ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

**Отчет об программном проекте на тему:**

# 3D игровой движок на C++

**Выполнили:**

студент группы БПМИ2110
Баландин Кирилл Андреевич — (подпись) — 13.05.2024 (дата)

студент группы БПМИ216
Рысин Денис Павлович — (подпись) — 13.05.2024 (дата)

**Принял руководитель проекта:**

Максименкова Ольга Вениаминовна
Доцент
Департамент программной инженерии, ФКН НИУ ВШЭ

(подпись) (дата)

**Москва 2024**

# Содержание

# Аннотация

# 1 Введение

В современном мире потребности в инструментах для производства интерактивного медиа контента становятся всё больше и больше, из-за чего появилась необходимость в специальных программах, содержащих обширные наборы различных инструментов для создания виртуальных миров. Для таких целей нужны игровые движки, используемые в различных сферах от создания визуализаций до графики в кино, но преимущественно для создания видеоигр. Исторически сложилось, что у игрового движка нету чёткого определения и границ, из-за чего игровым движком многие называют и небольшою библиотеку для создания игр (например raylib [3]), и огромную экосистему из взаимосвязанных программных продуктов (например Unreal Engine [4]) В рамках этой работы игровой движок будет определятся во многом так же как в книге Джейсона Грегори [2]. Игровой движок – это набор взаимосвязанных программных компонентов для создания видеоигр, который является фундаментом для её разработки, а также имеет возможность использоваться многократно для создания разных игр при минимальных изменениях. Далее в работе под словом движок будет подразумевается, конкретный движок под названием DummyEngine, разрабатываемый на протяжении 2 лет в учебных целях. Исходный код можно найти в репозитории [1]. Целью данной курсовой работы является улучшение и расширение функционала этого движка.

## 1.1 Разделение задач по участникам

**Рысин Денис Павлович:**

- Разработать FrontEnd дизайн рендерера, не требующий от пользователя знаний об используемом BackEnd и берущий на себе ответственность за синхранизацию и управление временем жизни ресурсов
- Реализовать инструмент описания процесса рендеринга RenderGraph как часть FrontEnd
- Реализовать BackEnd рендерера под Vulkan, совместимый с FrontEnd
- Интегрировать новый FrontEnd и Vulkan BackEnd в движок

**Баландин Кирилл Андреевич:**

- Переработка загрузки сущностей
- Реализовать отрисовку теней
- Реализовать отсечение геометрии вне поле зрения камеры

# 2 Термины

- **FrontEnd** - часть рендерера с которой взаимодействует пользователь. Предоставляет общий интрефейс независимый от используемого графического API.
- **BackEnd** - часть рендерера с реализующая интерфейс FrontEnd поверх одного графического API.
- **API** - (Application Programming Interface) набор протоколов, функций и описаний взаимодействия одной программы или её части с другой.
- **OpenGL** - графический API с средним уровнем абстракций.
- **Vulkan** - графический API с низким уровнем абстракций.
- **SwapChain** - объект представляющий из себя набор текстур, используемых для вывода изображения на экран и абстрагирующий этот механизм. Имеет два метода: aquire - получить одну текстуру для рендеринга изображения в неё, present - отдать эту тексуру обратно, для последующего вывод на экран

- **Очередь** - объект в Vulkan, в который отправляются командные буферы на исполнение.
- **Submission order** - порядок на командах внутри одной очереди, удовлетворяющий порядку комманд внутри командных буферов и порядку их отправки в очередь.
- **Процессор** - (CPU) устройство исполняющий множество различных операций и являющийся главным компонентом компьютера.
- **Видеокарта** - (GPU) устройство исполняющее операции с графикой.
- **Пайплайн** – (pipeline) своеобразный конвейер разделяющий выполнение задачи на независимые стадии.
- **Шейдер** - (shader) программа исполняемая на GPU.
- **Рендерер** - модуль отвечающий за обработку графики и взаимодействие с GPU.
- **Граф отрисовки** - (render graph) граф описывающий задачу по на GPU через множество ресурсов, проходов и зависимостей между ними.
- **Проход графа отрисовки** - (pass) представляет собой наименьшую еденицу работы в графе отрисовки.
- **Командный буфер** - буфер в который записываются команды для последующего исполнения.
- **RAM** - (Random Access Memory) оперативная память.
- **VRAM** - (Video Access Memory) видеопамять.
- **ModelLoader** - часть движка, отвечающая за загрузку моделей с диска.
- **Текстура** - изображение, хранимое в видеопамяти.
- **Кэш** - вид памяти используемый в процессорах и видеокартах, характеризуемый маленьким объёмом и большей скоростью
- **Буфер** - диапазон памяти
- **Текстура** - буфер, который хранит в себе какое-то изображение

# 3 Vulkan

## 3.1 Введение

Чтобы придумать хороший дизайн для FrontEnd рендерера, одновременно позволяющий BackEnd-ам эффективно исполнять поступающие задачи и удобный для пользователя, надо понимать основные термины, сущности и дизайны главных графических API.

## 3.2 Теория

### 3.2.1 Описание OpenGL и Vulkan

OpenGL и Vulkan являются двумя наиболее известными представителями графических API. OpenGL является представителем графического API старого поколения. Старое поколение можно охарактеризовать следующими признаками на примере OpenGL:

- **Поддержание состояния:** Всё состояние API является глобальным. В него входят все параметры графических пайплайнов и все настройки рендеринга. Такой подход сильно ограничивает возможность для многопоточного использования, а так же не позволяет средствами API изолировать независимые задачи друг от друга, что в свою очередь оставляет след на архитектуре приложений использующих такие API

- **Подход к описанию действий на GPU:** В OpenGL всё происходит через вызов функций API. Такой подход имеет следующие проблемы. Во-первых, так как OpenGL берёт на себя ответственность за коммуникацию с GPU и предоставляют пользователю по сути асинхронное API, вызов одной и той же функции пользователем, может привести как к дешёвому добавлению этой команды во внутренний буфер API, так и к дорогостоющей отправке этого буфера на GPU, из-за чего производительность для пользователя может быть непредсказуемой. Во-вторых, вызов практически всех функций не может производится многопоточно, что соответственно практически полностью убивает возможность для равномерного распределения задач по взаимодействию с API на ядра CPU.

- **Синхронизация:** Из-за того, что взаимодействие с OpenGL происходит посредством последовательного вызова ассинхронных функций(за исключением тех, которые запрашивают доступ к памяти GPU с CPU, например чтения пикселей текстуры), пользователь не думает о синхронизации, так как для него создаётся иллюзия последовательного

исполнения, но в реальности GPU это устройства максимально заточенные под многоядерные вычисления, имеющие тысячи ядер, а следовательно, требуюшие серьёзной синхронизации. Таким образом ответственность за всю синхронизацию на себя берёт API.

- **Подход к управлению временем жизни ресурсов:** OpenGL сильно упрощает жизнь пользователя так как берёт на себя управления временем жизни ресурсов. Чтобы понять насколько это важно есть следующий пример.
  - Создаётся текстура.
  - Запускается какая-то задача, модифицирующая память этой текстуры.
  - Увеличивается размер текстуры.
  - Запускается аналогичная задача.
  - Текстура удаляется.

  Все этапы этой цепочки реализуются через вызов одной функии графического API из-за чего пользователь 'наблюдает' последовательное исполнение этих команд. Но вспомнив, что OpenGL является ассинхронным, появляются следующие вопросы по реализации:
  - При исполнения шага 5, API не может сразу же освободить память текстуры, так как работа с ней на шаге 4 могла ещё не закончится. Следовательно шаг 5 не приводит у уничтожению текстуры, а лишь говорит API, что пользователю она больше не нужна, из чего следует что моментом уничтожения ресурсов управляет API
  - При исполнении шага 3, API должно выделить новую память для текстуры и освободить текущую, но с текущей памятью текстуры может ещё работать задача с шага 2. Тогда API может либо сразу выделить новую память для текстуры и пометить старую память для освобождения после завершения 2, либо подменить текстуру пользователя на новую, пометив старую на уничтожение, либо заблокироваться на вызове функции 3 до завершения шага 2.

  О всех вопросах такого рода пользователь не должен думать, из-за того что API управляет временем жизни ресурсов, а для пользователя создаётся иллюзия последовательного исполнения, в котором вышеописанных проблем нету.

Из-за всего вышеперечисленного API старого поколения являются крайне простыми в использовании, но накладывают сильные ограничения на архитектуру приложений из-за глобального состояния, а так же практически не поддаются улучшению производительности за счёт количества ядер CPU. Vulkan же является представителем графических API нового поколения. Новое поколение графических API можно сравнить со старым поколение на примере Vulkan:

- **Поддержание состояния:** Новое поколение API избавляется от идеи глобального состояния и даёт намного больше контроля пользователю. В Vulkan даже есть отдельный объект описывающий контекст использования, называющийся Instance. Таким образом один процесс может иметь 2 независимых друг от друга рендерера, когда же в старом поколении такое зачастую невозможно. Главным же хранилищем состояния являются командные буферы. Каждый командный буфер имеет своё локальное состояние независимое от других, что позволяет изолировать задачи средствами API. Остальное состояние хранится непосредственно в объектах, задаётся при их создании и зачастую является немодифицируемым.

- **Подход к описанию действий на GPU:** Большая часть функций в Vulkan не взаимодействует с GPU, а лишь с пользовательским контекстом. Таким образом уже открывается возможность многопоточно создавать и инициализировать ресурсы. Команды же которые приводят к исполнению на GPU, вызываются не на прямую, а записываются в командные буферы. Важно заметить, что такая запись тоже не приводит к исполнению на GPU, а следовательно тоже может происходить многопоточно. К исполнению на GPU приходит только отправка этих буферов на GPU и имеет ограничения по многопоточности. Такой подход во-первых даёт намного больше возможностей для многопоточного использования на CPU в сравнении со старым поколением и во-вторых даёт пользователю намного более предсказуемые длительности вызовов функций API.

- **Синхронизация:** В Vulkan вся ответственность за синхронизацию лежит на пользователе. По умолчанию все команды приводящие к исполнению на GPU, начинают исполнение в submission order, но исполняются парралельно и завершают исполнение в произвольном порядке. Более того произволен не только порядок исполнения, но и работа с памятью. Так же большинство функций не могут блокировать исполнение на CPU, а следовательно синхронизацию между GPU и CPU тоже должен выполнять пользователь. Чтобы создавать зависимости и порядок исполнение пользователь должен использовать инструменты синхронизации такие как semaphore, fence и barrier (определены и подробно описаны в главе 3.2.2).

- **Подход к управлению временем жизни ресурсов:** В Vulkan вся ответственность за время жизни объекта лежит на пользователе и все операции с объектом, такие как его уничтожение или изменение происходят сразу. Таким образом если вернутся к примеру из аналогичной секции про OpenGL, то 3 шаг сразу изменит текстуру, что приведёт к ошибке исполнения шага 2, и шаг 5 исполнится сразу из-за чего сломаются шаги 2 и 4.

### 3.2.2 Синхронизация в Vulkan

Также для принятия решений о дизайне FrontEnd и Vulkan BackEnd нужно знать как работает синхронизация в Vulkan, так как модель памяти в Vulkan является самой сложной среди графических API. Как уже было описано раньше, все команды приводящие к исполнению на GPU записываются в командные буферы и затем отправляются на GPU, но не было описано как именно отправляются.

Одним из важнейших объектов в Vulkan являются очереди. Очередь - объект в который отправляются командные буферы. В этот момент появляется первый порядок в синхронизации называемый submission order. Submission order - порядок на командах попавших в одну очередь, соответствующий порядку команд внутри одного командного буфера и порядку отправки командных буферов в очередь. То есть, если есть командный буфер **A**, в который сначала записали команду **CA** и затем **CB** и командный буфер **B**, с командами **CC** и **CD**, то если в одну очередь отправить буфер **A**, а затем **B**, то образуется submission order **CA** -> **CB** -> **CC** -> **CD**. Submission order определяет порядок запуска команд в пределах одной очереди, не давая гарантий синхронизации на процесс их исполнения и завершения.

Ещё одним важным концептом являются семейства очередей. Каждая очередь пренадлежит какому-то семейству очередей. Семейство очередей определяет набор команд, которые

могут исполнять очереди из этого семейства. Набор команд состоит из произовльного количества групп. Наиболее часто используемые группы - это GRAPHICS(все команды связанные с рендерингом), TRANSFER(команды связанные с перемещением данных между ресурсами, например копирование содержимого одного буфера в другой), COMPUTE(все команды связанные с исполнением compute shader-ов). Также семейство очередей может допускать или не допускать исполнение операций по презентации изображения (отправка содержания текстуры в систему окон операционный системы, для последующего вывода на экран)

Далее следует описать средства для синхронизации предоставляемые Vulkan:

- **Fence:** Fence - средство синхронизации от GPU к CPU. Fence по своей сути является флагом, который GPU может выставить равным True (такая операция называется fence signal), а CPU может выставить False (fence reset) или получить текущее значение флага (fence query). При отправке командных буферов в очередь, а также в некоторых других функциях, можно указать Fence. Это приведёт к тому, что когда очередь исполнит все команды отпарвленных буферов, а также все команды идущие ранне по submission order этой очереди, очередь выполнит fence signal операцию. В то же время CPU может начать ждать fence signal (называется fence wait), а затем сделать fence reset, чтобы вернуть fence в начальное состояние. Этот механизм синхронизации чаще всего нужен для управления временем жизни ресурсов. Например CPU не может уничтожить или начать менять отправленный в очередь командный буфер пока его команды исполняются, то есть до того как GPU выполнит fence signal, а CPU либо fence wait, либо fence query с результатом True

- **Semaphore:** Semaphore - средство синхронизации от GPU к GPU. Semaphore как и fence можно представить как флаг, но пользуется им только GPU. При отправке командных буферов в очередь, а также в некоторых других функциях, можно указать semaphore в качестве wait или signal. Если указать signal semaphore, то по завершении команды флаг будет True, а если wait semaphore, то команда не начнёт исполнение пока, кто-то не выполнит semaphore signal, а затем вернёт semaphore в изначальное состояние и начнёт исполнение. Semaphore чаще всего используется для синхронизации очередей друг с другом, например чтобы все команды из командного буфера **A** отправленного в очередь **A**, выполнились после всех команд командного буфера **B** отправленного в очередь **B**

- **Barrier:** Barrier (барьер) - самый часто используемый инструмент синхронизации в Vulkan с большой степенью гранулярности. Барьер записывается как команды в ко-

мандный буфер и при попадании буфера в очередь, потенциально выполняет синхронизацию между всеми командами до себя в submission order и после. Теперь следует более подробно объяснить, что значит потенциально и синхронизацию.

- **Зависимость по исполнению:** Все команды приводящие к исполнению на GPU имеют несколько стадий (stage). Например команда vkCmdDrawIndexed приводит к исполнению графического пайплайна, который включает в себя привычные стадии шейдеров такие как
  *VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT*,
  *VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT* и
  *VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT*
  У барьера есть два параметра: srcStages и dstStages предаставлюящие наборы стадий. Такой барьер сделает так, что все стадии указанные в dstStages команд далле в submission order после барьера будут исполнены после всех стадий указанных в srcStages команд ранее в submission order.
- **Зависимость по памяти:** Увы обойтись только зависимостями по исполнению в большинстве случаев не выйдет, так как в GPU есть кэш. Большинство команд будут работать с памятью через локальный кэш, а следовательно его надо когда-то сбрасывать (flush) и инавалидировать. Барьер позволяет неявно выполнять такие операции. У команд помимо конкретных стадий есть специальные виды доступа (access) к ресурсам. Например если при рендеринге используется текстура для цвета и буфер глубины, то команды по отрисовки будут выполнять доступы
  *VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT* и
  *VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT* к текстуре с цветом и
  *VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT* и
  *VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT* к буферу глубины. Итого у барьера есть ещё два параметра: srcAccess и dstAccess описывающие наборы доступов с помощью которых барьер выполняет availability и visibility операции. Availability операция делает условный flush кэшей для всех операций которые происходили на стадиях srcStages и производили один из srcAccess, до барьера в submission order. Visibility операция делает условную инвалидацию кэшей, для всех комбинаций dstStages и dstAccess команд после барьера в submission order

Так же барьеры бывают трёх видов:

- **Глобальный:** Распостраняется на все команды из вышеперечисленных условий.
- **Buffer Barrier:** Распостраняется только на команды взаимодействующие с конкретным буфером.
- **Image Barrier:** Распостраняется только на команды взаимодействующие с конкретной текстурой. У него есть 2 дополнительных параметра odlLayout и newLayout. Текстуры в Vulkan имеют свойство layout - то каким образом содержимое текстуры расположено в памяти и каждая команда работающая с текстурами требует, чтобы она была в конкретном layout в момент исполнения. Таким образом image barrier выполняет layout transition до visibility и после availability операций барьера, если oldLayout отличается от newLayout

## 3.3 Дизайн FrontEnd

Для того чтобы придумать дизайн надо установить, что от него требуется:

- **Независимость от BackEnd:** FrontEnd должен быть независимым от BackEnd и скрывать его от пользователя.
- **Синхронизация:** FrontEnd не должен давать пользователю контроля над синхронизацией. Для пользователя должна создавтся иллюзия последовательного исполнения команд как и в OpenGL.
- **Управление временем жизни ресурсов:** FrontEnd должен брать на себя ответственность за время жизни объектов и работать схоже с OpenGL, на примере с 5-ю шагами.
- **Хранение состояния:** FrontEnd должен давать пользователю средства для изоляции состояния и ресурсов в независимые еденицы исполнения.
- **Многопоточность:** FrontEnd должен давать возможность описывать свои действия для отправки на GPU многопоточно, но не обязан многопоточно отсылать эти действия на GPU

Исходя из этих требований предлогается следующий дизайн основанный на двух сущностях:

- **Task** - задача, независимая единица исполнения. Важно что создание задач и их наполнение должно допускать многопоточность и не должно приводить ни к какой работе на GPU. Исполнение работы на GPU в задаче должно описывается в отдельном методе, который может вызывать тольтко Executor. С точки зрения синхронизации задача должна отвечать за синхронизацию между своими операциями сама, а синхронизацию с другими задачами должен обеспечить Executor.

- **Executor** - исполнитель задач, которому отправляются задачи готовые для исполнения. Исполнитель при получении задачи должен вызвать её метод описывающий работу на GPU, предварительно убедившись в выполнении требуемой синхронизации. Так же исполнитель должен продлить время жизни задачи и её ресурсов до конца её исполнения. Отправление задач должно производится в однопоточном режиме, что является частью иллюзии последовательного исполнения.

Такой подход позволяет описывать различные задачи многопоточно и исполнять их не теряя иллюзии последовательного исполнения. Также он не ограничивает гранулярность задач, но как правило чем больше задача, тем больше она знает о требуемой внутри себя синхронизации и тем боллее эффективно можнт быть исполнена. Задачи могут быть как полностью служебные и недоступные для модификации пользователем(например метод SetData у буфера будет приводить к созданию служебной задачи по отправке данных в память GPU и отправки этой задаче в Executor), так и большие кастомизируемые пользователем. Таких задач есть 2 вида. Первый тип задач основан на концепте командного буфера. Она должна давать пользователю возможность записывать произвольные команды в буфер, но создавать прослойку между теми командами которые записывает пользователь и тем что на самом деле BackEnd записывает в командный буфер графического API если таковой есть. Второй тип задач - задачи которые выдаёт граф отрисовки, который описан в следующем разделе.

### 3.3.1 Граф отрисовки

Граф отрисовки - способ описать сложую последовательность графических операций и их зависимостей в виде однонаправленного ациклического графа. Идея взята с презентации GDC [7]. Веришины в графе бывают двух типов: ресурс и проход. Ресурсные вершины описывают какой-то ресурс(например текстуру или буфер), который необязательно ещё существует. Вершины с проходами описывают какую-то часть работы над ресурсами. Рёбра в

таком графе представляют из себя зависимость и могут быть только между ресурсной вершиной и проходной. Если ребро идёт из ресурса в проход, то оно может быть двух типов: только чтение или чтение и запись и означает, что проход использует в качестве входа этот ресурс и в процессе может делать только указанные доступы к нему. Очевидно, что из ресурса может выходить либо только одно ребро на чтение запись или произвольное количество только на чтение. Если ребро идёт из прохода в ресурс, то это говорит, что проход пишет в этот ресурс. Пользователь может создавать такие зависимости в пределах одного графа указав ресурс, проход, название зависимости и тип зависимости(Только чтение, чтение и запись или результат прохода). Создав такую зависимость проход получает информацию о внешнем для него ресурсе, наследующиеся проходы могут получать внешние ресурсы по имени зависимости. Это очевидно накладывает ограничение на уникальность имён зависимостей использующих один проход. Также в граф можно в качестве проходной вершины добавить в другой граф, через тот же самый механизм зависимостей. Важно отметить, что граф отрисовки являетс лишь 'чертежём' того, что надо отрисовать и соответственно имеет метод который создаёт задачу из графа, которую потом можно отправить в исполнитель. За реализации ресурсных и проходных вершин, а также за создание задачи по графу отвечает BackEnd.

Таким образом граф отрисовки позволяет описывать сложные процессы отрисовки в простом виде и позволяет переиспользовать графы друг в друге. Пользователь получает возможность строить этот граф из заранее заготовленных ресурсных и проходных вершин, а так же может создавать свои на базе некоторых из них. Например есть проход который внутри себя использует такой-же командный буфер как и в задаче первого типа, но позволяет отправить его не отдельной задачей, а частью большей, что позволяет BackEnd исполнить его более эффективно.

## 3.4 Vulkan BackeEnd

Главной задачей при реализации BackeEnd для Vulkan является автоматизация его синхронизации. Описание алгоритма синхронизации состоит из нескольких этапов по решению всё больших и больших проблем сценариев синхронизации.

- **Сценарий 1:** Допустим мы работает только с одним вершинными буфером внутри единственного командного буфера и нашей задачей является написать обёртку над командным буфером позволяющую выполнять операции с этим вершинными буфером создавая иллюзию последовательного исполнения. Для этого надо каким-то образом расставить барьеры между операциями по работе с вершинным буфером которые вы-

полняли бы нужную синхронизацию по исполнению и по памяти. Все операции можно разделить на две группы: только чтение (R) или чтение и запись(W). Допустим первая операция была из группы W(назовём её W1), а после неё идёт операция R1. Между ними надо выставить барьер B1, который во-первых сделает зависимость по исполнению от всех стадий операции W1 взаимодействующих с вершинным буфером до всех стадий операции R1 взаимодействующих с вершинным буфером, и во-вторых сделает availability операцию для всех доступов операции W1 и visibility операцию для всех доступов операции R1. Допустим дальше идёт точно такая же как R1 операция R2. Очевидно, что барьер для неё не нужен, так как все зависимости с W1 уже покрывает предыдущий барьер, а синхронизировать операции из группы R друг с другом не имеет смысла. Затем идёт операция W2. Для неё нужно обеспечить барьер B2 такой же как B1, но вместо стадий и доступов R1 взять их у W2. Так же надо убедится, что исполнение предыдущих стадий R1 и R2, читающих буфер, завершится раньше начала стадий W2, модифицирующих вершинный буфер, поэтому нужен ещё один барьер B3 содержащий только зависимость по исполнению между R1, R2 и W2.

Итого получаем следующий алгоритм:

- Храним для буфера стадии и доступы последней операции WL из группы W и все комбинации стадий и доступов операций группы R идущих после WL.
- Если приходит операция из группы R, то сначала проверяем наличие всех комбинаций её стадий и доступов в состоянии. Если все присутствуют, то барьер не требуется. Если есть отсутствующие комбинации, то вставляем барьер от сохранённых стадий и доступов WL до отсутствующие комбинации R и добавляем их в состояние.
- Если приходит операция из группы W, то вставляем 2 барьера как в примере, сбрасываем все комбинации стадий и доступов операций из R и заменяем стадии и доступы WL на аналогичные от текущей операции

Такой же алгоритм можно обобщить и для работы с текстурами. Для этого надо добавить к состоянию последний layout текстуры и при приходе операции R проверять совпадение нового layout со старым и в случае несовпадения относится к этой операции как к операции из группы W

Такой алгоритм хорош, но работает только в рамках одного командного буфера

- **Сценарий 2:** Допустим мы хотим решить такую же задачу как в сценарии 1, но допустить паралельную работу с вершинным буфером в разных командных буферах, а так же последовательную их отправку как задач в исполнитель. Такое расширение задачи требует дополнительной синхронизации между командами двух последовательно отправленных командных буферов (допустим сначала отправили B1, а после него B2). Введём понятие синхранизационный префикс и суффикс для использования вершинного буфера в этих командных буферах. Префиксом является множество операций от начала командного буфера до первой операции из группы W, а суффиксом операции от последней операции W до конца буфера. Тогда легко заметить, что достаточно лишь синхронизировать суффикс B1 (SB1) с префиксом B2 (PB2). Это можно сделать создав вспомогательный буфер B3 при отправке B2 в исполнитель и записать в него 3 барьера по вышеописанному алгоритму для синхронизации SB1 с PB2. Затем можно отправить в очередь B3, а затем B2. Чтобы расширить такой алгоритм на произвольное количество командных буфером, надо хранить последний суффикс классе вершинного буфера и при каждом новом буфере синхронизировать суффикс вершинного буфера с префиксом командного буфера, после чего заменять суффикс вершинного буфера на суффикс командного буфера.

- **Сценарий 3:** Решаем сценарий 2, но убираем из него ограничение на один вершинный буфер и позволяем использование любых множеств буферов и текстур. Тогда в каждом командном буфере храним множество всех когда либо использованных в нём ресурсов и поддерживаем их синхронизационное состояние. При отправке в исполнитель во вспомогательный буфер идут барьеры для всех используемых ресурсов, после чего для них же обновляются их суффиксы.

Таким образом получаем эффективный механизм полной автоматизации синхронизации внутри одной очереди. На этом автоматизация синхронизации в рамках данной работы почти заканчивается, но этого уже достаточно, чтобы поддержать все операции из групп COMPUTE, GRAPHICS и TRANSFER так как спецификация Vulkan гарантирует, что есть хотя бы одно семейство очередей поддерживающее эти 3 группы операций, но неотъемлимой частью любого графического приложения является вывод изображения на экран, поэтому обязательно поддерживать операции по презентации. Для таких операций по техническим причинам используется ещё одна вспомогательную очередь.

### 3.4.1 Vulkan RenderGraph

Выше мы описали механизм автоматизации синхронизации, но у него есть несколько проблем. Во-первых на каждую задачу может понадобится создать вспомогательный командный буфер для синхронизации, а во-вторых для текстуры отправляемой во вспомогательную очередь на презентацию, надо использовать семафоры для синхронизации между очередьми. Эти проблемы решает реализации графа отрисовки в BackEnd Vulkan.

Сам граф отрисовки реализован как часть FrontEnd, осталось только сделать реализации для вершин, проходов и алгоритм постройки задач. Сначала определимся с набором проходов под реализацию.

- **Командный проход:** Командный проход содержит в себе вышеописанную обёртку над командным буфером и позволяет наследующим проходам записывать туда произвольные команды со внутренней синхронизацией.
- **SwapChain Aquire:** Проход который выполняет операцию получения новой текстуры из swapchain
- **SwapChain Present:** Проход который выполняет операцию отправки текстуры из swapchain на презентацию

Если первый проход исполняется на основной очереди, то два последних исполняются на вспомогательной очереди, а следовательно между ними нужна синхронизация через semaphore.

Теперь надо описать алгоритм по которому из графа строится одна задача для отправки в исполнитель.

- **Шаг 1: Сбор струтуры графа**

  Для начала нужно собрать все вершины графа и зависимости между ними в одну структуру. Тут всё очевидно кроме одного момента. Граф хранит информацию только о своих внутренних вершинах и зависимостях, но в качестве любой проходной вершины может использовтся другой граф отрисовки, поэтому надо рекурсивно обойти все такие вершины и собрать требуемую информацию и из них

- **Шаг 2: Инстанциировать ресурсы и проходы**

  Как было описано ранее граф отрисовки является лишь чертежём того, что надо отрисовать и не обязан хранить в себе ресурсы требуемые для этого. Поэтому на этой стадии надо сначала для каждой ресурсной вершины вызвать метод Instantiate, который делает доступным для пользователя объект описываемого ресурса, а затем CreatePass у

каждой проходной вершины для создания объектов проходов и получения ими внешних ресурсов.

- **Шаг 3: Вспомогательные вычисления** Выполняем топологическую сортировку вершин графа и сохраняем результат. Строим вспомогательный однонаправленный ациклический граф P, содержащий только проходные вершины, в котором добавляем ребро между двумя проходами если оба используют общий ресурс в графе и у одного в качестве входа, у другого в качестве выхода. Данные вычисления понадобятся на следующих шагах.

- **Шаг 4: Кластеризация** Кластером является набор проходов исполняемых на одной и той же очереди, а так же удовлетворяющий следующему свойству: в графе P нету такой вершины, что она не принадлежит данному кластеру, но по рёбрам графа P можно дойти до неё из кластера, а так же можно дойти от неё до кластера. Алгоритм постройки такой кластеризации описан далее:

  - Найти все вершины графа P в которые не ведут никакие рёбра. Назовём это множество линией фронта.
  - Берём любую вершину из линии фронта и добавляем её в новый кластер. При добавлении любой вершины в любой кластер, добавляем в линию фронта все вершины, которые ещё не принадлежат никакому кластеру, но все вершины из которых в неё идут рёбра уже находятся в каком-то кластере.
  - Пока можем ищем в линии фронта вершины которые можно добавить в текущий кластер (то есть исполняемые на той же очереди) и расширяем текущий кластер.
  - Если ещё остались вершины идём на шаг 2. Иначе кластеризация завершена.

- **Шаг 5: Граф кластеров** Строим однонаправленный ациклический граф кластеров K, вершинами которого являются кластеры. Между двумя кластерами есть ребро, если между какими-то двумя вершинами этих кластеров есть ребро в графе P. На каждое ребро такого графа надо создать по семафору, которые будет синхронизировать отправку кластеров в разные очереди.

- **Шаг 6: Подготовка проходов** На данном шаге паралельно запускается метод Prepare у каждого прохода, который подготавливает содержимое прохода к отправке. В таких методах и будет содержатся большая часть нагрузки (например командные проходы будут именно тут записывать все команды в свой командный буфер)

- **Шаг 7: Полировка кластеров** На этом этапе для каждого кластера будет вызвана функция Finalize, которая завершит ‘полировку‘ кластера. Например в кластере командных проходов Finalize для всех командных буферов в порядке топологической сортировке генерирует промежуточные барьеры по синхронизации двух последовательных командных буферов и дописывает их в конец предыдущего, вместо создания отдельного как в случае с отправкой задач первого типа в исполнитель. Таким образом на n таких проходов вместо n вспомогательных буферов нужен только один.

Задача готова. Теперь осталось описать, что происходит при её отправке, что крайне просто. Каждый кластер в порядке топологической сортировки отправляется в свою очередь с учётом зависимостей из графа K реализуемых посредством semaphore.

## 3.5 Результаты

В итоге можно с уверенностью сказать, что задача была сильно недооценена в размерах и сложности, при выборе темы проекта из-за чего часть поставленных задач не была выполнена. Успешно получилось:

- Разработать приемлемый дизайн FrontEnd удовлетворяющий поставленным требованиям.
- Реализовать граф отрисовки как часть FrontEnd
- Реализовать BackEnd для Vulkan

Не вышло сделать:

- Интеграция нового FrontEnd и Vulkan BackEnd в движок

В связи с этим вся вышеописанная работа была реализована в отдельном репозитории [8] и будет интегрирована в движок, но не в рамках этой курсовой работы.

# 4 Переработка загрузки сущностей

## 4.1 Проблема

При попытке параллелизации загрузки сцен выявился ряд недостатков при проектировании, не позволяющих запускать загрузку сущностей параллельно. OpenGL по умолчанию имеет один контекст рендеринга на все потоки. Из-за этого при попытке подгружать текстуры из RAM в VRAM одновременно происходит состояние гонки. Архитектура ModelLoader не позволяла полноценно разделять загрузку моделей с диска в оперативную память и загрузку из оперативной памяти в видеопамять.

## 4.2 Решение

Для решения этой проблемы было решено ввести новую сущность Resource. Эта сущность хранит в себе следующее:

- ID
- Состояние
- Данные для загрузки с диска
- Данные ресурса
- Данные с GPU

*ID* - уникальный идентификатор ресурса.
Ресурс может быть в 3х состояниях: *NOT_LOADED*, *LOADED_TO_RAM*, *LOADED_TO_GPU*.
*NOT_LOADED* - состояние, когда ресурс не прогружен. В этом состоянии у ресурса уже существует ID и данные для загрузки с диска. На остальные данные память не выделена
*LOADED_TO_RAM* - состояние, когда данные ресурса загружены в оперативную память, но графическое API не знает ничего про этот ресурс.
*LOADED_TO_GPU* - ресурс готов к использованию в процессе рендера.
Соответственно при загрузке сцены игровым движком можно распаралеллить переход ресурса из *NOT_LOADED* в *LOADED_TO_RAM*.

# 5 Тени

## 5.1 Введение

Одной из основных составляющих освещения являются тени. Они добавляют реализма к изображению и помогают понять взаимное расположение объектов. Объект находится в тени когла луч света от источника не достигает его из-за пересечения с другим объектом.

Существует несколько различных реализаций рендеринга теней. Самой популярной техникой в современных видеоиграх является построение карты теней(shadow mapping).

## 5.2 Карты теней

Для отрисовки теней, при растеризации изображения нам необходимо знать, находится ли текущий пиксель в тени. Как мы знаем, пиксель находится в тени, если на пути от источника света до него нет никаких препятствий.

Чтобы определить, есть ли препятствия между источником света и пикселем давайте отрисуем сцену дважды. В первый раз отрисуем сцену от лица источника света и поместим в отдельный буфер. Заметим, что в этом буфере нас интересует не цвет объектов, а расстояние от источника света до объекта. Затем отрисуем сцену как обычно от лица камеры и, во время растеризации, в фрагментном шейдере получим соответствующее значение глубины из буфера полученного на прошлом этапе и сравним его с расстоянием до источника света. Если эти значения равны - пиксель находится на свету, иначе - в тени.

Рассмотрим этот процесс на примере:

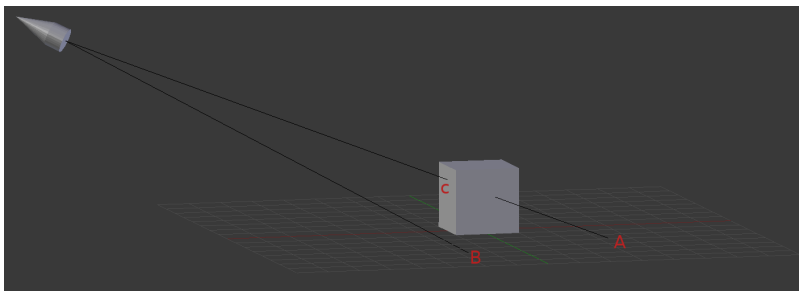


Рис. 5.1: Пример сцены

Пример состоит из 2х объектов: плоскости и куба. Во время первого прохода мы отрисовываем сцену от лица камеры. Рассмотрим точки A и C. Для буфера глубины они находятся в одном пикселе. Тогда в буфере глубины будет сохранено расстояние от камеры до точки C. При финальном рендере мы отрисовываем от лица камеры. При растеризации

точки A найдем ее координаты в буфере глубины и увидим, что глубина в буфере отличается от расстояния от точки A до источника света, а значит точка A в тени.

## 5.3 Построение карты теней

Для начала рассмотрим как построить карту теней для точечного направленного источника света. Для каждого источника света необходим дополнительный проход отрисовки сцены.

После добавления всей *непрозрачной* геометрии создается текстура, в которую будет производиться отрисовка буфера. Размер этой текстуры подбирается отдельно в зависимости от желаемого результата. Наиболее хорошее отношение результата к затрачиваемым ресурсам получается при использовании удвоенного размера окна.

Для отрисовки сцены в текстуру используется объект буфера кадра(Framebuffer Object). FBO хранит в себе буфер цвета, буфер глубины, буфер трафарета и прочие параметры. Нас интересует именно буфер глубины. При создании FBO мы привязываем созданную на прошлом этапе текстуру к буферу глубины и явно укзываем графическому API не строить буфер цвета.

Так как нас мы отключили построение буфера цвета, то фрагментный шейдер не будет использоваться. Но при этом тест глубины по-прежнему будет выполняться. Стоит заметить, что для построения карты теней для направленных источников света лучше использовать ортогональную проекцию вместо перспективной.

## 5.4 Использование карты теней

На прошлом этапе мы получили буфер глубины "от лица"источника освещения. Чтобы использовать его, нам необходимо при растеризации итоговой картинки узнать координату пикселя в текстуре буфера глубины.

Вспомним, как происходит преобразование глобальных координат в пространство камеры:

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ 1 \end{pmatrix} = projection \cdot view \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

где $projection$ - матрица проекции(ортогональная в случае отрисовки карты теней и перспективная в случае отрисовки финального кадра), $view$ - матрица, описывающая расположение

камеры в пространстве, $x, y, z$ - глобальные координаты, $x_c, y_c, z_c$ - координаты в пространстве камеры. Соответственно чтобы преобразовать глобальные координаты в координаты пикселя из карты теней необходимо вначале преобразовать его в пространство источника света, а затем в текстурные координаты:

$$\begin{pmatrix} x_t \\ y_t \\ z_t \\ 1 \end{pmatrix} = bias \cdot projection_{light} \cdot view_{light} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}, bias = \begin{pmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

где $bias$ - матрица сдвига из однородных координат($[-1; 1]$) в текстурные ($[0; 1]$), $projection_{light}$ - матрица проекции карты теней, $view_{light}$ - матрица, описывающая положение и направление источника света, $x_t, y_t$ - итоговые координаты пикселя в карте теней, $z_t$ - расстояние от источника света до точки в интервале $[0; 1]$, $x, y, z$ - глобальные координаты.

Зная координаты пикселя в карте теней мы можем проверить, находится ли он в тени. Для этого надо сравнить значение глубины из карты теней с $z_t$. Если они совпадают - значит объект освещен, иначе - находится в тени

## 5.5 Борьба с артефактами

### 5.5.1 Муаровый узор

При базовой реализации метода можно заметить много артефактов, похожих на муаровый узор. В анлоязычной литературе этот эффект называют "shadow acne"[TODO: ref]

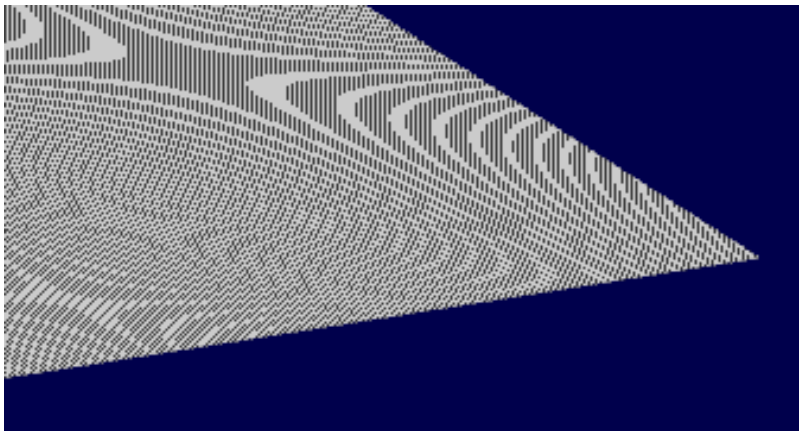

Рис. 5.2: Пример муаровых узоров в случае теней

Объясняется этот эффект тем, что карта теней имеет ограниченный размер и один

пиксель из карты теней может отвечать за множество пикселей итогового изображения. Для решения этой проблемы необходимо разрешить некоторую погрешность в сравнении значения из карты глубины и расстояния до источника освещения.

### 5.5.2 Объекты вне карты теней

Так же можно заметить, что объекты вдалеке находятся всегда в тени. Связано это с тем, что эти объекты находятся вне карты теней. Для решения этого артефакта есть два решения.

Первый, самый простой - отдалить источник освещения, чтобы объекты вдалеке попали в карту теней. Это даст быстрый результат, но при этом качество теней вблизи ухудшится.

Другой способ - игнорировать тени для объектов вдалеке. Для этого можно внутри шейдера проверять координаты пикселя в карте теней и проверить, что они находятся вне отрезка $[0; 1]$

### 5.5.3 Сглаживание краев тени

Если карта теней не имеет достаточного разрешения, то на границе тени можно заметить ее отдельные пиксели. Чтобы сгладить этот эффект используется техника PCF(percentage-closer filtering), которая заключается в том, чтобы сглаживать тень, используя значение не только конкретного пикселя в карте теней, но и его соседей.

## 5.6 Точечные источники освещения

С направленными источниками освещения нам достаточно построить одну карту теней, чтобы корректно симулировать тени. В случае с точечными всенаправленными источниками необходимо отрисовывать сцену во всех направлениях и обычная текстура не подойдет.

Кубическая карта(Cubemap) хранит данные об окружении с помощью 6 текстур. Соответственно можно отрисовывать всю сцену на каждую грань и передавать 6 карт теней в фрагментный шейдер.

Очевидный способ - повторить действия как для направленного источника 6 раз. Но такой подход оказывается очень требовательным к ресурсам, т.к. для обработки одного источника света используется много вызовов отрисовки(render call). Для оптимизации генерации кубической карты теней можно использовать геометрический шейдер.

Вначале создадим текстуру, и каждую грань привяжем как отдельный слой этой текстуры. В примитивном способе мы бы привязывали каждую грань к FBO и отрисовывали

бы сцену 6 раз, но с помощью геометрического шейдера мы можем за одно отрисовывание сцены вывести всю сцену на 6 граней.

Как и в генерации карты теней для направленного освещения сгенерируем *view*-матрицу, но на этот раз нам понадобится 6 матриц(по одной на каждую грань). Проекцию будем использовать перспективную с углом обзора в 90 градусов. Для отрисовки кубической карты будем использовать 3 шейдера: вершинный, геометрический и фрагментный. Вершинный просто возвращает глобальные координаты. Геометрический шейдер делает преобразование координат из глобальных в координаты камеры и дублирует все треугольники на каждый из слоев текстуры. В случае с направленным светом, ответственным за проверку глубины было графическое API. В случае точечного освещения глубину будет вычислять фрагментный шейдер. После отрисовки FBO, мы получим текстуру, содержащую все 6 граней кубической карты теней.

В использовании карты теней есть несколько несущественных изменений. Во-первых, источник освещения больше не имеет направления, поэтому на вход вершинного шейдера передается только положение источника света. Во-вторых поиск координат пикселя на карте теней происходит с помощью вектора направления.

# 6 Отсечение геометрии

## 6.1 Мотивация

При отрисовке сцены на каждую сущность сцены делаеся много вызовов отрисовки, что, в свою очередь увеличивает время отрисовки кадра. При этом случай, когда в кадре видно все сущности сцены, очень редок. Поэтому, для оптимизации используют отсечения геометрии.

## 6.2 Отсечение геометрии вне кадра

Поле зрения камеры с перспективной проекцией является усеченной пирамидой. При этом эта успеченная пирамида состоит из 6 плоскостей. Чтобы проверить, находится ли объект в поле зрения камеры достаточно проверить, находился ли он по "верную"сторону от всех этих 6 плоскостей.

Для проверки, давайте опишем сферу вокруг объекта и проверим ориентированное растояние от центра сферы до каждой из плоскостей. Если плоскость задана как $Ax + By + Cz + D = 0$ и вектор нормали $\vec{n} = (A, B, C)$ направлен внутрь плоскости, то

$$\rho(plane, point) = \frac{A \cdot point_x + B \cdot point_y + C \cdot point_z + D}{\sqrt{A^2 + B^2 + C^2 + D^2}}$$

Тогда, чтобы объект находился внутри поля зрения камеры для всех плоскостей усеченной пирамиды должно выполняться это условие:

$$\rho(plane, sphere_c) \geq -sphere_r$$

где $sphere_c$ - центр описанной сферы, $sphere_r$ - ее радиус.

Заметим, что для оптимизации вычислений можно нормализовать уравнение плоскости, чтобы $\sqrt{A^2 + B^2 + C^2 + D^2} = 1$

Другой оптимизацией вычисления является сохранение состояния сущности. Давайте каждый кадр будем проверять только те сущности, которые уже находятся за кадром и каждые 10 кадров проверять сущности, находящиеся внутри кадра. Тогда количество вычислений на CPU уменьшится, но возрастет количество вызовов отрисовки. Эта оптимизация хорошо подойдет для устройств, где процессор является "бутылочным горлышком"

# Список литературы


[1] Репозиторий DummyEngine(дата обр. 13.05.2024)

[2] Jason Gregory. Game Engine Architecture 1 st edition, 2009

[3] Сайт библиотеки raylib(дата обр. 13.05.2024)

[4] Сайт Unreal Engine(дата обр. 13.05.2024)

[5] Обучение по Vulkan(дата обр. 13.05.2024)

[6] Спецификация Vulkan(дата обр. 13.05.2024)

[7] Yuriy O'Donnel. FrameGraph: Extensible rendering architecture in Frostbite(дата обр. 13.05.2024)

[8] Репозиторий с кодом части проекта связанной с Vulkan(дата обр. 13.05.2024)

[9] Song Ho Ahn - Transformation & Coordinate Systems(дата обр. 13.05.2024)

[10] opengl-tutorial Shadow mapping(дата обр. 13.05.2024)

[11] Michael Bunnell - NVIDIA - Shadow Map Antialiasing(дата обр. 13.05.2024)

[12] LearnOpenGL - Frustum Culling(дата обр. 13.05.2024)