

Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт
(национальный исследовательский университет)»
Физтех-школа Прикладной Математики и Информатики
Центр обучения проектированию и разработке игр

Направление подготовки / специальность: 09.03.01 Информатика и вычислительная техника
Направленность (профиль) подготовки: Компьютерные науки и инженерия

ИМПЛЕМЕНТАЦИЯ И ПРИМЕНЕНИЕ КАДРОВЫХ ГРАФОВ В РЕНДЕРИНГЕ РЕАЛЬНОГО ВРЕМЕНИ

(бакалаврская работа)

Студент:

Мочалов Никита Андреевич

(подпись студента)

Научный руководитель:

Лесовой Алексей Игоревич

(подпись научного руководителя)

Москва 2024

Аннотация

Данная работа посвящена разработке кадрового графа и исследованию его связи со слоем абстракции над графическими API (Render Hardware Interface). Было предложено внедрить данный механизм непосредственно в слой абстракции. В результате работы была написана эффективная имплементация кадрового графа с использованием современного графического API Vulkan. Кроме того, был спроектирован и реализован высокуюровневый рендерер в качестве доказательства практической применимости выбранного дизайна.

Содержание

1 Введение	4
1.1 Обзор существующих решений	5
1.2 Цели работы	6
2 Имплементация кадрового графа	7
2.1 Описание программного интерфейса	7
2.1.1 Внедрение в Render Hardware Interface	8
2.1.2 Декларация кадрового графа	8
2.2 Алгоритм компиляции	10
2.2.1 Топологическая сортировка	10
2.2.2 Отсечение ребер	11
2.2.3 Синхронизация	13
2.3 Отладочная информация	14
2.4 Структура кадра	15
3 Применение в высокоуровневом рендеринге	17
3.1 Общий дизайн	17
3.2 Отрисовка геометрии	18
3.3 Кластерное отсечение света	19
3.4 Симуляция и отрисовка частиц	21
3.5 Эффекты постобработки	22
3.5.1 Bloom	22
3.5.2 Tone mapping	24
4 Результаты	25
5 Заключение	27

1. Введение

Рендеринг реального времени значительно изменился за последние несколько десятилетий благодаря достижениям в аппаратных возможностях и программных решениях. Отрисовка современных видеоигр включает большой объем вычислений, который еще недавно казался недостижимыми в интерактивных приложениях. С ростом сложности рендеринга и с появлением современных графических интерфейсов, таких как Vulkan, DirectX 12 и Metal, возникла необходимость в новых архитектурных решениях для декомпозиции рендеринга. Одним из таких решений является кадровый граф.

Кадровый граф представляет собой способ организации работы на GPU, представляя собой ориентированный ациклический граф, в котором вершины — это вычислительные задачи, а ребра между ними задают зависимости в порядке исполнения. Декомпозиция рендеринга на отдельные единицы исполнения упрощает процесс разработки новых графических эффектов, а полный граф дает глобальное представление о процессе отрисовки одного кадра (или его части), что особенно полезно при анализе производительности и отладке приложений. Важно отметить, что кадровые графы служат не только архитектурным решением для структурирования рендеринга, но и мощным инструментом оптимизации. Обладая полной картиной кадра до момента исполнения, можно эффективно использовать современные функции графических API, такие как, например, использование нескольких командных очередей [1] и многопоточная запись командных буферов [2]. Это позволяет автоматизировать и упростить процесс рендеринга, при этом повышая производительность приложений. Именно поэтому внедрение кадровых графов в процесс рендеринга является важным шагом в развитии технологий реального времени, обеспечивая более гибкие и мощные инструменты для разработчиков.

В данной работе рассматривается связь между кадровыми графиками и Render Hardware Interface (далее сокращенно RHI), представляющим собой слой абстракции, который инкапсулирует низкоуровневые детали взаимодействия с различными графическими API. Его основная задача - предоставить унифицированный программный интерфейс для отправки команд рендеринга на GPU, независимо от используемого графического API и/или аппаратной конфигурации. Благодаря абстракции низкоуровневых деталей, движок может быть легко портирован на различные платформы, требуя лишь реализации RHI для новой целевой платформы. Использование RHI также означает, что достаточно разработать всего один высокоуровневый рендерер. Поскольку современные игровые движки должны предоставлять

немалое количество графических эффектов, без RHI разработчики были бы не в состоянии быстро реализовывать сразу несколько версий этих эффектов под каждый интерфейс.

1.1. Обзор существующих решений

Первым широко известным описанием кадровых графов был доклад Юрия О’Доннелла об архитектуре рендеринга в проприетарном игровом движке Frostbite [3] в 2017 году. Их кадровый граф был разработан для решения проблем комплексности и взаимосвязанности в предыдущей архитектуре рендерера, а также в целях сокращения утилизации видеопамяти, с помощью переиспользования памяти для ресурсов, не использующихся одновременно, что особенно критично для консолей. Кадровый граф в Frostbite полностью задавался программно, а задачи связывались с помощью глобального типизированного хранилища – blackboard. Важно также отметить, что кадровый граф был глобальным и единственным на кадр. Представленное в этом докладе решение положило основу современным имплементациям кадровых графов.

В игровом движке Unreal Engine 5 также присутствует кадровый график, который называется Render Dependency Graph [4]. Данная имплементация очень похожа на решение в Frostbite, однако есть и несколько отличий – возможность создания нескольких графов и возможность автоматической декларации использования ресурсов в шейдерах при помощи препроцессинга полей структур в исходном коде.

Стремление к упрощению и более неявной декларации кадровых графов послужило главной мотивацией библиотеки, разработанной компанией AMD [5]. Ключевым отличием от других решений является то, что с данной библиотекой график создается внутри собственного шейдерного языка – RPSL, а потом внутри кода программы можно подключать функции к заданным в данных вершинам. Однако важно уточнить, что библиотека поддерживает разные фронтенды, и программная декларация все еще возможна при работе с данным решением.

	Frostbite	Unreal Engine 5	AMD RPS
Immediate или Retained Mode	Immediate	Immediate	Retained
Тип декларации	Программная	Программная	В данных или программная
Возможное количество графов	Один	Несколько	Несколько
Связь с RHI	Поверх	Поверх	Поверх

Таблица 1: Сравнение рассмотренных решений

1.2. Цели работы

При анализе существующих решений было обнаружено, что все они реализуют кадровые графы поверх RHI. В целом, такой подход логичен с точки зрения распределения человеческих ресурсов. Необходимо написать несколько имплементаций RHI (под каждый необходимый графический API), а после этого одну имплементацию кадрового графа, используя интерфейс RHI. В данной работе представлено альтернативное решение – сделать кадровые графы непосредственно частью RHI. Такое решение позволит утилизировать напрямую уникальные возможности каждого графического API и потенциально избавит от необходимости пытаться обобщить дизайн работы с командами, то есть абстракции над командными буферами, очередями и примитивами синхронизации. Таким образом целями данной работы являются:

1. Проектирование дизайна кадрового графа, встроенного напрямую в RHI.
2. Имплементация данного дизайна, используя современный графический API – Vulkan.
3. Внедрение и реализация высокогоуровневого рендерера для подтверждения валидности дизайна.

2. Имплементация кадрового графа

Концептуально работу с кадровыми графиками можно разбить на три этапа - *декларация, компиляция и исполнение*.

На этапе декларации разработчик определяет структуру графа, описывая узлы и зависимости между ними. Важно на этом этапе чётко определить зависимости между задачами, чтобы гарантировать правильную последовательность их выполнения.

Этап декларации — это этап, на котором определяется структура кадрового графа. В ходе этого этапа разработчики декларируют задачи, ресурсы и их зависимости. Это включает в себя указание, какие ресурсы читаются или записываются каждой задачей рендеринга. Четко определяя эти зависимости, кадровый график гарантирует, что ресурсы не будут одновременно использоваться конфликтующими способами, что могло бы привести к артефактам рендеринга или сбоям. Кроме того, этап декларации помогает организовать конвейер рендеринга модульным образом, что облегчает его управление и расширение.

После того как кадровый график задекларирован, он переходит на этап компиляции. На этом этапе кадровый график анализируется для определения оптимального порядка выполнения задач. Это включает в себя разрешение зависимостей, чтобы каждая задача выполнялась только после того, как все ее входные ресурсы были созданы предыдущими задачами. Этап компиляции также включает выделение ресурсов, где система решает, как эффективно использовать доступную память. Это может включать повторное использование ресурсов, где это возможно, для минимизации использования памяти.

Заключительный этап — это исполнение, на котором скомпилированный кадровый график используется для управления процессом рендеринга. В ходе исполнения задачи выполняются в порядке, определенном на этапе компиляции, обеспечивая соблюдение всех зависимостей. Этот этап включает в себя выдачу соответствующих команд отрисовки и вычислительных задач на GPU, управление выставлением ресурсов и обеспечение синхронизации между различными задачами.

2.1. Описание программного интерфейса

Прежде всего рассмотрим разработанный программный интерфейс как самого кадрового графа, так и RHI, чтобы иметь общую картину архитектуры и целей последующей имплементации.

2.1.1. Внедрение в Render Hardware Interface

Ключевая идея в дизайне RHI данной работы – не предоставлять доступ к созданию и отправке на исполнение командных буферов. Это заставляет выражать всю вычислительную работу с GPU посредством кадровых графов. Данный подход позволяет более точно оптимизировать каждую реализацию RHI с учетом особенностей конкретного графического API и аппаратного обеспечения, вместо того, чтобы пытаться обобщить работу с командными буферами между API. Помимо этого, разработчикам, использующим RHI для написания более высокоровневых модулей, будет проще и удобнее выражать свои идеи через граф, декомпозирируя сложные алгоритмы на небольшие задачи, имея глобальную визуализацию рендеринга и не погружаясь в низкоровневые детали работы с командами в графических интерфейсах.

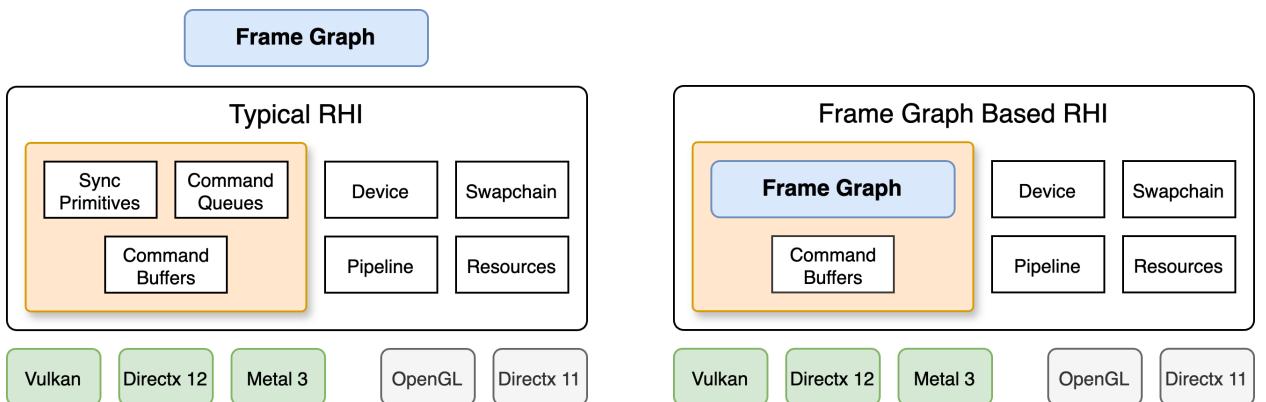


Рисунок 1: Сравнение типичного дизайна RHI (слева) и дизайна, предложенного в данной работе, внедряющего кадровый график напрямую в RHI (справа).

На рисунке 1 представлено сравнение типичной структуры RHI и структуры, предложенной в данной работе. Как видно из рисунка, получилось избавиться от абстракции примитивов синхронизации и командных очередей. Несмотря на то, что командные буферы остались как класс в RHI, методы их создания и отправки на исполнения не предоставляются – всем этим занимается кадровый график. Как уже было сказано, данный подход дает полную свободу при имплементации RHI, поскольку нет необходимости в попытках обобщить интерфейсы всех графических API, как это сделано в типичном дизайне.

2.1.2. Декларация кадрового графа

На этапе декларации графа задаются вершины, а также ресурсы, которые будут использоваться. Вершина графа может быть одного из трех типов – *graphics*, *compute* или же *transfer*. Ресурсы, в свою очередь, могут быть либо вспомогательными, либо импортированными.

Первыми владеет и управляет кадрового графа, вторыми же он только пользуется. Каждому использованию ресурса присваивается некоторое уникальное целое число – *версия*. Пример декларации графической вершины представлен в листинге 1. Компиляцией занимается конкретная имплементация RHI, на этом этапе создаются все необходимые объекты для исполнения кадрового графа. На этапе исполнения конкретная имплементация проходит в нужном порядке по вершинам, вызывает задачу, ассоциированную с данной вершиной на этапе декларации и занимается автоматически расстановкой синхронизации, контекста, отправляет работу на графический процессор и т.д.

```

1 auto rg_color = builder.DeclareImportTexture(...);
2 auto rg_depth = builder.DeclareTransientTexture(...);
3
4 builder.BeginRenderPass("Forward Pass");
5 builder.AddColorTarget(rg_color, ...);
6 builder.SetDepthStencil(rg_depth, ...);
7 builder.SetJob([](CommandBuffer& cmd) {
8     ...
9 });
10 builder.EndRenderPass();

```

C++ Листинг 1: Пример декларации графической вершины графа.

Часто бывает необходимо декларировать временные ресурсы, чьи параметры зависят от других ресурсов. Например, это может быть буфер глубины, ведь его размер должен совпадать с размерами окна, т.е. главного рендер таргета. В связи с этим функции декларации временных ресурсов принимают структуру, похожую на обычную спецификацию ресурса, но у которой некоторые поля (такие как напр. размер) могут помимо точного значения принимать версию ресурса кадрового графа, показывая зависимость. Пример такой декларации показан в 2.

```

1 auto rg_color = builder.DeclareImportTexture(...);
2
3 rhi::FrameGraph::DependentTextureInfo depth_info();
4 depth_info.extent.SetDependency(rg_color);
5 depth_info.format = rhi::Format::D32_SFLOAT;
6 depth_info.type   = rhi::TextureType::Texture2D;
7 depth_info.usage  = rhi::DeviceResourceState::DepthStencilTarget;
8 ...
9 depth_info.name   = "Depth buffer";
10
11 auto rg_depth = builder.DeclareTransientTexture(depth_info);

```

C++ Листинг 2: Пример декларации зависимых временных ресурсов.

2.2. Алгоритм компиляции

2.2.1. Топологическая сортировка

Первая часть компиляции кадрового графа – топологическая сортировка, она является универсальной для всех имплементаций. Предположим у нас есть V вершин графа и уже расставленные ребра между ними, где ребро (u, v) , означает что вершина v должна исполняться после u . В дальнейшем нам также понадобится глубина (или по-другому *dependency level*) каждой вершины, обозначим ее $d(u)$. Глубина вершины помогает определить какие вершины потенциально могут исполняться параллельно. Она определяется как минимальная из всех возможных реберной длины пути от некоторой вершины s , в которую не ведут никакие ребра. За один обход в глубину графа мы сможем получить как глубины всех вершин, так и некоторую топологическую сортировку. Однако произвольная топологическая сортировка нам не подойдет. Необходимо, чтобы в получившейся сортировке глубины вершин монотонно не убывали. Для этого необходима дополнительная сортировка массива. Результат данной части алгоритма для графа 2 проиллюстрирован на рисунке 3. Данная часть алгоритма позволяет преждевременно обнаружить циклические зависимости и обеспечивает корректный порядок всех задач в графе. Кроме того, монотонное неубывание глубин помогает в дальнейшем распределении ресурсов и синхронизации выполнения.

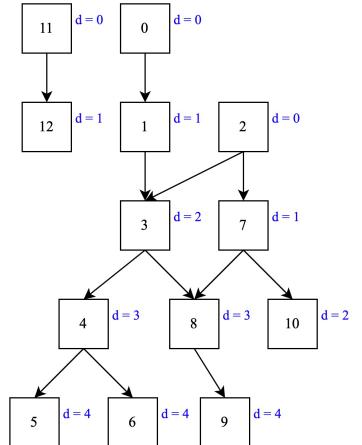


Рисунок 2: Граф зависимостей.

Sort Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Depth	0	1	0	1	2	2	3	3	4	4	4	0	1
Node Handle	0	1	2	7	3	10	4	8	5	6	9	11	12

Sort Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Depth	0	0	0	1	1	1	2	2	3	3	4	4	4
Node Handle	0	2	11	1	7	12	3	10	4	8	5	6	9

Рисунок 3: Первый массив – произвольная топологическая сортировка, второй – перестановка первого, упорядочивающая глубину по неубыванию (оставаясь при этом топологической сортировкой).

2.2.2. Отсечение ребер

Теперь избавимся от ненужных ребер графа для дальнейшего определения точек синхронизации командных очередей. Этот этап критически важен для оптимизации работы графа и уменьшения накладных расходов на синхронизацию. Данная часть алгоритма основана на статье [6] и идентична для всех современных графических API. Предположим, у нас есть Q командных очередей. Также для каждой вершины нам известен индекс очереди, на которой она будет исполняться – $queue_idx(u) \in \{0, \dots, Q - 1\}$. Прежде всего необходимо переиндексировать вершины так, чтобы внутри каждой очереди вершины шли по возрастанию, и для любой пары очередей $q_1, q_2 \in \{0, \dots, Q - 1\}$, таких что $q_1 < q_2$, все вершины очереди q_1 имели индексы меньше, чем индексы всех вершин очереди q_2 . Положим в качестве такой переиндексации $\sigma(u) = sort_idx(u) + queue_idx(u) \cdot V + 1$. Также для каждой вершины введем так называемый *Sufficient Synchronization Index Set* (сокращенно *SSIS*), для каждой очереди показывающий индекс σ вершины, с которой достаточно синхронизировать данную вершину. Формальное определение *SSIS* представлено в равенстве 2. Граф с посчитанными *SSIS* можно увидеть на рисунке 4.

$$i_q(u) = \begin{cases} \sigma(u) & \text{если } q = queue_idx(u) \\ 0 & \text{иначе, если } \forall v : (u, v) \notin E(G^R) \\ \arg \max_{\substack{(u, v) \in E(G^R) \\ d(v) \leq d(u)}} \{\sigma(v)\} & \text{иначе} \end{cases} \quad (1)$$

$$SSIS(u) = (i_0(u), \dots, i_{Q-1}(u)) \quad (2)$$

Строить новый граф без лишних ребер (назовем его G^*) будем итеративно. Для начала для каждой вершины посчитаем бинарный кортеж *cover* из Q элементов, каждый элемент которого показывает синхронизирована ли уже данная вершина с данной очередью. Дальше будем в цикле пытаться улучшить ситуацию, пока для всех вершин *cover* не станет полностью из единиц. На каждой итерации раскрываются все вершины u , чьи *cover* имеют нули. Раскрытие вершины u происходит следующим образом: рассматриваются все вершины v , в которые идут ребра реверс графа из u , считается промежуточный *cover* между только этими двумя вершинами, и если данное ребро улучшает глобальный $final_cover(u)$, то мы запоминаем вершину v . Выбрав из всех таких v наилучшую (по количеству единиц, которое она добавляет в $final_cover(u)$), мы обновляем $final_cover(u)$ и добавляем в граф G^* ребро (u, v) . Полный алгоритм представлен в листинге 3, а также первая итерация алгоритма проиллюстрирована на рисунке 5.

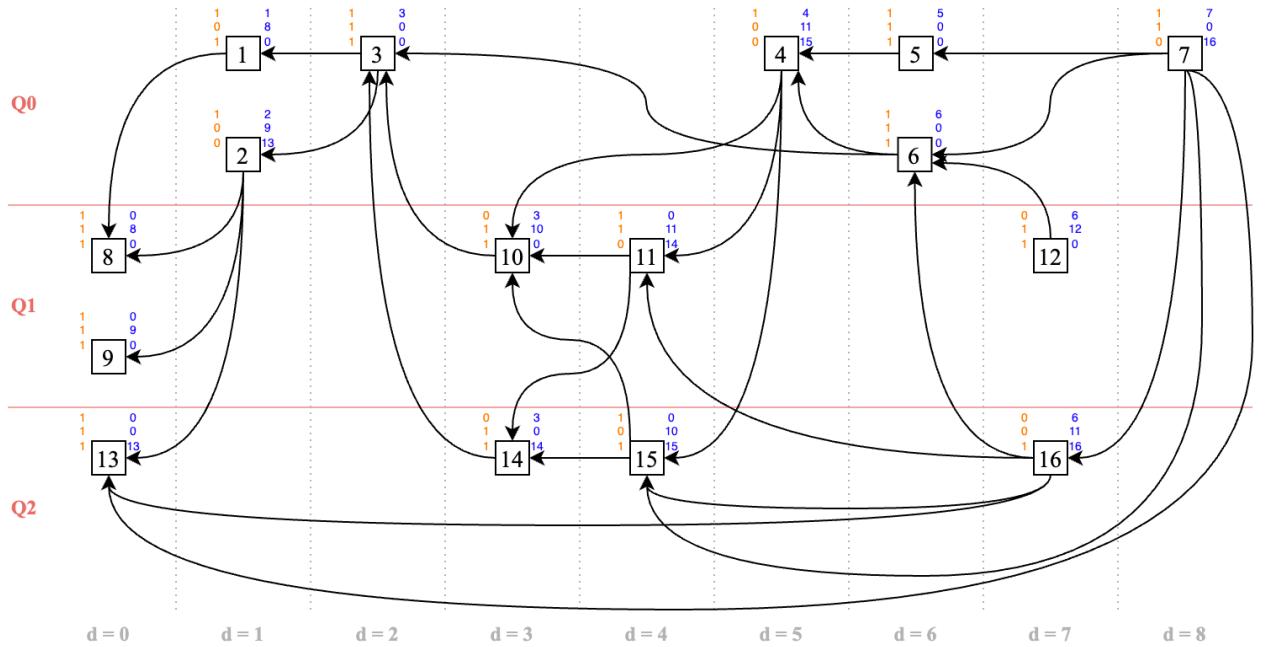


Рисунок 4: Изначальный реверс граф, где у каждой вершины справа синим светом подписан $SSIS$, слева оранжевым $final_cover$, а индексация вершин показывает $\sigma(u)$.

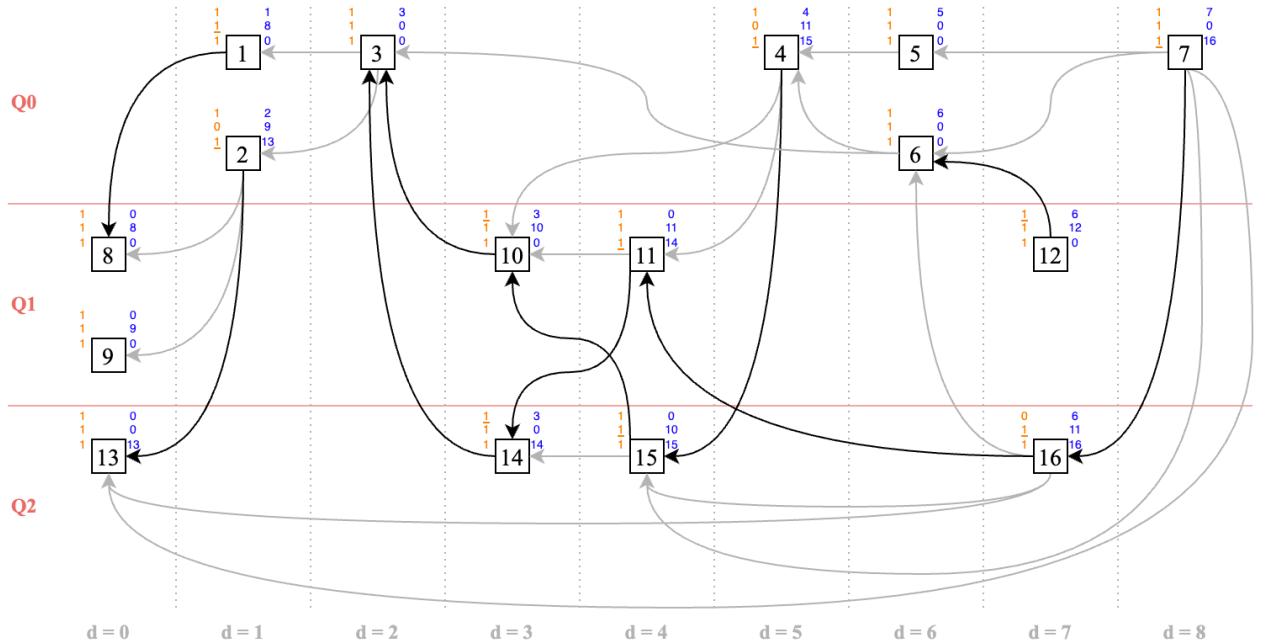


Рисунок 5: Первая итерация алгоритма построения G^* .

```

1   function CoverScore(cover):
2       return  $\sum_{q=0}^{Q-1} cover_q$ 
3
4        $\forall u : final\_cover(u) \leftarrow (1, \dots, 1)$ 
5        $\forall u \forall q \in \{0, \dots, Q-1\} : (\exists v : (u, v) \in E(G^R)) \implies final\_cover_q(u) \leftarrow 0$ 
6
7       covered_all  $\leftarrow \text{false}$ 
8       while not covered_all:
9           covered_all  $\leftarrow \text{true}$ 
10          for u  $\in G^R$ :
11              v*  $\leftarrow \emptyset$ 
12              cover*  $\leftarrow final\_cover(u)$ 
13              for v  $\in G^R, (u, v) \in E(G^R)$ :
14                  cover  $\leftarrow (\dots, SSIS_q(u) \leq SSIS_q(v), \dots)$ 
15                  if CoverScore(cover) > CoverScore(cover*):
16                      v*  $\leftarrow v$ 
17                      cover*  $\leftarrow cover$ 
18                  else if CoverScore(cover) = CoverScore(cover*) and  $\sigma(v) > \sigma(v^*)$ 
19                      v*  $\leftarrow v$ 
20                      cover*  $\leftarrow cover$ 
21                  if v*  $\neq \emptyset$ :
22                      final_cover(u)  $\leftarrow final\_cover(u)|cover^*$ 
23                       $G^* \leftarrow G^* \cup (u, v^*)$ 
24                  if CoverScore(final_cover(u))  $\neq Q$ :
25                      covered_all  $\leftarrow \text{false}$ 

```

Алгоритм 3: Построение графа G^* с достаточным минимальным набором ребер для межочередной синхронизации

2.2.3. Синхронизация

Дальше идет часть компиляции, во многом уникальная для Vulkan. Нам необходимо расставить сабмиты, а также синхронизацию между ними при помощи семафоров (примитивов синхронизации командных очередей). В новых версиях Vulkan появились так называемые *timeline semaphores* (временные семафоры) [7], способные хранить 64-битное число вместо бинарного флага. Поэтому в данной работе предлагается использовать именно их, ввиду их гибкости и сильного снижения количества необходимых примитивов синхронизации.

Сабмиты ставятся в конце каждой командной очереди, а также после каждого уровня глубины, на котором существует хотя бы одна вершина, в которую есть входящие ребра графа G^* . Такой подход гарантирует необходимую синхронизацию и при этом не влечет излишних задержек. На каждую командную очередь выделяется по одному временному семафору. Поскольку значения должны монотонно возрастать, каждое значение можно представить как

$$Timepoint(frame_idx, submit_idx) = frame_idx \cdot V + submit_idx.$$

В данном определении V – произвольное число, заведомо большее числа сабмитов на каждой очереди, а $submit_idx$ – индекс сабмита на конкретной очереди в 1-индексации. Дальше достаточно на основе ребер G^* расставить значения ожидания/сигнала на каждом сабмите. Финальную версию графа G^* с выставленными сабмитами можно увидеть на диаграмме 6.

Немаловажной частью работы с Vulkan также является ручная расстановка барьеров исполнения для ресурсов. Правильная расстановка барьеров критична для предотвращения ошибок гонки данных. Если ресурс используется исключительно в одной очереди, то автоматическая расстановка барьеров в таком случае тривиальна, хотя и требует внимательного рассмотрения всех параметров. Сложность возникает, когда ресурс используется на нескольких очередях. Самый простой вариант решения данной проблемы – это создание всех ресурсов с указанием автоматического кросс-очередного использования. Однако этот подход может повлиять на производительность, например, делая невозможным использование *delta color compression* [8]. Второй же подход – расставлять барьеры с *queue ownership transfer*. В целом, в текущей имплементации графа расстановка данных барьеров не требует больших усилий, поэтому этот подход более предпочтительный.

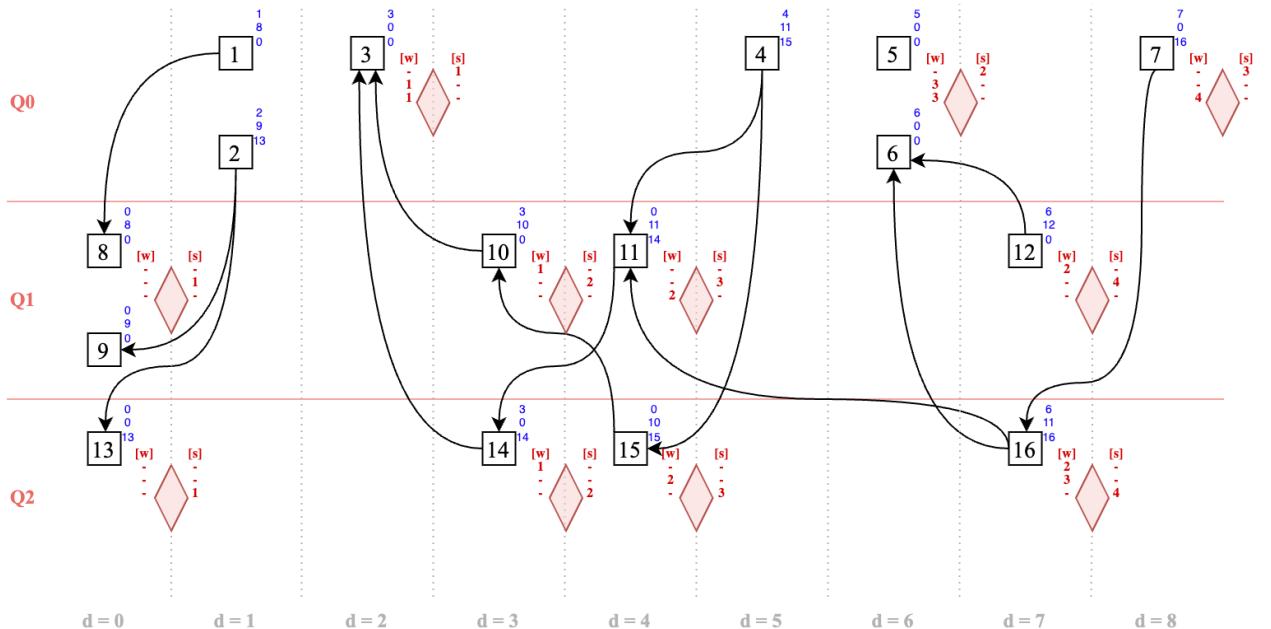


Рисунок 6: Финальный график G^* с выставленными сабмитами. Слева от каждого сабмита подписаны значения для ожидания на каждую очередь, а справа – значения для сигнала.

2.3. Отладочная информация

Для отладки были разработаны два вида визуализации конкретных кадровых графов с помощью инструмента Graphviz [9] – краткая и подробная. Краткая версия предостав-

ляет общую картину графа, что помогает быстро определить корректность "потока" ресурсов в графе. Подробная же отображает дополнительно все параметры ресурсов, типы их использования, а также барьеры. Эта информация особенно полезна при выявлении узких мест и оптимизации производительности рендеринга. Дополнительно, проставляются имена ресурсов и вершин, а также именные границы команд в командных буферах по тому, какой вершине они принадлежат, с помощью специального расширения для Vulkan – VK_EXT_debug_utils [10]. Данная информация помогает при графической отладке в приложениях, таких как RenderDoc [11]. Пример показан на рисунке 7. Представленные механизмы позволяют легко идентифицировать причины ошибок и конфликтующие зависимости между ресурсами, что значительно ускоряет процесс разработки.

7-11	➤ Clustered Light Prepare
13-16	➤ Static Mesh - Prepare
18-44	➤ Particle Emit
46-51	➤ Clustered Light Cull
53-58	➤ Static Mesh - Frustum Cull
54	vkCmdPipelineBarrier2({ StaticMeshFeature - Objects [⌚] , StaticMeshFeature - Batches [⌚] }, vkCmdBindPipeline(^{BuiltIn} .StaticMeshCull [⌚]) vkCmdBindDescriptorSets(0, { VulkanDescriptorManager::set_ [⌚] }) vkCmdPushConstants(VK_SHADER_STAGE_COMPUTE_BIT, (48 bytes)) vkCmdDispatch(2, 1, 1) vkCmdEndDebugUtilsLabelEXT()
55	
56	
57	
58	
59	
60-87	➤ Particle Update
89-211	➤ Forward - Opaque
213-258	➤ Forward - Transparent
260-292	➤ Bloom
294-299	➤ Bloom Compose
301-312	➤ Tonemap

Рисунок 7: Скриншот из RenderDoc, показывающий этапы встроенного рендерера (синий цвет означает transfer, зеленый означает compute и оранжевый – graphics), а также отображающий имена ресурсов (см. события 54-56).

2.4. Структура кадра

В отличие от оригинального решения, предложенного в [3], в данной работе была реализована возможность наличия нескольких кадровых графов. Существует два способа исполнения графов – последовательное и асинхронное. Последовательное исполнение синхронизируется при помощи временных семафоров, аналогично механизму синхронизации сабмитов, разобранному в пункте про компиляцию графов. В данном случае формула момента времени становится

$$Timepoint(frame_idx, graph_idx) = frame_idx \cdot G + graph_idx,$$

где G – произвольное число, заведомо большее числа графов на кадр. При асинхронном же исполнении синхронизации не происходит. Завершение асинхронного исполнения можно

проверить вручную, либо же указав callback при отправке графа на исполнение. Возможность наличия нескольких кадровых графов позволяет более гранулярно разбить разные модули игрового движка, занимающиеся работой с GPU. Так, например, на рисунке 8 представлено возможное разбиение работы на: рендеринг игры, рендеринг в инструментах редактора, отрисовка пользовательского интерфейса, а также асинхронный трансфер данных и генерация preview изображений для редактора. Такой подход позволяет легко масштабировать систему, добавляя новые графы для концептуально новых задач без изменений в уже существующих. Использование нескольких кадровых графов также обеспечивает изоляцию между различными подсистемами, что повышает стабильность и упрощает отладку.

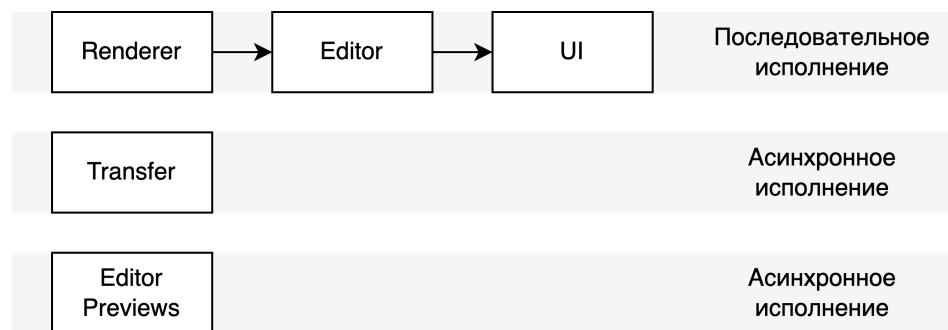


Рисунок 8: Виды композиции исполнения кадровых графов и примеры использования данной функции.

3. Применение в высокоуровневом рендеринге

В данной главе представлена имплементация высокоуровневого рендерера в целях тестирования дизайна и реализации RHI. Современный высокоуровневый рендерер должен быть модульным, удобным в использовании, а также в идеальном случае быть полностью конфигурируемым, включая возможность написания кастомного решения специально под нужды конкретного игрового проекта.

3.1. Общий дизайн

Дизайн, представленный в данной работе (см. рисунок 9) стремится решить все эти проблемы. Рендерер состоит из абстрактных компонент – IRenderFeature. Каждая из них декларирует вершины рендер графа, а также ECS системы (в основном предназначенные для агрегирования данных из компонент сущностей на сцене). Коммуникация между различными IRenderFeature сделана через абстрактный Context, способный содержать произвольные данные.

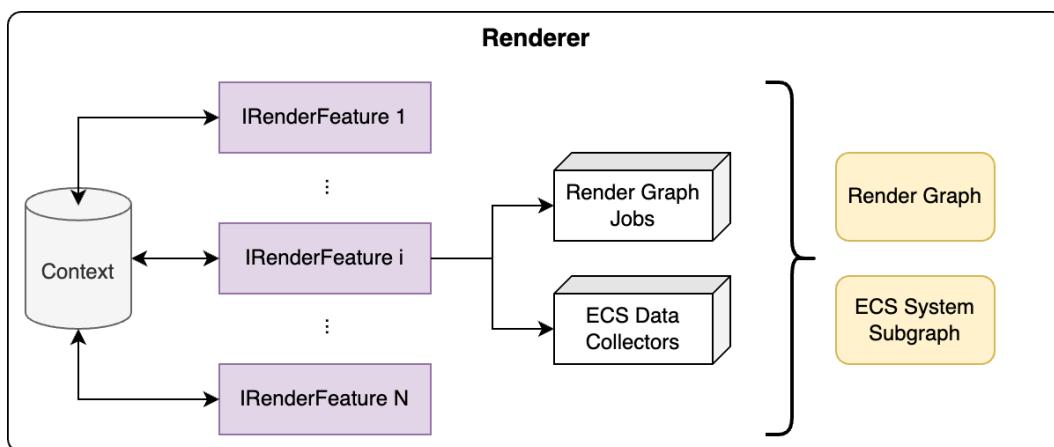


Рисунок 9: Дизайн рендерера

Конкретные имплементации IRenderFeature можно классифицировать на: отрисовку геометрии, отрисовку процедурных объектов, а также эффекты постобработки. Поэтому в качестве рендерера по-умолчанию были реализованы методы рендеринга из всех данных классов в качестве проверки дизайна. Список реализованных IRenderFeature и вершины графа каждой из них представлены схематично на 10.

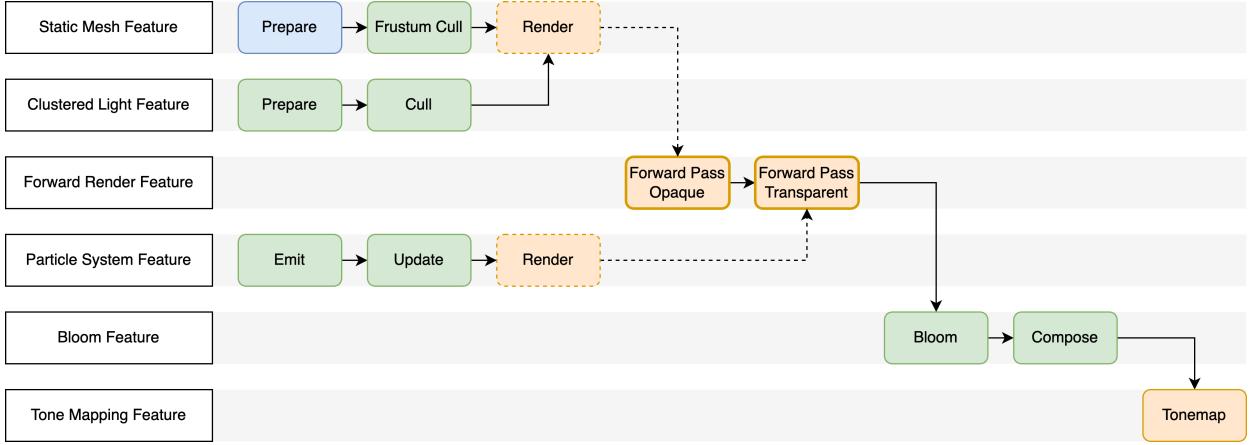


Рисунок 10: Схема имплементации рендерера, представленной в данной работе далее.

3.2. Отрисовка геометрии

Еще до появления современных графических API, было стремление к так называемому GPU-Driven Rendering [12] (рендерингу, управляемому GPU), который представляет собой ряд решений, возлагающих значительную работу по генерации команд отрисовки напрямую на GPU. С увеличением комплексности сцен в видеоиграх, возникла необходимость в перераспределении задач таким образом, чтобы максимально использовать возможности GPU. Этот подход использует возможности параллельной обработки современных графических процессоров для выполнения таких задач, как отсеивание, сортировка и генерация вызовов отрисовки. Рендеринг геометрии – основная цель данного метода. Поэтому было принято решение его задействовать.

У представленного метода два этапа – отсечение по пирамиде видимости и растеризация видимой геометрии. Первый этап происходит в вычислительном шейдере, и помимо непосредственной проверки видимости каждого объекта он также генерирует косвенный буфер команд на GPU. Предположим, есть N мешей, каждый из которых используется на сцене потенциально несколько раз. Соберем все экземпляры каждого меша в N групп. Каждая группа будет одним непрямым вызовом отрисовки экземпляров. В командный буфер запишется лишь одна команда – `multi draw instanced indirect`, которая на GPU будет представлена буфером из N команд отрисовки. Заметим, что на центральном процессоре размер работы в таком методе довольно небольшой – необходимо один раз разбить всю геометрию сцены на группы (препроцессинг), а дальше каждый кадр записать лишь две команды в командный буфер. Раньше вся эта работа (за исключением растеризации) происходила на CPU [13]. Детальная схема описанного алгоритма представлена на рисунке 11.

При самой отрисовке используются техники физически корректного рендеринга (PBR)

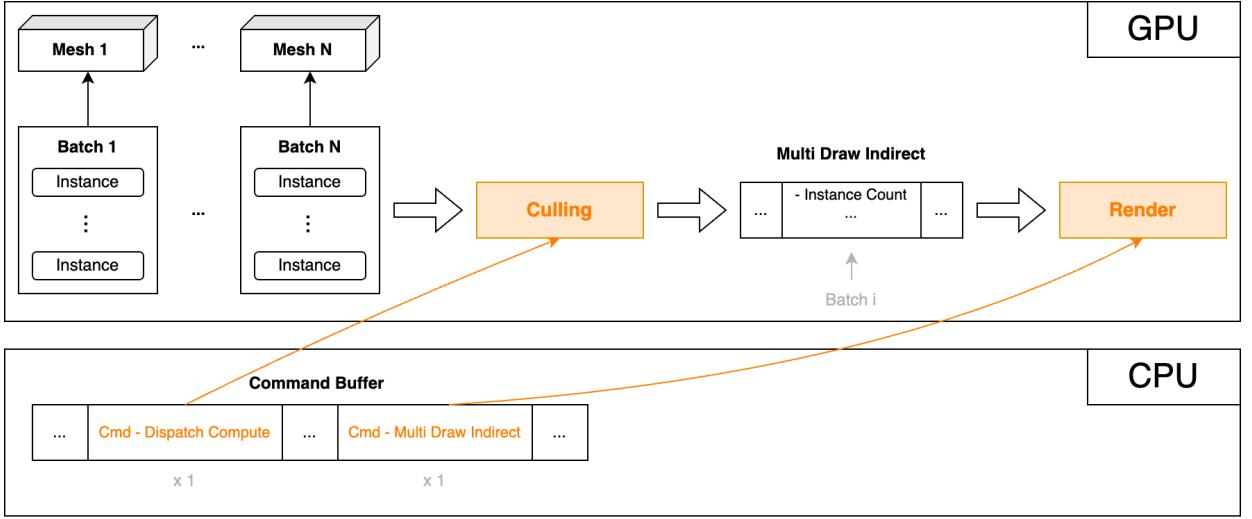


Рисунок 11: Схема процесса рендеринга мешей.

для достижения более визуально реалистичных результатов. PBR симулирует взаимодействие света с поверхностями в соответствии с физическими законами, что позволяет материалам выглядеть и вести себя как их реальные аналоги. Шейдерный код основан на решениях, представленных в [14].

3.3. Кластерное отсечение света

Одна из наиболее затратных (по объему необходимых вычислительных ресурсов) операций в графике является просчет освещения. Современные виртуальные сцены состоят из множества сложной геометрии и источников света. С физической точки зрения, свет, излучаемый источником света распространяется бесконечно далеко, однако в качестве вычислительной аппроксимации, учитывая что затухание света пропорционально $\frac{1}{d^2}$ (где d – расстояние от источника света), можно посчитать радиус точечного источника света, вне которого вклад будет пренебрежимо малым (см. коэффициент затухания 3; заметим, что $\text{Attenuation}(\text{LightRadius}) = 0$). Эта идея положила основу методам отсечения источников света [15], которые оптимизируют расчет освещения путем определения групп фрагментов, на которые не влияет определенный источник света.

$$\text{Attenuation}(d) \approx \frac{\max\left(1 - \frac{d}{\text{LightRadius}}, 0\right)}{d^2} \quad (3)$$

Метод кластерного отсечения света (clustered light culling) разделяет пирамиду видимости камеры на трехмерные кластеры, каждый из которых содержит информацию о всех источниках света, которые могут повлиять на пиксели в этом кластере. Кластеры расположены таким образом, что они полностью покрывают всю сцену и не пересекаются между

собой. Для быстрой проверки пересечений для каждого кластера, то есть усеченной пирамиды, считается минимальный параллельный осям ограничивающий параллелепипед (англ. axis-aligned bounding box или AABB), содержащий весь объем кластера. Ограничивающий же объем для точечных источников света – сфера.

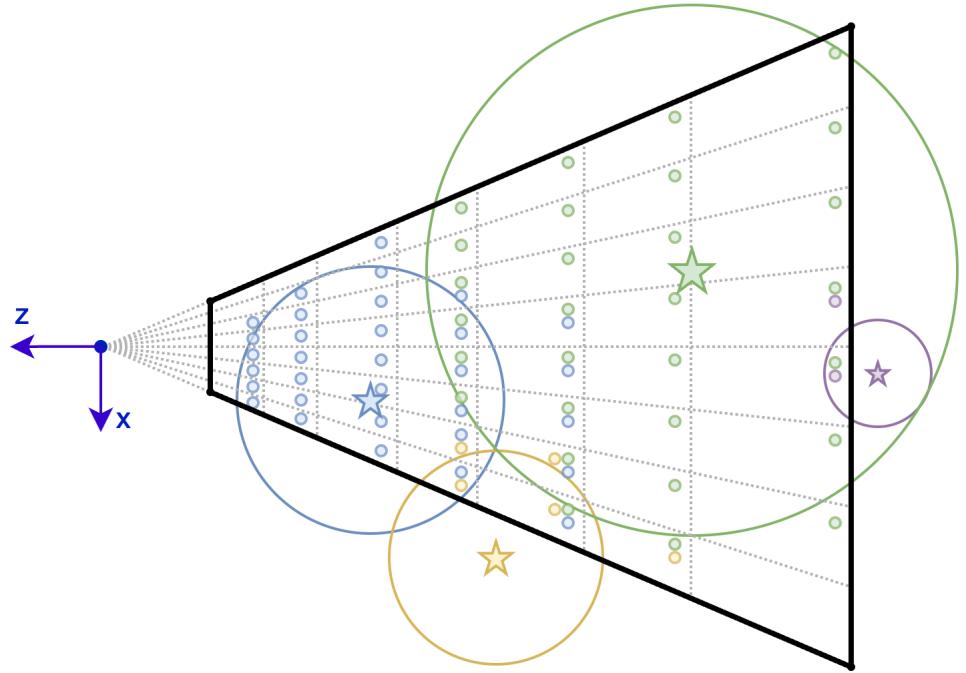


Рисунок 12: Отсечение точечных источников света относительно трехмерных кластеров (вид в проекции на плоскость X-Z)

Существует несколько способов разбиения на кластеры. Далее представлено разбиение, основанное на работе [16]. В плоскости X-Y экран разбивается равномерно на ячейки одинакового размера (в данной работе – 16x16 пикселей). По глубине (вдоль оси Z) же пирамида разделяется экспоненциально, однако была выбрана более простая вычислительно аппроксимация разделения, представленная в докладе [17]:

$$z = \text{Near} \cdot \left(\frac{\text{Far}}{\text{Near}} \right)^{\text{SliceIndex}/\text{NumSlices}} \quad (4)$$

Данный алгоритм работает в два этапа – построение ограничивающих объемов кластеров и непосредственное отсечение света. Первый шаг достаточно выполнять лишь при изменении размеров экрана, то есть когда пирамида видимости изменяется по форме. Второй шаг необходимо выполнять каждый раз когда изменяется положение и/или ориентация камеры, а также при движении источников света. Поскольку в современных видеоиграх преобладает непрерывная динамичность, на практике отсечение света вычисляется каждый кадр. Вызов вычислительного шейдера производится по количеству кластеров по соответствующим трехмерным индексам, то есть одна рабочая группа соответствует одному кластеру. Внутри

рабочей группы каждый поток проверяет пересечения данного кластера с $\frac{\text{NumLights}}{\text{WorkGroupSize}}$ источниками света. По итогу внутри рабочей группы собирается полный список источников света (а точнее их индексов, для сокращения необходимой алгоритму видеопамяти), влияющих на данный кластер. Дальше этот список переносится из shared memory в глобальный буфер, который в последствии будет использоваться при подсчете освещения.

Помимо самого алгоритма, также была реализована отладочная визуализация комплексности освещения, которая отображается поверх просчитанного изображения. На каждый фрагмент добавляется цвет, показывающий как много источников света было обработано в расчете освещения данного фрагмента. Эта визуализация полезна не только для отладки работы представленного алгоритма, но также и для работы дизайнеров сцен, которые могут благодаря ней оптимизировать расстановку источников света, избегая большого количества пересечений источников света.

3.4. Симуляция и отрисовка частиц

Системы частиц являются важным компонентом компьютерной графики, используемым для моделирования таких явлений, как огонь, дым, дождь, взрывы и др. Традиционно эти системы обрабатывались на центральном процессоре [18]. Это включало вычисление позиции, скорости и других свойств каждой частицы в системе для каждого кадра. CPU выполнял все необходимые физические расчеты, включая силы, такие как гравитация, ветер и столкновения. Однако ранние реализации систем частиц были вынуждены использовать упрощенную физику, методы рендеринга и малое количество частиц для достижения желаемых визуальных эффектов в рамках ограничений вычислительной мощности CPU, поскольку CPU не предназначены для высоко параллельной природы симуляций. Появление вычислительных шейдеров привело к переносу значительной части этой работы на GPU.

Симуляция частиц использует параллельную архитектуру GPU для обработки всего жизненного цикла частиц, от их "рождения" до "смерти". Подробная диаграмма системы частиц изображена на рисунке 13. Алгоритм разбит на три части:

1. Emission (инициализация): в вычислительном шейдере "излучаются" новые частицы и инициализируются их атрибуты, такие как положение, скорость, цвет и время жизни. Поскольку на GPU нет динамических массивов, используется один буфер фиксированного размера со всеми частицами. Для определения, какие элементы (частицы) этого буфера свободны, используется вспомогательный буфер – free list.
2. Simulation (симуляция): каждый кадр другой вычислительный шейдер обновляет ат-

рибуты частиц на основе физических расчетов. Это может включать применение сил, обновление скоростей и проверку на столкновения. В случае, если время жизни частицы закончилось, ее индекс добавляется в free list. Также, на этом этапе происходит запись индексов действующих и видимых частиц в еще один вспомогательный буфер.

3. Rendering (отрисовка): как и при отрисовке мешей здесь используется метод GPU-driven rendering. Этап симуляции генерирует команду отрисовки на GPU, где instance count равен числу действующих и видимых частиц. В качестве геометрии для отрисовки используются четырехугольники, ориентированные в сторону камеры [19], на которые потенциально накладывается текстурный атлас (по времени жизни выбирается текущая текстура).

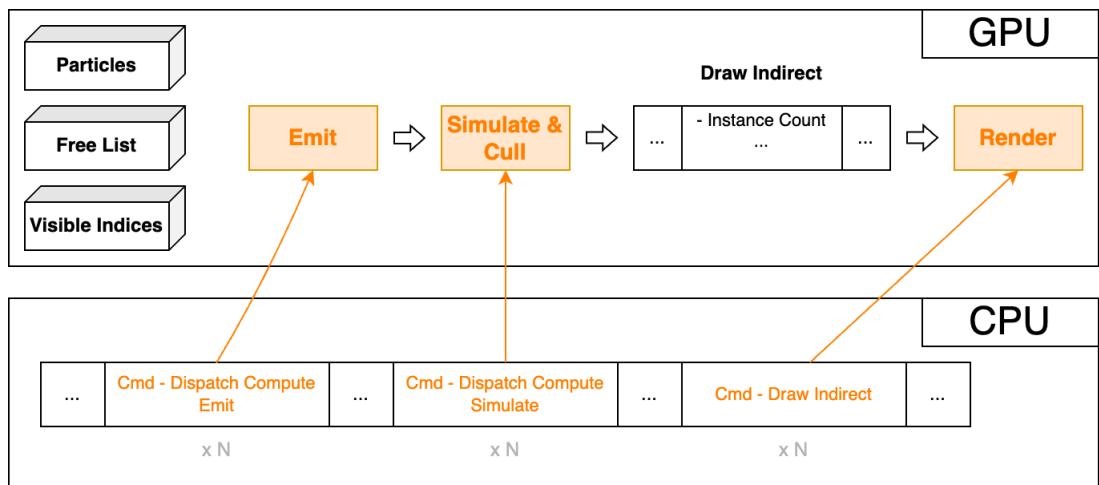


Рисунок 13: Схема процесса симуляции и рендеринга частиц

3.5. Эффекты постобработки

Следующим концептуально отличающимся элементом рендерера является стек эффектов постобработки. Данные эффекты преобразуют уже отрендеренное изображение. В данной работе были реализованы два эффекта – bloom и tone mapping.

3.5.1. Bloom

Эффект свечения (bloom) является популярной техникой постобработки, используемой для имитации рассеивания яркого света в человеческом глазу, создавая светящийся ореол вокруг ярких областей изображения. Этот эффект улучшает реалистичность сцен, особенно тех, которые включают интенсивные источники света, такие как солнце, огни или отражающие поверхности. Для данного эффекта необходимо, чтобы входное изображение было с

высоким динамическим диапазоном (HDR), где контраст между яркими и темными областями более выражен.

Алгоритм генерации данного эффекта был взят из доклада [20] и состоит из сначала последовательного *downsampling*, используя кастомный фильтр (см. рисунок 14), который был придуман для борьбы с негативными темпоральными артефактами, а потом последовательного аддитивного *upsampling* с тент-фильтром размера 3x3 (см. формулу 5) и композиции. Схематично этапы работы алгоритма представлены на рисунке 15.

$$T = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad (5)$$

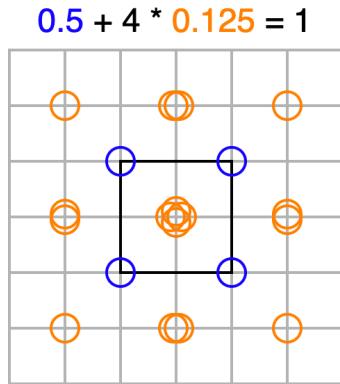


Рисунок 14: Кастомный *downsample*, митигирующий негативные темпоральные артефакты.

Первый *downsampling* отличается от последующих. На нем кастомный фильтр считается по взвешенным средним использованных семплов [21], используя luma: $w(L_{\text{HDR}}) = \frac{1}{1+\text{luma}(L_{\text{HDR}})}$, где $\text{luma}(v) = \text{dot}((0.2126, 0.7152, 0.0722), v^{-2.2})$, а после этого происходит отделение только "ярких" частей, что сделано для возможности художественной конфигурации эффекта при помощи параметров t (*threshold*) и s (*soft threshold*):

$$B(c) = \max(c.r, c.g, c.b) \quad \text{яркость} \quad (6)$$

$$S(c) = \frac{(clamp(B(c) - t + ts, 0, 2ts))^2}{4 \cdot (ts + bias)} \quad \text{смягчающая кривая} \quad (7)$$

$$P(c) = c \cdot \frac{\max(S(B(c)), B(c) - t)}{\max(B(c), bias)} \quad \text{результат префильтрации} \quad (8)$$

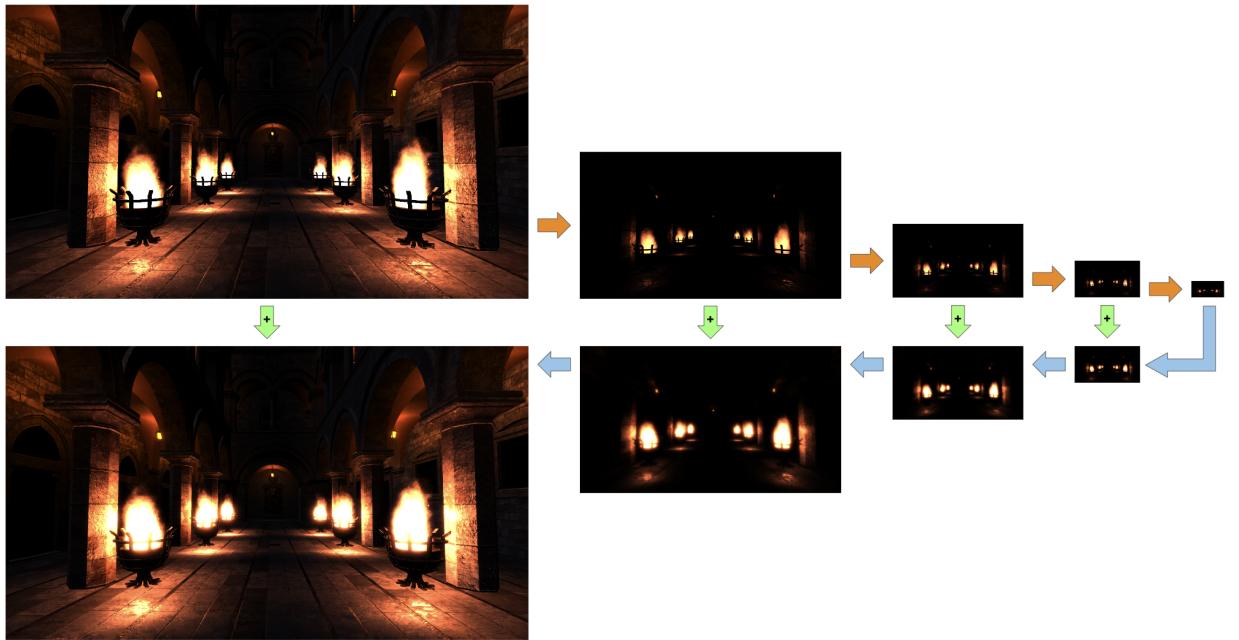


Рисунок 15: Схема просчета эффекта свечения. Изображения представлены без применения tone mapping.

3.5.2. Tone mapping

Тональное отображение (англ. tone mapping) является ключевым процессом в рендеринге с высоким динамическим диапазоном (HDR), позволяющим преобразовать изображения, которые имеют широкий диапазон значений яркости, в формат, подходящий для отображения на стандартных дисплеях с ограниченным динамическим диапазоном. В данной работе реализовано тональное отображение на основе экспозиции [22], динамически регулирующее экспозицию различных областей изображения для сохранения деталей по всему спектру яркости, обеспечивая видимость и правильное отображение как теней, так и высыплений (см. рисунки 16). Формула отображения представлена ниже:

$$L_{\text{mapped}}(x, y) = (1, 1, 1) - e^{-L_{\text{HDR}}(x, y) \cdot \text{exposure}} \quad (9)$$



Рисунок 16: Визуализация тонального отображения с $\text{exposure} = 0.6$; слева направо: $L_{\text{HDR}}(x, y)$ и $L_{\text{mapped}}(x, y)$.

4. Результаты

Имплементация описанного выше решения была написана на C++20 с использованием библиотек Assimp, EnTT, fmt, GLFW, GLM, glslang, magic-enum, stb-image, taskflow, VMA, volk, yaml-cpp. Исходный код имеет размер порядка 12 000 строк (не учитывая шейдерный код) и доступен по ссылке <https://github.com/tralf-strues/liger-engine>. В качестве тестовых моделей были выбраны сцена Sponza Atrium [23] и меш Medieval Brazier [24] с максимально возможным разрешением текстур.

В следующей таблице приведен анализ одного кадра тестового приложения. Замеры были произведены на компьютере с центральным процессором Intel Core i7-9750H (2.6Ghz) и видеокартой NVIDIA GeForce GTX 1660 Ti (Max-Q Design) при разрешении изображения 1920x1080 пикселей и MSAAx8. Для получения метрик использовался профилировщик NVIDIA Nsight Graphics [25].

Показатель	Значение
Меши: instances total	115
Меши: instances passed culling	81
Меши: triangles total	3 781 554
Меши: triangles passed culling	3 318 486
Кол-во источников света	28
Сред. кол-во источников / кластер	1.53
Макс. кол-во источников / кластер	6
Clustered Light Cull время	0.9 мс
Particle Emit время	0.1 мс
Particle Update время	0.12 мс
Particle Render время	0.12 мс
Mesh Frustum Cull время	<0.01 мс
Mesh Render время	3.11 мс
Bloom время	0.4 мс
Tone mapping время	0.07 мс
Время кадра, мс	5.2 мс
FPS (кадры в секунду)	192

Таблица 2: Анализ одного кадра исполнения тестового приложения.

Визуализация (без подробной информации, такой как барьеры и параметры ресурсов) кадрового графа реализованного рендерера, сгенерированная собственным отладочным модулем, представлена на рисунке 17. Более детальная визуализации слишком большая для размещения в данной работе.

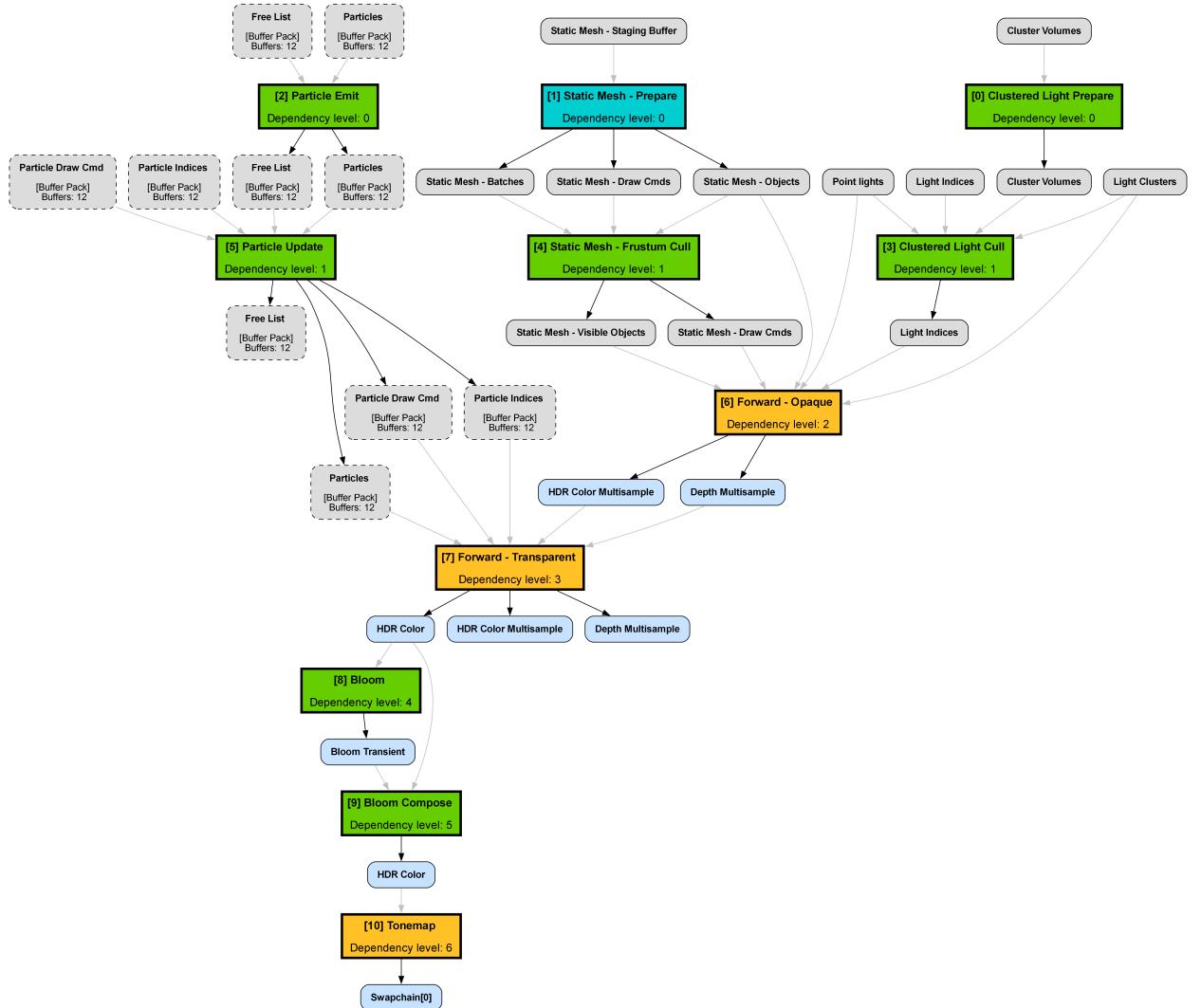


Рисунок 17: Кадровый график рендерера, представленного в данной работе.

5. Заключение

В рамках проведенной работы были уточнены детали устройства метода рендеринга с использованием кадровых графов, написана реализация кадрового графа с поддержкой нескольких командных очередей, предложен ряд новых архитектурных решений в проектировании Render Hardware Interface, а также написан рендерер, использующий современные подходы к графике, такие как вычислительные шейдеры и gpu-driven rendering, в качестве доказательства валидности нового дизайна.

В силу комплексности современных графических API, при дальнейших исследованиях планируется повышать производительность и качество алгоритма планировки задач кадрового графа. Планируется реализовать оптимизированную имплементацию компиляции в случае использования только одной или двух очередей, поскольку в таком случае сложность сильно понижается и нет необходимости рассматривать некоторые сценарии, ввиду их невозможности.

Еще одним крупным направлением для дальнейших исследований является написание фронтенда декларации графов в данных. Это может быть как интеграция в систему шейдеров (как это сделано в [5]), так и построение с помощью визуального редактора узлов, подобно современным редакторам игровых скриптов или материалов [26].

Подход интеграции кадровых графов в RHI весьма перспективен в применении к современным приложениям реального времени, поскольку на данном низком уровне абстракции держится весь высокоуровневый рендеринг, который становится все более комплексным с каждым годом. Данный подход позволит более эффективно использовать ресурсы GPU, позволяя графическим разработчикам концентрироваться на совершенствовании отдельных графических эффектов, делегируя связи и планировку кадровым графикам, а поэтому данный подход заслуживает дальнейшего исследования и усовершенствования.

Список литературы

1. *The Khronos Group Inc.* Vulkan Queues. — URL: <https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html#devsandqueues-queues> (дата обр. 24.05.2024).
2. *The Khronos Group Inc.* Vulkan Command Buffer Recording. — URL: <https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html#commandbuffers-recording> (дата обр. 24.05.2024).
3. *O'Donnell Y.* FrameGraph: Extensible Rendering Architecture in Frostbite. — URL: <https://www.gdcvault.com/play/1024612/FrameGraph-Extensible-Rendering-Architecture-in> (дата обр. 24.05.2024).
4. *Epic Games, Inc.* Render Dependency Graph, Unreal Engine 5. — URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/render-dependency-graph-in-unreal-engine> (дата обр. 24.05.2024).
5. *Advanced Micro Devices, Inc.* AMD Render Pipeline Shaders (RPS) SDK. — URL: <https://github.com/GPUOpen-LibrariesAndSDKs/RenderPipelineShaders> (дата обр. 24.05.2024).
6. *Muratov P.* Organizing GPU Work with Directed Acyclic Graphs. — 10.07.2020. — URL: <https://levelup.gitconnected.com/organizing-gpu-work-with-directed-acyclic-graphs-f3fd5f2c2af3>.
7. *The Khronos Group Inc.* Vulkan Timeline Semaphore. — URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_KHR_timeline_semaphore.html (дата обр. 24.05.2024).
8. *Kramer L.* Optimising a AAA Vulkan title on desktop. — 2019. — URL: <https://www.khronos.org/assets/uploads/developers/library/2019-vulkanised/06-Optimising-aaa-vulkan-title-on-desktop-May19.pdf>.
9. Graphviz, open source graph visualization software. — URL: <https://graphviz.org/> (дата обр. 24.05.2024).
10. *The Khronos Group Inc.* Vulkan Extension Debug Utils. — URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_EXT_debug_utils.html (дата обр. 24.05.2024).
11. RenderDoc, Graphics Debugger. — URL: <https://renderdoc.org/> (дата обр. 24.05.2024).

12. *Haar U., Aaltonen S.* GPU-Driven Rendering Pipelines at Ubisoft Entertainment. — 2015. — URL: <https://advances.realtimerendering.com/s2015/index.html>; SIGGRAPH, Advances in Real-Time Rendering in Games.
13. *Assarsson U., Moller T.* Optimized View Frustum Culling Algorithms for Bounding Boxes // Journal of Graphics Tools: JGT. — 2000. — Июль. — Т. 5. — DOI: [10.1080/10867651.2000.10487517](https://doi.org/10.1080/10867651.2000.10487517).
14. *Pharr M., Jakob W., Humphreys G.* Physically Based Rendering: From Theory to Implementation. — 3rd. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2016. — ISBN 0128006455.
15. *Harada T., McKee J., Yang J. C.* Forward+: Bringing Deferred Lighting to the Next Level // Eurographics. — 2012. — URL: <https://api.semanticscholar.org/CorpusID:4673120>.
16. *Olsson O., Billeter M., Assarsson U.* Clustered deferred and forward shading // High-Performance Graphics 2012, HPG 2012 - ACM SIGGRAPH / Eurographics Symposium Proceedings. — 2012. — Янв. — DOI: [10.2312/EGGH/HPG12/087-096](https://doi.org/10.2312/EGGH/HPG12/087-096).
17. *Sousa T., Geffroy J.* The devil is in the details: idTech 666. — 2016. — URL: <https://advances.realtimerendering.com/s2016/>; SIGGRAPH, Advances in Real-Time Rendering in Games.
18. *Reeves W. T.* Particle Systems—a Technique for Modeling a Class of Fuzzy Objects // ACM Trans. Graph. — New York, NY, USA, 1983. — Апр. — Т. 2, № 2. — С. 91—108. — ISSN 0730-0301. — DOI: [10.1145/357318.357320](https://doi.org/10.1145/357318.357320). — URL: <https://doi.org/10.1145/357318.357320>.
19. *Schaufler G.* Dynamically generated impostors // Proc. GI Workshop on Modeling, Virtual Worlds, Distributed Graphics. — 1995. — С. 129—136.
20. *Jimenez J.* Next Generation Post Processing in Call of Duty: Advanced Warfare. — 2014. — URL: <https://advances.realtimerendering.com/s2014/>; SIGGRAPH, Advances in Real-Time Rendering in Games.
21. *Karis B.* High-Quality Temporal Supersampling. — 2014. — URL: <https://advances.realtimerendering.com/s2014/>; SIGGRAPH, Advances in Real-Time Rendering in Games.
22. Tone mapping of HDR images: A review / Y. Ali [и др.] // . — 06.2012. — С. 368—373. — ISBN 978-1-4577-1967-7. — DOI: [10.1109/ICIAS.2012.6306220](https://doi.org/10.1109/ICIAS.2012.6306220).

23. Sponza Atrium 3D Scene. — URL: <https://www.cgtrader.com/free-3d-models/exterior/historic-exterior/sponza-atrium-2022> (дата обр. 24.05.2024).
24. Medieval Brazier 3D Mesh. — URL: <https://skfb.ly/6WSYs> (дата обр. 24.05.2024).
25. NVIDIA Nsight Graphics. — URL: <https://developer.nvidia.com/nsight-graphics> (дата обр. 24.05.2024).
26. *Epic Games, Inc.* Material Editor, Unreal Engine 5. — URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-material-editor-ui> (дата обр. 24.05.2024).