

Министерство науки и высшего образования Российской Федерации  
Санкт-Петербургский политехнический университет Петра Великого  
Физико-механический институт

Работа допущена к защите  
Руководитель ОП  
\_\_\_\_\_ К.Н. Козлов  
«\_\_\_\_\_» \_\_\_\_\_ 2023 г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА  
РАБОТА БАКАЛАВРА**

**ОПТИМИЗАЦИЯ КОЛИЧЕСТВА ВЫЗОВОВ ОТРИСОВКИ В  
СОВРЕМЕННЫХ ГРАФИЧЕСКИХ СИСТЕМАХ**

по направлению подготовки 01.03.02 "Прикладная математика и информатика"  
Направленность (профиль) 01.03.02\_02 "Системное программирование"

Выполнил

студент гр. 5030102/90201

В.А. Парусов

Руководитель

доцент ВШПМиВФ,

степень, звание

С.Ю. Беляев

Консультант

старший преподаватель, ВШПМиВФ

В.С. Чуканов

Консультант

по нормоконтролю

Л.А. Арефьева

Санкт-Петербург  
2023

**САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО**  
**Физико-механический институт**

**УТВЕРЖДАЮ**

**Руководитель образовательной программы «Прикладная математика и информатика»**

**К.Н. Козлов**

**«\_\_» \_\_\_\_\_ 202\_ г.**

**ЗАДАНИЕ**

**на выполнение выпускной квалификационной работы**  
студенту Парусову Владимиру Алексеевичу, гр. 5030102/90201

1. Тема работы: «Оптимизация количества вызовов отрисовки в современных графических системах»

2. Срок сдачи студентом законченной работы: июнь 2023 г.

3. Исходные данные по работе:

Инструментальные средства:

- язык программирования C++
- среда разработки Visual Studio 2022
- программная библиотека для работы с видеокартой DirectX 12
- система контроля версий git

Ключевые источники литературы:

1. Riccio C., Lilley S. Introducing the programmable vertex pulling rendering pipeline //GPU Pro. – 2013. – Т. 4. – С. 21-37

2. Park H., Han J. H. Fast rendering of large crowds using GPU //International Conference on Entertainment Computing. – Springer, Berlin, Heidelberg, 2008. – С. 197-202.

4. Содержание работы (перечень подлежащих разработке вопросов):

- 1) Введение. Обоснование актуальности
- 2) Постановка задачи
- 3) Обзор существующих решений
- 4) Предлагаемое решение
- 5) Результаты и их сравнительный анализ
- 6) Заключение

5. Дата выдачи задания 07.12.2022

Руководитель ВКР \_\_\_\_\_ С.Ю. Беляев  
(подпись)



Консультант ВКР \_\_\_\_\_ В.С. Чуканов  
(подпись)



Задание принял к исполнению

Студент \_\_\_\_\_ В.А. Парусов  
(подпись)



## **РЕФЕРАТ**

На 37 с., 36 рисунков, 1 таблицу, 0 приложений

**КЛЮЧЕВЫЕ СЛОВА:** INDIRECT DRAW, DIRECTX 12, PHYSICALLY BASED RENDERING, ORDER INDEPENDENT TRANSPARENCY.

Тема выпускной квалификационной работы: «Оптимизация количества вызовов отрисовки в современных графических системах»

Данная работа посвящена разработке и реализации архитектуры конвейера отрисовки трёхмерной компьютерной графики, снижающей нагрузку на центральный процессор, тем самым увеличивая объем ресурсов компьютера, доступных для разработчиков ПО.

В качестве основной демонстрационной сцены используется совокупность различных трёхмерных геометрических объектов, освещённых при помощи подхода, основанного на физической модели микрограней. Весь исходный код проекта написан на языке C++ с применением графической библиотеки DirectX 12. Средством для программирования шейдеров является HLSL.

Наиболее значимым результатом является система для вывода трёхмерных сцен, количество вызовов отрисовки в которой не зависит от числа выводимых объектов. В ходе работы был также разработан гибридный алгоритм отрисовки прозрачных объектов, который не требует их упорядочивания.

Предложенная архитектура пригодна для применения в современных графических приложениях и может быть развита в дальнейшем.

## ABSTRACT

37 pages, 36 figures, 1 tables, 0 appendices

**KEYWORDS:** INDIRECT DRAW, DIRECTX 12, PHYSICALLY BASED RENDERING, ORDER INDEPENDENT TRANSPARENCY.

The subject of the graduate qualification work is «Optimizing the number of draw calls on modern graphics systems».

The work aims to develop and implement graphics pipeline architecture, which will reduce usage of CPU and thereby increase amount of resources available to developers.

Main demonstration scene is a set of different geometric shapes, which uses physically-based lighting with microfacet model. All source code of project is written in C++ using the DirectX 12 graphics library. HLSL is used for shaders code.

The most significant result is a system for rendering three-dimentional scenes, the number of draw calls in which does not depend on the number of rendered objects. In the course of the work, a hybrid algorithm for rendering transparent objects was also developed, which does not require object sorting.

The proposed architecture is suitable for use in modern graphics applications and can be further developed to obtain more efficient results.

## СОДЕРЖАНИЕ

РЕФЕРАТ .....	4
ABSTRACT .....	5
СОДЕРЖАНИЕ .....	6
ВВЕДЕНИЕ.....	7
ГЛАВА 1. ПОСТАНОВКА ЗАДАЧИ .....	9
1.1. Техническое задание .....	9
1.2. Ожидаемый результат .....	9
ГЛАВА 2. ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ .....	9
2.1. Использование Geometry Instancing .....	9
2.2. Использование Programmable Vertex Pulling.....	12
ГЛАВА 3. ПРЕДЛАГАЕМОЕ РЕШЕНИЕ.....	13
3.1. Непрямая отрисовка .....	14
3.2. Общая структура .....	17
3.3. Этап Pre-pass .....	18
3.3.1. Frustum culling.....	19
3.3.2. Depth pre-pass.....	20
3.3.3. Occlusion culling.....	21
3.3.4. Построение карт теней .....	22
3.4. Этап Render pass .....	24
3.4.1. Непрозрачные объекты .....	24
3.4.2. Skybox .....	25
3.4.3. Полу-прозрачные объекты.....	26
3.5. Этап Post-process.....	30
ГЛАВА 4. Результаты и их сравнительный анализ .....	31
Заключение .....	35
Список использованных источников.....	36

## ВВЕДЕНИЕ. ОБОСНОВАНИЕ АКТУАЛЬНОСТИ

В большинстве современных компьютеров установлено и одновременно используются два вида вычислительных процессоров: центральный процессор (CPU) и графический (GPU). Из-за того что они работают одновременно, а также имеют обособленную друг от друга физическую память, возникает необходимость в синхронизации и передаче данных. Эту необходимость решает программа-драйвер, которую реализуют производители графических вычислительных процессоров, предоставляя разработчикам графических приложений специальные API, такие как OpenGL, Vulkan, DirectX 11 и тд. Однако, из-за того что заранее предугадать архитектурные особенности итоговых графических приложений невозможно, программа-драйвер может добавлять дополнительные синхронизации, что приводит к ухудшению производительности, так как центральный и графический процессоры начинают работать последовательно, как представлено на рис.0.1.

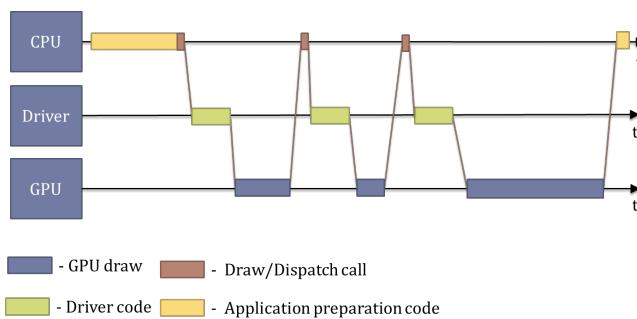


Рис.0.1. Схема работы приложения при неоптимальном драйвере DirectX 11

Во избежание подобных ситуаций, в более поздних и новых API (таких как DirectX 12 и Vulkan) задача синхронизации была снята с программы драйвера и была передана разработчику. Для этого было введено понятие “списка команд”, который разработчик мог заполнять и затем отправлять на исполнение, как показано на рис.0.2.

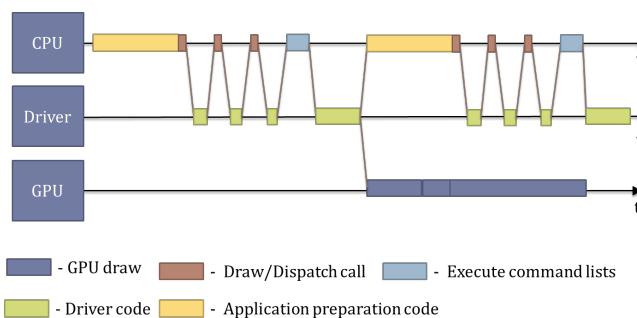


Рис.0.2. Схема работы приложения при DirectX 12

Однако несмотря на то, что данный подход даёт возможность явно управлять синхронизациями между процессорами, часть времени центрального процессора уходит на построение вышеупомянутого “списка”. Причём время построения и передачи будет зависеть от количества выводимых объектов (см. рис.0.3).

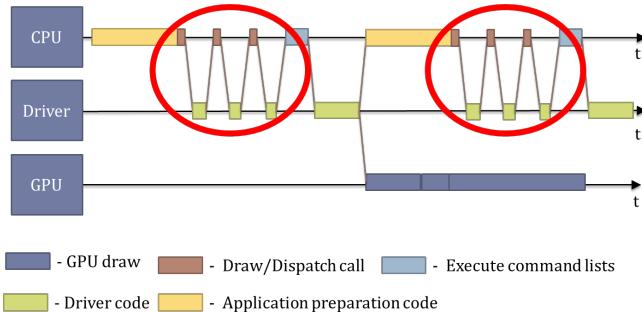


Рис.0.3. Схема работы приложения при DirectX 12, с выделенным временем на построение списка выводимых объектов

Целью работы является построение архитектуры графического приложения таким образом, чтобы время построения и передачи, а также размер “списка команд” не зависели от количества выводимых объектов (см. рис.0.4). Таким образом разработчику ПО будет предоставлено больше времени центрального процессора, а значит более трудоёмкие алгоритмы можно будет использовать для достижения желаемых результатов.

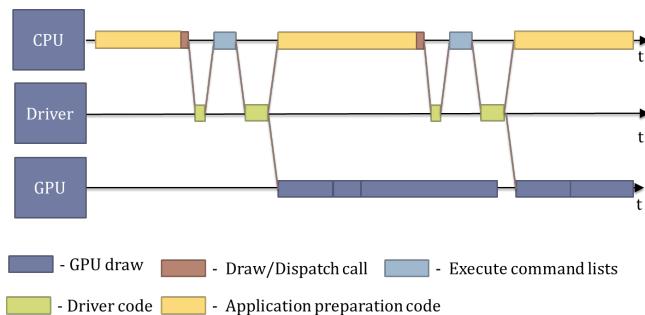


Рис.0.4. Схема работы приложения при DirectX 12 с использованием предлагаемого конвейера

## ГЛАВА 1. ПОСТАНОВКА ЗАДАЧИ

### 1.1. Техническое задание

Требуется:

- разработать архитектуру графического конвейера, использующего постоянное количество вызовов отрисовки относительно количества объектов сцены
- реализовать приложение, позволяющее использовать как разработанную архитектуру, так и традиционную

### 1.2. Ожидаемый результат

Ожидаемым результатом работы является демонстрационный проект, позволяющий сравнить производительность традиционной и разработанной архитектуры на тестовой сцене. Тестовая сцена должна содержать большое число объектов разной геометрии, положения которых изменяются на каждом кадре. Помимо этого, ожидаемым результатом работы является отчёт о производительности каждой архитектуры, и их сравнительный анализ.

## ГЛАВА 2. ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ

### 2.1. Использование Geometry Instancing

Рассмотрим традиционную архитектуру графического конвейера, на примере вывода только одного объекта. Тогда “список команд”, получающийся в результате работы конвейера, будет описывать алгоритм представленный на рис.2.1

### Algorithm

1. Установить формат хранения вершин
2. Установить формат соединения вершин
3. Установить программу-шейдер, в которой написан алгоритм обработки входных данных
4. Установить из какого буфера будут поступать вершины
5. Установить из какого буфера будут поступать номера вершин, для соединения
6. Установить из какого буфера брать текстуру
7. Установить преобразование объекта
8. Выполнить команду отрисовки

Рис.2.1. Примерный псевдокод алгоритма вывода одного объекта

Нетрудно заметить, что если повторить описанный выше алгоритм для большого числа различных объектов, то они отрисуются корректно, однако некоторые параметры выглядят излишними. Например, если используется только один формат хранения и соединения вершин, то нет необходимости выполнять эти(1 и 2 на рис.2.1) команды каждый раз. При рассмотрении крайнего случая, когда выводится один и тот же объект много раз, алгоритм будет выглядеть как показано на рис.2.2

### Algorithm

1. Установить формат хранения вершин
2. Установить формат соединения вершин
3. Установить программу-шейдер, в которой написан алгоритм обработки входных данных
4. Установить из какого буфера будут поступать вершины
5. Установить из какого буфера будут поступать номера вершин, для соединения
6. Установить из какого буфера брать текстуру
7. **for**  $\forall 1 \leq i \leq N$  **do**
8.     Установить преобразование объекта  $i$
9.     Выполнить команду отрисовки

Рис.2.2. Примерный псевдокод алгоритма вывода множества одинаковых объектов

Как можно заметить, количество команд в данном алгоритме, линейно зависит от количества объектов которые рисуются(7 на рис.2.2). Возникает желание уменьшить этот список, заменив многократные вызовы команд, одним вызовом, чтобы алгоритм выглядел как показано на рис.2.3

### **Algorithm**

1. Установить формат хранения вершин
2. Установить формат соединения вершин
3. Установить программу-шейдер, в которой написан алгоритм обработки входных данных
4. Установить из какого буфера будут поступать вершины
5. Установить из какого буфера будут поступать номера вершин, для соединения
6. Установить из какого буфера брать текстуру
7. Установить из какого буфера брать преобразования объектов
8. Выполнить команду отрисовки N раз

Рис.2.3. Примерный псевдокод алгоритма вывода множества одинаковых объектов при помощи Geometry Instancing

Именно поддержка такого алгоритма(рис.2.3) и называется Geometry Instancing[2], и его использование отрыывает большие возможности. Например в статье Park-a[13] описывается архитектура графического конвейера, позволяющего выводить большие толпы объектов, при помощи технологии Geometry Instancing (рис.2.4). Из особенностей можно отметить:

1. Все объекты имеют несколько типов геометрии, разнящиеся по уровню детализации(рис.2.5)
2. Для каждого объекта, при помощи вычислительных шейдеров, определяется уровень детализации и находится ли вообще объект в области видимости.
3. В "списке команд" находятся команды отрисовки для каждого типа геометрии, но количество раз отрисовки (для каждого типа) указывается уже исходя из работы предыдущего вычислительного шейдера.



Рис.2.4. Примеры работы приложения описанного в статье Park-a[13]

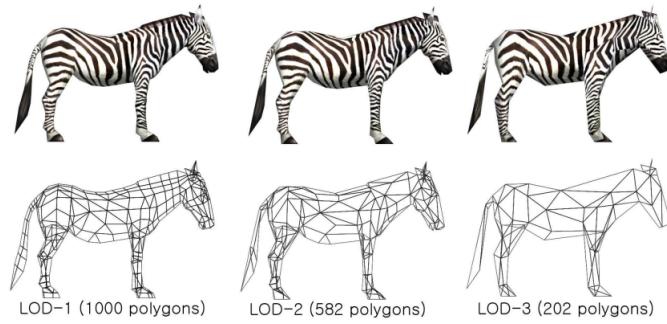


Рис.2.5. Уровни детализации, описанные в статье Park-a[13]

Основным недостатком является то, что предложенная Park-ом[13] архитектура требует добавлять в "список команд" команды отрисовки для каждого выводимого типа геометрии. В случае, если на экран выводится много разных объектов, предложенная архитектура ничем не будет отличаться от традиционной.

## 2.2. Использование Programmable Vertex Pulling

В своей статье Ricco[15] предлагает способ избавиться от основной проблемы Geometry Instancing - отрисовки объектов с одинаковой геометрией. Для этого он предлагает сделать следующее:

1. Ввести понятие “группы” треугольников - набор из  $N$  вершин (число  $N$  задаётся на центральном процессоре).
2. Все объекты, которые требуется вывести на экран, разделяются на “группы”.
3. Все вершины всех объектов складываются в один большой буфер.
4. В вычислительном шейдере для всех объектов определяется сколько “групп” соответствуют каждому объекту, и для каждой “группы” записываются индексы вершин в буфере со всеми вершинами.

5. В “списке команд” находится лишь одна команда отрисовки, которая рисует “группу” столько раз, сколько было решено на вычислительном шейдере.

Такой подход уменьшает количество вызовов команд отрисовок до одной, что является впечатляющим результатом. Однако этот подход не лишен недостатков:

1. Несмотря на уменьшение количества команд отрисовки в “списке команд”, количество вызовов отрисовки на графическом процессоре увеличивается (из-за дробления объектов на “группы”), что может негативно сказаться на производительности.
2. Число  $N$ , используемое для задания размера “группы”, должно подбираться эмпирически. Если сделать его слишком маленьким, то число вызовов отрисовки на графическом процессоре возрастет (см п.1). Если сделать его слишком большим, то будут отрисовываться “лишние” треугольники из “группы”.
3. Буфер со всеми вершинами всех объектов создаётся с точки зрения API не как буфер для вершин, а как буфер общего назначения. Это приводит к тому, что к нему не применяются оптимизации программы-драйвера (Например способ кеширования или расположение в видеопамяти).
4. Из-за возможной фрагментации видеопамяти, буфер со всеми вершинами может не поместиться в видеопамять, несмотря на то, что объем памяти достаточен для хранения объектов по отдельности.

## **ГЛАВА 3. ПРЕДЛАГАЕМОЕ РЕШЕНИЕ**

Одной из основных целей данной работы было разработать графический конвейер, максимально приближенный к существующим графическим конвейерам. Таким образом будут достигнуты две цели:

1. Эксперименты, проводимые при помощи разработанной архитектуры, будут содержать те же затратные (с точки зрения производительности) места, которые имеют современные графические системы. А значит, результаты, полученные в ходе экспериментов, будут более приближены к значениям, получаемым в реальных современных графических системах

2. Будут разработаны и реализованы алгоритмы, которые можно будет использовать другим графическим системам, если они возмут за основу предлагаемую архитектуру

### **3.1. Непрямая отрисовка**

Как было уже упомянуто ранее, основным способом отправки команд на графический процессор считается сбор “списка команд” на центральном процессоре и последующая передача получившегося “списка” на графический процессор. Однако, в современных API добавилась возможность дополнить вышеупомянутый “список команд” дополнительным массивом команд прямо из графического процессора. Эта технология называется непрямой отрисовкой, и в DirectX 12 эту технологию реализует команда ExecuteIndirect.

Для того чтобы воспользоваться командой ExecuteIndirect, ей необходимо передать следующие параметры:

1. “Сигнатуру вызова” - перечень команд, которые необходимо вызвать для каждого рисуемого объекта. Обязательно должна оканчиваться командой отрисовки.
2. Буфер из видеопамяти с параметрами (которые требует сигнатура), для каждого выводимого объекта.
3. Количество отрисовываемых объектов (число или указатель на буфер из видеопамяти).

Таким образом, простейший алгоритм отрисовки любого числа объектов будет выглядеть так: (рис.3.1)

## Algorithm

1. Загрузить все объекты
2. Создать сигнатуру
3. Создать буфер (*SRVbuffer*) и сохранить в него информацию для каждого объекта (например указатель на буфер вершин)
4. Установить формат хранения вершин
5. Установить формат соединения вершин
6. Установить программу-шейдер, для отрисовки
7. **for each frame do**
  - | ExecuteIndirect(*SRVbuffer*)**

Рис.3.1. Примерный псевдокод простейшего алгоритма использующего непрямую отрисовку

Таким образом мы добились поставленной цели, и количество команд в “списке команд” при данном подходе действительно константно относительно числа объектов. Но на практике этот алгоритм будет работать хуже, чем традиционный (см. рис.2.1). Причина в том, что традиционный подход позволяет заранее отбросить отрисовку части объектов, просто не добавив их в “список команд”, а данный (рис.3.1) алгоритм не даёт такой возможности. Однако, это можно было бы исправить, если буфер с параметрами (*SRVbuffer*) мог бы изменяться на каждом кадре работы приложения. Это приводит нас к следующему алгоритму (рис.3.2)

## Algorithm

1. Загрузить все объекты
2. Создать сигнатуру
3. Создать буфер (*SRVbuffer*) и сохранить в него информацию для каждого объекта (например указатель на буфер вершин)
4. Создать буфер (*UAVbuffer*), совпадающий размером с предыдущим,
5. Создать счётчик (*UAVcounter*)
6. Установить формат хранения вершин
7. Установить формат соединения вершин
8. **for each frame do**
  9. Установить программу-шейдер, для обработки команд
  10. Обнулить счётчик (*UAVcounter*)
  11. Запустить вычислительный шейдер, который скопирует из *SRVbuffer* в *UAVbuffer* параметры нужных команд и изменит *UAVcounter* на значение равное числу скопированных команд
  12. Установить программу-шейдер, для отрисовки
  13. ExecuteIndirect(*UAVbuffer*, *UAVcounter*)

Рис.3.2. Примерный псевдокод алгоритма использующего непрямую отрисовку с отбрасыванием команд

Отметим преимущества и недостатки описанного алгоритма(рис.3.2):

1. Преимущество: число вызовов отрисовки в “списке команд” всё ещё является константным, относительно числа объектов
2. Преимущество: число вызовов отрисовки на графическом процессоре не меняется (см. главу 2.2)
3. Преимущество: используются стандартные буфера, а значит оптимизации программы-драйвера не перестанут работать (см главу 2.2)
4. Преимущество: нет проблем с фрагментацией (см. главу 2.2)
5. Преимущество: проверки, проводимые при определении “нужных” команд (см. 11 на рис.3.2), будут происходить параллельно. А так как обычно графический процессор обладает большим числом вычислительных ядер, чем центральный процессор, предложенный алгоритм будет работать быстрее.
6. Недостаток: невозможно установить порядок, в котором объекты будут отрисовываться.

### 3.2. Общая структура

Большинство графических приложений имеют схожую структуру конвейера отрисовки, состоящую из 3-х этапов:

1. этап Pre-pass - выполнение задач, которые необходимо выполнить до рисования объектов на экран. Например: отбрасывание не видимых объектов, построение карт теней, предварительный подсчёт карты глубины.
2. этап Render pass - отрисовка всех объектов на кадр.
3. этап Post process - применение эффектов на получившийся кадр.

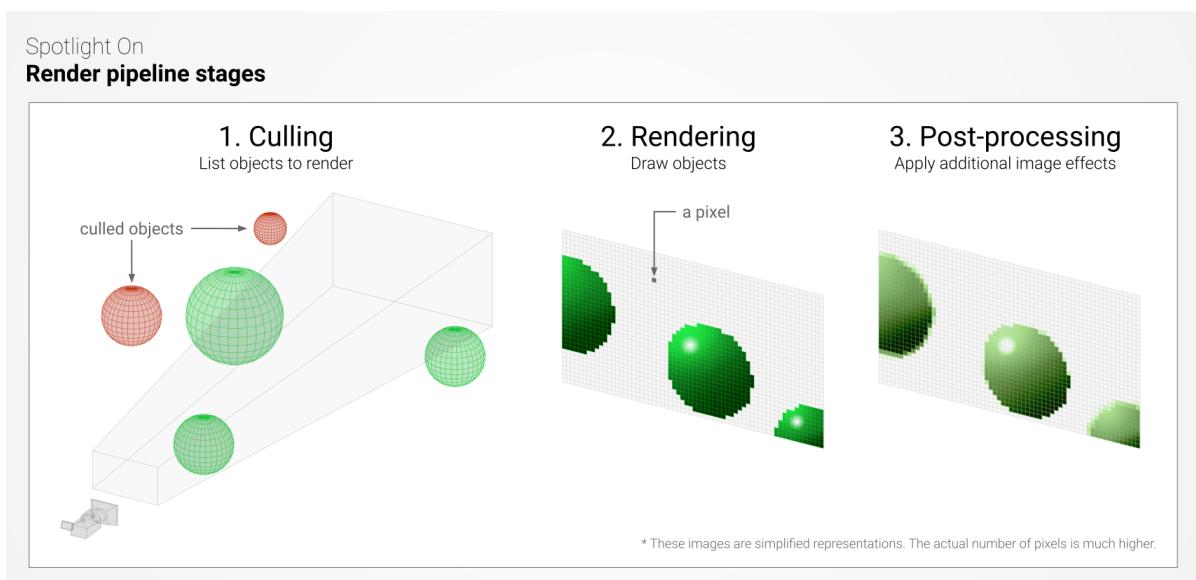


Рис.3.3. Упрощенная схема конвейера отрисовки Unity

Предлагаемый конвейер сохраняет эту структуру, несмотря на изменения в алгоритме отрисовки, и требует  $11 + S$  (где  $S$  - число источников света, генерирующих тень) вызовов отрисовки в “списке команд”, что видно по схеме конвейера на рис.3.4. Каждый этап представляет собой набор из подэтапов, где каждый подэтап решает ровно одну задачу.

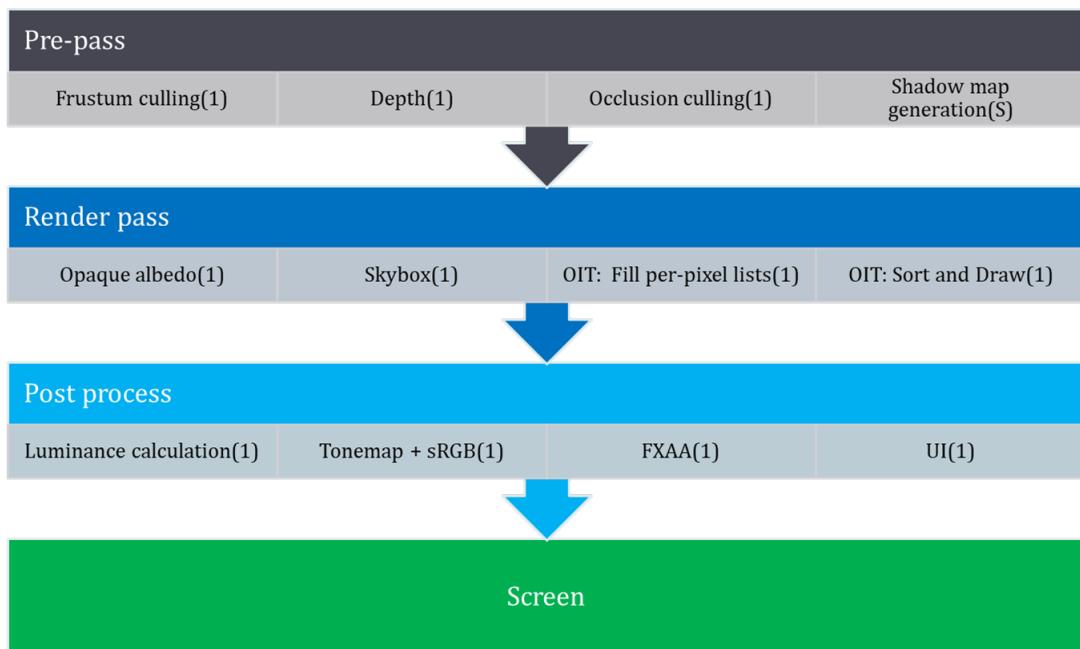


Рис.3.4. Схема предлагаемого конвейера. В скобках указано количество вызовов ExecuteIndirect.

### 3.3. Этап Pre-pass



Рис.3.5. Схема этапа Pre-pass предлагаемого конвеера.

Во время отрисовки кадра, на разных подэтапах могут требоваться команды, удовлетворяющие разным критериям. Для этого вводится несколько буферов с параметрами команд для ExecuteIndirect, каждый из которых имеет разное имя, работающее как фильтр:

1. All - буфер, в котором находятся параметры для всех объектов, присутствующих на сцене.
2. OpaqueAll - буфер, в котором находятся параметры всех непрозрачных объектов.
3. TransparentAll - буфер, в котором находятся параметры всех прозрачных объектов.
4. OpaqueFrustum - буфер, в котором находятся параметры всех непрозрачных объектов, пересекающих пирамиду видимости.
5. OpaqueCulled - буфер, в котором находятся параметры всех непрозрачных объектов, видимых пользователю.

6. TransparentCulled - буфер, в котором находятся параметры всех прозрачных объектов, видимых пользователю.

Основной задачей этапа Pre-pass является заполнение всех вышеописанных буферов (кроме All). Разберём подэтапы данного этапа:

### 3.3.1. *Frustum culling*

Данный подэтап отвечает за заполнение буферов OpaqueAll, TransparentAll и OpaqueFrustum при помощи алгоритма под названием Frustum culling[1]. Основная идея алгоритма заключается в том, чтобы для каждого объекта проверить, пересекает ли он усеченную пирамиду видимости камеры, или нет (см рис.3.6). Если объект и пирамида видимости камеры не пересекаются, то объект можно и не выводить.

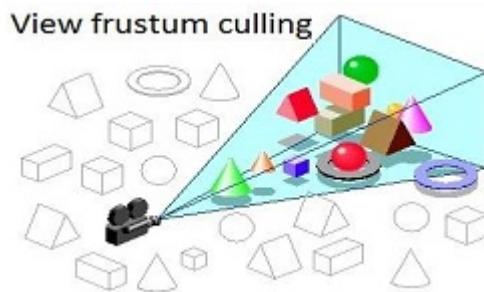
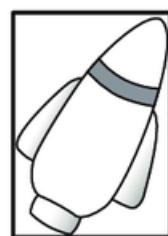


Рис.3.6. Схема работы алгоритма Frustum Culling.

Однако, зачастую, объекты представляют собой сложные невыпуклые геометрические формы, для которых считать пересечения является трудной вычислительной задачей. В связи с этим, в качестве оптимизации, для каждого объекта строится параллельный осям ограничивающий параллелепипед(см. рис.3.7), и проверяется пересечение не объектов с усечённой пирамидой видимости, а пересечение параллелепипедов и усечённой пирамиды.



AABB

Рис.3.7. Пример построения параллельного осям ограничивающего параллелепипеда в двухмерном случае.

Для того чтобы проверить, пересекаются ли параллелипипед и усеченная пирамида, достаточно задать уравнения плоскостей, содержащих грани усеченной пирамиды, и имеющие нормали, направленные “внутрь” пирамиды. Предположим, что плоскости задаются уравнениями 3.1, а вершины параллельного осям ограничивающего параллелепипеда задаются как 3.2.

$$A_i * x + B_i * y + C_i * z + D = 0, \forall 1 \leq i \leq 6 \quad (3.1)$$

$$x_j, y_j, z_j \forall 1 \leq j \leq 8 \quad (3.2)$$

Тогда выполнение условия 3.3 гарантирует то, что объект не попадёт в область видимости.

$$\exists i \forall j A_i * x_j + B_i * y_j + C_i * z_j + D < 0 \quad (3.3)$$

### 3.3.2. Depth pre-pass

Данный подэтап выполняет задачу построения иерархической карты глубины, используя объекты из буфера `OpaqueFrustum`.

Для начала, определим что такое карта глубины и что такое иерархическая карта глубины. *Картой глубины* называется монохромное изображение, где для каждого пикселя, вместо интенсивности цвета, хранится его расстояние до камеры. *Иерархической картой глубины* называется набор монохромных изображений, удовлетворяющих следующим условиям:

1. Первое изображение совпадает по размеру и содержимому с картой глубины
2. Ширина и высота каждого следующего изображения меньше ширины и высоты предыдущего в 2 раза
3. Для каждого следующего изображения, интенсивность пикселя считается как максимум из четырёх соответствующих пикселей предыдущего



Рис.3.8. Пример карты глубины.

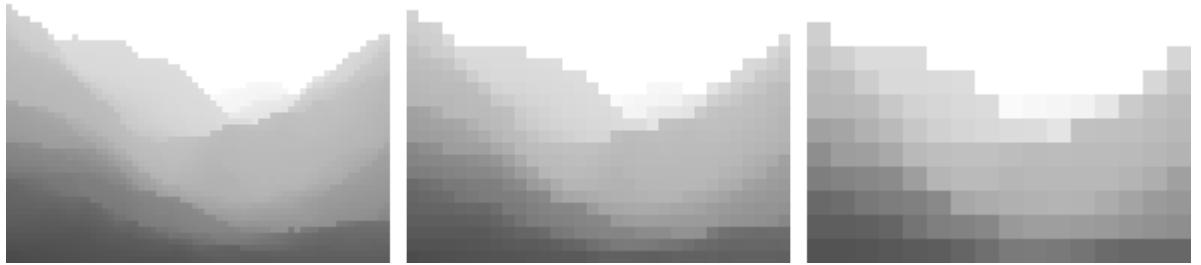


Рис.3.9. Пример иерархической карты глубины.

Построение такой карты предоставляет следующие преимущества:

1. Предварительный подсчёт карты глубины позволяет позднее высчитывать цвет пикселей только тех объектов, для которых расстояние до камеры совпадает со значением в карте глубины
2. Построенная иерархическая карты глубины позволяет применить алгоритм Occlusion culling, о котором будет сказано далее

### *3.3.3. Occlusion culling*

Данный подэтап отвечает за заполнение буферов OpaqueCulled и TransparentCulled при помощи алгоритма Occlusion culling[6]. Основная цель данного алгоритма заключается в отбрасывании тех объектов, которые оказываются перекрыты близлежащими объектами (см рис.3.10).

Для этого:

1. Каждый объект представляется в виде параллельного осям ограничивающего параллелепипеда (см. главу 3.3.1)
2. Среди вершин параллелепипеда находится ближайшая, и её расстояние до камеры сохраняется.
3. Каждая вершина параллелепипеда проецируется на экран, тем самым получаются координаты пикселей, соответствующих данной вершине.

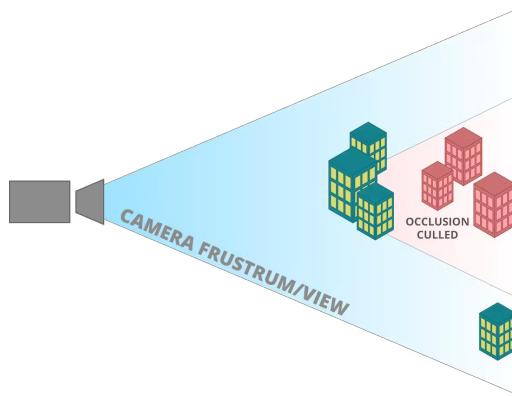


Рис.3.10. Схема работы алгоритма Occlusion culling.

4. Вокруг спроектированных вершин строится параллельный осям ограничивающий прямоугольник.
5. По размеру прямоугольника выбирается изображение из иерархической карты глубины (см. главу 3.3.2) таким образом, чтобы весь прямоугольник соответствовал одному пикселью изображения.
6. Если интенсивность в соответствующем пикселе изображения из иерархической карты глубины меньше, чем сохранённая глубина - то объект перекрывается другими объектами, и его копировать не надо.
7. Иначе, объект может быть виден, и тогда его необходимо скопировать в соответствующий буфер.

### *3.3.4. Построение карт теней*

Данный подэтап отвечает за построение карт теней, для реализации алгоритмов затенения[16]. Сами по себе карты теней очень похожи на “карту глубины” (см главу 3.3.2). Основное отличие заключается лишь в том, что в интенсивность записывается расстояние не от камеры до объекта, а от источника света до объекта.

При наличии карты теней, проверить находится ли точка в тени достаточно просто: необходимо лишь посчитать расстояние от точки до источника света, и если оно больше записанного в карте теней, то в данной точке присутствует тень.

Однако, в силу того, что изображение имеет конечное разрешение, при некотором приближении тени могут оказаться “блочными”, как показано на рис.3.11.

Чтобы избежать этого эффекта и необходимости затрачивать большое количество памяти на тени, используется алгоритм Percentage Closure Filtering[5]. Суть этого алгоритма заключается в том, чтобы сравнивать расстояние от точки до источника света не с одним пикселям, а с несколькими, рядом-лежащими



Рис.3.11. Пример "блочных" теней.

пикселями. При применении такого подхода, тени становятся “мягкими”, как показано на рис.3.12.



Рис.3.12. Пример мягких теней.

В связи с тем, что невозможно одновременно заполнить карты теней для всех источников света на сцене, этот подэтап требует отрисовки для каждого источника света, генерирующего тень.

### 3.4. Этап Render pass



Рис.3.13. Схема этапа Render pass предлагаемого конвеера.

На данном этапе, после всей подготовительной работы, поведённой в предыдущем этапе, происходит отрисовка объектов на кадр.

#### 3.4.1. Непрозрачные объекты

Данный подэтап отрисовывает все непрозрачные объекты, используя буфер OpaqueCulled, карту глубины из главы 3.3.2 и карты теней из главы 3.3.4.

Для отображения объектов с учётом их материала и расположения относительно источников света используются различные модели освещения. В предлагаемой архитектуре используется алгоритм освещения, называющийся *Physically based rendering*[14].

##### 3.4.1.1. Physically based rendering

Данный алгоритм освещения использует в своей основе физическую модель микрограней[11]. В этой физической модели поверхность любого объекта представляет собой множество идеальных зеркал, находящихся под разными углами друг к другу (см рис.3.14).

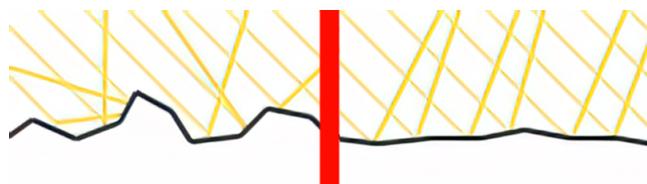


Рис.3.14. Схема физической модели микрограней. Слева - шершавая поверхность, справа - гладкая

Для понимания принципов работы алгоритма *Physically based rendering* необходимо рассмотреть “Основное уравнение рендеринга” (см. формулу 3.4), предложенное Джеймсом Каджия[8].

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega} f_r(x, \omega_i, \omega_o) L_i(x, \omega_i) (n_x * \omega_i) d\omega_i \quad (3.4)$$

Данное уравнение показывает, что интенсивность света в точке  $x$  по направлению  $\omega_o$  равна сумме излучемой интенсивности ( $L_e(x, \omega_o)$ ) и отраженной интенсивности, где последняя считается как интеграл по полусфере ( $\Omega$ ) произведения интенсивности падающего света ( $L_i(x, \omega_i)$ ), двунаправленной функции отражательной способности ( $f_r(x, \omega_i, \omega_o)$ ) и косинуса угла падения ( $(n_x * \omega_i)$ ). Двунаправленная функция отражательной способности  $f_r$  часто называется BRDF функцией.

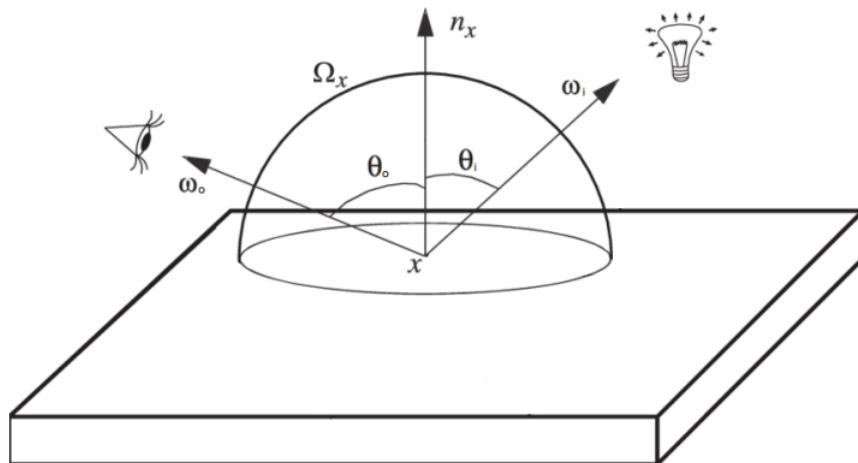


Рис.3.15. Обозначения используемые в "основном уравнении рендеринга"

В предлагаемом конвейере используется BRDF функция Кука-Торренса[3].

### 3.4.2. *Skybox*

На данном подэтапе в кадр выводится фоновое изображение, на котором изображено окружение сцены. Данный подэтап рисуется после отрисовки непрозрачных объектов, чтобы уменьшить число перерисовываемых пикселей кадра, так как фоновое изображение рисуется только в тех пикселях, где не были отрисованы объекты.

Описанное фоновое изображение представляет собой кубическую карту. *Кубической картой* называется набор из 6 изображений, снятых с 6-ти направлений и расположенных в развёртке куба, как показано на рис.3.16.

Таким образом, при необходимости вывести цвет в пикселе, в котором не были отрисованы объекты, достаточно будет посчитать направление, в котором этот пиксель находится и взять точку, соответствующую точке на кубе, в указаном направлении (см. рис.3.17).

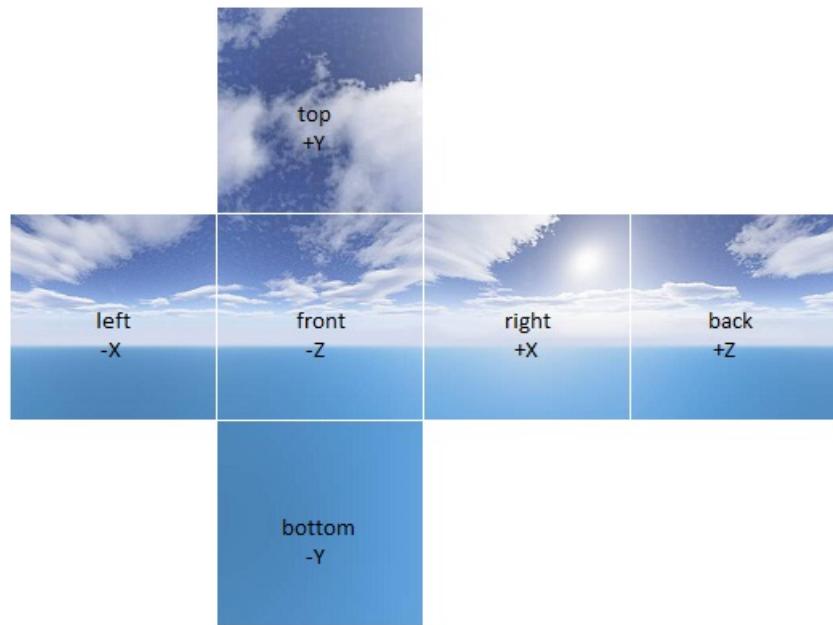


Рис.3.16. Пример кубической карты окружения

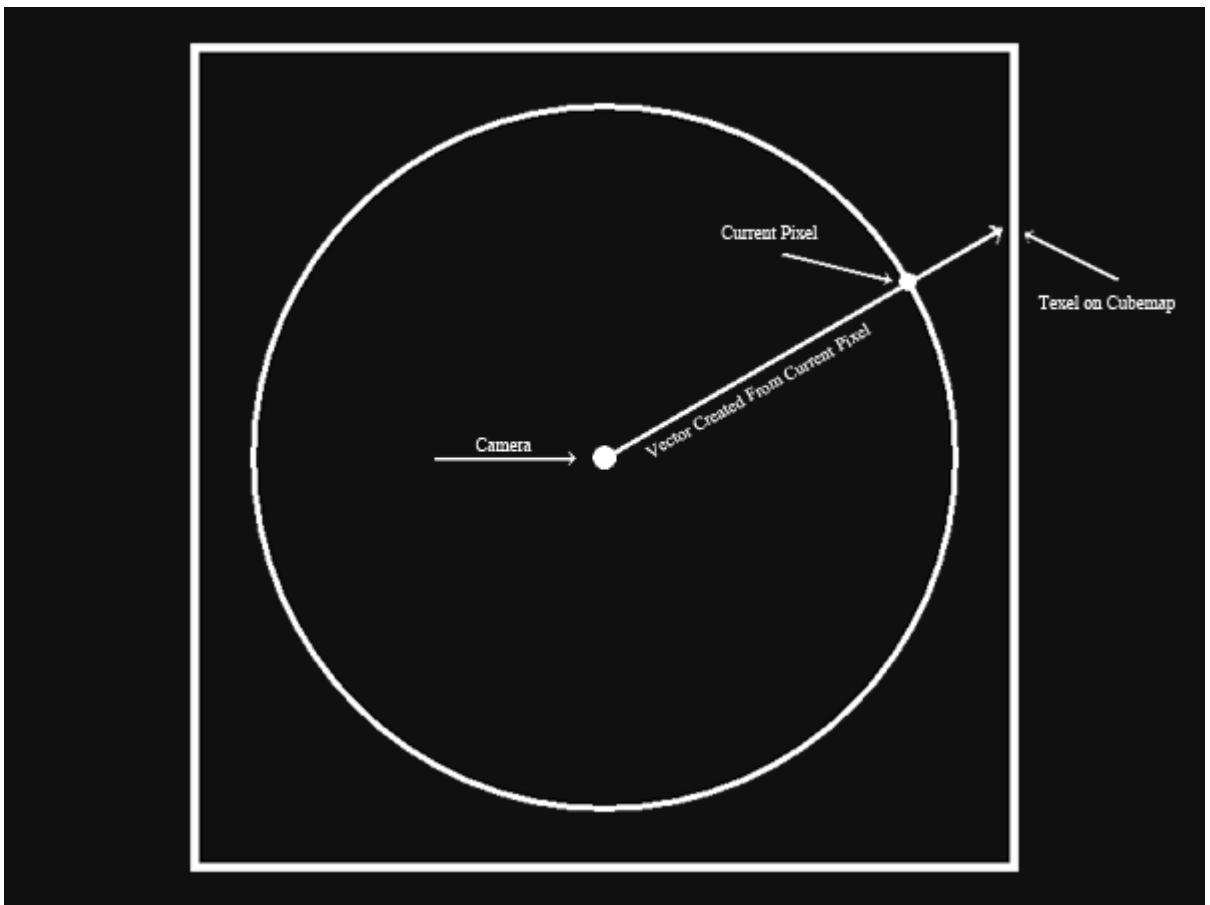


Рис.3.17. Схема работы вычисления цвета пикселя с использованием кубической карты

### 3.4.3. Полупрозрачные объекты

На данном подэтапе в кадр выводятся полупрозрачные объекты при помощи буфера TransparentCulled. Как понятно из названия, полу-прозрачные объекты отличаются от непрозрачных тем, что через них можно видеть объекты наход-

дящиеся позади. Из-за этого нельзя воспользоваться картой глубины (см. главу 3.3.2) для отрисовки только ближайшего пикселя, что может привести к ситуации, приведённой на рис.3.18.



Рис.3.18. Пример отрисовки прозрачных объектов в неправильном порядке

Чтобы избежать изображенной ситуации, необходимо производить сортировку объектов. Тогда, если выводить объекты начиная с самого дальнего, то цвет в пикселе можно вычислять по формуле 3.5.

$$C_{new} = \alpha * C_{transparent} + (1 - \alpha) * C_{pixel} \quad (3.5)$$

Где:

1.  $C_{new}$  - новый цвет пикселя
2.  $C_{transparent}$  - цвет полученный в результате применения алгоритма освещения, для данного пикселя
3.  $C_{pixel}$  - цвет, хранящийся в данном пикселе.
4.  $\alpha$  - коэффициент прозрачности. Значение 0 соответствует абсолютно прозрачному объекту, а значение 1 соответствует абсолютно непрозрачному.

Однако предлагаемый алгоритм неявной отрисовки (см. главу 3.1) не позволяет установить порядок отрисовки объектов. Из-за этого необходимо использовать особые алгоритмы отрисовки прозрачных объектов.

#### *3.4.3.1. Order Independent Transparency*

Первым из рассмотренных алгоритмов является алгоритм Order Independent Transparency with per-pixel linked lists [12]. В данном алгоритме для каждого пикселя экрана заводится список, в каждом элементе которого хранится: цвет, глубина и коэффициент прозрачности. Далее алгоритм работает в 2 запуска отрисовки:

1. Отрисовываются все полупрозрачные объекты, но результат отрисовки объекта записывается не в кадр, а добавляется в конец созданных списков

2. На экран отрисовывается прямоугольник, покрывающий весь экран. Для каждого пикселя прямоугольника берётся соответствующий ему список и значения в этом списке сортируются по глубине. Далее, используя отсортированный список, последовательно применим формулу 3.5 и получим формулу для итогового цвета 3.6

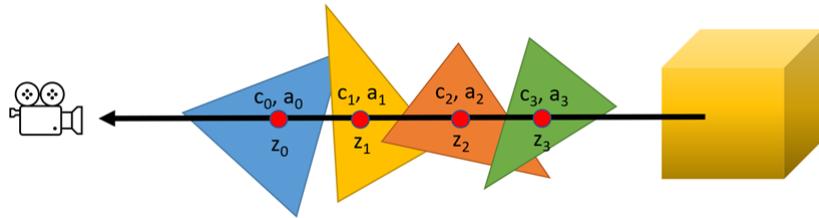


Рис.3.19. Изображение демонстрирующее первый этап работы алгоритма

$$C_{out} = C_1 \alpha_1 + \sum_{i=2}^N (C_i \alpha_i \prod_{j=1}^{i-1} (1 - \alpha_j)) + C_{opaque} \prod_{j=1}^N (1 - \alpha_j) \quad (3.6)$$

Где:

1.  $N$  - размер списка для данного пикселя
2.  $C_{out}$  - результирующий цвет пикселя
3.  $C_{opaque}$  - цвет непрозрачного объекта, полученный из пикселя кадра.
4.  $C_i$  - цвет, полученный из элемента отсортированного списка с номером  $i$ .
5.  $\alpha_i$  - коэффициент прозрачности, полученный из элемента отсортированного списка с номером  $i$ .

Нетрудно заметить, что благодаря переносу сортировки на второй этап, появляется возможность отрисовывать объекты на первом этапе в любом порядке. Однако время, требуемое на выполнение вышеописанной сортировки, сильно сказывается на производительности. Авторами статьи упоминается, что количество отрисовываемых кадров в секунду при использовании данного подхода, падает с 110 вплоть до 5. Это означает, что запуск отрисовки с сортировкой может занимать до 190 миллисекунд, что неприемлемо для современных приложений реального времени.

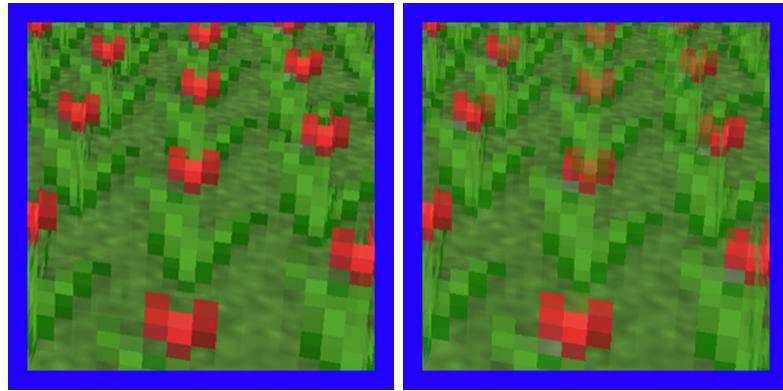
#### 3.4.3.2. Weighted Blended Order Independent Transparency

В 2013 году, в качестве улучшения предыдущего алгоритма, был представлен алгоритм Weighted Blended Order Independent Transparency[10]. Данный алгоритм

повторяет идею предыдущего алгоритма, однако, вместо сортировки и применения формулы 3.6, предлагаются использовать её аппроксимацию 3.7

$$C_{out} = \frac{\sum_{i=1}^N C_i}{\sum_{i=1}^N \alpha_i} \left(1 - \prod_{i=1}^N \alpha_i\right) + C_{opaque} \prod_{j=1}^N \left(1 - \alpha_j\right) \quad (3.7)$$

Очевидно, что данная аппроксимация работает гораздо быстрее подхода, описанного в оригинальном алгоритме. Однако, при коэффициентах  $\alpha_i$  близких к единице, погрешности аппроксимации становятся явно заметны человеческому глазу. Визуальное сравнение можно увидеть на рис.3.20.



a) Изображение полученное алгоритмом OIT.	b) Изображение полученное алгоритмом WBOIT.
Время построения кадра: 6.2ms	Время построения кадра: 1.9ms

Рис.3.20. Сравнение работы алгоритмов OIT и WBOIT

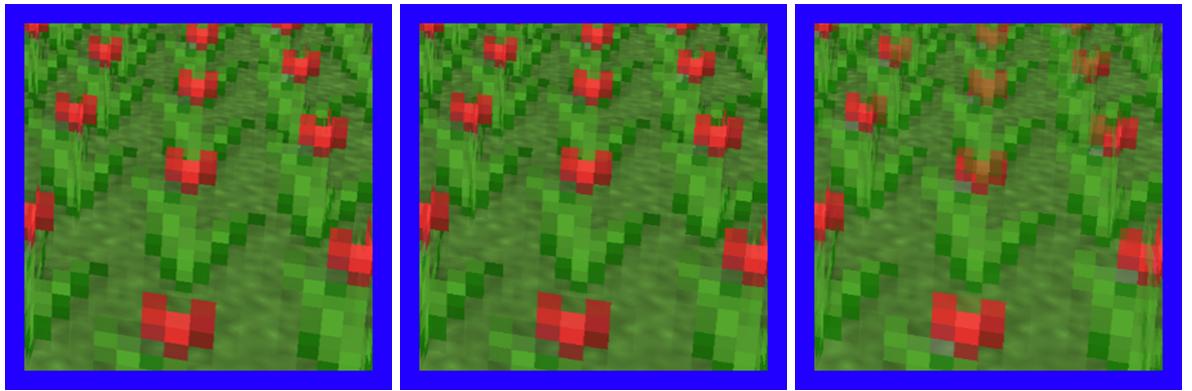
### 3.4.3.3. Hybrid Order Independent Transparency

В результате изучения описанных алгоритмов, было решено разработать новый, гибридный алгоритм, совмещающий в себе преимущества описанных алгоритмов. Как и предыдущие алгоритмы, он состоит из двух запусков отрисовки:

1. Как и в предыдущих алгоритмах, все полупрозрачные объекты отрисовываются в списки, созданные для каждого пикселя.
2. В списках для каждого пикселя, вместо сортировки всего списка, находятся  $K$  наименьших по параметру глубины, и они сортируются между собой. Затем результирующий цвет вычисляется по формуле 3.8

$$\begin{aligned}
 C_{out} = C_1 \alpha_1 + \sum_{i=2}^K (C_i \alpha_i \prod_{j=1}^{i-1} (1 - \alpha_j)) + \\
 \frac{\sum_{i=K+1}^N C_i}{\sum_{i=K+1}^N \alpha_i} (1 - \prod_{i=K+1}^N \alpha_i) \prod_{i=1}^K (1 - \alpha_i) + \\
 C_{opaque} \prod_{j=1}^N (1 - \alpha_j)
 \end{aligned} \tag{3.8}$$

Как можно заметить, данный алгоритм смешивает  $N - K$  элементов списка по формуле 3.7, а затем, считая получившийся цвет, как элемент списка с номером  $K + 1$ , применяет формулу 3.6, считая что список состоит из  $K + 1$  элемента. Визуальное сравнение всех трех методов можно увидеть на рис.3.21.



а) Изображение полученное алгоритмом OIT.  Время построения кадра: 6.2ms	б) Изображение полученное алгоритмом Hybrid OIT.  Время построения кадра: 2.1ms	в) Изображение полученное алгоритмом WBOIT.  Время построения кадра: 1.9ms
--	---	--

Рис.3.21. Сравнение работы алгоритмов OIT, WBOIT и Hybrid OIT

### 3.5. Этап Post-process

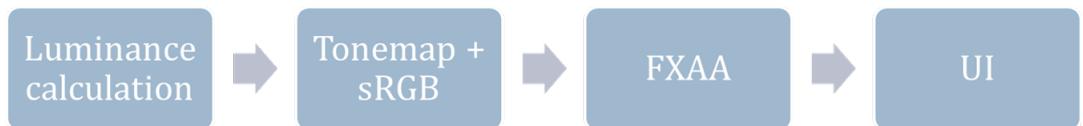


Рис.3.22. Схема этапа Post process предлагаемого конвеера.

На данном этапе происходит пост-обработка кадра, получившегося на предыдущем этапе. В качестве пост-обработки используются:

1. Алгоритм Filmic Tonemapping[4; 7]
2. Алгоритм Fast Approximate Anti-Aliasing[9]

Данный этап не претерпевает каких-либо изменений при использовании предлагаемого алгоритма непрямой отрисовки, потому что на данном этапе не используются объекты сцены.

## ГЛАВА 4. РЕЗУЛЬТАТЫ И ИХ СРАВНИТЕЛЬНЫЙ АНАЛИЗ

Для измерения производительности предлагаемого конвейера, было разработано приложение, поддерживающее динамическое переключение конвейера вывода. При помощи данного приложения была создана тестовая сцена, на которой были проведены эксперименты. Данная сцена представляет собой помещение, в котором находится аквариум и единственный источник света (генерирующий тень), находящийся над аквариумом. В аквариуме находятся рыбы, каждая из которых имеет уникальные геометрию и материал. С течением времени, количество рыб в аквариуме увеличивается. Примеры работы представлены на рис.4.1 и рис.4.2.



Рис.4.1. Тестовая сцена. Количество рыб: 10

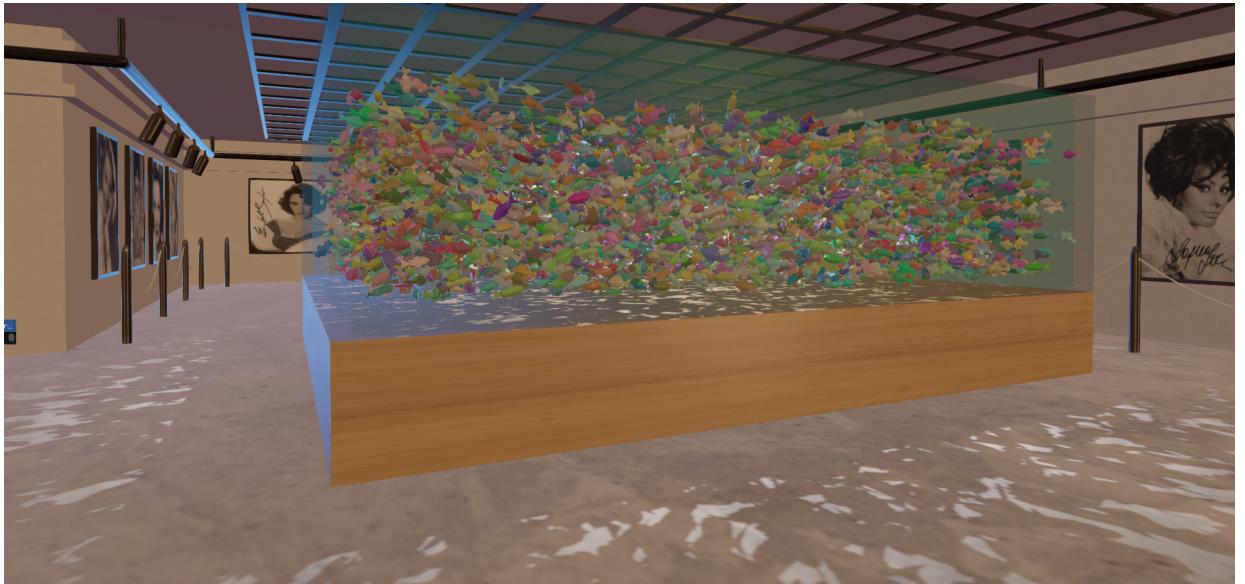


Рис.4.2. Тестовая сцена. Количество рыб: 8010

На данной сцене производились следующие измерения:

1. Зависимости количества кадров в секунду (FPS) от количества рыб
2. Зависимости времени обработки кадра на графическом процессоре (GPU Time) от количества рыб
3. Зависимости времени обработки кадра на центральном процессоре (CPU Time) от количества рыб

Результаты представлены на графиках рис.4.3, рис.4.4 и рис.4.5 соответственно. Значения “indirect” и соответствуют предлагаемой архитектуре, а значения “direct” соответствуют традиционной архитектуре. Эксперименты проводились на компьютере, с характеристиками представленными в таблице 4.1.

Таблица 4.1

Характеристики компьютера, на котором проводилось тестирование

CPU	Intel Core i5-9600K
GPU	nVidia RTX 2070 SUPER
RAM	32 Gb

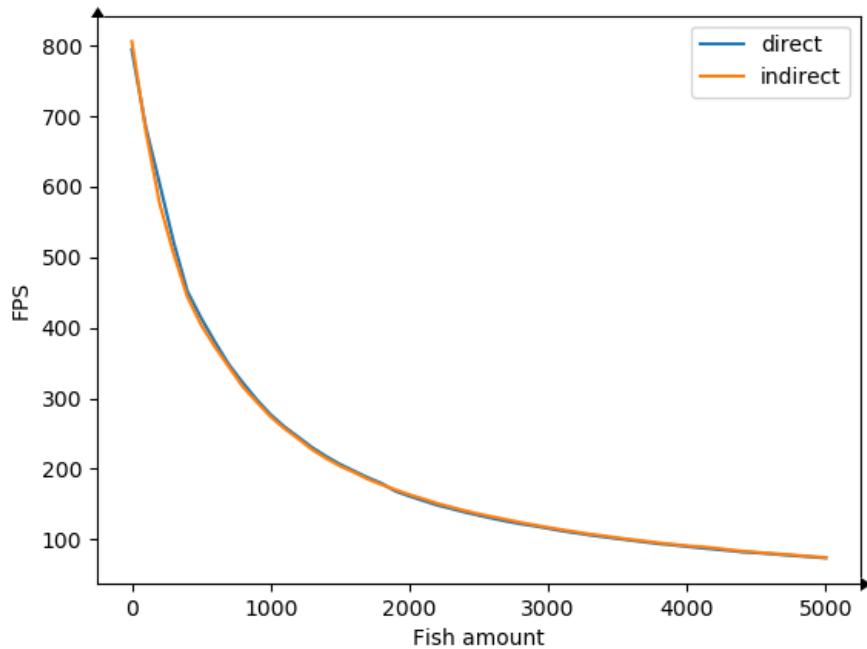


Рис.4.3. График зависимости количества кадров в секунду (FPS) от количества рыб.

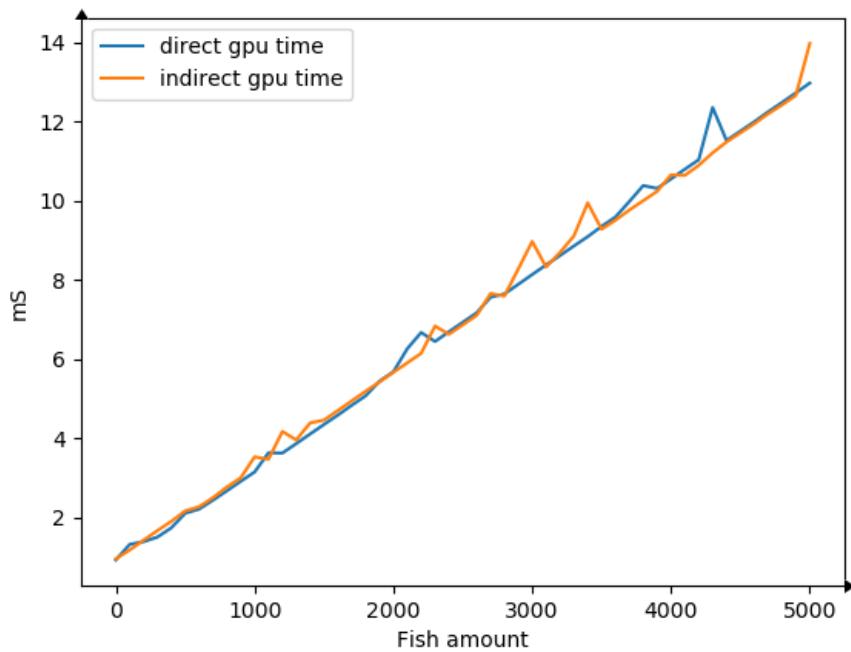


Рис.4.4. График зависимости времени обработки кадра на GPU (в миллисекундах) от количества рыб.

Как можно заметить по графикам рис.4.3 и рис.4.4, предлагаемая архитектура не вносит видимых изменений во время работы графического процессора. Однако, исходя из данных представленных на графике рис.4.5 можно заметить, что предлагаемая архитектура требует гораздо меньше времени центрального про-

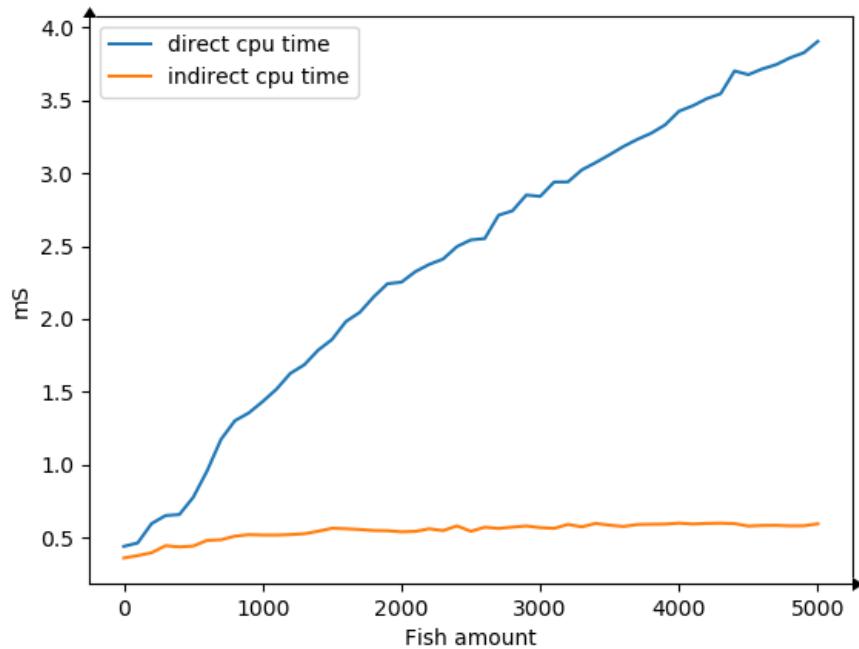


Рис.4.5. График зависимости времени обработки кадра на CPU (в миллисекундах) от количества рыб.

цессора для построения кадра, чем традиционная. Таким образом, получившиеся экспериментальные результаты соответствуют теоретическим.

## ЗАКЛЮЧЕНИЕ

В рамках проведённого исследования были выполнены все поставленные задачи. Были изучены алгоритмы, используемые в разработанной архитектуре. Помимо предлагаемой архитектуры был также разработан гибридный алгоритм для вывода полу-прозрачных объектов. Был реализован демонстрационный проект, использующий изученные современные алгоритмы компьютерной графики, и поддерживающий динамическую смену архитектуры конвейера вывода.

С использованием демонстрационного проекта был проведён эксперимент, в ходе которого были показаны преимущества предлагаемой архитектуры перед традиционной. Визуальные результаты работы демонстрационного проекта были приняты качественными.

Таким образом, благодаря сохранению основной структуры, предложенная архитектура может быть использована в современных графических системах, тем самым выделяя больше времени центрального процессора на выполнение различных алгоритмов, повышающих уровень интерактивности (например обработка физики твёрдых тел, или поведения искусственного интеллекта).

В качестве дальнейшей работы наиболее интересными представляются следующие два направления:

Первое - отрисовка полупрозрачных объектов. Несмотря на то, что гибридный алгоритм работает достаточно быстро, для его работы требуется создавать списки для каждого пикселя, что может занимать много видеопамяти. В связи с этим, стоит изучить (и возможно разработать) другие алгоритмы для отрисовки полу-прозрачных объектов, не требующие вывода в определённом порядке.

Второе - дальний перенос задач центрального процессора на графический. Благодаря предлагаемому конвейеру, с центрального процессора была снята задача отбрасывания невидимых объектов, однако существуют и другие задачи, которые можно было бы выполнять на графическом процессоре. Например, обработка положений объектов при использовании скелетной анимации.

Несмотря на всё вышеперечисленное, предлагаемая архитектура позволяет выводить сложные, высоконагруженные сцены с высоким уровнем реализма и высокой производительностью.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Assarsson U., Moller T. Optimized view frustum culling algorithms for bounding boxes // Journal of graphics tools. — 2000. — Т. 5, № 1. — С. 9—22.
2. Carucci F. Inside geometry instancing // GPU Gems. — 2005. — Т. 2. — С. 47—67.
3. Cook R. L., Torrance K. E. A reflectance model for computer graphics // ACM Transactions on Graphics (ToG). — 1982. — Т. 1, № 1. — С. 7—24.
4. Dille S., Fuhrmann A., Fischer G. Real-time tone mapping. — .
5. Fernando R. Percentage-closer soft shadows // ACM SIGGRAPH 2005 Sketches. — 2005. — 35—es.
6. Greene N., Kass M., Miller G. Hierarchical Z-buffer visibility // Proceedings of the 20th annual conference on Computer graphics and interactive techniques. — 1993. — С. 231—238.
7. Hable J. Uncharted 2: HDR lighting // Game Developers Conference. — 2010. — С. 56.
8. Kajiya J. T. The rendering equation // Proceedings of the 13th annual conference on Computer graphics and interactive techniques. — 1986. — С. 143—150.
9. Lottes T. Fast approximate anti-aliasing (FXAA) // NVIDIA Corporation, Santa Clara, CA, USA, Feb. — 2009.
10. McGuire M., Bavoil L. Weighted blended order-independent transparency // Journal of Computer Graphics Techniques. — 2013. — Т. 2, № 4.
11. Microfacet models for refraction through rough surfaces / B. Walter [и др.] // Proceedings of the 18th Eurographics conference on Rendering Techniques. — 2007. — С. 195—206.
12. Order independent transparency with per-pixel linked lists / P. Barta [и др.] // Budapest University of Technology and Economics. — 2011. — Т. 3.
13. Park H., Han J. Fast rendering of large crowds using GPU // Entertainment Computing-ICEC 2008: 7th International Conference, Pittsburgh, PA, USA, September 25-27, 2008. Proceedings 7. — Springer. 2009. — С. 197—202.
14. Pharr M., Jakob W., Humphreys G. Physically based rendering: From theory to implementation. — Morgan Kaufmann, 2016.
15. Riccio C., Lilley S. Introducing the programmable vertex pulling rendering pipeline // GPU Pro. — 2013. — Т. 4. — С. 21—37.

16. *Williams L.* Casting curved shadows on curved surfaces // Proceedings of the 5th annual conference on Computer graphics and interactive techniques. — 1978. — C. 270—274.