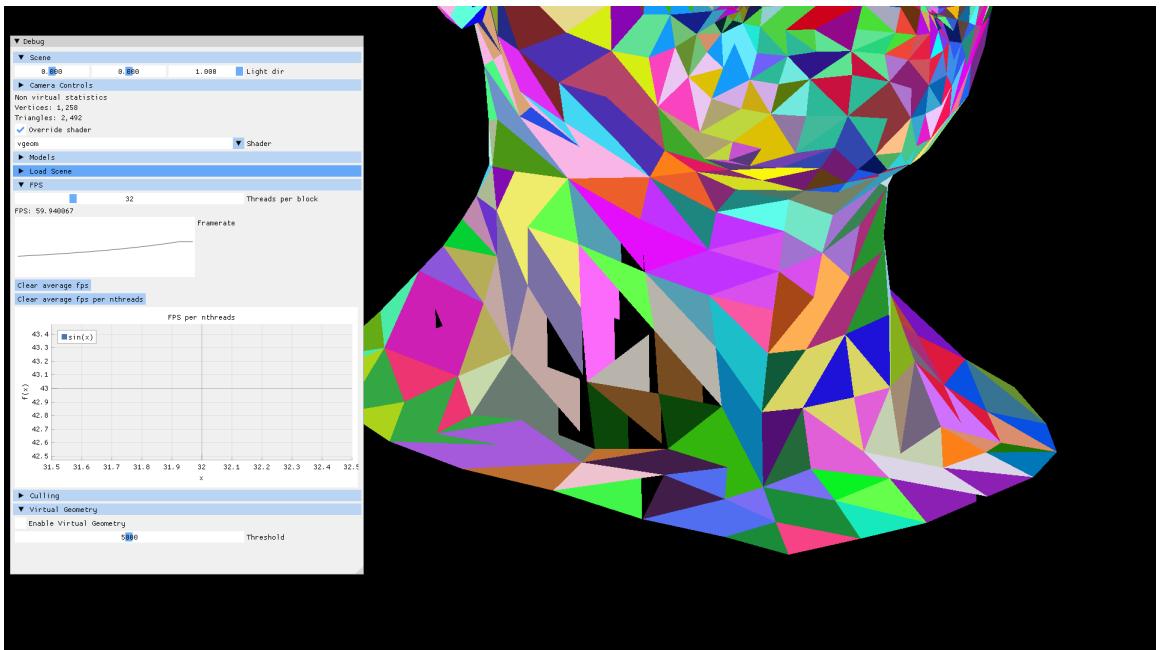


Рисунок 3.4 – Модель головы вблизи, геометрический шейдер, видны артефакты из-за разности площадей граней

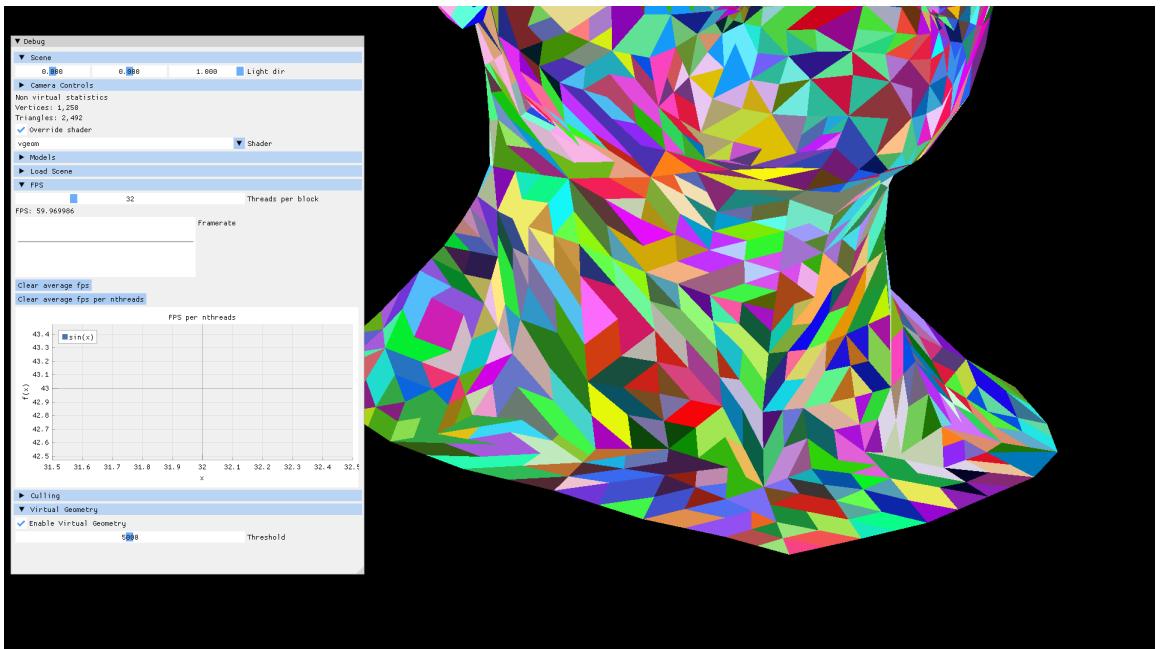


Рисунок 3.5 – Модель головы вблизи, геометрический шейдер, при виртуальной геометрии число треугольников увеличилось, а артефакты пропали.

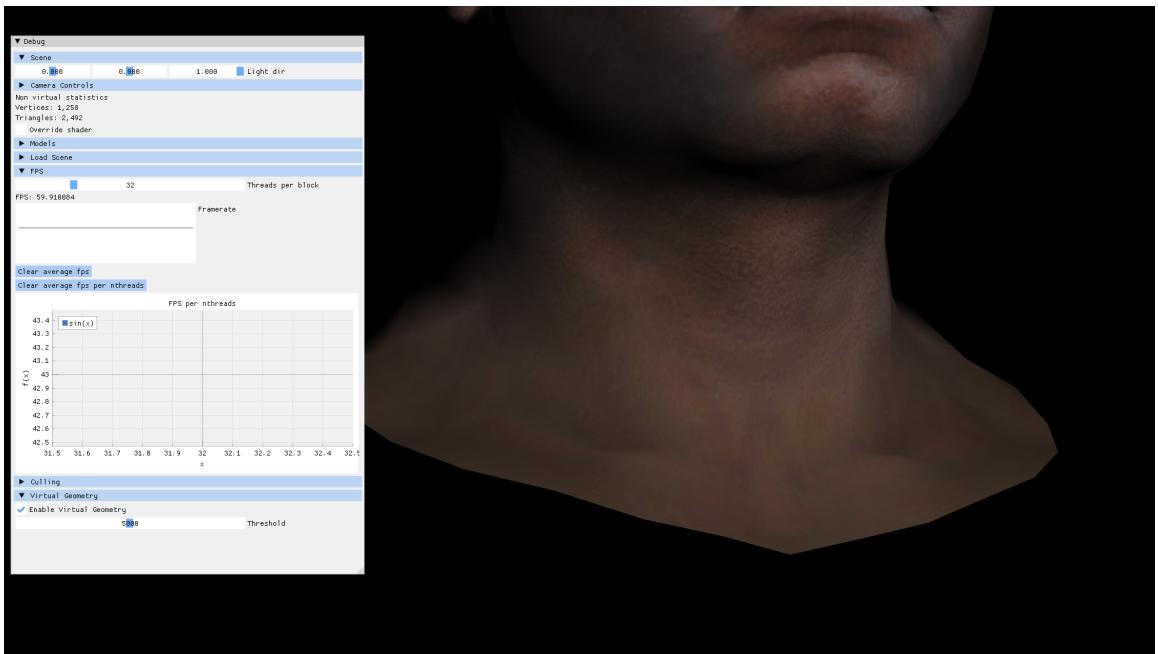


Рисунок 3.6 – Модель головы вблизи, при виртуальной геометрии, артефакты пропали а FPS возрос.

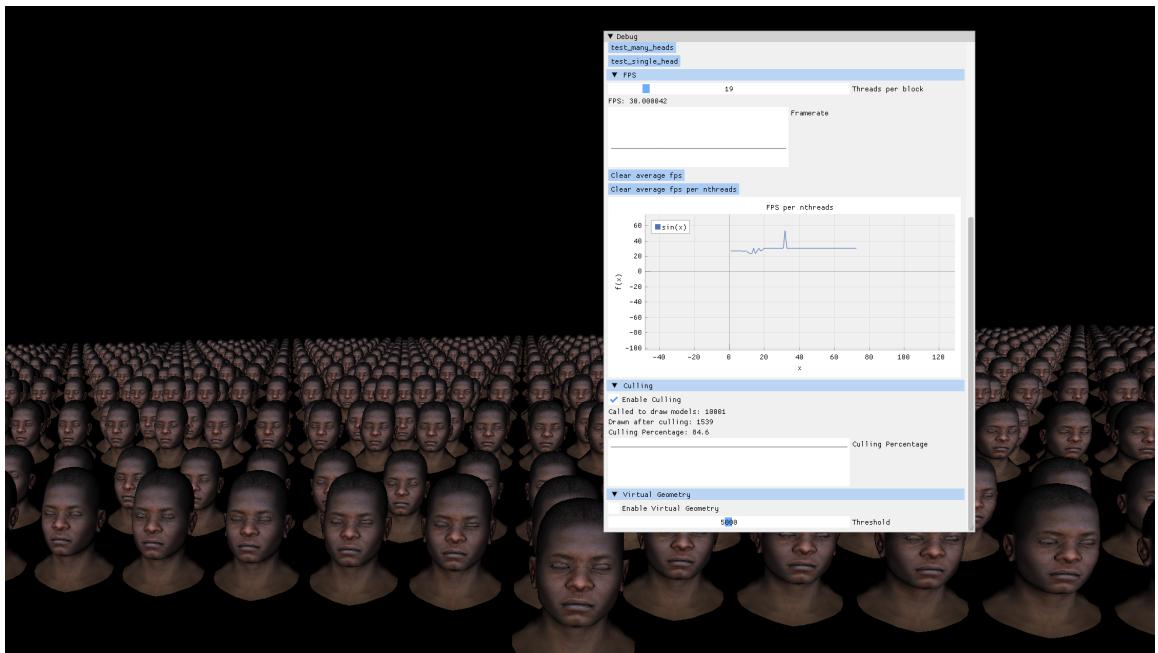


Рисунок 3.7 – Отрисовка 10.000 объектов (30 FPS при Culling, 20 без)

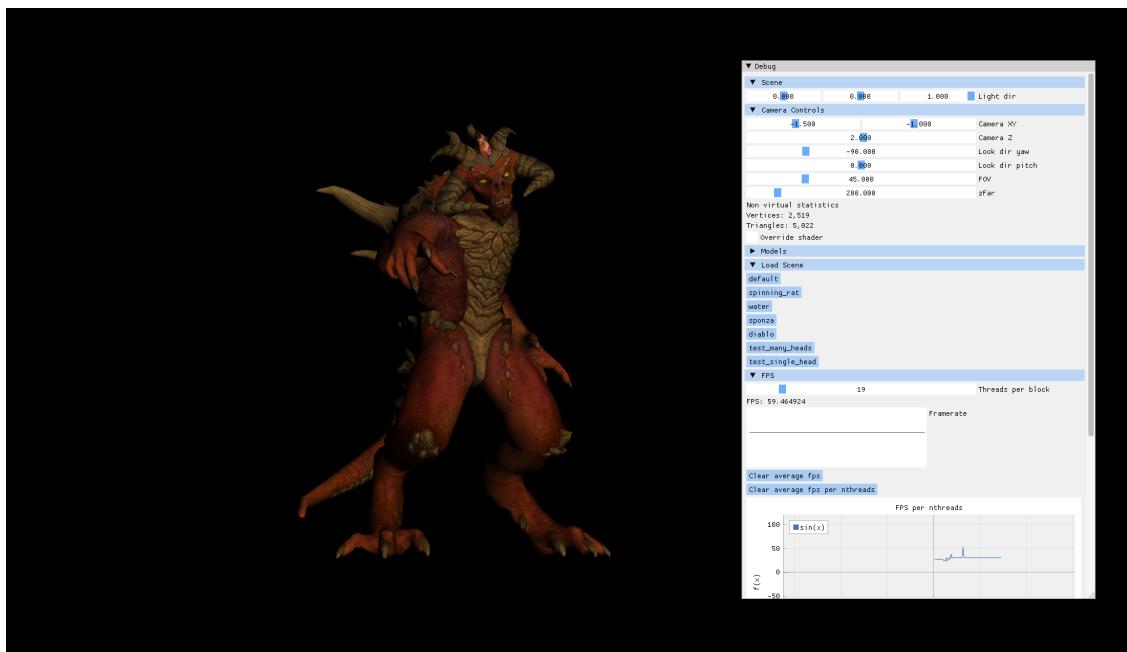


Рисунок 3.8 – Модель персонажа Diablo

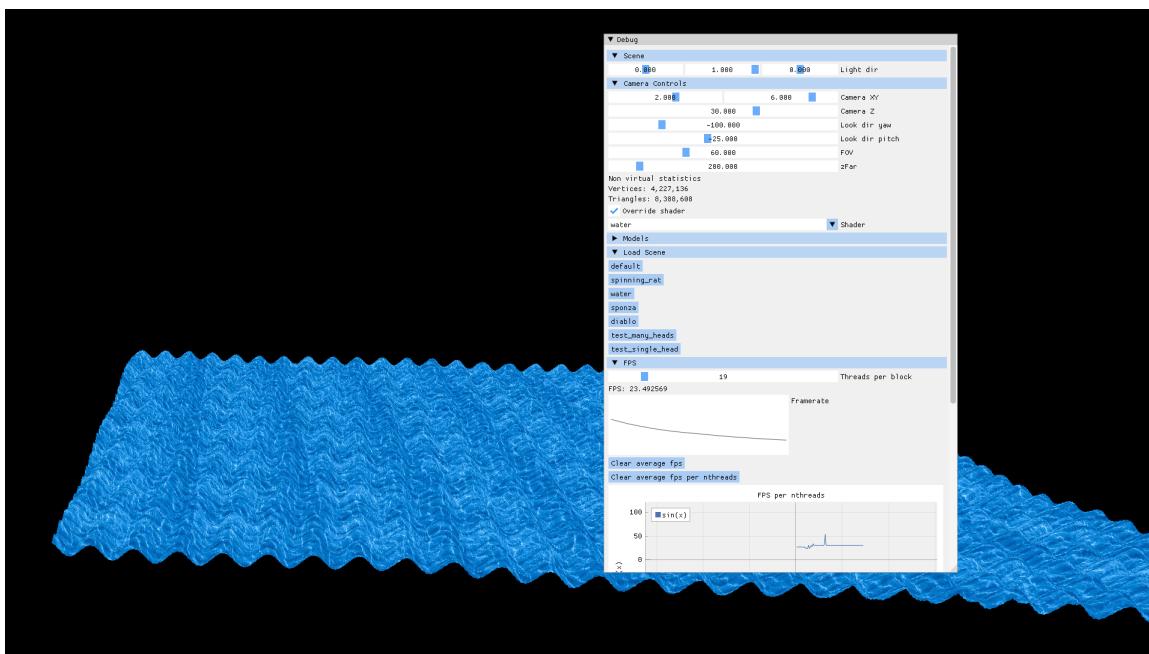


Рисунок 3.9 – Вершинный шейдер воды

Рисунок 3.9 показывает работу вершинного шейдера воды.

## **Вывод**

В этом разделе был выбран язык программирования и среда разработки, представлен код вызова отрисовки, подробно разобран интерфейс приложения и приведены результаты работы программы.

# 4 Экспериментальный раздел

В текущем разделе будет поставлена цель эксперимента, проведён сам эксперимент и сделаны соответствующие выводы.

## 4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование, следующие.

- Операционная система: Arch Linux [7] x86\_64.
- Память: 48 GiB.
- Процессор: 11th Gen Intel® Core™ i5-1135G7 @ 2.40GHz [8].
- 4 физических ядра и 8 логических ядра.

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

## 4.2 Постановка эксперимента 1

### 4.2.1 Цель эксперимента

Цель эксперимента – узнать зависимость количества времени на обработку кадра от количества потоков.

Гипотеза - чем больше потоков, тем лучше. При этом отключена всякая параллельность задач на видеокарте, разговор идет только про количество потоков запуска функции.

На рисунке 4.2 приведен график зависимости количества миллисекунд от количества потоков.

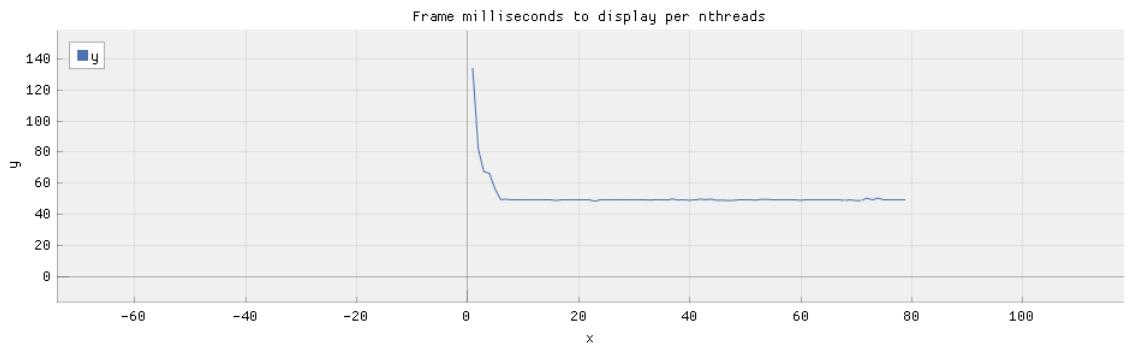


Рисунок 4.1 – Зависимость количества миллисекунд (по y) от количества потоков (по x).

## 4.3 Постановка эксперимента 1

### 4.3.1 Цель эксперимента

Цель эксперимента – узнать зависимость количества времени на обработку кадра от количества потоков.

Гипотеза - чем больше потоков, тем лучше. При этом отключена всякая параллельность задач на видеокарте, разговор идет только про количество потоков запуска функции.

На рисунке 4.2 приведен график зависимости количества миллисекунд от количества потоков.

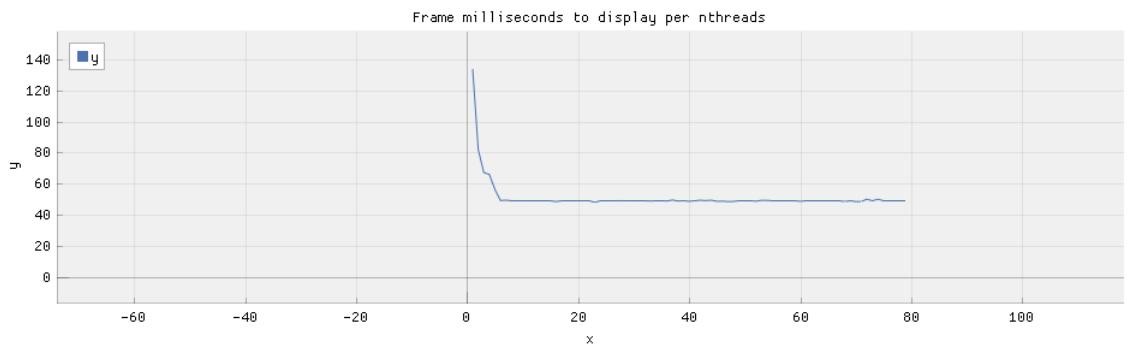


Рисунок 4.2 – Зависимость количества миллисекунд (по y) от количества потоков (по x).

## **Вывод**

В результате эксперимента было выявлено, что оптимизированный алгоритм обратной трассировки лучей на самом деле дает выигрыш во времени, так как требует меньшего количества лучей для построения сцены. Кроме того, расчитанные аналитическим способом результаты были подтверждены на практике.

# Заключение

В рамках данного курсового проекта были:

- описаны структуры трехмерной сцены, включая объекты, из которых состоит сцена;
- проанализированы и выбраны необходимые существующие алгоритмы для построения сцены;
- проанализированы и выбраны варианты оптимизации ранее выбранного алгоритма удаления невидимых линий;
- реализованы выбранные алгоритмы;
- разработано программное обеспечение, которое позволит отобразить трехмерную сцену и визуализировать удар молнии;
- проведены сравнение стандартного и реализованного оптимизированного алгоритма удаления невидимых линий.

В ходе выполнения эксперимента было установлено, что верное определение направлений лучей может значительно улучшить алгоритм. Было выявлено, что отрисовка сцены с помощью улучшенного алгоритма обратной трассировки лучей работает быстрее, чем с помощью стандартного.

# Литература

- [1] Алгоритм Z-буфера. [Электронный ресурс]. Режим доступа: <http://compgraph.tpu.ru/0glavlenie.htm> (дата обращения: 09.10.2021).
- [2] Трассировка лучей из книги Джека Проузиса [Электронный ресурс]. Режим доступа: <https://www.graphicon.ru/oldgr/courses/cg99/notes/lect12/prouzis/raytrace.htm> (дата обращения: 28.10.2021).
- [3] Алгоритм Робертса. [Электронный ресурс]. Режим доступа: <http://compgraph.tpu.ru/0glavlenie.htm> (дата обращения: 09.10.2021).
- [4] Алгоритм, использующие список приоритетов (алгоритм художника). [Электронный ресурс]. Режим доступа: <http://compgraph.tpu.ru/0glavlenie.htm> (дата обращения: 09.10.2021).
- [5] Алгоритм Варнока. [Электронный ресурс]. Режим доступа: <http://compgraph.tpu.ru/0glavlenie.htm> (дата обращения: 09.10.2021).
- [6] Модели освещения. [Электронный ресурс]. Режим доступа: <https://devburn.ru/2015/09/> (дата обращения: 09.10.2021).
- [7] Windows [Электронный ресурс]. Режим доступа: <https://www.microsoft.com/ru-ru/windows> (дата обращения: 30.09.2021).
- [8] Процессор Intel® Core™ i5-1135G7 [Электронный ресурс]. Режим доступа: <https://www.intel.ru/content/www/ru/ru/products/sku/208658/intel-core-i51135g7-processor-8m-cache-up-to-4-20-ghz-specifications.html> (дата обращения: 04.09.2021).