

Руководство по Вулкану

Александр Оверворт

Апрель 2023 года

Содержание

Введение	6
О программе	6
Электронная книга	7
Структура учебного пособия	7
Обзор	9
Происхождение Вулкана	9
Что нужно, чтобы нарисовать треугольник	10 10
Шаг 1 инстанции и физическое устройство, подбор	10
... Шаг 2 - логического устройства и семейства очереди....	10
... Шаг 3 - поверхности окна и цепочки обмена ..	11
... Шаг 4 - вид изображения и кадровых	11
буферов Шаг 5 - рендер пасы	11
... Шаг 6 - графический конвейер	12
... Шаг 7 - команду бассейны и	12
командные буфера Шаг 8 - основной цикл	12
... Резюме	13
Понятия API	14
Соглашения о кодировании	14
... Уровни проверки	14
Среда разработки	16
Окна	16
Vulkan SDK	16
GLFW	18
GLM	18
Настройка Visual Studio	19
Linux	26
Пакеты Vulkan	26
GLFW	27
GLM	28
Компилятор шейдеров	28
Настройка проекта makefile	28

macOS	31
Vulkan SDK	31
GLFW	33
GLM	33
Настройка Xcode	33
 Рисуем треугольник	 39
Настройка	39
Базовый код	39
Экземпляр Уровни	43
проверки Физические	47
устройства и семейства очередей	58
Логическое устройство и очереди	65
Представление	68
Поверхность окна	68
Цепочка подкачки Виды	72
изображений Основы	85
графического конвейера	87
Введение	87
Шейдерные модули	90
Исправленных функций	100
Проходы рендеринга	109
Заключение	113
Рисунок	116
Фреймбуферы	116
Командные буфера	118
и представление	124
Кадры в полете	136
Отдых по цепочке подкачки	
139 Введение	139
Воссоздание цепочки подкачки	139
Неоптимальная или устаревшая цепочка подкачки	
142 Исправление взаимоблокировок	
143 Явная обработка изменений размеров	143
Минимизация обработки	145
 Вершинные буфера	 146
Описание ввода вершин	146
Введение	146
Вершинный шейдер	146
Данные о вершинах	147
Описания привязок	147
Описания атрибутов	148
Ввод вершин конвейера	150
Создание буфера вершин	150
Введение	150

Создание буфера	
150 Требования к памяти	
152 Выделение памяти	
154 Заполнение вершинного буфера	
. 155	
Привязка вершинного буфера	
156 Промежуточный буфер	158 Введение
158	
Очередь передачи	158 Создание
буфера абстрагирования	159 Использование
промежуточного буфера	160
Заключение	163
Индексный буфер	164
Введение	164
Создание индексного буфера	165
Использование индексного буфера	167

Равномерные буфера

169

Схема дескриптора и буфер	169
Введение	169
Вершинный шейдер	
170 Макет набора дескрипторов	
171 Единый буфер	173
Обновление единообразных данных	
.... 175 Пул и наборы дескрипторов	
.. 177 Введение	177
Пул дескрипторов	177 Набор
дескрипторов	178 Использование
наборов дескрипторов	181
Требования к выравниванию	182
Несколько наборов дескрипторов	185

Отображение текстур

186

Изображения	186
Введение	186
Библиотека изображений	187
загрузка изображения	188
постановка буфер	190
Текстура Изображения	190
макет переходов	195
копирование в буфер изображения	198
подготовка текстуры изображения	199
переход барьера маски	200
Очистка	202 Просмотр
изображения и сэмплер	202 Просмотр
изображения текстуры	202
Пробоотборники	205

Функция устройства анизотропии	209
Комбинированный пробоотборник изображений	
.210 Введение	210
Обновление дескрипторов	210
Текстурные координаты	213
Шейдеры	214
Буферизация глубины	219
Введение	219
TREХМЕРНАЯ геометрия	219
Изображение глубины и вид	222
Явный переход к изображению глубины	222
Этап рендеринга	227
Буфер кадров	229
Очистить значения	229
Глубина и состояние трафарета	230
Изменение размера окна	231
Загружаемые модели	233
Введение	233
Библиотека	233
Пример сетки	234
Загрузка вершин и индексов	235
Дедупликация вершин	239
Генерация Mip-карт	242
Введение	242 Создание изображений
Mip-карт	243 Генерация линейной фильтрации
Сэмплер	244 Поддержка
Множественная дискретизация	254
Введение	254 Получение доступного количества выборок
Настройка цели рендеринга	257
Добавление новых вложений	259
Улучшение качества	262
Заключение	263
Вычислительный Шейдер	265
Введение	265
Преимущества	265
Конвейера Vulkan	266
Пример	266
Манипулирование данными	268

Объекты буфера хранения шейдеров (SSBO)	268
Образы хранилищ	269
Семейства вычислительных очередей	
270 Этап создания вычислительных шейдеров	
271 Загрузка вычислительных шейдеров	
271 Подготовка буферов хранения шейдеров	272
Дескрипторы	273
Вычислительные конвейеры	276
Вычислительное пространство	277
Вычислительные шейдеры	278
Выполнение вычислительных команд	280
Отправка	280
Отправка работы	281
Синхронизация графики и вычислений	281
Построение системы частиц	284
Заключение	285
	286
ЧАСТО ЗАДАВАЕМЫЕ ВОПРОСЫ	
Я получаю ошибку нарушения доступа на уровне проверки ядра	286
Я не вижу никаких сообщений от уровней проверки / Validation layers недоступны	286
vkCreateSwapchainKHR вызывает ошибку в SteamOverlayVulkan-Layer64.dll	286
Сбой vkCreateInstance с ошибкой VK_ERROR_INCOMPATIBLE_DRIVER	287

Введение

О нас

В этом руководстве вы познакомитесь с основами использования Vulkan graphics and compute API. Vulkan - это новый API от Khronos group (известный по OpenGL), который обеспечивает гораздо лучшую абстракцию современных видеокарт. Этот новый интерфейс позволяет вам лучше описать, что намеревается делать ваше приложение, что может привести к повышению производительности и менее неожиданному поведению драйвера по сравнению с существующими API, такими как OpenGL и Direct3D. Идеи Vulkan аналогичны идеям Direct3D 12 и Metal, но Vulkan обладает преимуществом в том, что он полностью кроссплатформенный и позволяет разрабатывать для Windows, Linux и Android одновременно . Однако цена, которую вы платите за эти преимущества, заключается в том, что вам придется работать со значительно более подробным API. Каждая деталь, связанная с графическим API, должна быть настроена вашим приложением с нуля, включая создание начального буфера кадров и управление памятью для таких объектов, как буфера и изображения текстур. Графический драйвер будет намного меньше занимать ресурсов, а это означает, что вам придется проделать больше работы в вашем приложении, чтобы обеспечить правильное поведение. Вывод здесь заключается в том, что Vulkan предназначен не для всех. Он предназначен для программистов, которые с энтузиазмом относятся к высокопроизводительной компьютерной графике, и готовы внести свой вклад. Если вас больше интересует разработка игр, а не компьютерная графика, то вы можете захотеть придерживаться OpenGL или Direct3D, которые в ближайшее время не будут устаревать в пользу Vulkan. Другой альтернативой является использование движка, такого как Unreal Engine или Unity, который будет способен использовать Vulkan, предоставляя вам API гораздо более высокого уровня. Покончив с этим, давайте рассмотрим некоторые необходимые условия для выполнения этого руководства:

- Видеокарта и драйвер, совместимые с Vulkan (NVIDIA, AMD, Intel, Apple Silicon (или Apple M1))
- Опыт работы с C++ (знакомство с RAII, списками инициализаторов) • Компилятор с достойной поддержкой функций C++ 17 (Visual Studio 2017+, GCC 7+ или Clang 5+) • Некоторый опыт работы с компьютерной графикой 3D

Это руководство не предполагает знания концепций OpenGL или Direct3D, но оно требует от вас знания основ компьютерной 3D-графики. Например, она не объясняет математику, лежащую в основе перспективного проектирования. Ознакомьтесь с этой онлайн-книгой, в которой рассказывается о концепциях компьютерной графики. Некоторые другие замечательные ресурсы по компьютерной графике:

- Трассировка лучей за один уикенд • Книга рендеринга на физической основе
- Vulkan используется в реальном движке Quake and DOOM с открытым исходным кодом

3

Вы можете использовать C вместо C++, если хотите, но вам придется использовать другую библиотеку линейной алгебры, и вы будете предоставлены сами себе в плане структурирования кода. Мы будем использовать функции C++, такие как классы и RAII, для организации логики и времени жизни ресурсов. Для Rust разработчиков также доступна альтернативная версия этого руководства. Чтобы разработчикам, использующим другие языки программирования, было проще следовать за ними, а также получить некоторый опыт работы с базовым API, мы будем использовать оригинальный C API для работы с Vulkan. Однако, если вы используете C++, вы можете предпочесть использовать более новые привязки Vulkan-Hpp, которые абстрагируют часть грязной работы и помогают предотвратить определенные классы ошибок.

Электронная книга

Если вы предпочитаете читать это руководство в виде электронной книги, то вы можете скачать версию EPUB или PDF здесь:

- EPUB
- PDF

Структура руководства

Мы начнем с обзора того, как работает Vulkan, и той работы, которую нам придется проделать, чтобы получить первый треугольник на экране. Назначение всех более мелких шагов станет более понятным после того, как вы поймете их основную роль во всей картине. Далее мы настроим среду разработки с помощью Vulkan SDK, библиотеки GLM для операций с линейной алгеброй и GLFW для создания окон. В руководстве будет рассказано, как настроить их в Windows с помощью Visual Studio и в Ubuntu Linux с помощью GCC. После этого мы внедрим все основные компоненты программы Vulkan, которые необходимы для визуализации вашего первого треугольника. Каждая глава будет примерно соответствовать следующей структуре:

- Представьте новую концепцию и ее назначение • Используйте все соответствующие вызовы API для ее интеграции в вашу программу

• Абстрагируйте его части во вспомогательные функции.

Хотя каждая глава написана как продолжение предыдущей, также возможно читать главы как отдельные статьи, рассказывающие об определенной функции Vulkan . Это означает, что сайт также полезен в качестве справочного материала. Все функции и типы Vulkan связаны со спецификацией, поэтому вы можете щелкнуть по ним, чтобы узнать больше. Vulkan - это очень новый API, поэтому в самой спецификации могут быть некоторые недостатки. Мы рекомендуем вам отправлять отзывы в этот репозиторий Khronos . Как упоминалось ранее, Vulkan API имеет довольно подробный API со множеством параметров, дающих вам максимальный контроль над графическим оборудованием. Это приводит к тому, что базовые операции, такие как создание текстуры, требуют большого количества шагов, которые приходится повторять каждый раз. Поэтому на протяжении всего урока мы будем создавать нашу собственную коллекцию вспомогательных функций. Каждая глава также завершается ссылкой на полный список кода до этого пункта. Вы можете обратиться к нему, если у вас есть какие-либо сомнения относительно структуры кода, или если вы столкнулись с ошибкой и хотите сравнить. Все файлы кода были протестированы на видеокартах нескольких производителей для проверки корректности. В конце каждой главы также есть раздел комментариев, где вы можете задать любые вопросы, которые имеют отношение к конкретной теме. Пожалуйста, укажите вашу платформу, версию драйвера, исходный код, ожидаемое и фактическое поведение, чтобы помочь нам помочь вам. Это руководство предназначено для сообщества. Vulkan по-прежнему является очень новым API, и лучшие практики на самом деле еще не были установлены. Если у вас есть какие-либо отзывы о руководстве и самом сайте, пожалуйста, не стесняйтесь отправлять вопрос или запрос на извлечение в репозиторий GitHub. Вы можете Смотреть репозиторий должен получать уведомления об обновлениях руководства. После того, как вы пройдете ритуал рисования вашего самого первого треугольника на экране с питанием от Vulkan , мы начнем расширять программу, включив в нее линейные преобразования, текстуры и 3D-модели. Если вы раньше играли с графическими API-интерфейсами, то знаете, что может быть много шагов, пока на экране не появится первая геометрия. В Vulkan есть много таких начальных шагов, но вы увидите, что каждый из них в отдельности прост для понимания и не кажется излишним. Также важно иметь в виду, что как только у вас получится этот скучный на вид треугольник, рисование полностью текстурированных 3D-моделей не потребует много дополнительной работы, и каждый шаг за пределы этого пункта приносит гораздо больше пользы. Если вы столкнетесь с какими-либо проблемами при выполнении руководства, то сначала ознакомьтесь с часто задаваемыми вопросами, чтобы узнать, указана ли там уже ваша проблема и ее решение. Если вы все еще застряли после этого, то не стесняйтесь обращаться за помощью в разделе комментариев к ближайшей по теме главе. Готовы погрузиться в будущее высокопроизводительных графических API? Поехали!

Обзор

Эта глава начнется с ознакомления с Vulkan и проблемами, которые он решает. После этого мы рассмотрим ингредиенты, необходимые для первого треугольника. Это даст вам общую картину, в которой вы сможете разместить каждую из последующих глав. В заключение мы рассмотрим структуру Vulkan API и общие шаблоны использования.

Происхождение Vulkan

Как и предыдущие графические API, Vulkan разработан как кроссплатформенный абстракция поверх графических процессоров. Проблема большинства этих API заключается в том, что в эпоху, в которую они были разработаны, графическое оборудование было в основном ограничено настраиваемой фиксированной функциональностью. Программистам приходилось предоставлять данные вершин в стандартном формате, и они зависели от производителей графических процессоров в отношении параметров освещения и затенения. По мере развития архитектур видеокарт они начали предлагать все больше и больше программируемой функциональности. Всю эту новую функциональность нужно было каким-то образом интегрировать с существующими API. Это привело к далеко не идеальным абстракциям и большому количеству догадок на стороне графического драйвера, чтобы сопоставить возможности программиста с современными графическими архитектурами. Вот почему существует так много обновлений драйверов для повышения производительности в играх, иногда за счет значительных потерь. Из-за сложности этих драйверов разработчикам приложений также необходимо иметь дело с несоответствиями между поставщиками, такими как синтаксис, который принят для шейдеров. Помимо этих новых функций, в последнее десятилетие также наблюдался приток мобильных устройств с мощным графическим оборудованием. Эти мобильные графические процессоры имеют различную архитектуру в зависимости от их требований к энергопотреблению и пространству. Одним из таких примеров является плиточный рендеринг, который выигрывает от повышения производительности за счет предоставления программисту большего контроля над этой функциональностью. Другим ограничением, обусловленным возрастом этих API, является ограниченная поддержка многопоточности, что может привести к узкому месту на стороне процессора. Vulkan решает эти проблемы, разрабатываясь с нуля для современных графических архитектур. Это снижает нагрузку на драйвер, позволяя программистам четко указывать свои намерения, используя более подробный API, и позволяет нескольким потокам

создавайте и отправляйте команды параллельно. Это уменьшает несогласованность в компиляции шейдеров за счет переключения на стандартизованный формат байт-кода с помощью единого компилятора. Наконец, он подтверждает универсальные вычислительные возможности современных видеокарт, объединяя графику и вычислительные функции в единый API.

Что нужно, чтобы нарисовать треугольник

Теперь мы рассмотрим обзор всех шагов, необходимых для визуализации треугольника в хорошо работающей программе Vulkan. Все представленные здесь концепции будут подробно рассмотрены в следующих главах. Это просто для того, чтобы дать вам общую картину, с которой можно соотнести все отдельные компоненты.

Шаг 1 - Выбор экземпляра и физического устройства

Приложение Vulkan запускается с настройкой Vulkan API через а `vkInstance`. Экземпляр создается путем описания вашего приложения и любых расширений API, которые вы будете использовать. После создания экземпляра вы можете запросить оборудование, поддерживаемое портированным Vulkan, и выбрать одно или несколько `VkPhysicalDevice`-ов для использования в операциях. Вы можете запросить такие параметры, как размер видеопамяти и возможности устройства, чтобы выбрать нужные устройства, например, предпочесть использовать выделенные видеокарты.

Шаг 2 - Семейства логических устройств и очередей.

После выбора подходящего аппаратного устройства для использования вам необходимо создать `VkDevice` (логическое устройство), где вы более конкретно опишите, какие функции `VkPhysicalDevice`- вы будете использовать, такие как рендеринг с несколькими видовыми экранами и 64-разрядные функции с плавающей точкой. Вам также необходимо указать, какие семейства очередей вы хотели бы использовать. Большинство операций, выполняемых с помощью Vulkan, таких как команды рисования и операции с памятью, выполняются асинхронно путем отправки их в `VkQueue`. Очереди выделяются из семейств очередей, где каждое семейство очередей поддерживает определенный набор операций в своих очередях. Например, могут существовать отдельные семейства очередей для операций с графикой, вычислениями и передачей памяти. Доступность семейств очередей также может использоваться в качестве отличительного фактора при выборе физического устройства. Устройство с поддержкой Vulkan может не предлагать никаких графических функций, однако все видеокарты с поддержкой Vulkan на сегодняшний день, как правило, поддерживают все интересующие нас операции с очередью.

Шаг 3 - Поверхность окна и цепочка подкачек

Если вас не интересует только закадровый рендеринг, вам нужно будет создать окно для представления отрисованных изображений. Windows можно создавать с помощью собственных платформенных API или библиотек, таких как GLFW и SDL. Мы будем использовать GLFW в этом руководстве, но подробнее об этом в следующей главе.

Нам нужны еще два компонента для реального рендеринга в окне: поверхность окна (`VkSurfaceKHR`) и цепочка подкачки (`VkSwapchainKHR`). Обратите внимание на `KHR` постфиксный, который означает, что эти объекты являются частью расширения Vulkan. Сам по себе Vulkan API полностью не зависит от платформы, вот почему нам нужно использовать стандартизированное расширение WSI (Window System Interface) для взаимодействия с оконным менеджером. Surface представляет собой кроссплатформенную абстракцию поверх Windows для визуализации и обычно создается путем предоставления ссылки на собственный дескриптор окна, например, в Windows. К счастью, библиотека GLFW имеет встроенную функцию для обработки деталей, зависящих от конкретной платформы.

Цепочка обмена - это набор целей рендеринга. Его основная цель - гарантировать, что изображение, которое мы в данный момент визуализируем, отличается от того, которое в данный момент находится на экране. Это важно, чтобы убедиться, что отображаются только полные изображения. Каждый раз, когда мы хотим нарисовать рамку, мы должны попросить цепочку обмена предоставить нам изображение для рендеринга. Когда мы заканчиваем рисовать рамку, изображение возвращается в цепочку подкачки, чтобы в какой-то момент оно было представлено на экране.

Количество целей рендеринга и условий для вывода готовых изображений на экран зависит от текущего режима. Распространенными существующими режимами являются двойная буферизация (vsync) и тройная буферизация. Мы рассмотрим это в главе о создании цепочки обмена. Некоторые платформы позволяют выполнять визуализацию непосредственно на дисплее без взаимодействия с каким-либо оконным менеджером через `VK_KHR_display_swapchain` который отображает весь экран и может быть использован напрямую для визуализации в собственного оконного менеджера.

Шаг 4 - Просмотр изображений и фреймбуфера

Чтобы отрисовать изображение, полученное из цепочки обмена, мы должны обернуть его в `VkImageView` и `VkFramebuffer`. Вид изображения ссылается на определенную часть используемого изображения, а фреймбуфер ссылается на виды изображения, которые должны использоваться для определения цвета, глубины и целевого трафарета. Поскольку в цепочке подкачки может быть много разных изображений, мы предварительно создадим вид изображения и фреймбуфер для каждого из них и выберем нужное во время рисования.

Шаг 5 - Этапы рендеринга

Этапы рендеринга в Vulkan описывают тип изображений, которые используются во время операций рендеринга, то, как они будут использоваться, и как следует обрабатывать их содержимое. В нашем первоначальном приложении для рендеринга треугольника мы сообщим Vulkan, что будем использовать одно изображение в качестве целевого цвета и что мы хотим, чтобы оно было очищено до сплошного цвета прямо перед операцией рисования. В то время как проходит рендеринг описывает только тип изображений, `VkFramebuffer` фактически привязывает определенные изображения к этим слотам.

Шаг 6 - Графический конвейер

Графический конвейер в Vulkan настраивается путем создания объекта VkPipeline. Он описывает настраиваемое состояние видеокарты, такое как размер области просмотра и работа буфера глубины, а также программируемое состояние с использованием объектов VkShaderModule . Объекты VkShaderModule создаются из байтового кода шейдера. Драйверу также необходимо знать, какие цели рендеринга будут использоваться в конвейере, которые мы указываем, ссылаясь на этап рендеринга.

Одной из наиболее отличительных особенностей Vulkan по сравнению с существующими API является то, что почти вся конфигурация графического конвейера должна быть настроена заранее. Это означает, что если вы хотите переключиться на другой шейдер или немного изменить расположение вершин, то вам нужно полностью воссоздать графический конвейер. Это означает, что вам придется заранее создать множество объектов VkPipeline для всех различных комбинаций, необходимых вам для ваших операций рендеринга. Только некоторые базовые настройки, такие как размер области просмотра и четкий цвет, могут быть изменены динамически. Все состояния также должны быть описаны явно, например, нет состояния наложения цветов по умолчанию.

Хорошой новостью является то, что, поскольку вы выполняете эквивалент компиляции с опережением времени по сравнению с компиляцией точно в срок, появляется больше возможностей оптимизации настроек для драйвера, а производительность среды выполнения более предсказуема, потому что большие изменения состояния, такие как переключение на другой графический конвейер, делаются очень явными.

Шаг 7 - Пулы команд и командные буферы

Как упоминалось ранее, многие операции в Vulkan, которые мы хотим выполнить, например, операции рисования, должны быть отправлены в очередь. Сначала эти операции необходимо записать в VkCommandBuffer, прежде чем их можно будет отправить. Эти командные буферы выделяются из VkCommandPool – это связано с определенным семейством очередей. Чтобы нарисовать простой треугольник, нам нужно записать командный буфер со следующими операциями:

- Начать прохождение рендеринга
- Привязать графический конвейер
- Нарисовать 3 вершины
- Завершить прохождение рендеринга

Поскольку изображение в фреймбуфере зависит от того, какое конкретное изображение даст нам цепочка подкачки , нам нужно записать буфер команд для каждого возможного изображения и выбрать правильное во время рисования. Альтернативой может быть повторная запись командного буфера в каждом кадре, что не так эффективно.

Шаг 8 - Основной цикл.

Теперь, когда команды рисования перенесены в буфер команд, основной цикл довольно прост. Сначала мы получаем изображение из

цепочка с `vkAcquireNextImageKHR`. Можно выбрать упражнения или в буфер команд для изображения и выполнить ее с `vkQueueSubmit`. Наконец, мы возвращаем изображение в цепочку подкачки для представления на экран с помощью `vkQueuePresentKHR`. Операции, отправляемые в очереди, выполняются асинхронно. Поэтому для обеспечения правильного порядка выполнения нам приходится использовать объекты синхронизации, такие как семафоры. Выполнение буфера команды рисования должно быть настроено так, чтобы ожидать завершения получения изображения, в противном случае может случиться так, что мы запустим повторную обработку переход к изображению, которое все еще считывается для представления на экране. Вызов `vkQueuePresentKHR`, в свою очередь, необходимо дождаться завершения рендеринга, для чего мы будем использовать второй семафор, который сигнализируется после завершения рендеринга.

Краткие сведения

Этот обзорный тур должен дать вам общее представление о предстоящей работе по рисованию первого треугольника. Реальная программа содержит больше шагов, таких как выделение буферов вершин, создание однородных буферов и загрузка изображений текстур, которые будут рассмотрены в последующих главах, но мы начнем с простого, потому что у Vulkan и так достаточно сложная кривая обучения. Обратите внимание, что мы немного обманем, изначально вложив координаты вершин в вершинный шейдер вместо использования вершинного буфера. Это потому, что для управления буферами вершин требуется некоторое знакомство с буферами команд. Короче говоря, чтобы нарисовать первый треугольник, нам нужно:

- Создайте `VkInstance` • Выберите поддерживаемую видеокарту (`VkPhysicalDevice`) • Создайте `VkDevice` и `VkQueue` для рисования и презентации • Создайте окно, оконную поверхность и цепочку подкачки • Перенесите изображения цепочки подкачки в `VkImageView` • Создайте этап рендеринга, который определяет цели рендеринга и использование • Создайте фреймбуфера для этапа рендеринга
 - Настройте графический конвейер • Выделите и запишите командный буфер с командами рисования для каждого возможной цепочки подкачки изображения • Рисуйте рамки, получая изображения, отправляя правильную команду рисования `buffer` и возвращая изображения обратно в цепочку подкачки
- Это много шагов, но цель каждого отдельного шага будет сделана очень простой и понятной в следующих главах. Если вы не уверены в отношении отдельного шага по сравнению со всей программой, вам следует вернуться к этой главе.

Концепции API

Эта глава завершится кратким обзором того, как устроен Vulkan API структурирован на более низком уровне.

Соглашения о кодировании.

Все функции, перечисления и структуры Vulkan определены в `vulkan.h` заголовок, который включен в Vulkan SDK, разработанный LunarG. Мы рассмотрим установку этого SDK в следующей главе. Функции имеют нижний регистр `vk` префикс, такие типы, как перечисления и структуры, имеют а `vk` значения префикса и перечисления имеют префикс `vk_`. API в значительной степени использует структуры для предоставления параметров функциям. Например, создание объекта обычно выполняется по следующему

```
шаблону: VkXXXXCreateInfo CreateInfo{};  
1 CreateInfo.sType = VK_STRUCTURE_TYPE_XXX_CREATE_INFO;  
2 CreateInfo.pNext = nullptr;  
3 CreateInfo.foo = ...;  
4 CreateInfo.bar = ...;  
5  
6  
7 Объект VkXXXX;  
8 если (vkCreateXXX(&CreateInfo, nullptr, &object) != VK_SUCCESS) {  
9     std::cerr << "не удалось создать объект" <<  
10    std::endl; возвращает false;  
11 }
```

Многие структуры в Vulkan требуют, чтобы вы явно указывали тип структуры Участник Р. В `pNext` элемент может указывать на структуру расширения и всегда будет иметь значение `nullptr` в этом руководстве. Функции, которые создают или уничтожают объект, будут иметь параметр `VkAllocationCallbacks`, который позволяет вам использовать пользовательский распределитель для памяти драйвера, который также будет оставлен в руководстве.

`nullptr` В ЭТОМ

Уровни проверки

Как упоминалось ранее, Vulkan рассчитан на высокую производительность и низкие затраты драйверов. Поэтому по умолчанию он будет включать в себя очень ограниченные возможности проверки на ошибки и отладки. Драйвер часто выходит из строя вместо того, чтобы возвращать код ошибки, если вы делаете что-то неправильно, или, что еще хуже, кажется, что он работает на вашей видеокарте и полностью выходит из строя на других.

Vulkan позволяет вам проводить обширные проверки с помощью функции, известной как *уровни проверки*. Уровни проверки - это фрагменты кода, которые могут быть вставлены между API и графическим драйвером для выполнения таких действий, как выполнение дополнительных проверок функциональных параметров и отслеживание проблем с управлением памятью. Приятно то, что вы можете включить их во время разработки, а затем полностью отключить их при выпуске вашего приложения с нулевыми накладными расходами. Любой может написать свои собственные уровни проверки, но Vulkan SDK от LunarG предоставляет стандартный набор уровней проверки, которые мы будем использовать в этом руководстве. Вам также необходимо зарегистрировать функцию обратного вызова для получения отладочных сообщений от слоев. Поскольку Vulkan настолько четко описывает каждую операцию, а уровни проверки настолько обширны, на самом деле может быть намного проще выяснить, почему ваш экран черный по сравнению с OpenGL и Direct3D! Остался только один шаг, прежде чем мы начнем писать код, и это настройка среды разработки.

Среда разработки

В этой главе мы настроим вашу среду для разработки приложений Vulkan и установим несколько полезных библиотек. Все инструменты, которые мы будем использовать, за исключением компилятора, совместимы с Windows, Linux и macOS, но шаги по их установке немного отличаются, поэтому они описаны здесь отдельно.

Windows

Если вы разрабатываете для Windows, то я предположу, что вы используете Visual Studio для компиляции своего кода. Для полной поддержки C++ 17 вам необходимо использовать либо Visual Studio 2017, либо 2019. Описанные ниже шаги были написаны для VS 2017.

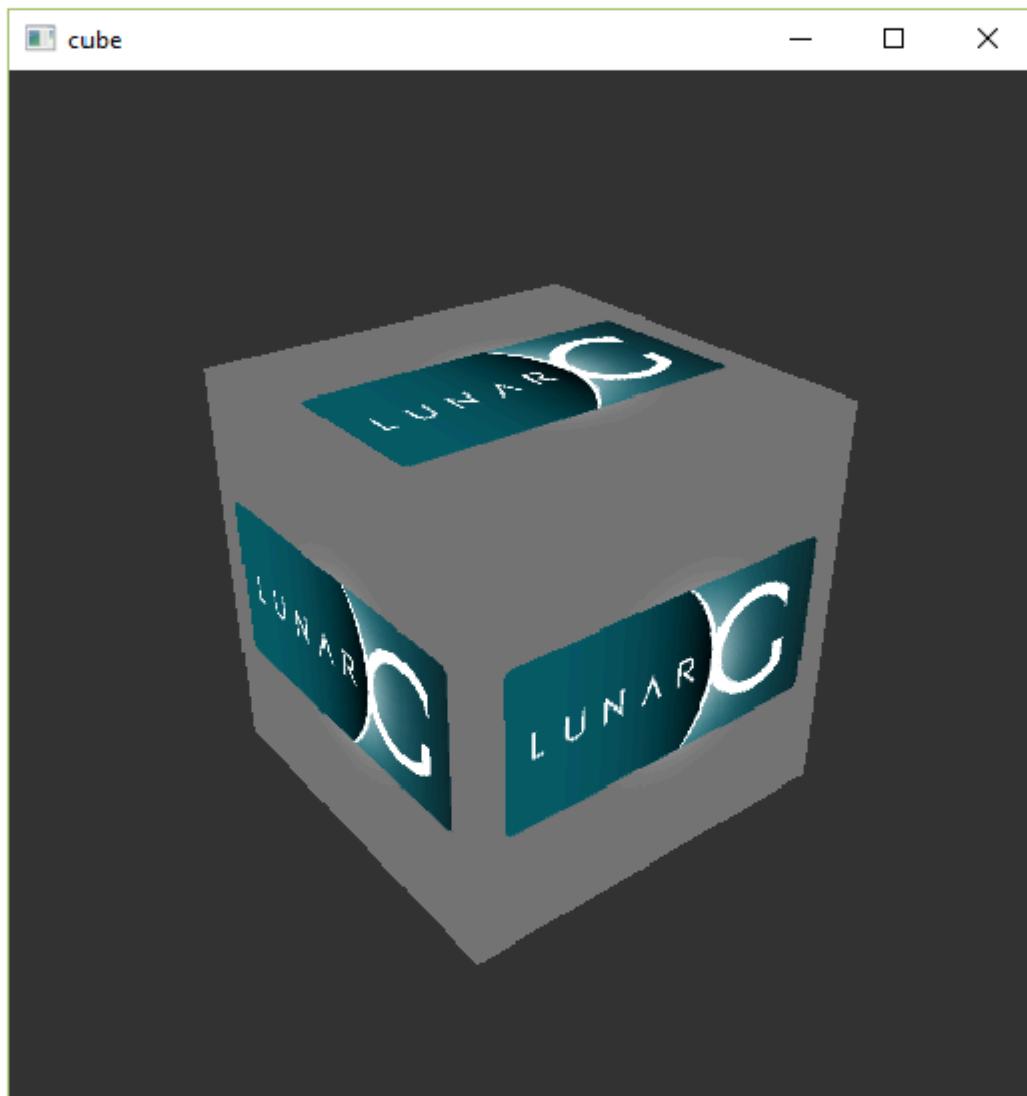
Vulkan SDK

Наиболее важным компонентом, который вам понадобится для разработки приложений Vulkan, является SDK. Он включает заголовки, стандартные уровни проверки, инструменты отладки и загрузчик для функций Vulkan. Загрузчик ищет функции в драйвере во время выполнения, аналогично GLEW для OpenGL - если вы с этим знакомы. SDK можно загрузить с веб-сайта LunarG, используя кнопки в нижней части страницы. Вам не нужно создавать учетную запись, но это даст вам доступ к некоторой дополнительной документации, которая может быть вам полезна.



Продолжайте установку и обратите внимание на место установки SDK. Первое, что мы сделаем, это убедимся, что ваша видеокарта и драйвер правильно поддерживают Vulkan. Перейдите в каталог, в котором вы установили SDK,

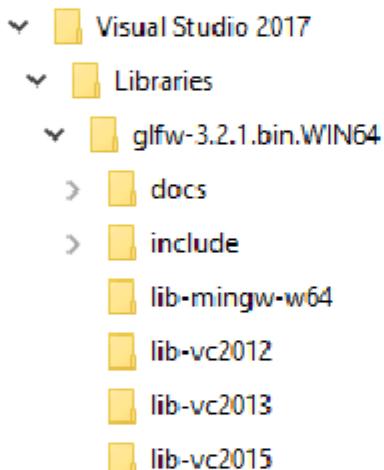
откройте `bin` каталог и запустите `vkcube.exe` ДЕМОНСТРАЦИЯ.
Вы должны увидеть следующее:



Если вы получаете сообщение об ошибке, убедитесь, что ваши драйверы обновлены, включают среду выполнения Vulkan и что ваша видеокарта поддерживается. Смотрите в вводной главе ссылки на драйверы основных производителей. В этом каталоге есть еще одна программа, которая будет полезна для разработки. The `vkdevinfo` программы будут использоваться для компиляции шейдеров из удобочитаемого GLSL в байт-код. Об этом мы поговорим в глубины шейдере модули главе. В `bin` каталог также содержит двоичные файлы загрузчика Vulkan и уровни проверки, в то время как каталог `lib` содержит библиотеки. Наконец, есть `include` каталог, содержащий заголовки Vulkan. Не стесняйтесь изучить другие файлы, но они нам не понадобятся в этом руководстве.

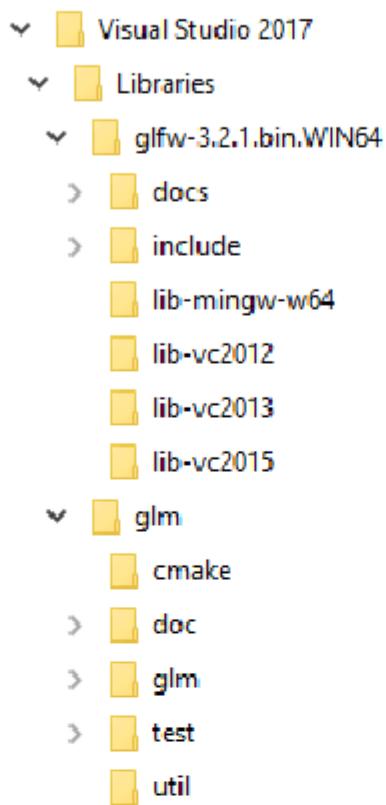
GLFW

Как упоминалось ранее, Vulkan сам по себе не зависит от платформы API и не включает инструменты для создания окна для отображения отрисованных результатов. Чтобы воспользоваться кроссплатформенными преимуществами Vulkan и избежать ужасов Win32, мы будем использовать библиотеку GLFW для создания окна, поддерживающего Windows, Linux и macOS. Для этой цели доступны другие библиотеки, например SDL, но преимущество GLFW в том, что он также абстрагирует от некоторых других, специфичных для платформы функций в Vulkan, помимо простого создания окон. С последней версией GLFW вы можете ознакомиться на официальном сайте. В этом руководстве мы будем использовать 64-разрядные двоичные файлы, но вы, конечно, также можете выбрать сборку в 32-разрядном режиме. В этом случае обязательно установите ссылку на двоичные файлы Vulkan SDK в каталоге `Lib\z\2` вместо `Lib\z`. После загрузки распакуйте архив в удобное место. Я решил создать каталог `Libraries` в каталоге Visual Studio в разделе `documents`.



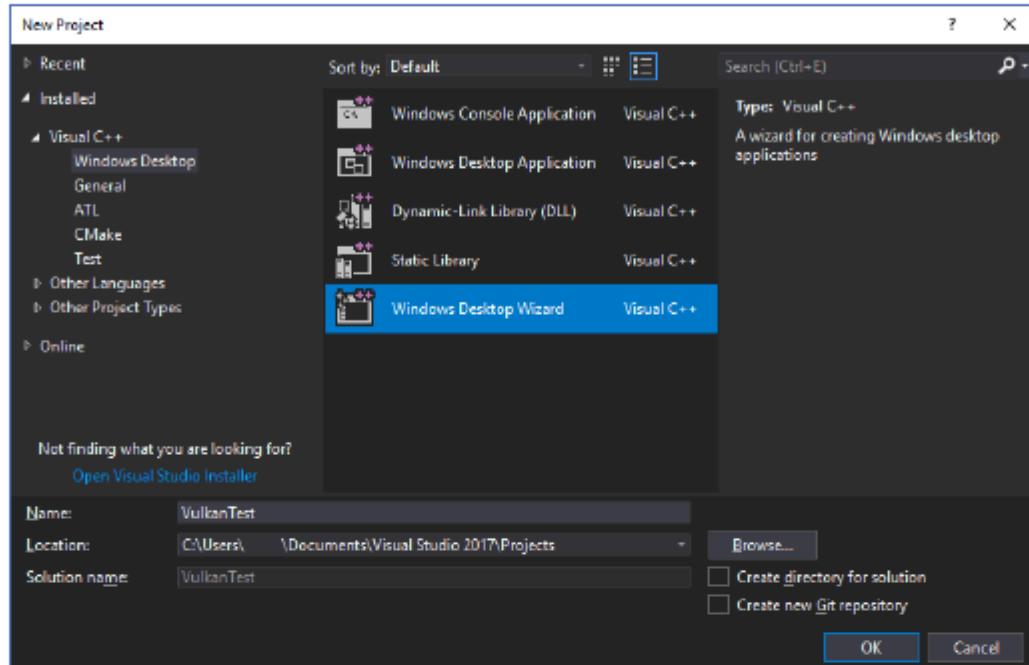
GLM

В отличие от DirectX 12 и Vulkan, который не включает библиотека для линейной алгебры операций, так что придется скачать один. GLM - это хорошая библиотека, которая предназначена для использования с графическими API, а также обычно используется с OpenGL. GLM - это библиотека только для заголовков, поэтому просто скачайте последнюю версию и храните ее в удобном месте. Теперь у вас должна быть структура каталогов, подобная следующей:

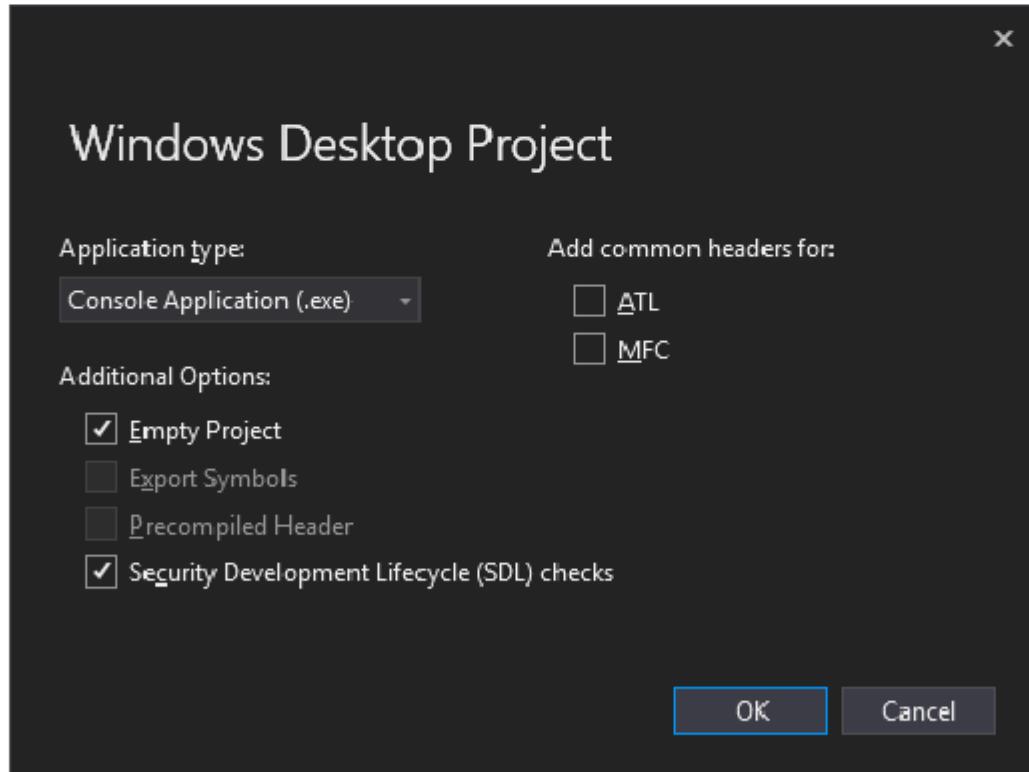


Настройка Visual Studio

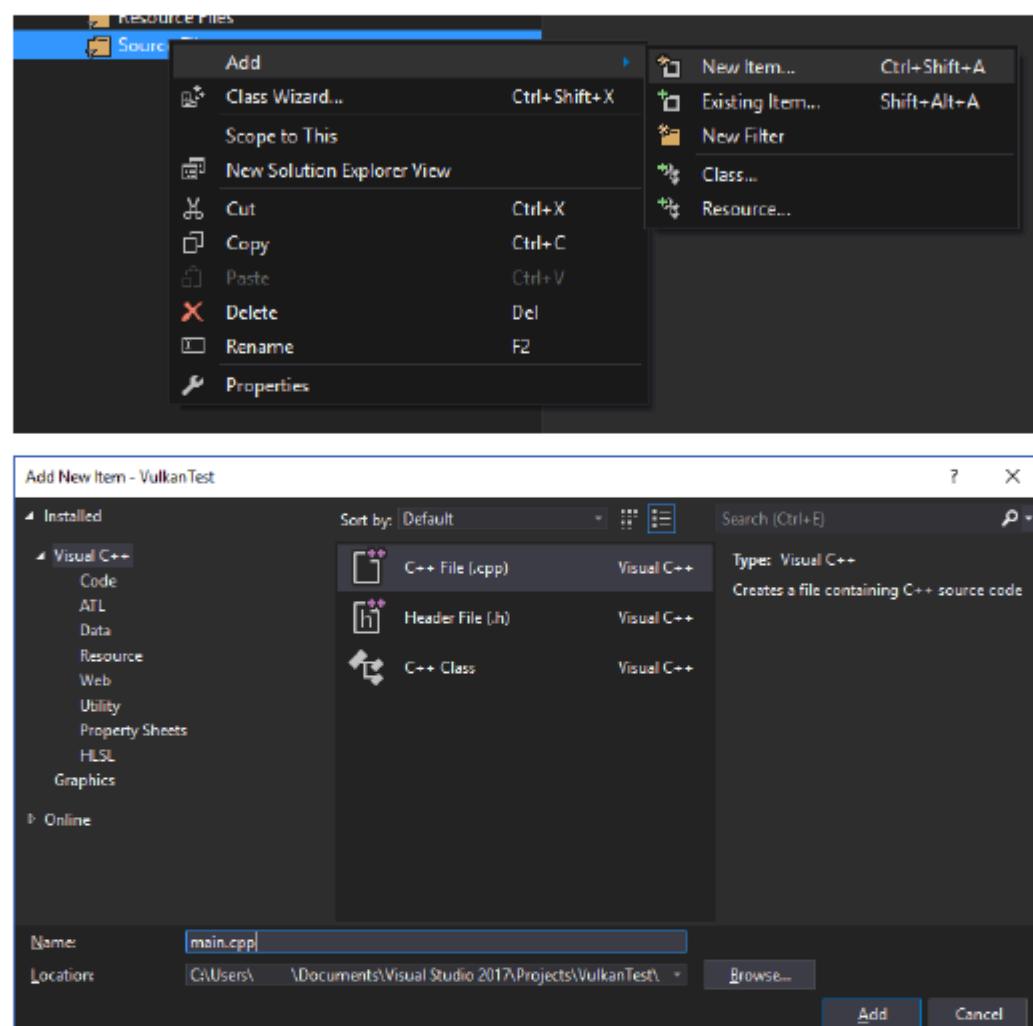
Теперь, когда вы установили все зависимости, мы можем настроить базовый проект Visual Studio для Vulkan и написать немного кода, чтобы убедиться, что все работает. Запустите Visual Studio и создайте новый мастер создания рабочего стола Windows проект, введя имя и нажав OK .



Убедитесь, что Консольное Приложение (.exe-файл) выбран тип заявки чтобы у нас было место для печати отладочных сообщений, и установите флагок Пустой проект чтобы Visual Studio не добавляла шаблонный код.



Пресса **OK** – создать проект и добавить исходный файл C++. Вы уже должны знать, как это сделать, но шаги приведены здесь для полноты картины.



Теперь добавьте следующий код в файл. Не беспокойтесь о том, чтобы попытаться понять это прямо сейчас; мы просто убеждаемся, что вы можете компилировать и запускать приложения Vulkan. В следующей главе мы начнем с

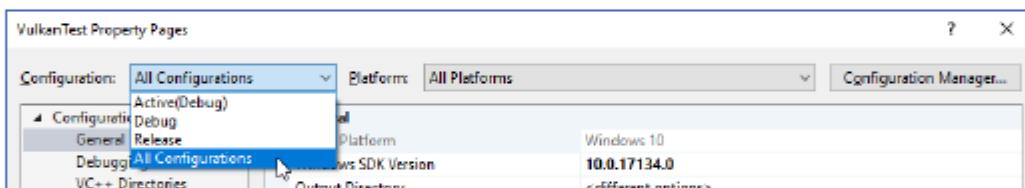
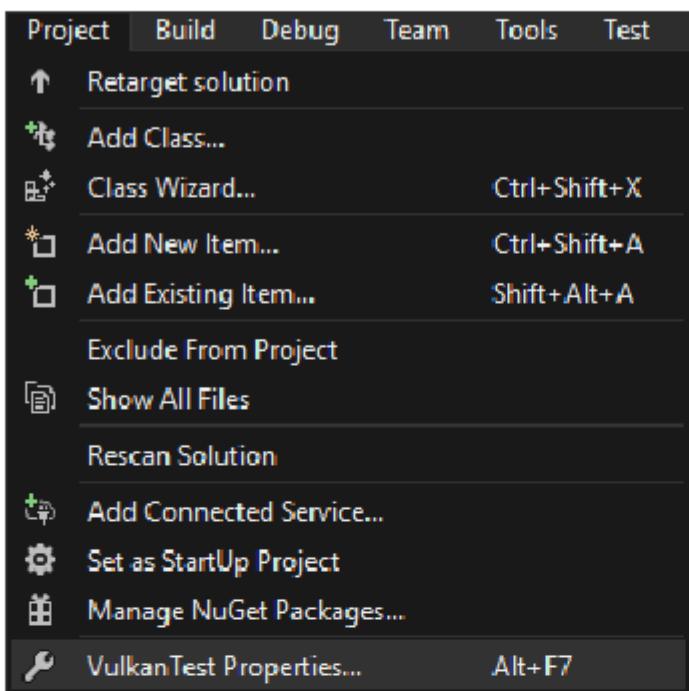
```
1 Нуля. # определение GLFW_INCLUDE_VULKAN
2 # включить <GLFW/glfw3.h>
3 # определить GLM_FORCE_RADIANS
4 # определить GLM_FORCE_DEPTH_ZERO_TO_ONE
5 # включить <glm/vec4.hpp>
6 # включить <glm/mat4x4.hpp>
7 # включить <iostream>
```

```

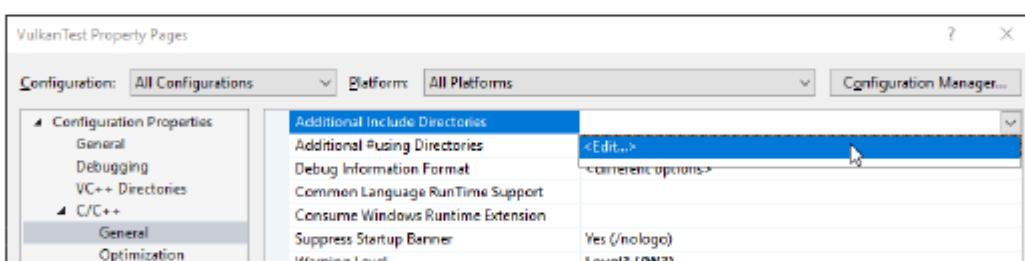
11     Инт    главный()   {
12         glfwInit();
13
14         glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
15         GLFWwindow * window = glfwCreateWindow(800, 600, "Окно вулкана",
16                                         nullptr, nullptr);
17
18         uint32_t extensionCount = 0;
19         vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount,
20                                         nullptr);
21
22         glm::mat4 matrix;
23         glm::vec4 vec;
24         авто тест = матрица * век;
25
26         пока (!glfwWindowShouldClose(окно)) {
27             glfwPollEvents();
28         }
29
30         glfwDestroyWindow(окно);
31
32         glfwTerminate();
33
34         Возврат 0 ;
35     }

```

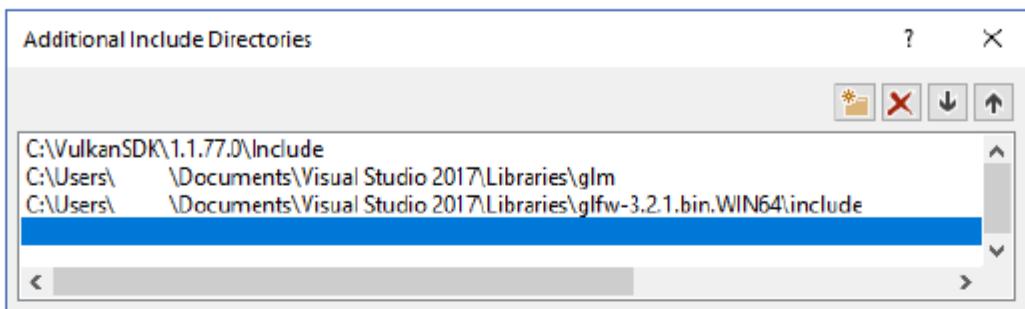
Давайте теперь настроим проект, чтобы избавиться от ошибок. Откройте диалоговое окно "Параметры проекта" и убедитесь, что выбрано значение "Все конфигурации" , поскольку большинство настроек применяются как к режиму Debug , так и к режиму Release .



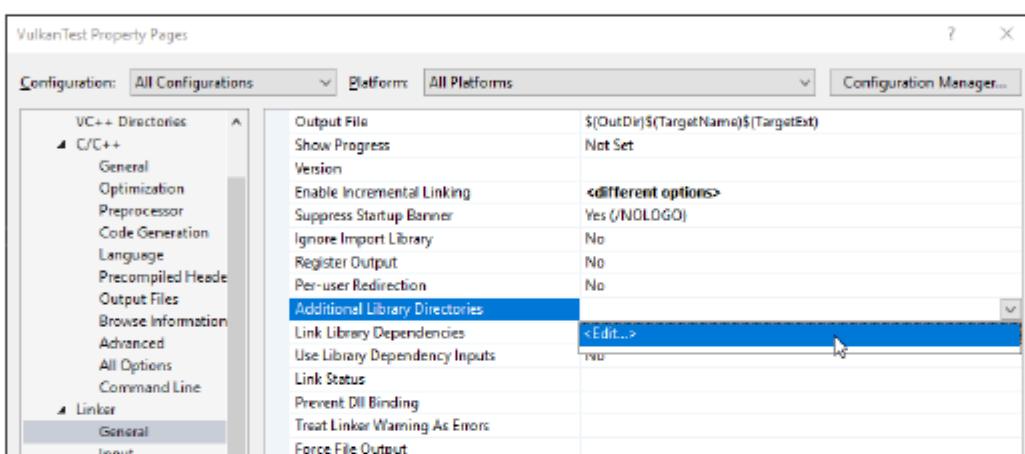
Перейти к ++ -> Общие -> Дополнительные каталоги включения и нажмите <Редактировать ...> в раскрывающемся списке.



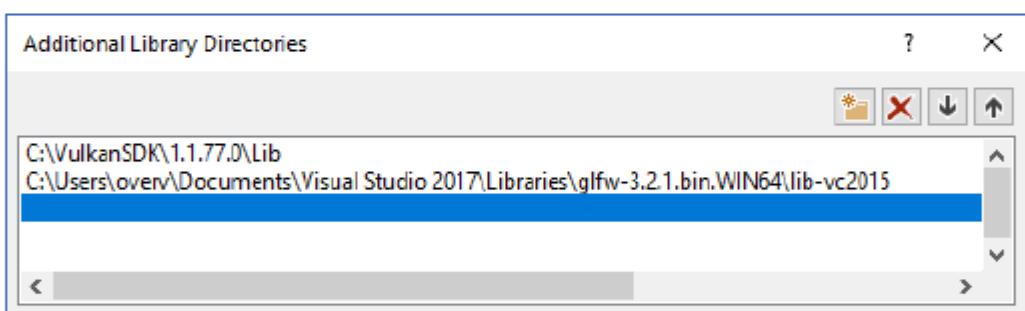
Добавьте каталоги заголовков для Vulkan, GLFW и GLM:



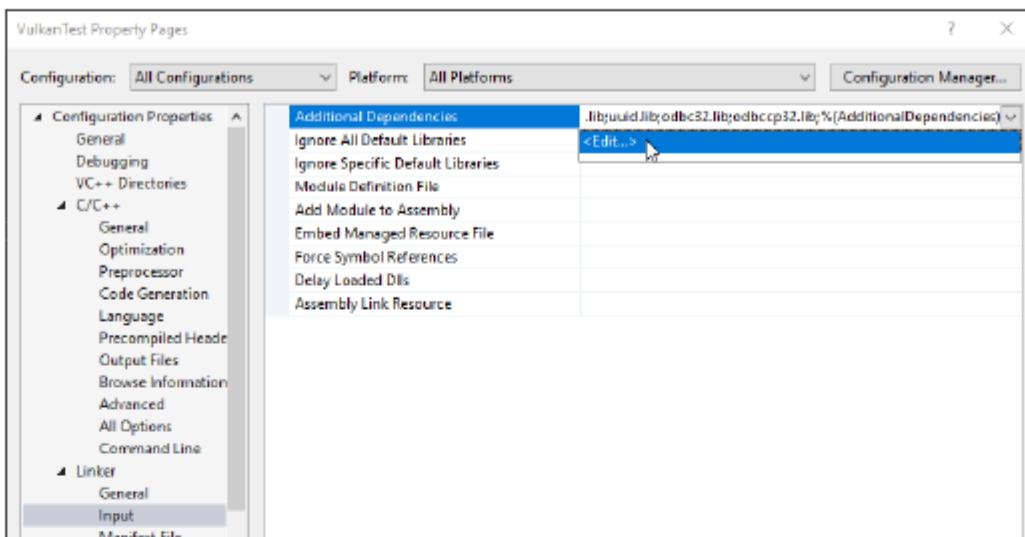
Затем откройте редактор каталогов библиотеки в разделе Компоновщик -> общие:



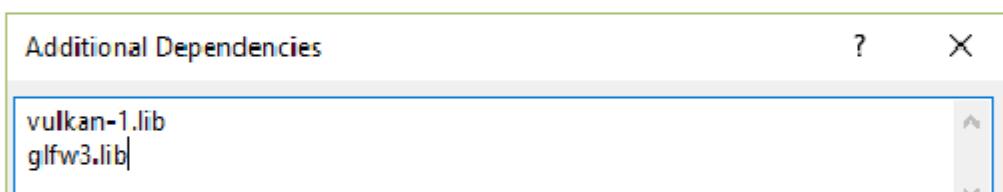
И добавьте расположение объектных файлов для Vulkan и GLFW:



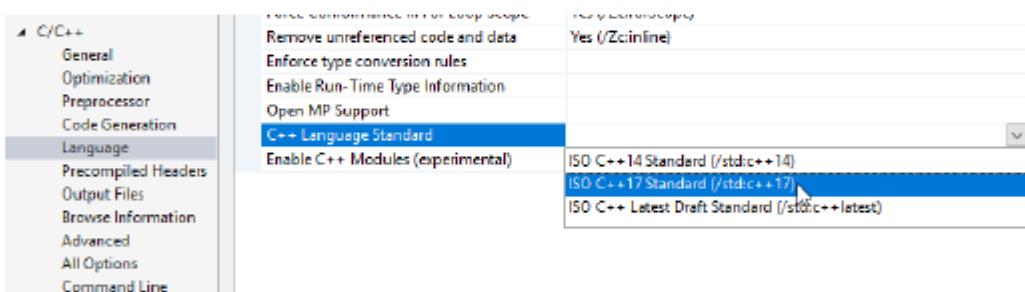
Перейдите в раздел Компоновщик -> Ввод и Прессы <Редактировать...> в Дополнительном
Зависимости раскрывающийся список.



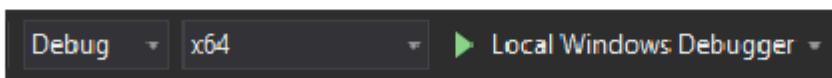
Введите имена объектных файлов Vulkan и GLFW:



И, наконец, измените компилятор, чтобы он поддерживал функции C++ 17:



Теперь вы можете закрыть диалоговое окно свойств проекта. Если вы все сделали правильно, то вы больше не должны видеть никаких ошибок, выделяемых в коде. Наконец, убедитесь, что вы действительно компилируете в 64-разрядном режиме:



Пресса **F5** чтобы скомпилировать и запустить проект, вы должны увидеть командную строку и всплывающее окно, подобное этому:



Количество расширений должно быть отличным от нуля. Поздравляем, у вас все готово для игры с Vulkan!

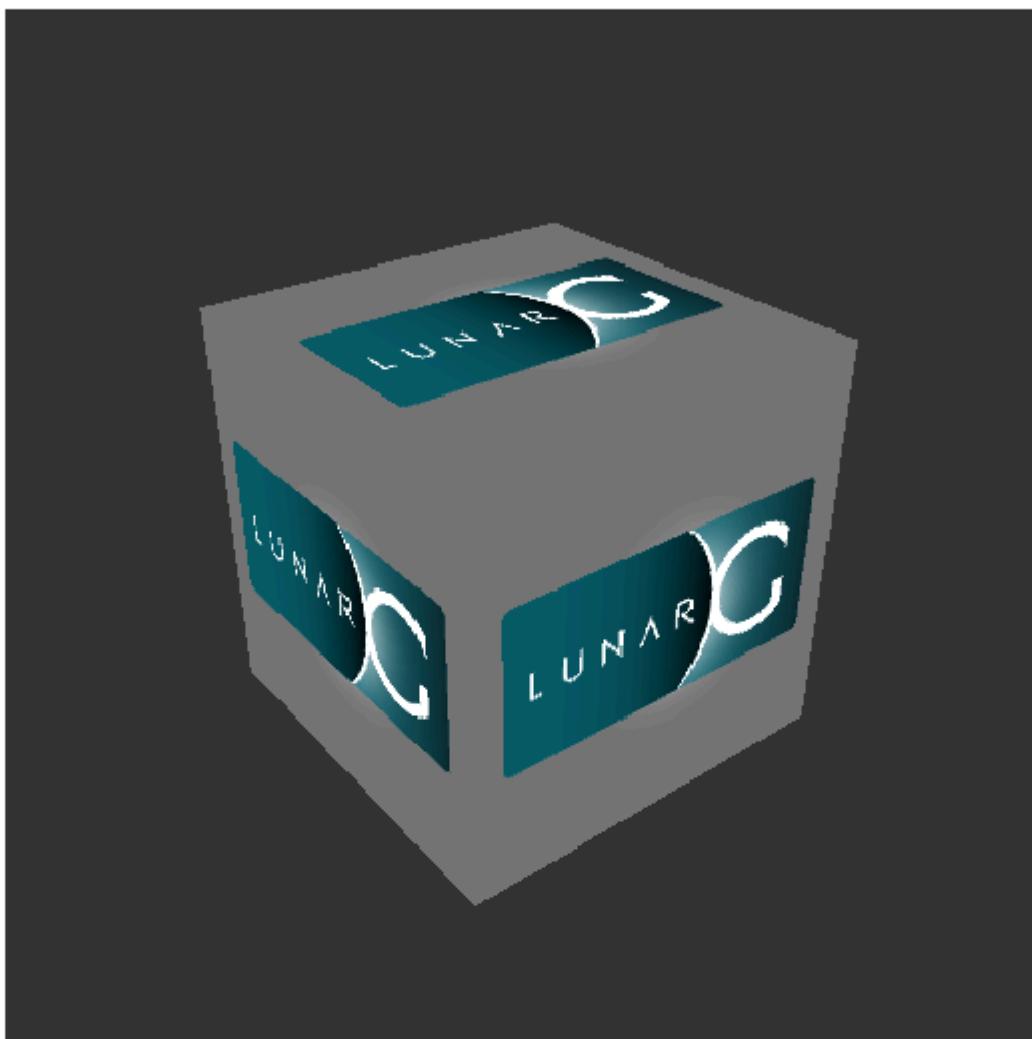
Linux

Эти инструкции будут предназначены для пользователей Ubuntu, Fedora и Arch Linux, но вы можете следовать им, изменив команды, специфичные для диспетчера пакетов, на те, которые подходят именно вам. У вас должен быть компилятор, поддерживающий C++ 17 (GCC 7+ или Clang 5+). Вам также понадобится [создать](#).

Пакеты услуг Vulkan

Наиболее важными компонентами, которые вам понадобятся для разработки приложений Vulkan в Linux, являются загрузчик Vulkan, уровни проверки и пара утилит командной строки для проверки того, поддерживает ли ваша машина Vulkan:

```
• sudo apt установит vulkan-tools          или sudo dnf установит vulkan-tools ..  
Утилиты командной строки, самое главное          vulkaninfo      и      vkcube      Запустите  
это для подтверждения того, что ваш компьютер поддерживает Vulkan.  
• sudo apt установит libvulkan-dev или sudo dnf устанавливает vulkan-loader-devel  
:Устанавливает Vulkan loader. Загрузчик ищет функции в драйвере  
во время выполнения, аналогично GLEW для OpenGL - если вы с этим знакомы. •  
sudo apt устанавливает vulkan-validationlayers-dev spirv-инструменты или  
sudo dnf устанавливает mesa-vulkan-разработка vulkan-проверка-слои-devel:  
Устанавливает стандартные уровни проверки и необходимые инструменты SPIR-V. Они  
имеют решающее значение при отладке приложений Vulkan, и мы обсудим их  
в следующей главе.  
В Arch Linux вы можете запускать sudo pacman -S vulkan-devel для установки всех  
необходимых инструментов, описанных выше.  
Если установка прошла успешно, у вас все должно быть готово к работе с частью Vulkan.  
Не забудьте запустить vkcube и убедитесь, что вы видите следующее всплывающее окно:
```



Если вы получаете сообщение об ошибке, убедитесь, что ваши драйверы обновлены, включите Vulkan runtime и что ваша видеокарта поддерживается. Ссылки на драйверы основных производителей приведены в вводной главе.

GLFW

Как упоминалось ранее, Vulkan сам по себе не зависит от платформы API и не включает инструменты для создания окна для отображения отрисованных результатов. Чтобы воспользоваться кроссплатформенными преимуществами Vulkan и избежать ужасов X11, мы будем использовать библиотеку GLFW для создания окна, поддерживающего Windows, Linux и macOS. Для этой цели доступны другие библиотеки, например SDL, но преимущество GLFW в том, что он также абстрагирует от некоторых других, специфичных для платформы функций в Vulkan, помимо простого создания окон. Мы будем устанавливать GLFW с помощью следующей команды:

```
1 sudo apt установит libglfw 3 -dev
```

или
sudo dnf устанавливает glfw-devel 1

или
судо пакман -S glfw-вэйленд #
glfw-x 1 1 для пользователей X 1 1 1

GLM

В отличие от DirectX 12 и Vulkan, который не включает библиотека для линейной алгебры операций, так что придется скачать один. GLM - это хорошая библиотека, которая предназначена для использования с графическими API, а также обычно используется с OpenGL.

Это библиотека только для заголовков, которую можно установить с libglm-разработчик или

glm-devel пакет:

sudo apt устанавливает libglm-dev
1

или

sudo dnf устанавливает glm-devel
1

или

sudo pacman -S glm
1

Компилятор шейдеров

У нас есть практически все, что нам нужно, за исключением того, что нам понадобится программа для компиляции шейдеров из удобочитаемого GLSL в байт-код.

Двумя популярными компиляторами шейдеров являются Khronos Group glslangValidator и Google glslc. Последний имеет знакомое использование, подобное GCC и Clang, поэтому мы воспользуемся этим: в Ubuntu, загрузите неофициальные двоичные файлы Google и скопируйте glslc в свой файл /usr/local/bin. Примечание: возможно, вам потребуется sudo в зависимости от ваших конкретных задач. В Fedora используйте sudo dnf установите glslc, а в Arch Linux запустите sudo pacman -S shaderc. Чтобы протестировать, запустите glslc и он должен по праву пожаловаться мы не передавали никаких шейдеров для компиляции:

glslc: ошибка: нет входных файлов

Мы подробно рассмотрим glslc в главе "Модули шейдеров".

Настройка проекта makefile

Теперь, когда вы установили все зависимости, мы можем настроить базовый проект makefile для Vulkan и написать немного кода, чтобы убедиться, что все работает.

Создайте новый каталог в удобном месте с именем, подобным `VulkanTest`.
Создайте исходный файл с именем `main.cpp` и вставьте следующий код. Не беспокойтесь о попытках разобраться в этом прямо сейчас; мы просто следим за тем, чтобы вы могли компилировать и запускать приложения Vulkan. В следующей главе мы начнем с нуля.

```
1 # определение GLFW_INCLUDE_VULKAN
2
3 # включить <GLFW/glfw3.h>
4
5 # определить GLM_FORCE_RADIANS
6
7 # определить GLM_FORCE_DEPTH_ZERO_TO_ONE
8
9 # включить <glm/vec4.hpp>
10
11 # включить <glm/mat4x4.hpp>
12
13 # включить <iostream>
14
15 int main()
16 {
17     glfwInit();
18
19     glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
20     GLFWwindow * window = glfwCreateWindow(800, 600, "Окно вулкана",
21                                             nullptr, nullptr);
22
23     uint32_t extensionCount = 0;
24     vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount,
25                                             nullptr);
26
27     std::cout << extensionCount << " поддерживаемые расширения \n";
28
29     glm::mat4 matrix;
30     glm::vec4 vec;
31
32     auto test = матрица * век;
33
34     пока (!glfwWindowShouldClose(окно)) {
35         glfwPollEvents();
36     }
37
38     glfwDestroyWindow(окно);
39
40     glfwTerminate();
41
42     Возврат 0 ;
43 }
```

Далее мы напишем `makefile` для компиляции и запуска этого базового кода Vulkan. Создайте новый пустой файл с именем `Makefile`. Я предположу, что у вас уже есть некоторые

базовый опыт работы с makefile, например, с тем, как работают переменные и правила.

Если нет, то вы можете очень быстро освоиться с этим руководством. Сначала

мы определим пару переменных, чтобы упростить оставшуюся часть файла. Определите

а `CFLAGS` переменная, которая будет указывать основные флаги компилятора:

```
1 CFLAGS = -std= c ++ 1 7 -O 2
```

Мы собираемся использовать современный C++ (`-std= c ++ 1 7`), и мы установим

уровень оптимизации равным O2. Мы можем удалить `-O2`, чтобы быстрее компилировать программы, но мы должны помнить, чтобы поместить его обратно для релизных сборок.

Аналогично, определите флаги компоновщика в а `LDFLAGS` переменная:

```
1 LDFLAGS = -lglfw -lvulkan -ldl -lpthread -lx 1 1 -lXxf 8 6 vm -lXrandr  
-lXi
```

Флаг `-lglfw` предназначен для GLFW, `-lvulkan` ссылки с загрузчиком функций Vulkan и остальные флаги - это низкоуровневые системные библиотеки, которые нужны GLFW. Остальные флаги являются зависимостями от самого GLFW: потоковой обработки и управления окнами .

Возможно, что `Xxf 8 6 vm` и `Xi` библиотеки еще не установлены в вашей системе. Вы можете найти их в следующих пакетах:

```
1 sudo apt install libxxf 8 6 vm-dev libxi-dev
```

или

```
sudo dnf устанавливает  
libXi-devel libXxf 8 6 vm-devel 1
```

или

```
sudo pacman -S  
libxi libxxf 8 6 vm 1
```

Указание правила для компиляции `VulkanTest` теперь все просто. Убедитесь, что для отступов вместо пробелов используются табуляции.

```
1 ВулканТест: main.cpp  
2 g ++ $(CFLAGS) -o VulkanTest main.cpp $(LDFLAGS)
```

Убедитесь, что это правило работает, сохранив makefile и запустив

make в каталоге

с помощью `main.cpp` и `Makefile`. Результатом должен стать `VulkanTest`

исполняемый файл. Теперь

мы определим еще два правила, тестировать и очищать, где первый запустит

исполняемый файл, а второй удалит встроенный исполняемый файл:

```
1 .ФАЛЬШИВЫЙ: проверка чистоты  
2  
3 test: VulkanTest
```

```
4     ./ВулканТест
5
6 очистка:
7     rm -f VulkanTest
```

Выполняется `сделайте тест` должно отображаться, что программа успешно запущена, и она исчезает- воспроизведите количество расширений Vulkan. Приложение должно завершиться с кодом возврата `успеха ()` при закрытии пустого окна. Теперь у вас должен быть полный `makefile`,

который выглядит следующим образом: `CFLAGS = -std=`

```
t++ 1 7 -O 2 LDFLAGS = -lglfw -lvulkan -ldl -lpthread
2 lx 1 1 -lXxf 8 6 vm -lXrandr
    -lxI
3
4 ВулканТест: main.cpp
5     g++ $(CFLAGS) -проверка работоспособности main.cpp $(LDFLAGS)
6
7 .ФАЛЬШИВЫЙ: тест чистый
8
9 тест: ВулканТест
10     ./VulkanTest
11
12 очистка:
13     rm -f VulkanTest
```

Теперь вы можете использовать этот каталог в качестве шаблона для своих проектов Vulkan. Создайте копию, переименуйте ее во что-то вроде `HelloTriangle` и удалите весь код из `main.cpp`.

Теперь вы готовы к настоящему приключению.

macOS

В этих инструкциях предполагается, что вы используете Xcode и Homebrew package manager. Также имейте в виду, что вам понадобится как минимум macOS версии 10.11, и ваше устройство должно поддерживать Metal API.

Vulkan SDK

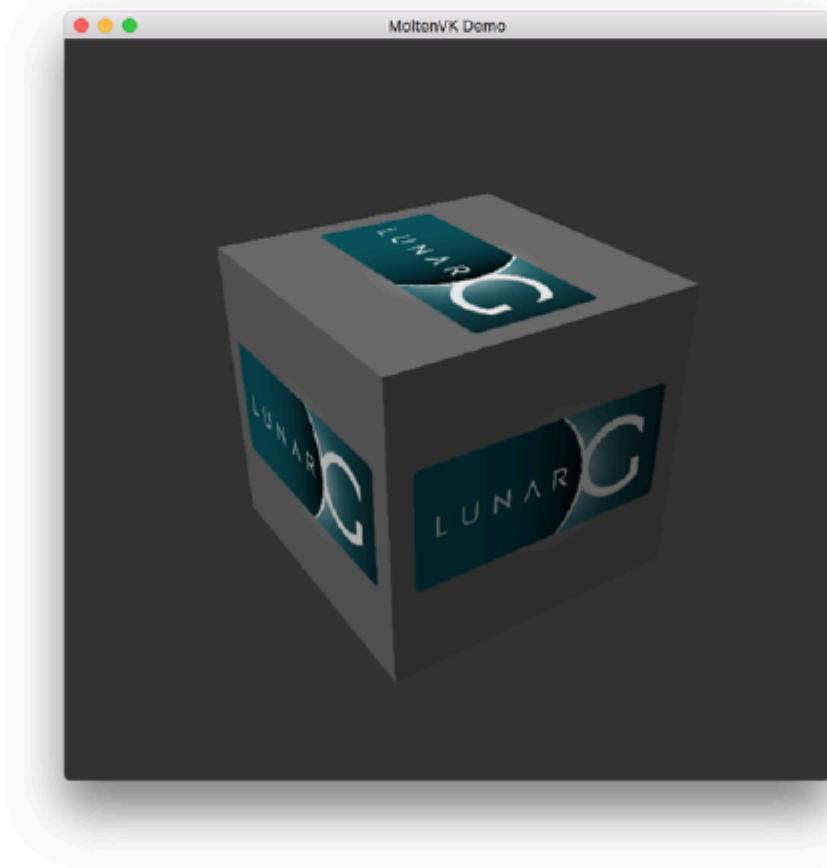
Наиболее важным компонентом, который вам понадобится для разработки приложений Vulkan, является SDK. Он включает заголовки, стандартные уровни проверки, инструменты отладки и загрузчик для функций Vulkan. Загрузчик ищет функции в драйвере во время выполнения, аналогично GLEW для OpenGL - если вы с этим знакомы. SDK можно загрузить с веб-сайта Lunarg, используя кнопки в нижней части страницы. Вам не нужно создавать учетную запись, но это даст вам доступ к некоторой дополнительной документации, которая может быть вам полезна.



Версия SDK для macOS внутренне использует MoltenVK. В macOS нет собственного вспомогательного порта для Vulkan, поэтому MoltenVK фактически действует как слой, который переводит вызовы Vulkan API в Metal graphics Framework от Apple. С помощью этого приложения вы сможете воспользоваться преимуществами Apple в области отладки и производительности Metal Framework.

После загрузки, просто извлеките содержимое в папку по вашему выбору (имейте в виду, что вам нужно будет ссылаться на нее при создании ваших проектов в Xcode).

Внутри извлеченной папки, в разделе `Приложения` в папке у вас должно быть несколько исполняемых файлов, которые позволят запустить несколько демонстраций с использованием SDK. Запустите `vkcube` исполняемый файл, и вы увидите следующее:



GLFW

Как упоминалось ранее, Vulkan сам по себе не зависит от платформы API и не включает инструменты для создания окна для отображения отрисованных результатов. Мы будем использовать библиотеку GLFW для создания окна, поддерживающего Windows, Linux и macOS. Для этой цели доступны другие библиотеки, например SDL, но преимущество GLFW в том, что он также абстрагирует от некоторых других специфичных для платформы функций в Vulkan, помимо простого создания окон. Чтобы установить GLFW на macOS, мы воспользуемся менеджером пакетов Homebrew, чтобы получить пакет `glfw` :

```
1 brew установит glfw
```

GLM

В Vulkan нет библиотеки для операций с линейной алгеброй, поэтому нам придется скачать ее. GLM - это хорошая библиотека, предназначенная для использования с графическими API, а также широко используемая с OpenGL.

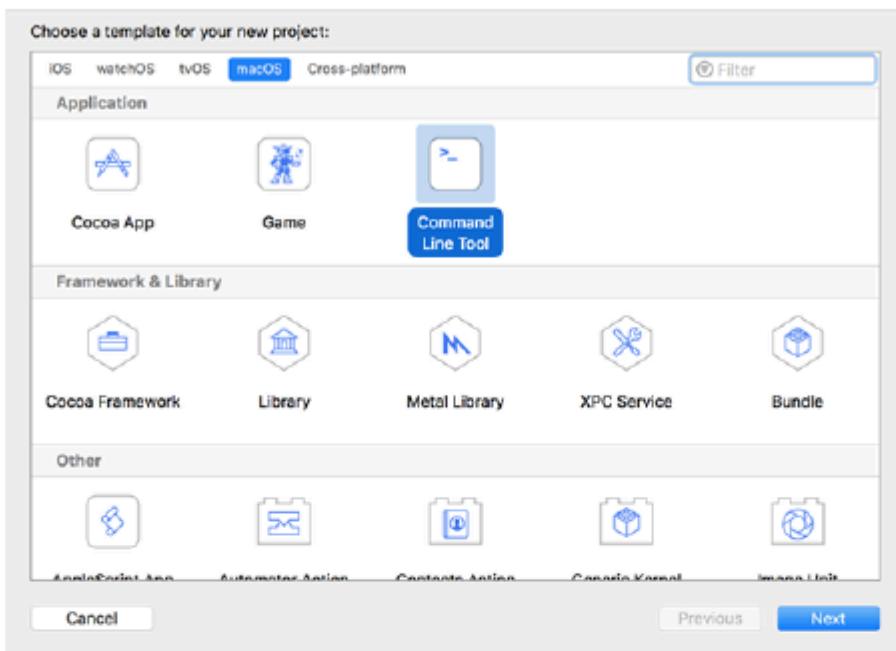
Это библиотека только для заголовков, которую можно установить с `glm` пакет:

```
1 brew install glm
```

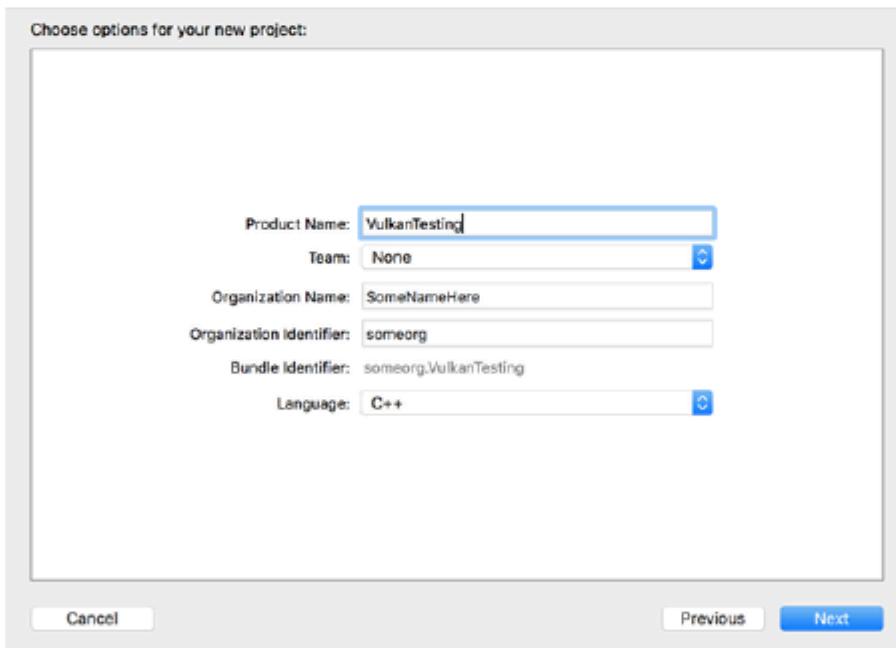
Настройка Xcode

Теперь, когда все зависимости установлены, мы можем настроить базовый проект Xcode для Vulkan. Большинство приведенных здесь инструкций, по сути, представляют собой множество "сантехнических работ", так что мы можем получить все зависимости, связанные с проектом. Также имейте в виду, что в следующих инструкциях всякий раз, когда мы упоминаем папку `vulkansdk` мы ссылаемся на папку, из которой вы извлекли Vulkan SDK.

Запустите Xcode и создайте новый проект Xcode. В открывшемся окне выберите Приложение > Инструмент командной строки.



Выберите далее, напишите название для проекта и для языка – выберите C++.



Пресса далее и проект должен был быть создан. Теперь давайте изменим

код в сгенерированном main.cpp файле на следующий код:

```
1 # define GLFW_INCLUDE_VULKAN
2 # включить <GLFW/glfw 3 .h>
3
4 # определить GLM_FORCE_RADIANS
```

```

5  # определить GLM_FORCE_DEPTH_ZERO_TO_ONE
6  # включить <glm/vec 4.hpp>
7  # включить <glm/mat 4x4.hpp>
8
9  # включить <iostream>
10
11 int main()
12 {
13     glfwInit();
14
15     glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
16     GLFWwindow * window = glfwCreateWindow(800, 600, "Окно вулкана",
17                                           nullptr, nullptr);
18
19     uint32_t extensionCount = 0;
20     vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount,
21                                           nullptr);
22
23     std::cout << extensionCount << " поддерживаемые расширения \n";
24
25     glm::mat4 matrix;
26     glm::vec4 vec;
27     auto test = матрица * век;
28
29     пока (!glfwWindowShouldClose(окно)) {
30         glfwPollEvents();
31     }
32
33     glfwDestroyWindow(окно);
34
35     Возврат 0;
}

```

Имейте в виду, что от вас пока не требуется понимать все, что делает этот код, мы просто настраиваем некоторые вызовы API, чтобы убедиться, что все работает. Xcode уже должен показывать некоторые ошибки, такие как библиотеки, которые он не может найти. Теперь мы начнем настраивать проект, чтобы избавиться от этих ошибок. В *Навигаторе проекта* на панели выберите свой проект. Откройте вкладку *Настройки сборки* и затем:

- Найдите поле **Пути поиска в заголовке** и добавьте ссылку на /usr/local/include (именно здесь Homebrew устанавливает заголовки, поэтому файлы заголовков glm и glfw3 должны быть там) и ссылка на vulkansdk/macOS/include для заголовков Vulkan.
- Найдите поле **Пути поиска в библиотеке** и добавьте ссылку на /usr/local/lib

(опять же, именно здесь Homebrew устанавливает библиотеки, поэтому там должны быть файлы библиотек `glm` и `glfw3`) и ссылка на `vulkansdk / macOS / lib`. Это должно выглядеть примерно так (очевидно, пути будут разными в зависимости от того, где вы разместили свои файлы):



Теперь, на вкладке **Этапы сборки**, на **Связите двоичный файл с библиотеками** мы добавим как `glfw` , так и `vulkan` фреймворки. Чтобы упростить задачу, мы будем добавлять динамические библиотеки в проект (вы можете ознакомиться с документацией по этим библиотекам, если хотите использовать статические фреймворки).

· Для `glfw` откройте папку `/usr/local/lib` и там вы найдете файл с таким именем, как `libglfw.3.x.dylib` ("x" - это номер версии библиотеки, он может отличаться в зависимости от того, когда вы загрузили пакет с Homebrew). Просто перетащите этот файл на вкладку **Связанные фреймворки** и библиотеки в Xcode. · Чтобы найти `vulkan`, перейдите на страницу `vulkansdk / macOS / lib`. Сделайте то же самое для обоих файлов

`libvulkan.1.dylib` и `libvulkan.1.x.xx.dylib` (где "x" будет номер версии загруженного вами SDK).

После добавления этих библиотек на той же вкладке на странице **Скопируйте файлы** измените назначение чтобы перейти к разделу "Фреймворки", очистите подпуть и снимите флажок "Только копировать при установке". Нажмите на знак "+" и добавьте все эти три фреймворка также сюда.

Ваша конфигурация Xcode должна выглядеть следующим образом:

Link Binary With Libraries (3 items)

Name	Status
libglfw.3.3.dylib	Required
libvulkan.1.dylib	Required
libvulkan.1.1.73.dylib	Required

+ - Drag to reorder frameworks

Copy Files (3 items)

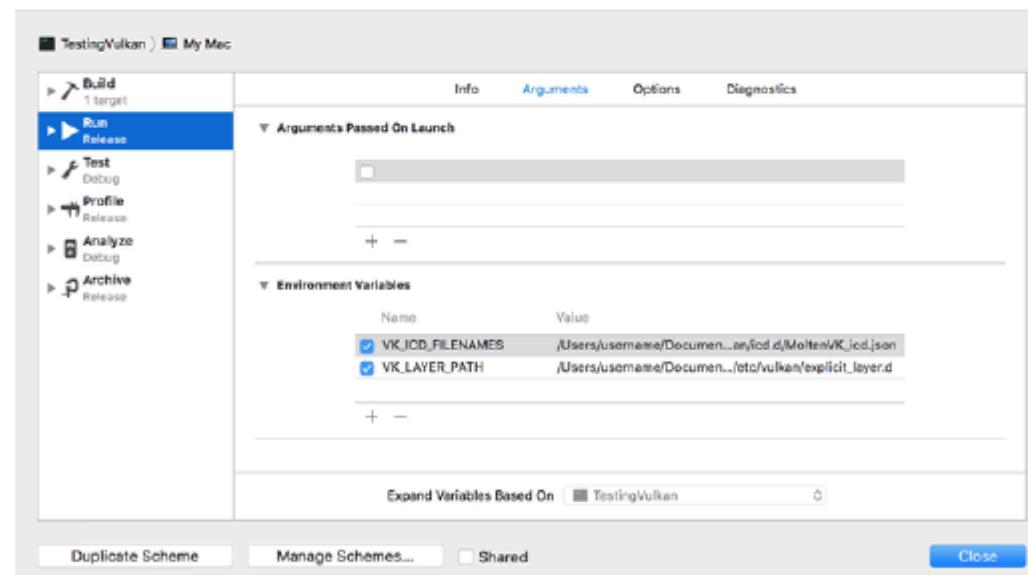
Destination	Frameworks
Subpath	
<input type="checkbox"/> Copy only when installing	
Name	Code Sign On Copy
libvulkan.1.1.73.dylib ...in ../../DevelopmentTools...	<input type="checkbox"/>
libvulkan.1.dylib ...in ../../DevelopmentTools/vulk...	<input type="checkbox"/>
libglfw.3.3.dylib ...in ../../../../../../usr/local/lib	<input type="checkbox"/>

+ -

Последнее, что вам нужно настроить, - это пара переменных среды. На панели инструментов Xcode перейдите К Продукт > Схема > Отредактируйте схему..., и на вкладке Аргументы добавьте две следующие переменные окружения:

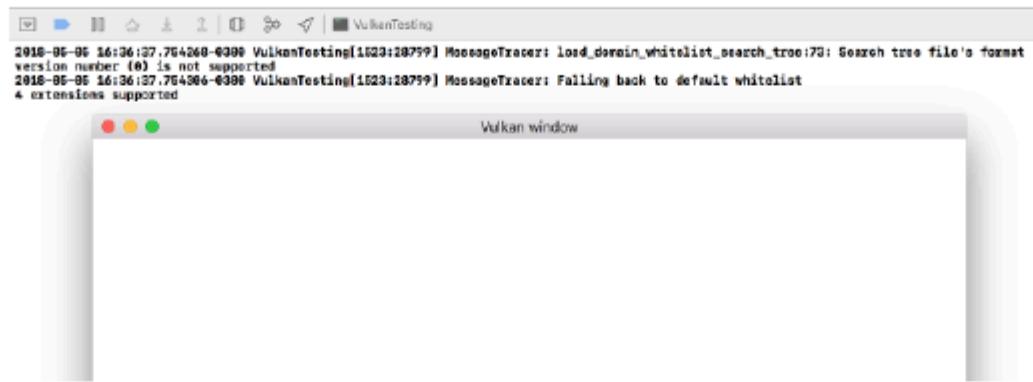
- **VK_ICD_FILenames** = vulkansdk/macOS/поделиться/vulkan/icd.d/MoltenVK_icd.json
- **VK_LAYER_PATH** = vulkansdk/macOS/поделиться/vulkan/explicit_layer.d

Это должно выглядеть примерно так:



Наконец, все должно быть готово! Теперь, если вы запустите проект (не забыв настроить конфигурацию сборки на отладку или выпуск в зависимости от конфигурации).

вы выбрали) вы должны увидеть следующее:



Количество расширений должно быть отличным от нуля.

Остальные журналы взяты из библиотек, вы можете получать из них разные сообщения в зависимости от вашей конфигурации.

Теперь у вас все готово для настоящего дела.

Рисование треугольника

Настройка

Базовый код

Общая структура

В предыдущей главе вы создали проект Vulkan со всей необходимой конфигурацией и протестировали его с помощью примера кода. В этой главе мы начнем с нуля со следующего кода:

```
1 # включить <vulkan/vulkan.h>
2
3 # включить <iostream>
4
5 # включить <stdexcept>
6
7 класс Приложение hellotriangle {
8
9     общедоступное:
10
11     анулирование выполнить() {
12
13         initVulkan();
14
15         mainLoop();
16
17         очистка();
18     }
19
20     частное:
21
22     анулирование initVulkan() {
23
24         анулирование mainLoop() {
25
26             анулирование очистка() {
```

```

26     }
27 };
28
29 int main() {
30     Привет, приложение Triangle application;
31
32     попробуйте {
33         app.run();
34     } перехватите (const std:: исключение и e) {
35         std::cerr << e.what() <<
36         std::endl; Возврат EXIT_FAILURE;
37     }
38
39     Возврат EXIT_SUCCESS ( выход_УСПЕХ );
40 }

```

Сначала мы включаем заголовок Vulkan из LunarG SDK, который предоставляет функции, структуры и перечисления. В `std::exception` и `iostream` заголовки включены для сообщения и распространения ошибок. Заголовок `cstdlib` предоставляет `EXIT_SUCCESS` и `EXIT_FAILURE` макросы.

Сама программа упакована в класс, где мы будем хранить объекты Vulkan как частные члены класса и добавлять функции для инициализации каждого из них, которые будут вызываться из функции. Как только все будет подготовлено, мы войдем в основной цикл, чтобы начать рендеринг кадров. Мы заполним основной цикл функция, включающая цикл, который повторяется до тех пор, пока окно не закроется через мгновение. Как только окно будет закрыто, и `mainLoop` в противном случае мы обязательно освободим ресурсы, которые мы использовали в функции `cleanup`. Если во время выполнения произойдет какая-либо фатальная ошибка, мы выдадим `std::runtime_error` исключение с описательным сообщением, которое передаст обратно в главная функционировать и выводиться в командную строку. Чтобы обрабатывать также различные стандартные типы исключений, мы перехватываем более общие `std::exception`. Один из примеров ошибки, с которым мы скоро разберемся, - это выяснение того, что определенное требуемое расширение не поддерживается.

Примерно в каждой главе, следующей за этой, будет добавлена одна новая функция, которая будет вызываться из, и один или несколько новых объектов Vulkan к членам частного класса, которые необходимо освободить в конце в очистка.

Управление ресурсами

Точно так же, как каждый фрагмент памяти, выделенный с помощью `malloc` требует вызова `free` бесплатно, каждый объект Vulkan, который мы создаем, должен быть явно уничтожен, когда он нам больше не нужен. В C++ возможно выполнять автоматическое управление ресурсами с использованием RAII или интеллектуальных указателей, предоставленных в выбранном . Однако, У меня есть чтобы быть явным в отношении выделения и освобождения объектов Vulkan в этом заголовке

Учебник. В конце концов, ниша Vulkan заключается в том, чтобы четко описывать каждую операцию, чтобы избежать ошибок, поэтому полезно четко указывать время жизни объектов, чтобы узнать, как работает API. Следуя этому руководству, вы могли бы реализовать автоматическое управление ресурсами, написав классы C++, которые получают объекты Vulkan в своем конструкторе и освобождают их в своем деструкторе, или предоставив пользовательский метод удаления для любого из них std::unique_ptr или std::shared_ptr , в зависимости от ваших требований к владельцу. улучшения. RAII - рекомендуемая модель для более крупных программ Vulkan, но для учебных целей всегда полезно знать, что происходит за кулисами. Объекты Vulkan либо создаются напрямую с помощью функций типа vkCreateXXX , либо выделяются через другой объект с помощью функций типа vkAllocateXXX . После того, как объект больше нигде не используется, вам нужно уничтожить его с помощью аналогов vkDestroyXXX И vkFreeXXX . Параметры для этих функций обычно различаются для разных типов объектов, но есть один параметр, который является общим для всех них: pAllocator . Это необязательный параметр, который позволяет вам указывать обратные вызовы для пользовательского ptr распределителя памяти. Мы будем игнорировать этот параметр в руководстве и всегда передавать в качестве аргумента. **Интеграция GLFW**

Vulkan отлично работает без создания окна, если вы хотите использовать его для отрисовки за кадром, но на самом деле показывать что-то гораздо интереснее!

Сначала замените строку # include <vulkan/vulkan.h> на

```
1 # define GLFW_INCLUDE_VULKAN
2 # включить <GLFW/glfw3.h>
```

Таким образом, GLFW включит свои собственные определения и автоматически загрузит с ними заголовок Vulkan. Добавьте функцию и вызов к ней из the run функция перед другими вызовами. Мы будем использовать эту функцию для инициализации GLFW и создания окна.

```
void run() {
1
2     initWindow();
3     initVulkan();
4     mainLoop();
5     очистка();
6 }
7
8 Частное:
9     void initWindow() {
10
11 }
```

Самый первый вызов в initWindow должен быть , который инициализирует glfwInit()

Библиотека GLFW. Потому что GLFW изначально был разработан для создания OpenGL

контекст, нам нужно сообщить ему, чтобы он не создавал контекст OpenGL с последующим

вызовом:

```
1 glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
```

Поскольку обработка окон с измененным размером требует особой осторожности, которую мы рассмотрим позже, отключите это сейчас с помощью другого вызова подсказки window:

```
1 glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
```

Все, что теперь осталось, это создать само окно. Добавьте `GLFWwindow * window;`

частный член класса для сохранения ссылки на него и инициализации окна с помощью:

```
1 window = glfwCreateWindow(800, 600, "Vulkan", nullptr, nullptr);
```

Первые три параметра определяют ширину, высоту и заголовок окна. Четвертый параметр позволяет вам при необходимости указать монитор для открытия окна, а последний параметр имеет отношение только к OpenGL. Рекомендуется использовать константы вместо жестко заданных значений ширины и высоты, потому что в будущем мы будем пару раз обращаться к этим значениям.

Я добавил следующие строки над приложением `hellotriangle` определение класса

```
1 :  
2 const uint32_t WIDTH = 800;  
3 const uint32_t HEIGHT = 600;
```

и заменил вызов создания окна на

```
1 window = glfwCreateWindow(ШИРИНА, ВЫСОТА, "Vulkan", nullptr, nullptr);
```

Теперь у вас должно быть `initWindow` функция, которая выглядит следующим образом:

```
1 void initWindow() {  
2     glfwInit();  
3  
4     glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);  
5     glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);  
6  
7     window = glfwCreateWindow(ШИРИНА, ВЫСОТА, "Vulkan", nullptr,  
8                               nullptr);  
9 }
```

Чтобы приложение работало до тех пор, пока не произойдет ошибка или окно не будет закрыто, нам нужно добавить цикл событий в функцию `mainLoop` следующим

```
1 образом: void Основной_цикл() {  
2     пока (!glfwWindowShouldClose(окно)) {  
3         glfwPollEvents();  
4     }  
5 }
```

Этот код должен быть достаточно понятным. Он выполняет цикл и проверяет наличие событий, таких как нажатие кнопки X, пока окно не будет закрыто пользователем. Это также цикл, в котором мы позже вызовем функцию для рендеринга одного кадра. Как только окно будет закрыто, нам нужно очистить ресурсы, уничтожив его и завершив

очистка

```
1 void cleanup() {  
2     glfwDestroyWindow(окно);  
3  
4     glfwTerminate();  
5 }
```

Когда вы запустите программу сейчас, вы должны увидеть окно под названием `Vulkan` показывать до тех пор, пока приложение не будет завершено путем закрытия окна. Теперь, когда у нас есть скелет для приложения Vulkan, давайте создадим первый объект Vulkan!

Код на C++

Экземпляр

Создание экземпляра

Самое первое, что вам нужно сделать, это инициализировать Вулкан библиотеку, созданную экземпляром. Экземпляр представляет собой соединение между вашим приложением и библиотекой Vulkan, и для его создания необходимо указать некоторые сведения о вашем приложении в драйвере.

Начните с добавления `CreateInstance` функция и ее вызов в `initVulkan` функция.

```
1 void initVulkan() {  
2     CreateInstance();  
3 }
```

Дополнительно добавьте элемент данных для хранения дескриптора

```
1 экземпляра: частное:  
2 Экземпляр VkInstance;
```

Теперь, чтобы создать экземпляр, нам сначала нужно будет заполнить структуру некоторой информацией о нашем приложении. Эти данные технически необязательны, но они могут предоставить некоторую полезную информацию драйверу для оптимизации нашего конкретного приложения (например, потому, что оно использует хорошо известный графический движок с определенным поведением). Эта структура называется `VkApplicationInfo` :

```
1 анулирование CreateInstance() {  
2     VkApplicationInfo appInfo{};  
3  
4     appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;  
5     appInfo.pApplicationName = "Привет, Треугольник";
```

```
5     appInfo.applicationVersion = VK_MAKE_VERSION( 1 ,
6         0 , 0 ); appInfo.pEngineName = "Нет движителя";
7     appInfo.EngineVersion = VK_MAKE_VERSION( 1 , 0 , 0 );
8     appInfo.apiVersion = VK_API_VERSION_1_0 ;
9 }
```

Как упоминалось ранее, многие структуры в Vulkan требуют, чтобы вы явно указывали тип в поле `sType`. Участник. Это также одна из многих структур с `pNext` членом, который может указывать на информацию о расширении в будущем. Мы используем инициализацию значения здесь, чтобы оставить его как `nullptr`. Большая часть информации в Vulkan передается через структуры, а не через параметры функции, и нам придется заполнить еще одну структуру, чтобы обеспечить достаточную информацию для создания экземпляра. Эта следующая структура не является необязательной и сообщает драйверу Vulkan, какие глобальные расширения и уровни проверки мы хотим использовать. Глобальный характер здесь означает, что они применимы ко всей программе, а не к конкретному устройству, что станет ясно из следующих нескольких глав.

```
1 VkInstanceCreateInfo CreateInfo{};
2 CreateInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
3 CreateInfo.pApplicationInfo = &appInfo;
```

Первые два параметра просты. Следующие два уровня определяют желаемые глобальные расширения. Как упоминалось в главе "Обзор", Vulkan - это независимый от платформы API, что означает, что вам нужно расширение для взаимодействия с оконной системой. GLFW имеет удобную встроенную функцию, которая возвращает расширения, необходимые для выполнения

```
1 того, что мы можем передать в структуру: uint32_t glfwExtensionCount =
2 0 ; // постоянный символ * * glfwExtensions;
3
4 Расширенные возможности glfw =
5 glfwGetRequiredInstanceExtensions(&glfwExtensionCount);
6
7 CreateInfo.enabledExtensionCount = glfwExtensionCount;
CreateInfo.pEnabledExtensionName = glfwExtensions;
```

Последние два элемента структуры определяют глобальные уровни проверки, которые нужно включить. Мы поговорим об этом более подробно в следующей главе, поэтому пока просто

```
1 оставьте их пустыми. CreateInfo.enabledLayerCount = 0 ;
```

Теперь мы указали все, что нужно Vulkan для создания экземпляра, и мы можем, наконец, выполнить вызов `vkCreateInstance`:

```
1 Результат VkResult = vkCreateInstance(&CreateInfo, nullptr, &instance);
```

Как вы увидите, общий шаблон, которому следуют параметры функции создания объекта в Vulkan::

- Указатель на struct с информацией о создании • Указатель на обратные вызовы пользователя распределителя, всегда `nullptr` в этом руководстве
- Указатель на переменную, в которой хранится дескриптор нового объекта

Если все прошло хорошо, то дескриптор экземпляра был сохранен в член класса. Почти все функции Vulkan возвращают значение типа `VkResult` это либо `VK_SUCCESS`, либо код ошибки. Чтобы проверить, был ли экземпляр создан успешно, нам не нужно сохранять результат, и вместо этого мы можем просто использовать проверку значения успеха:

```
1 если (vkCreateInstance(&CreateInfo, nullptr, &instance) != VK_SUCCESS)
{
    выбросить std::runtime_error("не удалось создать экземпляр!");
}
```

Теперь запустите программу, чтобы убедиться, что экземпляр создан успешно.

Столкнулся с VK_ERROR_INCOMPATIBLE_DRIVER:

При использовании macOS с последней версией MoltenVK sdk вы можете получить `VK_ERROR_INCOMPATIBLE_DRIVER` возвращено из `vkCreateInstance`. Согласно инструкциям по началу работы. Начиная с версии 1.3.216 Vulkan SDK, `VK_KHR_PORTABILITY_subset` расширение обязательно. Чтобы устранить эту ошибку, сначала добавьте `VK_INSTANCE_CREATE_ENUMERATE_PORTABILITY_BIT_KHR` измените значение на `VkInstanceCreateInfo` флаги структуры, затем добавьте `VK_KHR_PORTABILITY_ENUMERATION_EXTENSION_NAME` для экземпляра включен список расширений.

Обычно код может быть таким:

```
1 ...
2 ...
3 std::vector<постоянный символ * > Требуемые расширения;
4
5 для(uint 3 2 _t i = 0; i < glfwExtensionCount; i++) {
6     requiredExtensions.emplace_back(glfwExtensions[i]);
7 }
8
9 Требуемые расширения.emplace_back(VK_KHR_PORTABILITY_ENUMERATION_EXTENSION_NAME);
10
11 CreateInfo.flags |= VK_INSTANCE_CREATE_ENUMERATE_PORTABILITY_BIT_KHR;
12
13 CreateInfo.enabledExtensionCount = (uint 3 2 _t)
14     Требуемые расширения.size();
15 CreateInfo.pppenedextensionname = Требуемые расширения.data();
```

```
16     если (vkCreateInstance(&CreateInfo, nullptr, &instance) != VK_SUCCESS)
17     {
18         выбросить std::runtime_error ("не удалось создать экземпляр!");
19     }
```

Проверка поддержки расширений

Если вы посмотрите на `vkCreateInstance` документация, затем вы увидите это из возможных кодов ошибок

просто укажите требуемые нам расширения и завершите работу, если этот код ошибки вернется. Это имеет смысл для основных расширений, таких как системный интерфейс Windows, но что, если мы захотим проверить наличие дополнительной функциональности? Чтобы получить список поддерживаемых расширений перед созданием экземпляра,

существует `vkEnumerateInstanceExtensionProperties` функция.

указатель на переменную, которая хранит количество расширений, и массив из `VkExtensionProperties` для хранения сведений о расширениях. Он также принимает необязательный первый параметр, который позволяет нам фильтровать расширения по определенному уровню проверки, который мы пока проигнорируем.

Чтобы выделить массив для хранения сведений о расширении, нам сначала нужно знать, сколько их. Вы можете запросить только количество расширений, оставив последний параметр пустым:

```
1 uint32_t extensionCount = 0;
2 vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount,
3                                         nullptr);
```

Требуется

Теперь выделите массив для хранения сведений о расширении (включить):

```
1 расширения <vector> std::vector<VkExtensionProperties>(extensionCount);
```

Наконец, мы можем запросить информацию о расширении:

```
1 vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount,
2                                         расширения.данные());
```

Каждое из них свойства `VkExtensionProperties` структура содержит имя и версию

выражения `tension`. Мы можем перечислить их с помощью простого цикла for (\t -

```
1 табуляция для отступа): std::cout << "доступные расширения:\n";
2
3 для (const auto и расширения : расширения) {
4
5     std::cout << '\t' << расширение.extensionName << '\n';
6 }
```

Вы можете добавить этот код в функцию `CreateInstance`, если хотите предоставить некоторые подробности о службе поддержки Vulkan. создать функцию, которая проверяет, все ли расширения, возвращаемые

В качестве проблемы попробуйте

`glfwGetRequiredInstanceExtensions` включены в список поддерживаемых экспортов.

Очистка

Значение `VkInstance` следует уничтожать только непосредственно перед завершением программы. Это может быть уничтожено в `cleanup` с помощью `vkDestroyInstance` функции:

```
1 void cleanup() {
2     vkDestroyInstance(экземпляр, nullptr);
3
4     glfwDestroyWindow(окно);
5
6     glfwTerminate();
7 }
```

Параметры для функции `vkDestroyInstance` просты. Как упомянутые в предыдущей главе функции распределения и освобождения в Vulkan имеют необязательный обратный вызов распределителя, который мы проигнорируем, передав ему `nullptr`. Все остальные ресурсы Vulkan, которые мы создадим в следующих главах, должны быть очищены перед уничтожением экземпляра.

Прежде чем перейти к более сложным шагам после создания экземпляра, пришло время оценить наши возможности отладки, проверив уровни проверки.

Код на C++

Уровни проверки

Что такое уровни проверки?

API Vulkan разработан вокруг идеи минимальной нагрузки на драйвер, и одним из проявлений этой цели является то, что в API по умолчанию очень ограничена проверка ошибок. Даже такие простые ошибки, как присвоение перечислению неправильных значений или передача нулевых указателей на требуемые параметры, как правило, не обрабатываются явно и просто приводят к сбоям или неопределенному поведению. Поскольку Vulkan требует от вас предельной ясности во всем, что вы делаете, легко допустить множество мелких ошибок, таких как использование новой функции графического процессора и забывание запросить ее во время создания логического устройства. Однако это не означает, что эти проверки нельзя добавлять в API. Vulkan представляет элегантную систему для этого, известную как *уровни проверки*. Уровни проверки являются необязательными компонентами, которые подключаются к вызовам функций Vulkan для применения дополнительных операций. Обычными операциями в уровнях проверки являются:

- Сверка значений параметров со спецификацией для обнаружения ошибок
- Использование отслеживания создания и уничтожения объектов для обнаружения утечек ресурсов
- Проверка потокобезопасности путем отслеживания потоков, из которых исходят вызовы

- Протоколирование каждого вызова и его параметров в стандартный вывод
- Отслеживание вызовов Vulkan для профилирования и воспроизведения

Вот пример того, как может выглядеть реализация функции на уровне диагностики проверки:

```

1 VkResult vkCreateInstance(
2     постоянный VkInstanceCreateInfo *
3     pCreateInfo, const Обратные вызовы vkAllocation *
4     pAllocator, VkInstance * экземпляр) {
5
6     если (pCreateInfo == nullptr || экземпляр == nullptr) {
7         log("Нулевой указатель передан на требуемый параметр!");
8         Возврат ОШИБКА VK_ERROR_INITIALIZATION_FAILED;
9     }
10
11     Возврат real_vkCreateInstance(pCreateInfo, pAllocator, экземпляр);
12 }
```

Эти уровни проверки можно свободно укладывать друг на друга, чтобы включить все интересующие вас функции отладки. Вы можете просто включить уровни проверки для отладочных сборок и полностью отключить их для релизных сборок, что дает вам лучшее из обоих миров! Vulkan не поставляется со встроенными уровнями проверки, но LunarG Vulkan SDK предоставляет хороший набор слоев, которые проверяют наличие распространенных ошибок. У них также полностью открытый исходный код, так что вы можете проверить, какие ошибки они проверяют, и внести свой вклад. Использование уровней проверки - лучший способ избежать того, чтобы ваше приложение зависало на разных драйверах, случайно полагаясь на неопределенное поведение. Уровни проверки могут использоваться только в том случае, если они были установлены в системе. Например, уровни проверки LunarG доступны только на компьютерах с установленным Vulkan SDK. Ранее в Vulkan существовало два разных типа уровней проверки: для конкретного экземпляра и для конкретного устройства. Идея заключалась в том, что уровни экземпляра будут проверять только вызовы, связанные с глобальными объектами Vulkan, такими как экземпляры, а уровни для конкретного устройства будут проверять вызовы, связанные только с определенным графическим процессором. Уровни, относящиеся к конкретному устройству, теперь устарели, что означает, что уровни проверки экземпляра применяются ко всем вызовам Vulkan. Документ спецификации по-прежнему рекомендует включить уровни проверки также на уровне устройства для обеспечения совместимости, что требуется некоторыми реализациями. Мы просто укажем те же слои, что и экземпляр, на уровне логического устройства, что мы увидим позже.

Использование слоев проверки

В этом разделе мы увидим, как включить стандартные уровни диагностики, предоставляемые Vulkan SDK. Как и расширения, уровни проверки должны быть включены

указав их имя. Вся полезная стандартная проверка включена в слой, включенный в SDK, который известен как `VK_LAYER_KHRONOS_validation`. Давайте сначала добавим в программу две переменные конфигурации, чтобы указать слои, которые нужно включить, и включать их или нет. Я решил основывать это значение на том, компилируется ли программа в режиме отладки или нет. The `NDEBUG` макрос является частью стандарта C++ и означает "не отлаживается".

```

1 const uint32_t WIDTH = 800;
2 const uint32_t HEIGHT = 600;
3 const std::vector<const char*> validationLayers = {
4     "VK_LAYER_KHRONOS_validation"
5 };
6
7
8 #ifdef ОШИБКА_ПОИСКА
9     const bool enableValidationLayers = false;
10 #else
11     const bool enableValidationLayers = true;
12 #endif

```

Мы добавим новую функцию `checkValidationLayerSupport`, который проверяет, все ли из доступных запрошенные слои. Сначала перечислите все доступные слои с помощью функции `vkEnumerateInstanceExtensionProperties`. Ее использование идентично свойству из свойств `vkEnumerateInstanceExtensionProperties`, который обсуждался в главе о создании экземпляра.

```

1 bool Проверка validationlayersupport() {
2     uint32_t layerCount;
3     vkEnumerateInstanceLayerProperties(&layerCount, nullptr);
4
5     стандартный формат::vector<VkLayerProperties> Доступные слои(layerCount);
6     vkEnumerateInstanceLayerProperties(&layerCount,
7         Доступные слои.данные());
8
9     возвращает false;
}

```

Затем проверьте, все ли слои в `validationLayers` существуют в `availableLayers`. Список. Возможно, вам потребуется включить `<cstring>` для `strcmp`.

```

1 для (const char* layerName : validationLayers) {
2     bool layerFound = false;
3
4     для (const auto& layerProperties : доступные слои) {
5         if (strcmp(layerName, layerProperties.layerName) == 0) {
6             layerFound = true;
7             перерыв;
}

```

```

8         }
9     }
10
11     if (!layerFound)
12     {
13         возвращает false;
14     }
15
16     возвращает true;

```

Теперь мы можем использовать эту функцию в `CreateInstance`:

```

1 пустота CreateInstance() {
2     if (Включить Validationlayers && !Проверить validationlayersupport()) {
3         выбросить std::runtime_error("запрошены уровни проверки, но
4             недоступны!");
5     }
6     ...
7 }

```

Теперь запустите программу в режиме отладки и убедитесь, что ошибка не возникает.

Если это произойдет, то ознакомьтесь с часто задаваемыми вопросами.

Наконец, измените `vkInstanceCreateInfo` создайте структуру экземпляра,

чтобы включить имена уровней проверки, если они включены:

```

1 if (enableValidationLayers) {
2     CreateInfo.enabledLayerCount =
3         static_cast<uint32_t>(validationLayers.size());
4     CreateInfo.pEnabledLayerNames
5     = validationLayers.data(); } еще {
6     CreateInfo.enabledLayerCount = 0;
}

```

Если проверка прошла успешно,

то `VK_ERROR_LAYER_NOT_PRESENT`

`vkCreateInstance`

никогда не должен возвращать а

ошибку, но вы должны запустить программу, чтобы сделать

уверен.

Обратный вызов сообщения

Уровни проверки будут выводить отладочные сообщения на стандартный вывод по

умолчанию ошибки, но мы также можем обработать их самостоятельно, предоставив явный обратный вызов в нашей программе. Это также позволит вам решить, какие сообщения вы хотели

главы.

Чтобы настроить обратный вызов в программе для обработки сообщений и связанных с ними деталей, мы должны настроить debug messenger с обратным вызовом, используя `VK_EXT_DEBUG_UTILS` расширение. Сначала мы создадим `getRequiredExtensions` функция, которая вернет требуемый список расширений в зависимости от того, включены уровни проверки или нет:

```
1 std::vector<const char*> getRequiredExtensions() {
2     uint32_t glfwExtensionCount = 0;
3     const void* glfwExtensions;
4     glfwExtensions =
5         glfwGetRequiredInstanceExtensions(&glfwExtensionCount);
6
7     std::vector<const void*> extensions(glfwExtensions,
8                                         glfwExtensions + glfwExtensionCount);
9
10    if (EnableValidationLayers) {
11        extensions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
12    }
13
14    return extensions;
15 }
```

Расширения, указанные GLFW, требуются всегда, но расширение debug messenger добавляется условно.

Обратите внимание, что я использовал

`VK_EXT_DEBUG_UTILS_EXTENSION_NAME` здесь используется макрос, который равен буквенной строке "VK_EXT_debug_utils". Использование этого макроса позволяет избежать опечаток.

Теперь мы можем использовать эту функцию в `CreateInstance`:

```
1 автоматический расширения = getRequiredExtensions();
2CreateInfo.enabledExtensionCount =
3     static_cast<uint32_t>(расширения.размер());
3CreateInfo.pEnabledExtensionName = расширения.данные();
```

Запустите программу, чтобы убедиться, что вы не получаете `VK_ERROR_EXTENSION_NOT_PRESENT` ошибку. На самом деле нам не нужно проверять существование этого расширения, потому что это должно подразумеваться доступностью уровней проверки. Теперь давайте посмотрим, как выглядит функция обратного вызова отладки. Добавьте новую статическую функцию памяти с именем `ber_debugCallback` с помощью `PFN_vkDebugUtilsMessengerCallbackEXT` прототипа. The `VKAPI_ATTR` и `VKAPI_CALL` убедитесь, что функция имеет правильную сигнатуру, позволяющую Vulkan вызывать ее.

```
1 статический VKAPI_ATTR VkBool32 VKAPI_CALL debugCallback(
2     VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
3     VkDebugUtilsMessageTypeFlagsEXT messageType,
4     const vkdebugutilsmessengercallbackdata* text *
5     pCallbackData, void* pUserData) {
```

```

6
7     std::cerr << "уровень проверки: " << pCallbackData->pMessage <<
8         std::endl;
9
10    Возврат VK_FALSE;
11 }
```

Первый параметр определяет серьезность сообщения, которое является одним из следующих флагов:

<ul style="list-style-type: none"> • <code>VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT</code>: Сообщение • <code>VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT</code>: Информационное сообщение • <code>VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT</code>: Сообщение о поведении, которое не обязательно является ошибкой, но весьма вероятно, что это ошибка в вашем приложении • <code>VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT</code>: Сообщение о недопустимое поведение, которое может привести к сбоям 	Диагностика
---	--------------------

Значения этого перечисления настроены таким образом, что вы можете использовать операцию сравнения, чтобы проверить, является ли сообщение равным или хуже по сравнению с некоторым уровнем серьезности, например: `если` (Серьезность сообщений \geq 1

```

1 // Сообщение достаточно важно, чтобы показать
2
3 }
```

Параметр `MessageType` может иметь следующие значения:

<ul style="list-style-type: none"> • <code>VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT</code>: Произошло какое-то событие - значение, не имеющее отношения к спецификации или производительности • <code>VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT</code>: Что-то изменилось произошло событие, нарушающее спецификацию или указывающее на возможную ошибку • <code>VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT</code>: Потенциальное отсутствие-оптимальное использование Vulkan 	Произошло какое-то событие Что-то изменилось Потенциальное отсутствие-оптимальное использование Vulkan
---	---

Параметр `pCallbackData` относится к структура, содержащая подробную информацию о самом сообщении, причем наиболее важными элементами являются:

<ul style="list-style-type: none"> • <code>pMessage</code>: Сообщение отладки в виде строки, заканчивающейся нулем • <code>pObjects</code>: Массив дескрипторов объектов Vulkan, связанных с сообщением • <code>objectCount</code>: Количество объектов в массиве 	текст vkdebugutilsmessengercallbackdata
--	--

Наконец, параметр `pUserData` содержит указатель, который был указан во время настройки обратного вызова, и позволяет вам передавать ему ваши собственные данные.

Обратный вызов возвращает логическое значение, указывающее, следует ли прервать вызов Vulkan, который вызвал сообщение уровня проверки. Если обратный вызов возвращает true, то вызов прерывается с ошибкой. Это обычно используется только для тестирования самих уровней проверки, поэтому вы всегда должны возвращаться . Все, что теперь остается, - сообщить Vulkan о функции обратного вызова. Возможно, несколько удивительно, что даже обратный вызов debug в Vulkan управляется с помощью дескриптора, который необходимо явно создавать и уничтожать. Такой обратный вызов является частью отладчика messenger и у вас может быть столько их, сколько вы хотите. Добавьте член класса для этого дескриптора прямо в разделе экземпляра:

```
1 VkDebugUtilsMessengerEXT отладчик сообщений;
```

Теперь добавьте функцию setupDebugMessenger вызывается из initVulkan сразу

после CreateInstance:

```
1 анилирование Инициализированный вулкан() {
2     CreateInstance();
3     setupDebugMessenger();
4 }
5
6 анилирует действие setupDebugMessenger() {
7     если (!enableValidationLayers) Возврат;
8
9 }
```

Нам нужно будет заполнить структуру подробностями о мессенджере и его

```
1 обратном вызове: VkDebugUtilsMessengerCreateInfoEXTCreateInfo{};
2 CreateInfo.Тип =
3     VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
4 create_info.messageSeverity =
5     VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT
| VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT
| VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
6 CreateInfo.MessageType = VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT
| VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT
| VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
5 CreateInfo.pfnUserCallback = debugCallback;
6 CreateInfo.pUserData = nullptr; // Необязательно
```

В messageSeverity поле позволяет указать все типы серьезности, для которых вы хотели бы, чтобы был вызван ваш обратный вызов. Я указал все типы, кроме для VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT здесь, чтобы получать уведомления о возможных проблемах, опуская подробную общую информацию об отладке. Аналогично, поле MessageType позволяет фильтровать типы сообщений, которые ваш

отображается уведомление об обратном вызове. Я просто включил здесь все типы. Вы всегда можете отключить некоторые, если они вам не нужны.

Наконец, функция `pfnUserCallback` поле задает указатель на функцию обратного вызова.

При желании вы можете передать указатель на поле `pUserData`, которое будет передано функции обратного вызова через параметр `pUserData`. Вы могли бы использовать это, например, для передачи указателя на класс `HelloTriangleApplication`. Обратите внимание, что существует множество других способов настройки сообщений уровня проверки и обратных вызовов отладки, но это хорошая настройка для начала работы с этим руководством. Смотрите спецификацию расширения для получения дополнительной информации о возможностях.

Эта структура должна быть передана в `vkCreateDebugUtilsMessengerEXT` функция для создания объекта `VkDebugUtilsMessengerEXT`. К сожалению, поскольку эта функция является функцией расширения, она не загружается автоматически. Мы должны сами найти его адрес с помощью `vkGetInstanceProcAddr`. Мы собираемся создать нашу собственную прокси-функцию, которая обрабатывает это в фоновом режиме. Я добавил это прямо над `HelloTriangleApplication` определение класса.

```

1 VkResult CreateDebugUtilsMessengerEXT(экземпляр VkInstance, постоянный
2                                     VkDebugUtilsMessengerCreateInfoEXT * pCreateInfo, const
3                                     VkAllocationCallbacks * pAllocator, vkdebugutilsmessenger
4                                     (текст * pDebugMessenger) {
5
5     автоматически функция = (PFN_vkCreateDebugUtilsMessengerEXT)
6
6     vkGetInstanceProcAddr(экземпляр,
7                           "vkCreateDebugUtilsMessengerEXT");
8
8     if (функция != nullptr) {
9
9         возврат функция (экземпляр, pCreateInfo, pAllocator,
10                      pDebugMessenger);
11
11     } еще {
12
12         возврат VK_ERROR_EXTENSION_NOT_PRESENT;
13
13     }
14 }
```

Функция `vkGetInstanceProcAddr` вернет результат `nullptr` если функция не удалось загрузить. Теперь мы можем вызвать эту функцию для создания объекта расширения, если он доступен:

```

1 если (CreateDebugUtilsMessengerEXT(экземпляр, &CreateInfo, nullptr,
2                                     &debugMessenger) != VK_SUCCESS) {
3
3     выбросить std::runtime_error("не удалось настроить debug messenger!");
4 }
```

Предпоследним параметром снова является необязательный обратный вызов распределителя, для которого мы установили значение `nullptr`, в остальном параметры довольно просты. Поскольку отладочный мессенджер специфичен для нашего экземпляра Vulkan и его слоев, его необходимо явно указать в качестве первого аргумента. Вы также увидите этот шаблон с другим дочерним объектом позже.

Объект `VkDebugUtilsMessengerEXT` также необходимо очистить с помощью вызова

для `vkDestroyDebugUtilsMessengerEXT`. Аналогично `vkCreateDebugUtilsMessengerEXT` функция должна быть загружена явно.

Создайте другую прокси-функцию прямо ниже `CreateDebugUtilsMessengerEXT` :

```
1 аннулирование DestroyDebugUtilsMessengerEXT(экземпляр VkInstance,
2
3     vkdebugutilsmessenger - текстовый отладчик-мессенджер, const
4     VkAllocationCallbacks * pAllocator) {
5
6     авто func = (PFN_vkDestroyDebugUtilsMessengerEXT)
7
8         vkGetInstanceProcAddr(экземпляр,
9         "vkDestroyDebugUtilsMessengerEXT");
10
11    if (функция != nullptr) {
12
13        функция(экземпляр, debugMessenger, pAllocator);
14    }
15 }
```

Убедитесь, что эта функция является либо функцией статического класса, либо функцией вне класса. Затем мы можем вызвать его в

```
1 void cleanup() {
2
3     if (Включить Validationlayers) {
4
5         DestroyDebugUtilsMessengerEXT(экземпляр, debugMessenger,
6             nullptr);
7
8     }
9
10
11     vkDestroyInstance(экземпляр, nullptr);
12
13     glfwDestroyWindow(окно);
14
15     glfwTerminate();
16 }
```

Создание и уничтожение отладочного экземпляра

Хотя теперь мы добавили в программу отладку со слоями проверки мы еще не охватили все полностью. Текст сообщения `vkCreateDebugUtilsMessenger` для вызова требуется создание действительного экземпляра и `vkDestroyDebugUtilsMessengerEXT` должен быть вызван до уничтожения экземпляра. В настоящее время это не позволяет нам отлаживать какие-либо проблемы в вызовах `vkCreateInstance` и `vkDestroyInstance`. Однако, если вы внимательно прочтете документацию по расширению, вы увидите, что есть способ создать отдельный debug utils messenger специально для

этих двух вызовов функций. Для этого требуется, чтобы вы просто передали указатель на `VkDebugUtilsMessengerCreateInfoEXT` структурировать в поле расширения `pNext` `VkInstanceCreateInfo`. Сначала извлеките информацию о создании messenger в отдельную функцию:

```

1 пустота

    populateDebugMessengerCreateInfo(VkDebugUtilsMessengerCreateInfoEXT&
CreateInfo) {
2     CreateInfo = {};
3     CreateInfo.тип =
        VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
4     create_info.messageSeverity =
        VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT |
        VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT |
        VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
5     CreateInfo.MessageType =
        VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |
        VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT |
        VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
6     CreateInfo.pfnUserCallback = обратный вызов отладки;
7 }
8
9 ...
10
11 анулирование setupDebugMessenger()
{
12     если (!enableValidationLayers) Возврат;
13
14     VkDebugUtilsMessengerCreateInfoEXT CreateInfo;
15     Заполненная ошибка messengerCreateInfo(CreateInfo);
16
17     если (createdebugutilsmessenger(экземпляр, &CreateInfo, nullptr,
18         &debugMessenger) != VK_SUCCESS) {
19         выбросить std::runtime_error("не удалось настроить debug
20             посыльный!");
21     }
22 }

```

Теперь мы можем повторно использовать

CreateInstance

функция:

```

1 ЭТО В void CreateInstance() {
2     ...
3
4     VkInstanceCreateInfo CreateInfo{};
5     CreateInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
6     CreateInfo.pApplicationInfo = &appInfo;
7
8     ...
9
10    VkDebugUtilsMessengerCreateInfoEXT
11    debugCreateInfo(); если (enableValidationLayers) {
12        CreateInfo.enabledLayerCount =

```

```

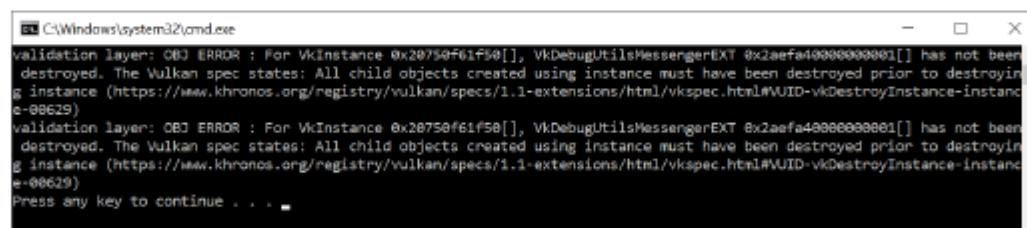
    static_cast<( статическая передача)>uint 3 2_t>(validationLayers.size());
13 CreateInfo.ppEnabledLayerNames = validationLayers.data();
14
15     populateDebugMessengerCreateInfo(debugCreateInfo);
16     CreateInfo.pNext = (VkDebugUtilsMessengerCreateInfoEXT *) 
17         &debugCreateInfo;
18 } ещё {
19     CreateInfo.enabledLayerCount = 0;
20
21     CreateInfo.pNext = nullptr;
22 }
23 если (vkCreateInstance(&CreateInfo, nullptr, &instance) != 
24     VK_SUCCESS) {
25     выбросить std::runtime_error("не удалось создать экземпляр!");
26 }

```

переменная помещается вне инструкции if, чтобы гарантировать
 что она не будет уничтожена перед vkCreateInstance вызовите. Создав
 дополнительный отладочный мессенджер, таким образом, он будет автоматически использоваться
 во время vkCreateInstance и vkDestroyInstance и после этого был очищен.

Тестирование

Теперь давайте намеренно допустим ошибку, чтобы увидеть уровни проверки в действии. Временно удалите вызов `DestroyDebugUtilsMessengerEXT` в функции `очистка` и запустите вашу программу. После завершения работы вы должны увидеть что-то вроде этого:



Если вы не видите никаких сообщений, проверьте свою установку.

Если вы хотите узнать, какой вызов вызвал сообщение, вы можете добавить точку останова в обратный вызов сообщения и посмотреть трассировку стека.

Конфигурация

Существует гораздо больше настроек для поведения слоев проверки, чем просто флаги, указанные в `VkDebugUtilsMessengerCreateInfoEXT` структура. Обзор

перейдите в Vulkan SDK и перейдите в каталог config . Там вы найдете vk_layer_settings.txt файл, объясняющий, как настроить слои. Чтобы настроить параметры слоя для вашего собственного приложения, скопируйте файл в Каталоги Debug и Release вашего проекта и следуйте инструкциям, чтобы установить желаемое поведение. Однако в оставшейся части этого руководства я буду исходить из того, что вы используете настройки по умолчанию. На протяжении всего этого урока я буду допускать пару преднамеренных ошибок, чтобы показать вам, насколько полезны уровни проверки при их обнаружении, и научить вас, насколько важно точно знать, что вы делаете с Vulkan. Теперь пришло время взглянуть на устройства Vulkan в системе. Код на C++

Физические устройства и семейства очередей

Выбор физического устройства

После инициализации библиотеки Vulkan через VkInstance нам нужно найти и выбрать видеокарту в системе, которая поддерживает нужные нам функции. На самом деле мы можем выбрать любое количество видеокарт и использовать их одновременно, но в этом руководстве мы остановимся на первой видеокарте, которая соответствует нашим потребностям.

Мы добавим функцию Выберите physicaldevice и добавьте к ней вызов в функции initVulkan

```
1 void Инициализирующий вулкан() {
2     CreateInstance();
3     setupDebugMessenger(); pickPhysicalDevice();
4 }
5
6
7 анулирование pickPhysicalDevice() {
8
9 }
```

Видеокарта, которую мы в конечном итоге выберем, будет сохранена в дескрипторе VkPhysicalDevice, который добавляется как новый член класса. Этот объект будет неявно уничтожен при уничтожении VkInstance, поэтому нам не нужно будет делать ничего нового в функции cleanup .

```
1 VkPhysicalDevice физическое устройство = VK_NULL_HANDLE;
```

Перечисление видеокарт очень похоже на перечисление расширений и начинается с запроса только номера.

```
1 uint 3 2 _t deviceCount = 0;
2 vkEnumeratePhysicalDevices(экземпляр, &deviceCount, nullptr);
```

Если есть 0 устройств с поддержкой Vulkan, то нет смысла

```
1 идти дальше. if (Количество устройств == 0) {
2     выбросить std::runtime_error("не удалось найти графические процессоры с поддержкой Vulkan
! ");
3 }
```

В противном случае теперь мы можем выделить массив для хранения всех

дескрипторов VkPhysicalDevice.

```
1 std::vector<VkPhysicalDevice> devices(deviceCount); vkEnumeratePhysicalDevices(экземпляр
2 иdeviceCount, devices.data());
```

Теперь нам нужно оценить каждую из них и проверить, подходят ли они для операций, которые мы хотим выполнить, потому что не все видеокарты созданы равными. Для этого мы представим новую функцию:

```
1 bool isDeviceSuitable(VkPhysicalDevice device) {
2     верните значение true;
3 }
```

И мы проверим, соответствует ли какое-либо из физических устройств требованиям, которые мы добавим к этой функции.

```
1 для const auto и device : устройства) {
2     если (isDeviceSuitable(устройство)) {
3         Физическое устройство
4         = device; сломать;
5     }
6 }
7
8 если (физическое устройство == VK_NULL_HANDLE) {
9     выбросить std::runtime_error("не удалось найти подходящий графический процессор!");
10 }
```

В следующем разделе будут представлены первые требования, которые мы проверим в isDeviceSuitable функция. Как мы начнем использовать больше Vulkan особенности в последующих главах мы расширим эту функцию, чтобы включить дополнительные проверки.

База пригодности устройства проверки

Чтобы оценить пригодность устройства, мы можем начать с запроса некоторых деталей. Основные свойства устройства, такие как имя, тип и поддерживаемая версия Vulkan, могут запрашиваться с помощью vkGetPhysicalDeviceProperties.

```
1 VkPhysicalDeviceProperties deviceProperties;
2 vkGetPhysicalDeviceProperties(устройство и девайсные свойства);
```

Поддержка дополнительных функций, таких как сжатие текстур, 64-битные переходы с плавающей точкой и рендеринг с несколькими видовыми экранами (полезно для виртуальной реальности), может быть запрошена с помощью

```
1 vkGetPhysicalDeviceFeatures: VkPhysicalDeviceFeatures функции
2 устройства; vkGetPhysicalDeviceFeatures(устройство и deviceFeatures);
```

С устройств можно запросить дополнительные сведения, которые мы обсудим позже, касающиеся памяти устройств и семейств очередей (см. Следующий раздел). В качестве примера предположим, что мы считаем наше приложение пригодным только для выделенных видеокарт, поддерживающих геометрические шейдеры. Тогда `isDeviceSuitable` функция будет выглядеть следующим образом:

```
1 bool isDeviceSuitable(VkPhysicalDevice device) {
2     VkPhysicalDeviceProperties свойства устройства;
3     VkPhysicalDeviceFeatures характеристики устройства;
4     vkGetPhysicalDeviceProperties(устройство, & deviceProperties);
5     vkGetPhysicalDeviceFeatures(устройство, & deviceFeatures);
6
7     Возврат Свойства устройства.Тип устройства == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU &&
8         Характеристики устройства.geometryShader;
9 }
```

Вместо того чтобы просто проверять, подходит ли устройство или нет, и выбирать первое, вы также могли бы присвоить каждому устройству оценку и выбрать самое высокое. Таким образом, вы могли бы отдать предпочтение выделенной видеокарте, дав ей более высокий балл, но вернуться к встроенному графическому процессору, если это единственный доступный. Вы могли бы

```
1 реализовать что-то подобное следующим образом: # включить <карта>
2
3 ...
4
5 пустота pickPhysicalDevice() {
6     ...
7
8     // Используйте упорядоченную карту для
9     // автоматической сортировки кандидатов по
10    // возрастающему баллу
11    std::multimap<int, VkPhysicalDevice> кандидаты;
12
13    для (const auto& device : устройства) { 11
14        инт оценка = rateDeviceSuitability(устройство);
15        кандидаты.вставить(std::make_pair(оценка, устройство));
16    }
17
18    // Проверьте, подходит ли вообще лучший кандидат
```

```

17     если (кандидаты.rbegin()-> первый > 0) {
18         physicalDevice =
19             кандидаты.rbegin()-> второе; } еще {
20             выбросить std::runtime_error("не удалось найти подходящий графический процессор!");
21     }
22 }
23
24 int rateDeviceSuitability (VkPhysicalDevice device) {
25     ...
26
27     int оценка = 0;
28
29     // Дискретные графические процессоры имеют значительное преимущество
30     в производительности если (deviceProperties.DeviceType ==
31         VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU)
32         { оценка += 1000;
33     }
34
35     // Максимально возможный размер текстур влияет на качество графики
36     оценка += deviceProperties.limits.maxImageDimension_2D;
37
38     // Приложение не может функционировать без
39     геометрических шейдеров if (!deviceFeatures.geometryShader) {
40         Возврат 0;
41     }
42
43     Возврат оценка;
}

```

Вам не нужно реализовывать все это в этом руководстве, но оно должно дать вам представление о том, как вы могли бы спроектировать процесс выбора устройства. Конечно, вы можете также просто отобразить названия вариантов и разрешить пользователю выбирать. Поскольку мы только начинаем, поддержка Vulkan - это единственное, что нам нужно, и поэтому мы остановимся на любом графическом

```

1     процессоре:bool isDeviceSuitable(физическое устройство VK) {
2         возвращает значение true;
3     }

```

В следующем разделе мы обсудим первую действительно необходимую функцию для проверки.

Семейства очередей

Ранее вкратце упоминалось, что почти каждая операция в Vulkan, любая, от рисования до загрузки текстур, требует отправки команд в очередь. Существуют различные типы очередей, которые возникают из разных источников.

семейства очередей и каждое семейство очередей допускает только подмножество команд. Для примера, может существовать семейство очередей, которое разрешает обработку только команд compute или одно, которое разрешает только команды, связанные с передачей памяти. Нам нужно проверить, какие семейства очередей поддерживаются устройством и какое одно из них поддерживает команды, которые мы хотим использовать. Для этой цели мы добавим новую функцию, которая ищет все семейства очередей, которые нам нужны. Прямо сейчас мы будем искать только очередь, поддерживающую графические команды., таким образом, функция могла бы выглядеть

```
1     uint 3 2 _t findQueueFamilies(VkPhysicalDevice device) {  
2         // Логика для поиска семейства графических очередей  
3     }
```

Однако в одной из следующих глав мы уже собираемся искать еще одну очередь, так что лучше подготовиться к этому и объединить индексы в

```
1     структуру: struct QueueFamilyIndices {  
2         uint 3 2 _t Графическое семейство;  
3     };  
4  
5     QueueFamilyIndices Найти queuefamilies(VkPhysicalDevice device) {  
6         Индексы QueueFamilyIndices;  
7         // Логика поиска индексов семейства очередей для заполнения  
8         struct Возврат индексы;  
9     }
```

Но что, если семейство очередей недоступно? Мы могли бы создать исключение в `findQueueFamilies`, но эта функция на самом деле не подходит для принятия решений о пригодности устройства. Например, мы можем предполагать устройства с выделенным семейством очередей передачи, но не требующие этого. Следовательно, нам нужен какой-то способ указать, было ли найдено конкретное семейство очередей. На самом деле невозможно использовать магическое значение, которое может быть допустимым семейством очередей для указания на несуществование семейства очередей, поскольку любое значение, включающее индекс. К счастью, в C++ 17 была введена структура данных, позволяющая различать, существует значение или нет:

```
2 # включить <необязательно>  
3 ...  
4  
5 std::optional<uint 3 2 _t> Графическое семейство;  
6  
7 std::cout << std::boolalpha << Графическое семейство.has_value() <<  
8     std::endl; // ложный
```

```

9 Графическая семья = 0 ;
10 std::cout << std::boolalpha <<
graphicsFamily.has_value() << 11
    std::endl; // true

```

std:: необязательно - это оболочка, которая не содержит значения, пока вы не присвоите
ей что-либо. В любой момент вы можете запросить, содержит ли он значение или нет,

вызвав его функция-член has_value(). Это означает, что мы можем изменить логику на:

```

1 # включить <необязательно>
2 ...
3 ...
4 структура QueueFamilyIndices {
5
6     std::необязательно<uint 3 2 _t> Графическое семейство;
7 };
8
9 QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
10     Индексы QueueFamilyIndices;
11     // Присвоить индекс семействам очередей, которые
12     можно было бы найти Возврат индексы;
13 }

```

Теперь мы можем приступить к фактической реализации findQueueFamilies :

```

1 QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
2     Индексы QueueFamilyIndices;
3
4     ...
5
6     Возврат индексы;
7 }

```

Процесс получения списка семейств очередей - это именно то, что вы ожидаете

и использует свойства vkGetPhysicalDeviceQueueFamilyProperties:

```

1 uint 3 2 _t queueFamilyCount = 0 ;
2 vkGetPhysicalDeviceQueueFamilyProperties(устройство, &queueFamilyCount,
    nullptr);
3
4 std::вектор<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
5 vkGetPhysicalDeviceQueueFamilyProperties(устройство, &queueFamilyCount,
    queueFamilies.data());

```

Структура VkQueueFamilyProperties содержит некоторые сведения о семействе
queue, включая тип поддерживаемых операций и количество очередей,
которые могут быть созданы на основе этого семейства. Нам нужно найти хотя
бы одно семейство очередей, поддерживающее VK_QUEUE_GRAPHICS_BIT.

```

1 int i = 0;
2 для (const auto& queueFamily : семьи очередей) {
3     if (Семья очередей.queueFlags & VK_QUEUE_GRAPHICS_BIT) {
4         indexes.graphicsFamily = i;
5     }
6
7     i++;
8 }

```

Теперь, когда у нас есть эта необычная функция поиска семейства очередей, мы можем использовать ее как проверку в `isDeviceSuitable` функция, гарантирующая, что устройство может обрабатывать команды, которые мы хотим

```

1 используеться: bool isDeviceSuitable (VkPhysicalDevice device) {
2     Индексы QueueFamilyIndices = findQueueFamilies(устройство);
3
4     Возврат indexes.graphicsFamily.has_value();
5 }

```

Чтобы сделать это немного удобнее, мы также добавим общую проверку к самой структуре:

```

1 структура QueueFamilyIndices {
2     std::необязательно<uint 3 2 _t> Графическое семейство;
3
4     bool IsComplete() {
5         Возврат graphicsFamily.has_value();
6     }
7 };
8
9 ...
10
11 bool isDeviceSuitable(VkPhysicalDevice
device) { 11
12     Индексы QueueFamilyIndices = findQueueFamilies(устройство);
13
14     Возврат indexes.IsComplete();
15 }

```

Теперь мы также можем использовать это для раннего выхода из `findQueueFamilies`:

```

1 для (const auto& queueFamily : семьи очередей) {
2     ...
3
4     if (indexes.IsComplete()) {
5         разбить;
6     }
7 }

```

```
8     я++;
9 }
```

Отлично, это все, что нам сейчас нужно, чтобы найти подходящее физическое устройство! Следующий шаг - создать логическое устройство для взаимодействия с ним. Код на C++

Логическое устройство и очереди

Введение

После выбора физического устройства для использования нам нужно настроить *логическое устройство для взаимодействия с ним*. Процесс создания логического устройства аналогичен процессу создания экземпляра и описывает функции, которые мы хотим использовать. Нам также нужно указать, какие очереди создавать теперь, когда мы запросили, какие семейства очередей доступны. Вы даже можете создать несколько логических устройств из одного физического устройства, если у вас разные требования. Начните с добавления нового члена класса для хранения дескриптора логического устройства.

```
Устройство VkDevice;
```

Затем добавьте функцию `createLogicalDevice`, которая вызывается из `initVulkan`.

```
1 анулирование Инициализированный вулкан() {
2
3     CreateInstance();
4
5     setupDebugMessenger(); pickPhysicalDevice();
6
7
8     анулирование Создайте логическое устройство() {
9
10    }
```

Указание очередей, которые будут созданы

Создание логического устройства включает в себя указание множества деталей в структурах снова, из которых первой будет `VkDeviceQueueCreateInfo`. Эта структура описывает количество очередей, которое мы хотим для одного семейства очередей. Прямо сейчас нас интересует только очередь с графическими возможностями.

```
1 Индексы QueueFamilyIndices = findQueueFamilies(физическое устройство);
2
3 VkDeviceQueueCreateInfo queueCreateInfo{};
4 queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
5 queueCreateInfo.queueFamilyIndex = индексы.graphicsFamily.value();
6 queueCreateInfo.queueCount = 1;
```

Доступные в настоящее время драйверы позволяют вам создать лишь небольшое количество очередей для каждого семейства очередей, и на самом деле вам не нужно больше одной. Это потому что вы можете создать все буферы команд в нескольких потоках, а затем отправить их все сразу в основной поток одним вызовом с низкими накладными расходами. Vulkan позволяет вам назначать приоритеты очередям, чтобы влиять на планирование выполнения командного буфера, используя числа с плавающей запятой между ними. 0 . 0 и 1 . 0 .

Это требуется, даже если имеется только одна очередь:

```
1 float queuePriority = 1.0f;
2 queueCreateInfo.pQueuePriorities = &queuePriority;
```

Указание функций используемого устройства

Следующая информация, которую необходимо указать, - это набор функций устройства, которые мы будем использовать. Это функции, для которых мы запросили поддержку с помощью `vkGetPhysicalDeviceFeatures` в предыдущей главе, например, геометрия шейдеры. Прямо сейчас нам не нужно ничего особенного, поэтому мы можем просто определить это и оставить все как есть `VK_FALSE`. Мы вернемся к этой структуре, как только мы начнем делать с Vulkan более интересные вещи.

```
1 VkPhysicalDeviceFeatures features = {};
```

Создание логического устройства

Установив две предыдущие структуры, мы можем приступить к заполнению основного `VkDeviceCreateInfo` структура.

```
1 VkDeviceCreateInfoCreateInfo{};
2 CreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
```

Сначала добавьте указатели на информацию о создании очереди и структуры функций устройства:

```
1 CreateInfo.pQueueCreateInfos =
2 &queueCreateInfo; CreateInfo.queueCreateInfoCount = 1;
3
4 CreateInfo.pEnabledFeatures = &deviceFeatures;
```

Остальная информация имеет сходство с `VkInstanceCreateInfo` структурой и требует от вас указания расширения и слоя проверки. Отличаются друг от друга. Ниже заключается в том, что эти конкретные устройства в этот раз. Примером расширения, зависящего от конкретного устройства, является `VK_KHR_swapchain`, который позволяет вам представлять отрисованные изображения с этого устройства в Windows. Возможно, что в системе есть устройства Vulkan, которым не хватает этой возможности, например, потому, что они поддерживают только вычислительные операции. Мы вернемся к этому расширению в главе о цепочке обмена.

Предыдущие реализации Vulkan проводили различие между уровнями проверки, зависящими от экземпляра, и уровнями, зависящими от де-вице, но это больше не так. Это означает, что `enabledLayerCount` и `ppEnabledLayerNames` поля `VkDeviceCreateInfo` игнорируются современными реализациями. Тем не менее, все равно рекомендуется установить их в любом случае, чтобы они были совместимы со старыми реализациями:

```
1 CreateInfo.enabledExtensionCount = 0;
2
3 если (enableValidationLayers) {
4     CreateInfo.enabledLayerCount =
5         static_cast<uint32_t>(validationLayers.size());
6     CreateInfo.ppEnabledLayerNames
7     = validationLayers.data(); } еще {
8     CreateInfo.enabledLayerCount = 0;
}
```

Пока нам не понадобятся никакие расширения для конкретного устройства.

Все, теперь мы готовы создать экземпляр логического устройства с помощью вызова соответствующего имени `vkCreateDevice` функция.

```
1 если (vkCreateDevice(физическое устройство, &CreateInfo, nullptr, &устройство) != VK_SUCCESS) {
2     выбросить std::runtime_error("не удалось создать логическое устройство!");
3 }
```

Параметрами являются физическое устройство для взаимодействия, только что указанная нами информация об очереди и использовании, необязательный указатель обратного вызова выделения и указатель на переменную для хранения дескриптора логического устройства. Аналогично функции создания экземпляра, этот вызов может возвращать ошибки, основанные на включении несуществующих расширений или указании желаемого использования неподдерживаемых функций.

Устройство должно быть уничтожено в очистке с помощью `vkDestroyDevice` функция:

```
1 аннулировать очистка () {
2     vkDestroyDevice(устройство,
3         nullptr); ...
4 }
```

Логические устройства не взаимодействуют напрямую с экземплярами, поэтому это не включено в качестве параметра.

Получение дескрипторов очереди

Очереди создаются автоматически вместе с логическим устройством, но у нас пока нет дескриптора для взаимодействия с ними. Сначала добавьте член класса для хранения дескриптора графической очереди:

```
1 VkQueue graphicsQueue;
```

Очереди устройств неявно очищаются при уничтожении устройства, поэтому нам не нужно ничего делать в очистке. Мы можем использовать `vkGetDeviceQueue` функция для извлечения дескрипторов очереди для каждого семейства очередей. Параметрами являются логическое устройство, семейство очередей, индекс очереди и указатель на переменную, в которой хранится дескриптор очереди. Поскольку мы создаем только одну очередь из этого семейства, мы просто будем использовать

0

```
1 index.vkGetDeviceQueue(устройство, индексы.graphicsFamily.value(), 0,
    &Графическая очередь);
```

С логическим устройством и дескрипторами очередей мы теперь можем фактически начать использовать видеокарту для выполнения задач! В следующих нескольких главах мы настроим ресурсы для представления результатов в оконной системе. Код на C++

Презентация

Поверхность окна

Поскольку Vulkan не зависит от платформы API, он не может напрямую взаимодействовать с оконной системой сам по себе. Чтобы установить соединение между Vulkan и оконной системой для отображения результатов на экране, нам необходимо использовать расширения WSI (Window System Integration). В этой главе мы обсудим первый, который называется `VK_KHR_surface`. Он предоставляет `VkSurfaceKHR` объект, который представляет абстрактный тип поверхности для представления визуализированных изображений. Поверхность в нашей программе будет поддерживаться окном, которое мы уже открывали с помощью GLFW.

The `VK_KHR_surface` extension - это расширение уровня экземпляра, и мы фактически уже включили его, потому что оно включено в список, возвращаемый `glfwGetRequiredInstanceExtensions`. В список также включены некоторые другие расширения WSI, которые мы будем использовать в следующих двух главах.

Поверхность окна необходимо создать сразу после создания экземпляра, потому что это действительно может повлиять на выбор физического устройства. Причина, по которой мы отложили это, заключается в том, что поверхности окон являются частью более широкой темы целей рендеринга и представления, объяснение которой загромоздило бы базовую настройку. Следует также отметить, что поверхности окон являются полностью необязательным компонентом в Vulkan, если вам просто нужен рендеринг за пределами экрана. Vulkan позволяет вам сделать это без таких хитростей, как создание невидимого окна (необходимого для OpenGL). **Создание поверхности окна**

Начните с добавления `Поверхность` член класса прямо под отладочным обратным вызовом.

```
1 VkSurfaceKHR surface;
```

Хотя этот `VkSurfaceKHR` объект и его использование не зависят от платформы, его создание не связано с деталями системы `Windows`. Например, ему нужны `HWND` и `HMODULE` обрабатывает в `Windows`. Поэтому существует дополнение к расширению, зависящее от платформы, которое в `Windows` называется `VK_KHR_win_3_2_surface` и также автоматически включается в список из `glfwGetRequiredInstanceExtensions`. Я продемонстрирую, как это расширение для конкретной платформы можно использовать для создания `surface` в `Windows`, но на самом деле мы не будем использовать его в этом руководстве. Не имеет никакого смысла использовать библиотеку, подобную `GLFW`, а затем все равно переходить к использованию кода, зависящего от платформы. На самом деле в `GLFW` есть `glfwCreateWindowSurface` который обрабатывает различия платформ для нас. Тем не менее, полезно посмотреть, что он делает за кулисами, прежде чем мы начнем полагаться на него.

Чтобы получить доступ к собственным функциям платформы, вам необходимо обновить `includes`

```
вверху: # define VK_USE_PLATFORM_WIN_3_2_KHR
1 # define GLFW_INCLUDE_VULKAN
2 # включить <GLFW/glfw3.h>
3 # определить GLFW_EXPOSE_NATIVE_WIN_3_2
4 # включить <GLFW/glfw3native.h>
```

Поскольку поверхность окна является объектом `Vulkan`, она поставляется с `VkWin_3_2_SurfaceCreateInfoKHR` структура, которую необходимо заполнить. У нее есть два важных параметра: `hwnd` и `помеха`. Это дескрипторы окна и процесса.

```
1 VkWin_3_2_SurfaceCreateInfoKHR CreateInfo{};
2 CreateInfo.sType = VK_STRUCTURE_TYPE_WIN_3_2_SURFACE_CREATE_INFO_KHR;
3 CreateInfo(hwnd = glfwGetWin_3_2_Window(окно);
4 CreateInfo.hinstance = GetModuleHandle(nullptr);
```

Функция `glfwGetWin_3_2_Window` используется для получения необработанного файла `HWND` из объекта `GLFW window`. Вызов `GetModuleHandle` возвращает препятствие дескриптор текущего процесса.

После этого поверхность можно создать с помощью `vkCreateWin_3_2_SurfaceKHR`, который включает параметр для экземпляра, сведения о создании поверхности, пользовательские распределители и переменную для дескриптора поверхности, в которой будет сохранен. Технически это функция расширения `WSI`, но она настолько широко используется, что ее включает стандартный загрузчик `Vulkan`, поэтому, в отличие от других расширений, вам не нужно загружать ее явно.

```
1 if (vkCreateWin_3_2_SurfaceKHR(экземпляр, &CreateInfo, nullptr,
2 &surface) != VK_SUCCESS) {
3     выбросить std::runtime_error("не удалось создать поверхность окна!");
}
```

Процесс аналогичен для других платформ, таких как `Linux`, где `vkCreateXcbSurfaceKHR` принимает подключение `XCB` и окно в качестве сведений о создании с помощью `X11`.

Функция `glfwCreateWindowSurface` выполняет именно эту операцию с разной реализацией для каждой платформы. Теперь мы интегрируем ее в нашу программу. Добавьте функцию, которая будет вызываться из `initVulkan` сразу после создания экземпляра и `setupDebugMessenger`.

```
1 пустота initVulkan() {
2     CreateInstance(); setupDebugMessenger();
3     createSurface();
4     pickPhysicalDevice();
5     createLogicalDevice();
6 }
7 }
8
9 void Создать поверхность() {
10 }
11 }
```

Вызов GLFW принимает простые параметры вместо структуры, что делает реализацию функции очень простой:

```
1 void createSurface() {
2     if (glfwCreateWindowSurface(экземпляр, окно, nullptr и поверхность)
3         != VK_SUCCESS) {
4         выбросить std::runtime_error("не удалось создать поверхность окна!");
5     }
6 }
```

Параметрами являются следующие: `VkInstance`, указатель окна GLFW, пользовательские распределители и указатель на переменную `VkSurfaceKHR`. Он просто проходит через `VkResult` из соответствующего вызова платформы. GLFW не предлагает специальной функции для уничтожения поверхности, но это можно легко сделать с помощью оригинального API:

```
1 void cleanup() {
2     ...
3     vkDestroySurfaceKHR(экземпляр, поверхность,
4         nullptr); vkDestroyInstance(экземпляр,
5         nullptr); ...
6 }
```

Убедитесь, что `surface` уничтожен до создания экземпляра.

Запрос на поддержку презентаций

Хотя реализация Vulkan может поддерживать системную интеграцию Windows, это не означает, что каждое устройство в системе поддерживает ее. Следовательно, нам нужно расширить создание, чтобы гарантировать, что устройство может отображать изображения на `surface`. Поскольку представление зависит от очереди, то

проблема на самом деле заключается в поиске семейства очередей, которое поддерживает представление на созданной нами поверхности.

На самом деле возможно, что семейства очередей, поддерживающие команды рисования, и те, которые поддерживают представление, не перекрываются. Следовательно, мы должны принять во внимание, что может существовать отдельная очередь представления путем изменения

`QueueFamilyIndices` структура:

```
1 struct QueueFamilyIndices {  
2     std::optional<uint32_t> GraphicsFamily;  
3     std::optional<uint32_t> PresentFamily;  
4  
5     bool IsComplete() const {  
6         return GraphicsFamily.has_value() &&  
7             PresentFamily.has_value();  
8     }  
};
```

Далее мы модифицируем

`findQueueFamilies`

функцию для поиска семейства очередей

это может отображаться в нашем окне `surface`. Функция для

проверки этого - это физическое устройство,

, которое принимает

индекс семейства очередей и `surface` в качестве параметров. Добавьте к

нему вызов в том же цикле, что и

`VK_QUEUE_GRAPHICS_BIT`

```
1 VkBool32 presentSupport = false;  
2 vkGetPhysicalDeviceSurfaceSupportKHR(device, i, surface,  
    &presentSupport);
```

Затем просто проверьте значение логического значения и сохраните

индекс очереди семейства презентаций:

```
1 if (presentSupport) {  
2     indices.PresentFamily = i;  
3 }
```

Обратите внимание, что очень вероятно, что в конечном итоге это будет одно и то же семейство очередей, но на протяжении всей программы мы будем

обращаться с ними так, как если бы они были отдельными очередями для единого подхода. Тем не менее, вы могли бы добавить логику, чтобы явно предпочесть физическое устройство, поддерживающее рисование и презентацию в одной очереди, для повышения производительности. **Создание очереди презентаций**

Единственное, что остается, это изменить процедуру создания логического устройства, чтобы создать очередь презентаций и получить дескриптор `VkQueue`. Добавьте член

переменной для дескриптора:

`VkQueue presentQueue;`

Далее, нам нужно иметь несколько файлов `VkDeviceQueueCreateInfo` структуры

для создания очереди из обоих семейств. Элегантный способ сделать это - создать набор всех уникальных семейств очередей, которые необходимы для требуемых

```
1 // очередь: # включить <установить>
2 ...
3 ... Индексы
4 QueueFamilyIndices = findQueueFamilies(физическое устройство);
5 std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
6
7 std::set<uint32_t> Уникальные семейства =
8     {indexes.graphicsFamily.value(), indexes.presentFamily.value()};
9
10 значение с плавающей точкой Приоритет очереди = 1.0f;
11 для (uint32_t queueFamily : Уникальные семейства) {
12
13     VkDeviceQueueCreateInfo queueCreateInfo{};
14     queueCreateInfo.sType =
15         VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
16     queueCreateInfo.queueFamilyIndex =
17         queueFamily; queueCreateInfo.queueCount = 1;
18     queueCreateInfo.pQueuePriorities = &queuePriority;
19     queueCreateInfos.push_back(queueCreateInfo);
20 }
```

И изменить `VkDeviceCreateInfo` чтобы указать на

```
1 ВЕКТОР:CreateInfo.queueCreateInfoCount =
2     static_cast<const uint32_t>(queueCreateInfos.size());
3 CreateInfo.pQueueCreateInfos = queueCreateInfos.data();
```

Если семейства очередей совпадают, то нам нужно передать их индекс только один раз. Наконец, добавьте вызов для получения дескриптора очереди:

```
1 vkGetDeviceQueue(устройство, индексы.настоящее семейство.значение(), 0,
2     &presentQueue);
```

В случае, если семейства очередей совпадают, два дескриптора, скорее всего, теперь будут иметь одинаковое значение. В следующей главе мы рассмотрим цепочки подкачки и как они дают нам возможность отображать изображения на поверхности. Код на C++

Цепочка подкачки

В Vulkan нет концепции "фреймбуфера по умолчанию", следовательно, для него требуется инфраструктура, которая будет владеть буферами, которые мы будем визуализировать до того, как мы визуализируем их на экране. Эта инфраструктура известна как *цепочка обмена* и должна

должны быть созданы явно в Vulkan. Цепочка обмена - это, по сути, очередь изображений, которые ожидают вывода на экран. Наше приложение получит такое изображение, чтобы нарисовать его, а затем вернет его в очередь. Как именно работает очередь и условия представления изображения из очереди зависят от того, как настроена цепочка подкачки, но общая цель цепочки подкачки - синхронизировать представление изображений с частотой обновления экрана.

Проверка поддержки цепи подкачки

Не все видеокарты способны выводить изображения непосредственно на экран по различным причинам, например, потому, что они предназначены для серверов и не имеют каких-либо выходов на дисплей. Во-вторых, поскольку представление изображений сильно привязано к оконной системе и поверхностям, связанным с окнами, это не фактически часть ядра Vulkan. Вы должны включить `VK_KHR_swapchain` расширение устройства после запроса о его поддержке. Для этой

цели мы сначала расширим `isDeviceSuitable` функция проверки, поддерживается ли это расширение. Ранее мы видели, как составить список расширений,

которые поддерживаются а, поэтому сделать это должно быть довольно просто.

Обратите внимание, что заголовочный файл Vulkan предоставляет удобный макрос

`VK_KHR_SWAPCHAIN_EXTENSION_NAME` который определяется как `VK_KHR_swapchain`.

Преимущество использования этого макроса в том, что компилятор будет улавливать орфографические ошибки. Сначала объявите список необходимых расширений устройства, аналогичный списку уровней проверки, которые нужно включить.

```
1 const std::vector<const char*> Расширения устройства = {  
2     VK_KHR_SWAPCHAIN_EXTENSION_NAME ( Имя_расширения )  
3 };
```

Затем создайте новую функцию

`checkDeviceExtensionSupport`

которая вызывается из

```
isDeviceSuitable  
    bool isDeviceSuitable(VkPhysicalDevice  
device) {  
    Индексы QueueFamilyIndices = findQueueFamilies(устройство);  
  
    bool extensionsSupported = checkDeviceExtensionSupport(устройство);  
  
    Возрат индексы.IsComplete() && поддерживаются расширения;  
}  
  
bool checkDeviceExtensionSupport(VkPhysicalDevice device) {  
    возвращает true;  
}
```

Измените тело функции, чтобы перечислить расширения и проверить, есть ли среди них все требуемые расширения.

```

1  бул Проверьте поддержку deviceextensionsupport(VkPhysicalDevice device) {
2      uint 3 2 _t extensionCount;
3      vkEnumerateDeviceExtensionProperties(устройство, nullptr,
4                                              и extensionCount, nullptr);
5
6      std::vector<VkExtensionProperties>
7          доступные расширения(extensionCount);
8      vkEnumerateDeviceExtensionProperties(устройство, nullptr,
9                                              &extensionCount, доступные расширения.data());
10
11     std::set<std::string>
12         требуемые расширения(deviceExtensions.begin(),
13                               deviceExtensions.end());
14
15     для (const auto& расширение : доступные расширения) {
16         требуемые расширения.erase(extension.extensionName);
17     }
18
19     Возврат требуемые расширения.empty();
20 }
```

Я решил использовать здесь набор строк для представления неподтвержденных требуемых расширений. Таким образом, мы можем легко отмечать их галочками при перечислении последовательности доступных расширений. Конечно, вы также можете использовать вложенный цикл, как в checkValidationLayerSupport. Разница в производительности не имеет значения. Теперь запустите код и убедитесь, что ваша видеокарта действительно способна создавать цепочку подкачки. Следует отметить, что доступность очереди представления, как мы проверяли в предыдущей главе, подразумевает, что должно поддерживаться расширение цепочки обмена. Тем не менее, по-прежнему полезно четко указывать вещи, и расширение действительно должно быть явно включено.

Включение расширений устройств

Использование цепочки обмена требует включения VK_KHR_swapchain сначала расширение. Для расширения требуется лишь небольшое изменение в структуре создания логического устройства:

```

1 CreateInfo.enabledExtensionCount =
2     static_cast<uint 3 2 _t>(Расширения устройства.размер());
3 CreateInfo.pppenabledextensionname = deviceExtensions.data();
```

Обязательно замените существующую строку CreateInfo.enabledExtensionCount = 0 ; когда вы это сделаете.

Запрос подробной информации о поддержке цепочки обмена

Просто проверить, доступна ли цепочка подкачки, недостаточно, потому что она может на самом деле быть несовместима с нашей поверхностью окна. Создание цепочки обмена также требует гораздо большего количества настроек, чем создание экземпляра и устройства, поэтому нам нужно запросить дополнительные сведения, прежде чем мы сможем продолжить. В основном нам нужно проверить три типа свойств:

- Базовые возможности surface (минимальное / максимальное количество изображений в цепочке подкачки, min / – максимальная ширина и высота изображений)
- Форматы поверхности (формат пикселей, цветовое пространство)
- Доступные режимы представления,

анalogичные `findQueueFamilies`, мы будем использовать структуру для передачи этих сведений, как только они будут запрошены. Три вышеупомянутых типа свойств представлены в виде

```
1 struct SwapChainSupportDetails {
2     Возможности VkSurfaceCapabilitiesKHR;
3     std::vector<VkSurfaceFormatKHR> форматы;
4     std::vector<VkPresentModeKHR> Текущие режимы;
5 };
```

Теперь мы создадим новую функцию `querySwapChainSupport` которая будет заполнять эту структуру.

```
1 SwapChainSupportDetails запрашивает swapchainsupport(VkPhysicalDevice
2                     устройство) {
3     SwapChainSupportDetails details;
4     Возврат Подробные сведения;
5 }
```

В этом разделе описывается, как запрашивать структуры, содержащие эту информацию. Значение этих структур и какие именно данные они содержат, обсуждается в следующем разделе. Давайте начнем с базовой поверхностью возможности.

Эти свойства удобно для запроса и возвращаются в единое

`VkSurfaceCapabilitiesKHR`

структуре.

```
1 vkGetPhysicalDeviceSurfaceCapabilitiesKHR(устройство, поверхность,
2                                            &детали.возможности);
```

Эта функция учитывает указанные параметры `VkPhysicalDevice` и `VkSurfaceKHR` `window` `surface` при определении поддерживаемых возможностей. Все

функции запроса поддержки имеют эти два параметра в качестве первых, потому что они являются основными компонентами цепочки обмена.

Следующий шаг касается запроса поддерживаемых форматов surface. Поскольку это список структур, он следует знакомому ритуалу из двух вызовов функций.:

```

1 uint 3 2 _t formatCount;
2 vkGetPhysicalDeviceSurfaceFormatsKHR(устройство, поверхность и formatCount,
3                                     nullptr);
4
5 if (formatCount != 0) {
6     Подробные сведения.форматы.изменение размера (formatCount);
7     vkGetPhysicalDeviceSurfaceFormatsKHR(устройство, поверхность,
8                                         и количество форматов, подробные сведения.форматы.данные());
9 }

```

Убедитесь, что размер вектора изменен таким образом, чтобы он соответствовал всем доступным форматам. И наконец, запрос поддерживаемых режимов представления работает точно так же как `vkGetPhysicalDeviceSurfacePresentModesKHR`:

```

1 uint 3 2 _t presentModeCount;
2 vkGetPhysicalDeviceSurfacePresentModesKHR(устройство, поверхность,
3                                            &presentModeCount, nullptr);
4
5 if (presentModeCount != 0) {
6     подробности.Настоящие режимы.изменение размера (presentModeCount);
7     vkGetPhysicalDeviceSurfacePresentModesKHR(устройство, поверхность,
8                                                 &presentModeCount, details.presentModes.data());
9 }

```

Все детали теперь в структуре, так что давайте расширим `isDeviceSuitable` еще раз использовать эту функцию, чтобы убедиться в адекватности поддержки цепочки обмена. Поддержка цепочки подкачки достаточна для этого руководства, если существует хотя бы один поддерживаемый формат изображения и один поддерживаемый режим представления с учетом

```

1 имеющейся у нас поверхности окна.bool swapChainAdequate = значение false;
2 if (Поддерживаемые расширения) {
3     SwapChainSupportDetails swapChainSupport =
4         Запрос к swapchainsupport(устройству);
5     swapChainAdequate = !swapChainSupport.formats.empty() &&
6                         !swapChainSupport.presentModes.empty();
7 }

```

Важно, чтобы мы пытались запросить поддержку цепочки обмена только после проверки доступности расширения. Последняя строка функции изменяется на:

```

1 Возврат indexes.IsComplete() && поддерживаются расширения &&
2 swapChainAdequate;

```

Выбор правильных настроек для цепочки

обмена Если `swapChainAdequate` **условия были выполнены, тогда поддержка определенно доступна** **достаточна, но все еще может быть много различных режимов с различной оптимальностью.**

Теперь мы напишем пару функций для поиска правильных настроек для наилучшей возможной цепочки обмена. Необходимо определить три типа настроек.:

- Формат поверхности (глубина цвета)
- Режим представления (условия "подкачки" изображений на экран)
- Масштаб подкачки (разрешение изображений в цепочке подкачек)

Для каждого из этих параметров мы будем иметь в виду идеальное значение, которое мы будем использовать, если оно доступно, а в противном случае мы создадим некоторую логику для поиска следующего наилучшего варианта.

Формат поверхности Функция для этой настройки начинается следующим образом. Позже мы передадим `formats` элемент `SwapChainSupportDetails struct` в качестве аргумента `ment`.

```
1 VkSurfaceFormatKHR выбирает wapsurfaceformat(const 1
2     std::vector<VkSurfaceFormatKHR> и доступные форматы) {
3 }
```

`VkSurfaceFormatKHR` Каждый запись содержит а `format` и `colorSpace` и `memm-` файл. Элемент `format` задает цветовые каналы и типы. Например,

`VK_FORMAT_B 8 G 8 R 8 A 8 _SRGB` означает, что мы сохраняем B, G, R и альфа-канал выполняется в таком порядке с 8-битным целым числом без знака, всего 32 бита на пиксель. Элемент указывает, поддерживается ли цветовое пространство SRGB или не используется

Отметить. Обратите внимание, что этот флаг

раньше назывался `VK_COLORSPACE_SRGB_NONLINEAR_KHR` в старых версиях спецификации. Для цветового пространства мы будем использовать SRGB, если оно доступно, потому что это приводит к более точному восприятию цветов.

Это также в значительной степени стандартное цветовое пространство для изображений, как и текстуры, которые мы будем использовать позже.

Из-за этого мы должны также использовать цветовой формат SRGB, одним из наиболее распространенных из которых является `VK_FORMAT_B 8 G 8 R 8 A 8 _SRGB`.

Давайте пройдемся по списку и посмотрим, доступна ли предпочтительная

```
1 комбинация: для (const auto& available Format : доступные форматы) {
2     if (доступный формат.format == VK_FORMAT_B 8 G 8 R 8 A 8 _SRGB &&
3         доступный формат.Цветовое пространство
4             == VK_COLOR_SPACE_SRGB_NONLINEAR_KHR)
5             { Возврат доступный формат;
6     }
7 }
```

Если это также не удастся, мы могли бы начать ранжирование доступных форматов на основе того, насколько они "хороши", но в большинстве случаев можно просто остановиться на первом указанном формате.

```

1 VkSurfaceFormatKHR выбирает wapsurfaceformat(const
2     std::vector<VkSurfaceFormatKHR>и
3     доступные форматы) { для (const auto и
4     доступный формат: доступные форматы) { 2
5         if (доступный формат.format == VK_FORMAT_B8G8R8A8_SRGB &&
6             Доступный формат.Цветовое пространство
7             == VK_COLOR_SPACE_SRGB_NONLINEAR_KHR)
8             { Возврат доступный формат;
9         }
10    }
11
12    Возврат Доступные форматы[ 0 ];
13 }

```

Режим представления Режим презентации, пожалуй, является наиболее важной настройкой для цепочки обмена, поскольку он отражает фактические условия отображения изображений на экране. В Vulkan доступны четыре возможных режима работы:

: Изображения, отправленные вашим приложением-
изображения сразу переносятся на экран, что может привести к разрыву.
VK_PRESENT_MODE_FIFO_KHR • получает изображение из [цепочки](#) обмена - это очередь, в которой отображается предыдущая, если заявка подана с опозданием и очередь была пустой на момент поступления. Вместо ожидания следующего вертикального пробела, программа вставляет отрисованные изображения в конец очереди. Если очередь заполнена, программе придется подождать. Это больше всего похоже на вертикальную синхронизацию, которая встречается в современных играх. Момент обновления дисплея называется "вертикальным пробелом".
..VK_PRESENT_MODE_FIFO_RELAXED_KHR: Этот режим отличается только от предыдущая, если заявка подана с опозданием, и очередь была пустой на момент последней . Вместо ожидания следующего вертикального пробела, изображение переносится сразу после его получения. Это может привести к видимым разрывам. •
VK_PRESENT_MODE_MAILBOX_KHR: Это еще одна разновидность второго режима. Вместо блокировки приложения при заполнении очереди, изображения, которые уже находятся в очереди, просто заменяются более новыми.
Этот режим можно использовать для максимально быстрой визуализации кадров, избегая при этом разрывов, что приводит к меньшему количеству проблем с задержкой, чем при стандартной вертикальной синхронизации. Это обычно известно как "тройная буферизация", хотя наличие только трех буферов не обязательно означает, что частота кадров разблокирована.
Только the VK_PRESENT_MODE_FIFO_KHR режим гарантированно будет доступен, поэтому нам снова придется написать функцию, которая ищет наилучший

```

1 доступный режим: vkPresentModeKHR выбирает wappresentmode(const
2     std::vector<VkPresentModeKHR>и доступные
3     presentmodes) { Возврат VK_PRESENT_MODE_FIFO_KHR;
4 }

```

Я лично считаю, что `VK_PRESENT_MODE_MAILBOX_KHR` это очень хороший компромисс, если потребление энергии не вызывает беспокойства. Это позволяет нам избежать разрыва при сохранении довольно низкой задержки за счет рендеринга новых изображений, которые являются максимально актуальными вплоть до вертикального пробела. На мобильных устройствах, где потребление энергии более важно, вы, вероятно, захотите использовать вместо этого `VK_PRESENT_MODE_FIFO_KHR`. Теперь давайте просмотрим список, чтобы увидеть, есть ли `VK_PRESENT_MODE_MAILBOX_KHR` доступно:

```
VkPresentModeKHR выбирает wappresentmode(const 1
    std::vector<VkPresentModeKHR>& available presentmodes) {
    для (const auto& availablePresentMode : доступные представленные режимы) {
        2

        если (доступный текущий режим == VK_PRESENT_MODE_MAILBOX_KHR) {
            3
            Возврат доступный текущий режим;
            4
        }
        5
    }
    6
}
8     Возврат VK_PRESENT_MODE_FIFO_KHR;
9 }
```

Поменять местами экстент Остается только одно основное свойство, для которого мы добавим последнюю функцию:

```
VkExtent2D выбирает wapextent(const VkSurfaceCapabilitiesKHR& 1
    возможности) {
    2
}
3 }
```

Масштаб подкачки - это разрешение изображений цепочки подкачки, и оно почти всегда точно равно разрешению окна, в котором мы рисуем в пикселях (подробнее об этом чуть позже). Диапазон возможных разрешений определен в `VkSurfaceCapabilitiesKHR` структуре. Vulkan предлагает нам соответствовать разрешению окна, установив ширину и высоту в поле `currentExtent`. Участник. Однако некоторые оконные менеджеры позволяют нам отличаться здесь, и на это указывает установка ширины и высоты в максимальное значение. В этом случае мы выберем наилучшее разрешение. `currentExtent` соответствует окну в пределах границ `minImageExtent` и `maxImageExtent`. Но мы должны указать разрешение в правильной единице измерения.

GLFW использует две единицы измерения при измерении размеров: пиксели и координаты экрана. Для примера, разрешение {ширина, высота} то, что мы указали ранее при создании окна, измеряется в экранных координатах. Но Vulkan работает с пикселями, поэтому протяженность цепочки подкачки также должна быть указана в пикселях. К сожалению, если вы используете дисплей с высоким разрешением (например, Retina display от Apple), экранные координаты не соответствуют пикселям. Вместо этого, из-за более высокой плотности пикселей, разрешение окна в пикселях будет больше, чем разрешение в координатах экрана. Так что, если Vulkan не исправит для нас экстент подкачки, мы не можем просто использовать оригинал {ширина, высота}. Вместо, мы должны использовать `glfwGetFramebufferSize`

запрашивать разрешение окна в пикселях перед сопоставлением его с
минимальным и максимальным размером изображения.

```
1 # включить <cstdint> // Необходимо для uint32_t
2 # включить <ограничения> // Необходимо для std::числовые ограничения
3 # включить <алгоритм> // Необходимо для std::clamp
4 ...
5 VkExtent2D выбирает wapextent(const VkSurfaceCapabilitiesKHR&
6
7     возможности) {
8
9     если (возможности.текущий.ширина != std::числовые
10         ограничения<uint32_t>::max()) { Возврат возможности.currentExtent;
11     } еще {
12         int ширина, высота;
13         glfwGetFramebufferSize(окно, &ширина, &высота);
14
15         VkExtent2D actualExtent = {
16             static_cast<uint32_t>(ширина),
17             static_cast<uint32_t>(высота)
18         };
19
20         actualExtent.width = std::зажим(фактический.размер.ширина,
21             возможности.Минимальный.размер.ширина,
22             возможности.Максимальный.размер.ширина);
23         фактическая.протяженность.высота = std::зажим(фактическая.длина.высота,
24             возможности.Минимальная.протяженность.высота,
25             возможности.Максимальная.протяженность.высота);
26
27         Возврат actualExtent;
28     }
29 }
30 }
```

высота В функция clamp здесь используется для привязки значе

между разрешенными минимальным и максимальным экстентами, которые поддерживаются реализацией.

Создание цепочки подкачки.

Теперь, когда у нас есть все эти вспомогательные функции, помогающие нам с выбором, который мы должны сделать во время выполнения, у нас, наконец, есть вся информация, необходимая для создания рабочей цепочки обмена. Создайте а CreateSwapChain функцию, которая запускается с результатами этих вызовов, и обязательно вызывайте ее из initVulkan после создания логического устройства. аннулировать Инициирующий вулкан() {

```

2     CreateInstance();  setupDebugMessenger();
3
4     createSurface();
5
6     pickPhysicalDevice();
7
8     createLogicalDevice();
9
10    CreateSwapChain();
11
12
13    анулирование CreateSwapChain() {
14
15        SwapChainSupportDetails swapChainSupport =
16
17            Запрос swapchainsupport(физическое устройство);
18
19        VkSurfaceFormatKHR Формат поверхности =
20
21            Выбирает wapsurfaceformat(swapChainSupport.formats);
22
23        VkPresentModeKHR presentMode =
24
25            Выбирает wappresentmode(swapChainSupport.presentModes);
26
27        Экстент VkExtent2D =
28
29            Выбирает waextent(swapChainSupport.capabilities);
30
31    }

```

Помимо этих свойств, мы также должны решить, сколько изображений мы хотели бы иметь в цепочке обмена. Реализация определяет минимальное число, которое требуется для функционирования:

```
1 uint32_t imageCount = swapChainSupport.capabilities.minImageCount;
```

Однако простое соблюдение этого минимума означает, что иногда нам может потребоваться дождаться завершения внутренних операций драйвером, прежде чем мы сможем получить другое изображение для рендеринга. Поэтому рекомендуется запрашивать по крайней мере на одно изображение больше минимального:

```
1 uint32_t imageCount = swapChainSupport.capabilities.minImageCount +
2
3 1;
```

Мы также должны следить за тем, чтобы не превышать максимальное количество изображений при выполнении этого, где 0 является специальным значением, которое означает, что максимального

```
1 значение нет: если (swapChainSupport.capabilities.maxImageCount > 0 && imageCount >
2
3     swapChainSupport.capabilities.maxImageCount) {
4
5     imageCount = swapChainSupport.capabilities.maxImageCount;
6
7 }
```

Как и в случае с объектами Vulkan, для создания объекта swap chain требуется заполнение большой структуры. Начинается все очень знакомо:

```
1 VkSwapchainCreateInfoKHR CreateInfo{};
2
3 CreateInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
4
5 CreateInfo.surface = поверхность;
```

После указания того, к какой поверхности должна быть привязана цепочка

подкачки, указываются детали изображений цепочки подкачки:

```
1 CreateInfo.minImageCount = imageCount;
2 CreateInfo.ImageFormat = surfaceFormat.format;
3 CreateInfo.imageColorSpace = surfaceFormat.colorSpace;
4 CreateInfo.imageExtent = extent;
5 CreateInfo.imageArrayLayers = 1;
6 CreateInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
```

В `imageArrayLayers` определяет количество слоев, из которых состоит каждое изображение.

Это всегда 1 если только вы не разрабатываете стереоскопическое 3D-приложение. The `imageUsage` битовое поле определяет, для каких операций мы будем использовать изображения в цепочке обмена. В этом уроке мы собираемся визуализировать непосредственно их, что означает, что они используются в качестве цветного вложения. Также возможно, что вы сначала отобразите изображения в отдельном изображении для выполнения операций, подобных постобработке. В этом случае вы можете использовать значение типа `VK_IMAGE_USAGE_TRANSFER_DST_BIT` вместо этого и используйте операцию с памятью для переноса отрисованного изображения в цепочку подкачки изображения.

```
1 Индексы QueueFamilyIndices = findQueueFamilies(физическое устройство);
2 uint 3 2 _t queueFamilyIndices[] = {indexes.graphicsFamily.value(),
3                                     индексы.Настоящее семейство.значение()};
4
5 если (indexes.graphicsFamily != индексы.Настоящее семейство) {
6     CreateInfo.imageSharingMode = VK_SHARING_MODE_CONCURRENT;
7     CreateInfo.queueFamilyIndexCount = 2;
8     CreateInfo.pQueueFamilyIndices = queueFamilyIndices;
9 } еще {
10     CreateInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
11     CreateInfo.queueFamilyIndexCount = 0; // Необязательно
12     CreateInfo.pQueueFamilyIndices = nullptr; // Необязательно
}
```

Далее нам нужно указать, как обрабатывать изображения цепочки подкачки, которые будут использоваться в нескольких семействах очередей. Это будет иметь место в нашем приложении, если семейство графических очередей отличается от очереди презентаций. Мы будем рисовать на изображениях в цепочке подкачки из графической очереди, а затем отправлять их в очередь презентаций. Существует два способа обработки изображений, к которым обращаются из нескольких очередей:

• `VK_SHARING_MODE_EXCLUSIVE`: Изображение принадлежит одному семейству очередей одновременно, и право собственности должно быть явно передано перед использованием в другом семействе очередей. Этот параметр обеспечивает наилучшую производительность.

• `VK_SHARING_MODE_CONCURRENT`: Изображения могут использоваться в нескольких очередях семейства без явной передачи прав собственности.

Если семейства очередей отличаются, то в этом мы будем использовать параллельный режим

учебное пособие, позволяющее избежать необходимости читать главы о владении, поскольку они включают в себя некоторые концепции, которые лучше будут объяснены позже. Параллельный режим требует, чтобы вы заранее указали, между какими семействами очередей будет распределено право собственности используя `queueFamilyIndexCount` и `queueFamilyIndices` параметры. Если семейство графических очередей и семейство очередей презентаций одинаковы, что будет иметь место на большинстве аппаратных средств, тогда мы должны придерживаться эксклюзивного режима, поскольку параллельный режим требует, чтобы вы указали по крайней мере два разных семейства очередей.

```
1 CreateInfo.Предварительная трансформация =
    swapChainSupport.capabilities.currentTransform;

```

Мы можем указать, что к изображениям в цепочке подкачки должно быть применено определенное преобразование, если оно поддерживается ([Поддерживаемые преобразования в возможностях](#)), например, поворот на 90 градусов по часовой стрелке или горизонтальное сальто. Чтобы указать, что вы не хотите никаких преобразований, просто укажите текущее преобразование.

```
1 CreateInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
```

Поле `compositeAlpha` указывает, следует ли использовать альфа-канал для наложения-взаимодействие с другими окнами в оконной системе. Вы почти всегда захотите просто игнорировать альфа-канал, следовательно `VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR`.

```
1 CreateInfo.presentMode = текущий режим;
2 CreateInfo.clipped = VK_TRUE;
```

Параметр `presentMode` говорит сам за себя. Если для параметра обрезанный установлено значение `VK_TRUE` тогда это означает, что нас не волнует цвет пикселей, которые затемнены, например, потому, что перед ними находится другое окно. Если вам действительно не нужно иметь возможность считывать эти пиксели обратно и получать предсказуемые результаты, вы добьетесь наилучшей производительности, включив

```
1 отсечение.CreateInfo.oldSwapchain = VK_NULL_HANDLE;
```

Остается последнее поле, . С Vulkan возможно, что ваш `oldSwapChain` цепочка подкачки становится недействительной или неоптимизированной во время работы вашего приложения, например, из-за изменения размера окна. В этом случае цепочку обмена фактически необходимо воссоздать с нуля, и в этом поле должна быть указана ссылка на старую. Это сложная тема, о которой мы узнаем больше в следующей главе. Пока предположим, что мы создадим только одну

цепочку обмена. Теперь добавьте член класса для хранения объекта `VkSwapchainKHR` :

```
1 VkSwapchainKHR swapChain;
```

Создание цепочки обмена теперь так же просто, как вызов `vkCreateSwapchainKHR`:

```
1 если (vkCreateSwapchainKHR(устройство, &CreateInfo, nullptr, & swapChain)
        != VK_SUCCESS) {
```

```
2     бросок std::runtime_error("не удалось создать цепочку обмена!");
3 }
```

Параметрами являются логическое устройство, информация о создании цепочки подкачки, необязательные пользовательские распределители и указатель на переменную, в которой хранится дескриптор. **Никаких сюрпризов.** Его следует

```
1 очистить перед использованием устройства: аннулирование очистка() {
2     vkDestroySwapchainKHR (устройство, цепочка
3     обмена, nullptr); ...
4 }
```

Теперь запустите приложение, чтобы убедиться, что цепочка обмена создана успешно!

Если на этом этапе вы получите сообщение об ошибке нарушения доступа в `vkCreateSwapchainKHR` или увидите сообщение типа Не удалось найти 'vkGetInstanceProcAddress' в `layer SteamOverlayVulkanLayer.dll`, тогда ознакомьтесь с часто задаваемыми вопросами о поверхности Steam.

Попробуйте удалить `CreateInfo.imageExtent = extent;` строка с включенными слоями проверки. Вы увидите, что один из слоев проверки немедленно распознает ошибку и выводится полезное сообщение.:

```
validation layer: vkCreateSwapchainKHR() called with pCreateInfo->imageExtent - (0,0), which is not equal to the currentExtent - (800,600) returned by vkGetPhysicalDeviceSurfaceCapabilitiesKHR().
```

Извлечение изображений цепочки подкачки

Цепочка подкачки уже создана, поэтому все, что остается, - это извлечение дескрипторов `VkImage` -ов в ней. Мы будем ссылаться на них во время операций рендеринга в последующих главах. Добавьте член класса для

```
1 хранения дескрипторов: std::vector<VkImage> swapChainImages;
```

Изображения были созданы реализацией для цепочки обмена, и они будут автоматически очищаться после уничтожения цепочки обмена, поэтому нам не нужно добавлять какой-либо код очистки. Я добавляю код для извлечения дескрипторов в конец цепочки обмена. `CreateSwapChain` функция, сразу после вызова `vkCreateSwapchainKHR`.
Их извлечение очень похоже на другие случаи, когда мы извлекали массив дескрипторов. Мы извлекали только минимальное количество изображений в цепочке подкачки, поэтому реализации разрешено создавать цепочку подкачки с большим количеством.
Вот почему мы сначала запросим конечное количество изображений с помощью `vkGetSwapchainImagesKHR` чтобы получить дескрипторы., затем измените размер контейнера и, наконец, вызовите его СНОВА `vkGetSwapchainImagesKHR(устройство, цепочка обмена и количество изображений, nullptr); swapChainImages.resize(количество изображений);`

```
3 vkGetSwapchainImagesKHR(устройство, цепочка обмена и количество изображений,
                           swapChainImages.data());
```

И последнее, сохраните формат и экстент, которые мы выбрали для цепочки подкачки изображений в переменных-членах. Они понадобятся нам в

```
1 следующих главах. VkSwapchainKHR swapChain;
2 std::vector<VkImage> swapChainImages;
3 VkFormat swapChainImageFormat;
4
5 VkExtent2D swapChainExtent;
6 ...
7 swapChainImageFormat = surfaceFormat.format;
8 swapChainExtent = extent;
```

Теперь у нас есть набор изображений, которые могут быть нарисованы на окне и представлены в нем. В следующей главе будет рассказано, как мы можем настроить изображения в качестве целей рендеринга, а затем мы начнем изучать сам графический конвейер и команды рисования!

Код на C++

Просмотр изображений

Для использования любых `VkImage`, включая те, что находятся в цепочке обмена, в конвейере рендеринга мы должны создать объект `VkImageView`. Просмотр изображения - это в буквальном смысле просмотр изображения. В нем описывается, как получить доступ к изображению и к какой части изображения получить доступ, например, следует ли его обрабатывать как текстуру глубины 2D-текстуры без каких-либо уровней `mipmapping`. В этой главе мы напишем а `createImageViews` функция, которая создает базовый вид изображения для каждого изображения в цепочке подкачки, чтобы мы могли использовать их в качестве цветовых целей позже.

1 Сначала добавьте элемент класса для хранения представлений изображений в:

```
std::vector<VkImageView> swapChainImageViews;
```

Создайте `createImageViews` функция и вызовите ее сразу после создания цепочки

```
обмена. void initVulkan() {
1
2     CreateInstance(); setupDebugMessenger();
3     createSurface();
4     pickPhysicalDevice();
5     createLogicalDevice();
6     CreateSwapChain();
```

```
    createImageViews();
9}8
10
аннулирование createImageViews()
{ 11
12
13 }
```

Первое, что нам нужно сделать, это изменить размер списка, чтобы он соответствовал всем видам изображений, которые мы будем создавать:

```
1 аннулировать createImageViews() {
2
3     swapChainImageViews.resize(swapChainImages.size());
4 }
```

Затем настройте цикл, который выполняет итерацию по всем изображениям

```
1 цепочки подкачки. для (size_t i = 0; i < swapChainImages.size(); i++) {
2
3 }
```

Параметры для создания вида изображения указаны в `VkImageViewCreateInfo` структура. Первые несколько параметров просты.

```
1 VkImageViewCreateInfo CreateInfo{};
2 CreateInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
3 CreateInfo.image = swapChainImages[i];
```

В полях `viewType` и `format` указывается, как должны быть интерпретированы данные изображения. В поле `viewType` параметр позволяет обрабатывать изображения как одномерные текстуры, 2D текстуры, 3D текстуры и

```
1 кубические карты. CreateInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
2 CreateInfo.format = swapChainImageFormat;
```

В компоненты поле позволяет изменять цветовые каналы по кругу. Для например, вы можете сопоставить все каналы с красным каналом для получения монохромной текстуры. Вы также можете сопоставить постоянные значения 1 с каналом. В нашем случае мы будем придерживаться сопоставления по умолчанию.

```
1 CreateInfo.components.r = VK_COMPONENT_SWIZZLE_IDENTITY;
2 CreateInfo.components.g = VK_COMPONENT_SWIZZLE_IDENTITY;
3 CreateInfo.components.b = VK_COMPONENT_SWIZZLE_IDENTITY;
4 CreateInfo.components.a = VK_COMPONENT_SWIZZLE_IDENTITY;
```

Поле `вложенный ресурс` описывает назначение изображения и к какой части изображения следует получить доступ. Наши изображения будут использоваться в качестве цветных мишней без каких-либо уровней `mipmapping` или нескольких слоев.

```
1 CreateInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
2 CreateInfo.subresourceRange.baseMipLevel = 0;
3 CreateInfo.subresourceRange.levelCount = 1;
4 CreateInfo.subresourceRange.baseArrayLayer = 0;
5 CreateInfo.subresourceRange.Количество слоев = 1;
```

Если бы вы работали над стереографическим 3D-приложением, то вы бы создали цепочку подкачки с несколькими слоями. Затем вы могли бы создать несколько видов изображений для каждого изображения, представляющего виды для левого и правого глаза, получив доступ к разным слоям.

Создание вида изображения теперь сводится к вызову `vkCreateImageView`:

```
1 if (vkCreateImageView(устройство, &CreateInfo, nullptr,
2     &swapChainImageViews[i]) != VK_SUCCESS) {
3     вбросить std::runtime_error("не удалось создать представления изображений!");
4 }
```

В отличие от изображений, представления изображений были явно созданы нами, поэтому нам нужно добавить аналогичный цикл, чтобы снова уничтожить

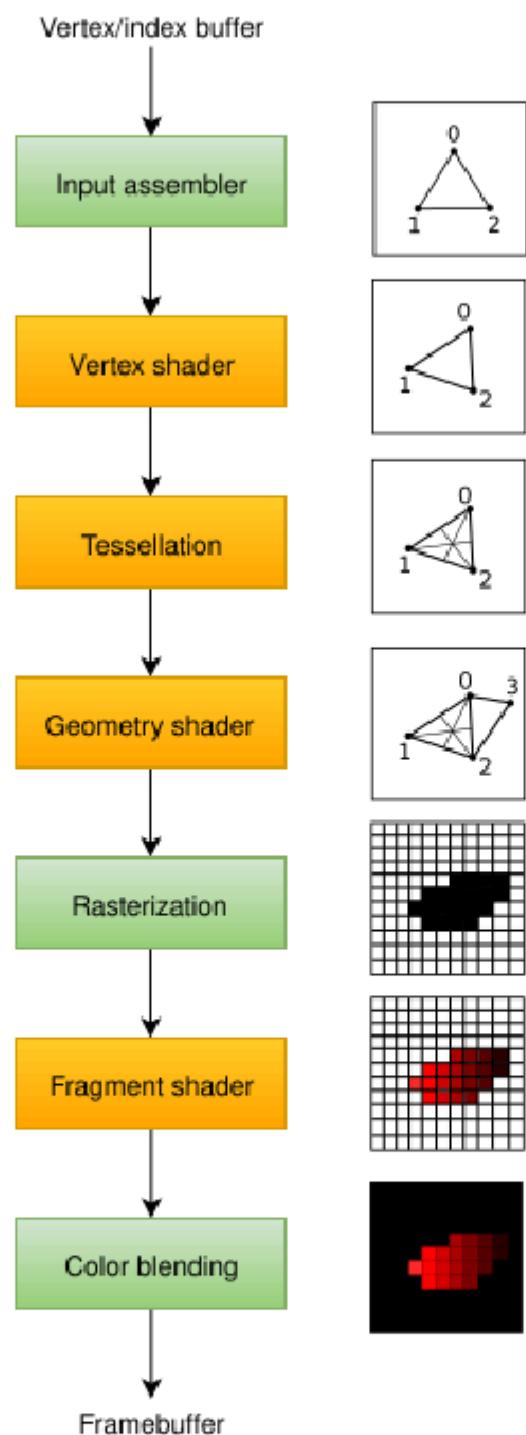
```
1 иx в конце программы: void cleanup() {
2     для (автоматического ImageView : swapChainImageViews) {
3         vkDestroyImageView(устройство, ImageView, nullptr);
4     }
5     ...
6 }
```

Просмотра изображения достаточно, чтобы начать использовать изображение в качестве текстуры, но оно еще не совсем готово к использованию в качестве цели рендеринга. Для этого требуется еще один шаг косвенного управления, известный как фреймбуфер. Но сначала нам нужно настроить графический конвейер. Код на C++

Основы графического конвейера

Введение

В течение следующих нескольких глав мы будем настраивать графический конвейер, который настроен для рисования нашего первого треугольника. Графический конвейер модель-SE -quence операций, которые принимают вершин и текстур в модели все пути до точек расставлять приоритеты. Упрощенный обзор отображаемые ниже:



В сборщик входных данных собирает необработанные данные о вершинах из указанных вами буферов, а также может использовать индексный буфер для повторения определенных элементов без дублирования самих данных о вершинах. The вершинный шейдер запускается для каждой вершины и обычно применяет преобразования

чтобы преобразовать положения вершин из пространства модели в пространство экрана. Он также передает данные для каждой вершины по конвейеру.

В *шейдеры тесселяции* позволяют подразделять геометрию на основе определенных правил для повышения качества сетки. Это часто используется для изготовления поверхностей, таких как кирпичные стены и лестницы выглядят менее плоские, когда они находятся поблизости.

В *геометрия шейдеров* выполняется на каждом примитиве (треугольник, линия, точка) и может отбросить его или вывести больше примитивов, чем было введено. Это похоже на мозаичный шейдер, но гораздо более гибкий. Тем не менее, он мало используется в современных приложениях, потому что производительность не так высока на большинстве видеокарт, за исключением встроенных графических процессоров Intel.

Растеризация stage разбивает примитивы на фрагменты. Это

пиксельные элементы, которые они заполняют в кадровом буфере. Все фрагменты, которые выпадают за пределы экрана, отбрасываются, а атрибуты, выводимые вершинным шейдером, интерполируются по фрагментам, как показано на рисунке. Обычно фрагменты, которые находятся за фрагментами других примитивов, также отбрасываются здесь из-за тестирования глубины.

В *Шейдер фрагментов* вызывается для каждого сохранившегося фрагмента и определяет, в какой фреймбуфер (ы) записаны фрагменты и с какими значениями цвета и глубины.

Это можно сделать, используя интерполированные данные из вершинного шейдера, которые могут включать такие вещи, как координаты текстуры и нормали для освещения. Функция *Смешивание цветов stage* применяет операции для смешивания различных фрагментов, которые отображаются на один и тот же пиксель в кадровом буфере. Фрагменты могут просто перезаписывать друг друга, складываться или смешиваться в зависимости от прозрачности.

Этапы с зеленым цветом известны как *этапы с фиксированной функцией*. Эти этапы позволяют настраивать их работу с помощью параметров, но способ их работы предопределен. Этапы,

выделенные оранжевым цветом, с другой стороны, являются *программируемыми*, что означает, что вы можете загрузить свой собственный код на видеокарту, чтобы применить именно те операции, которые вы хотите. Это позволяет вам использовать фрагментные шейдеры, например, для реализации чего угодно, от текстурирования и освещения до трассировки лучей.

Эти программы запускаются на многих ядрах графического процессора одновременно для параллельной обработки множества объектов, таких как вершины и фрагменты. Если вы раньше использовали старые API, такие как OpenGL и Direct3D, то вы привыкнете к возможности изменять любые настройки конвейера по желанию с помощью вызовов типа `glBlendFunc` и `OMSetBlendState`. Графический конвейер в Vulkan почти полностью неизменяем, поэтому вам необходимо воссоздать конвейер с нуля, если вы хотите изменить шейдеры, привязать разные фреймбуфера или изменить функцию наложения. Недостатком является то, что вам придется создать несколько конвейеров, представляющих все различные комбинации состояний, которые вы хотите использовать в своих операциях рендеринга. Однако, поскольку все операции, которые вы будете выполнять в конвейере, известны заранее, драйвер может оптимизировать их гораздо лучше.

Некоторые из программируемых этапов являются необязательными в зависимости от того, что вы собираетесь делать. Например, этапы тесселяции и геометрии могут быть отключены, если вы просто рисуете простую геометрию. Если вас интересуют только значения глубины, то вы можете отключить этап шейдера фрагментов, который полезен для генерации карты теней. В следующей главе мы сначала создадим два программируемых этапа, необходимых для вывода треугольника на экран: вершинный шейдер и фрагментный шейдер. Конфигурация с фиксированными функциями, такая как режим наложения, область просмотра, растирование, будет настроена в следующей главе. Заключительная часть настройки графического конвейера в Vulkan включает в себя спецификацию входных и выходных фреймбуферов. Создайте а `createGraphicsPipeline` функция, вызываемая сразу после `createImageViews` в `initVulkan`. Мы будем работать над этой функцией на протяжении следующих глав.

```
void Инициализирующий вулкан() {
1
2     CreateInstance();    setupDebugMessenger();
3     createSurface();
4     pickPhysicalDevice();
5     createLogicalDevice();
6     CreateSwapChain();
7     createImageViews();
8     Создайте графический конвейер();
9
10    }
11
12    ...
13
14    пустота createGraphicsPipeline() {
15
16    }
```

Код на C++

Модули шейдеров

В отличие от более ранних API, шейдерный код в Vulkan должен быть указан в формате байт-кода в отличие от удобочитаемого синтаксиса, такого как GLSL и HLSL. Этот формат байтового кода называется SPIR-V и предназначен для использования как с Vulkan, так и с OpenCL (оба API Khronos). Это формат, который можно использовать для написания графики и вычисления шейдеров, но в этом руководстве мы сосредоточимся на шейдерах, используемых в графических конвейерах Vulkan. Преимущество использования формата байт-кода заключается в том, что компиляторы, написанные производителями графических процессоров для преобразования шейдерного кода в машинный, значительно менее сложны. опыт прошлого показал, что благодаря удобочитаемому синтаксису, такому как GLSL, некоторые производители графических процессоров были довольно гибкими в своей интерпретации стандарта. Если вам случится

пишите нетривиальные шейдеры с графическим процессором от одного из этих поставщиков, тогда вы рискуете, что драйверы других поставщиков отклонят ваш код из-за синтаксических ошибок или, что еще хуже, ваш шейдер будет работать по-другому из-за ошибок компилятора. С помощью простого формата байт-кода, такого как SPIR-V, этого, надеюсь, удастся избежать.

Однако это не означает, что нам нужно писать этот байт-код вручную. Компания Khronos выпустила свой собственный, независимый от поставщика компилятор, который компилирует GLSL в SPIR-V. Этот компилятор предназначен для проверки того, что ваш шейдерный код полностью соответствует стандартам и создает один двоичный файл SPIR-V, который вы можете отправить с вашей программой. Вы также можете включить этот компилятор в качестве библиотеки для создания SPIR-V во время выполнения, но в этом руководстве мы не будем этого делать. Хотя мы можем использовать этот компилятор напрямую через `glslangValidator.exe`, мы будем использовать `glslc.exe` вместо этого от Google. Преимущество `glslc` заключается в том, что он использует тот же формат параметров, что и известные компиляторы, такие как GCC и Clang, и включает некоторые дополнительные функции, такие как `включает`.

Оба они уже включены в Vulkan SDK, поэтому вам не нужно загружать ничего дополнительного.

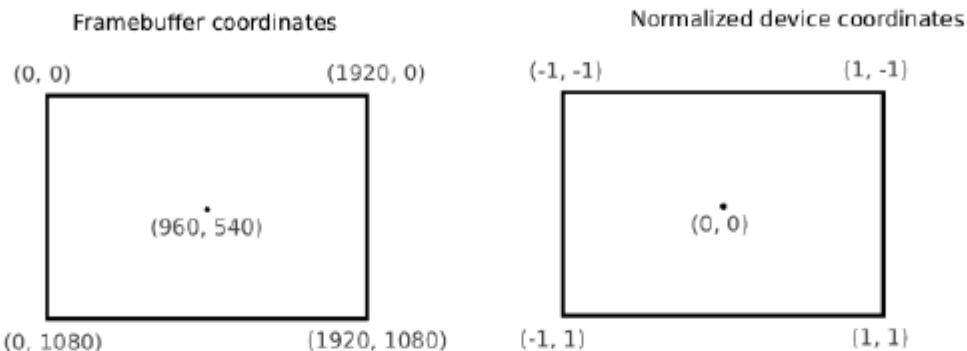
GLSL - это язык затенения с синтаксисом в стиле C. Программы, написанные на нем, имеют функцию, которая вызывается для каждого объекта. Вместо использования параметров

[главная](#)
для ввода и возвращаемого значения в качестве выходных данных GLSL использует глобальные переменные для обработки ввода и вывода. Язык включает в себя множество функций, помогающих в графическом программировании, таких как встроенные векторные и матричные примитивы. Включены функции для операций, таких как перекрестные произведения, матрично-векторные произведения и отражения вокруг вектора. Тип вектора называется `vec` числом, указывающим количество элементов . Например, трехмерная позиция будет сохранена в `a vec 3`. Возможно получить доступ к отдельным компонентам через элементы, такие как `.x`, но также возможно создать новый вектор из нескольких компонентов одновременно. Например, выражение `vec 3 (1 . 0 , 2 . 0 , 3 . 0) . xy` привело бы к `vec 2`. Конструкторы векторов могут также принимать комбинации векторных объектов и скалярных значений. Например, `a vec 3` может быть сконструирован `C vec 3 (vec 2 (1 . 0 , 2 . 0), 3 . 0).` Как упоминалось в предыдущей главе, нам нужно написать вершинный шейдер и фрагментный шейдер, чтобы получить треугольник на экране. В следующих двух разделах будет рассмотрен GLSL-код каждого из них, а после этого я покажу вам, как создать два двоичных файла SPIR-V и загрузить их в программу.

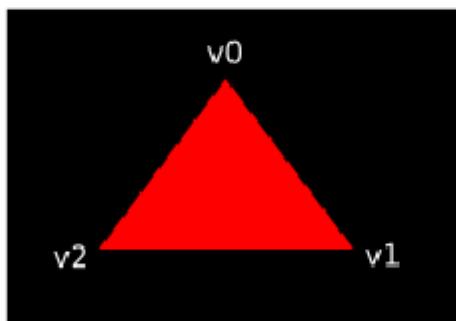
Вершинный шейдер

Вершинный шейдер обрабатывает каждую входящую вершину. Он принимает ее атрибуты, такие как мировое положение, цвет, нормальные и текстурные координаты в качестве входных данных. Результатом является конечная позиция в координатах клипа и атрибуты, которые необходимо передать шейдеру фрагмента, например, координаты цвета и текстуры. Эти значения будут затем интерполированы растрировщиком по фрагментам для получения плавного градиента. А *координата клипа* представляет собой четырехмерный вектор из вершинного шейдера, который впоследствии преобразуется в *нормализованную координату устройства* путем деления всего вектора на его последнюю составляющую. Эти нормализованные координаты устройства являются *homogenized coordinates*.

общие координаты, которые отображают фреймбуфер в систему координат $[-1, 1]$ by $[-1, 1]$, которая выглядит следующим образом:



Вы уже должны быть знакомы с ними, если вы раньше увлекались компьютерной графикой. Если вы раньше использовали OpenGL, то вы заметите, что знак координат Y теперь перевернут. Координата Z теперь использует тот же диапазон, что и в Direct3D, от 0 до 1. Для нашего первого треугольника мы не будем применять никаких преобразований, мы просто укажем положения трех вершин непосредственно как нормализованные координаты устройства, чтобы создать следующую фигуру:



Мы можем напрямую выводить нормализованные координаты устройства, выводя их в виде координат клипа из вершинного шейдера¹, для последнего компонента которого установлено значение . Таким образом, разделение для преобразования координат клипа в нормализованные координаты устройства ничего не изменит. Обычно эти координаты хранятся в буфере вершин, но создать буфер вершин в Vulkan и заполнить его данными нетривиально. Поэтому я

1

решили отложить это до тех пор, пока мы не получим удовольствие от того, что на экране появится треугольник. Тем временем мы собираемся сделать кое-что немного неортодоксальное: включить координаты непосредственно внутрь вершинного шейдера. Код выглядит

```
1 следующим образом: # версия 4 5 0
2
3 позиции vec 2 [ 3 ] = vec 2 [](
4     vec 2 ( 0 . 0 , - 0 . 5 ),
5     vec 2 ( 0 . 5 , 0 . 5 ),
6     vec 2 ( - 0 . 5 , 0 . 5 )
7 );
8
9 аннулировать main() {
10     gl_Position = vec 4 (позиции[gl_VertexIndex], 0 . 0 , 1 . 0 );
11 }
```

Функция `Главная` Вызывается для каждой вершины. Встроенный переменная содержит индекс текущей вершины. Обычно это индекс в буфер вершин, но в нашем случае это будет индекс жестко закодированного массива данных вершин. Доступ к положению каждой вершины осуществляется из постоянного массива в шейдере и объединяется с фиктивным `w` компоненты для получения положения в координатах клипа. Встроенная переменная `gl_Position` функционирует как вывод.

`gl_VertexIndex`

Фрагментный шейдер

Треугольник, образованный позициями вершинного шейдера, заполняет область экрана фрагментами. Для этих фрагментов вызывается шейдер фрагмента для создания цвета и глубины фреймбуфера (или фреймбуферов).

Простой фрагментный шейдер, который выводит красный

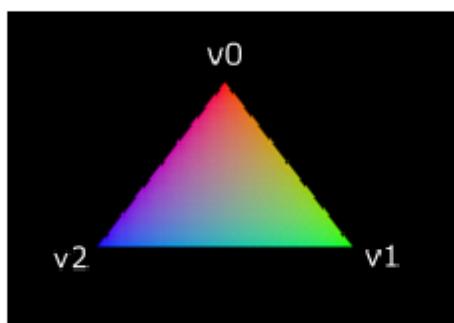
```
1 цвет для всего треугольника, выглядит так: # версия 4 5 0
2
3 макет (location = 0) out vec 4 outColor;
4
5 силы основных() {
6     outColor = vec 4 ( 1 . 0 , 0 . 0 , 0 . 0 , 1 . 0 );
7 }
```

В `Главная` функция вызывается для каждого фрагмента точно так же, как вершинный шейдер `Главная` функция вызывается для каждой вершины. Цвета в GLSL представляют собой 4-компонентные векторы с R, G, B и альфа-каналы в пределах диапазона [0, 1]. В отличие от `gl_Position` в вершинном шейдере нет встроенной переменной для вывода цвета для текущего фрагмента. Вы должны указать свою собственную выходную переменную для каждого кадрового буфера

где `the_макет(location = 0)` модификатор задает индекс фреймбуфера. Для этого записывается красный цвет `outColor` переменная, которая связана с первым (и единственным) фреймбуфером по индексу `0`.

Цвета для каждой вершины

Делать весь треугольник красным не очень интересно, не будет ли что-то вроде следующего выглядеть намного приятнее?



Для этого нам нужно внести пару изменений в оба шейдеров.

Во-первых, нам нужно указать отдельный цвет для каждой из трех вершин.

Вершинный шейдер теперь должен включать массив с цветами, как это

```
1 делается для позиций: цвета vec 3 [ 3 ] = vec 3 [ ] ( 
2     vec 3 ( 1 . 0 , 0 . 0 , 0 . 0 ) ,
3     vec 3 ( 0 . 0 , 1 . 0 , 0 . 0 ) ,
4     vec 3 ( 0 . 0 , 0 . 0 , 1 . 0 )
5 );
```

Теперь нам просто нужно передать эти цвета для каждой вершины в шейдер

фрагментов, чтобы он мог выводить их интерполированные значения в Главная функция:
фреймбуфер. Добавьте вывод для цвета в вершинный шейдер и запишите в

```
1 него в layout(location = 0) out vec 3 FragColor;
2 void main() {
3
4     gl_Position = vec 4 (позиции[gl_VertexIndex], 0 . 0 , 1 . 0 );
5     FragColor = цвета[gl_VertexIndex];
6 }
```

Далее нам нужно добавить соответствующие входные данные в

```
1 шейдер фрагмента: layout(location = 0) in vec 3 FragColor;
```

```
3 void main() {  
4     Внешний цвет = vec 4 (FragColor, 1.0);  
5 }
```

Входная переменная не обязательно должна использовать одно и то же имя, они будут связаны друг с другом с использованием индексов, указанных `расположение` директивы.
Главная функция была изменена для вывода цвета вместе с альфа-значением.
Как показано на изображении выше, значения для `FragColor` будут автоматически интерполированы фрагменты между тремя вершинами, что приведет к плавному градиенту.

Компиляция шейдеров

Создайте каталог под названием `шейдеры` в корневом каталоге вашего проекта и сохраните вершинный шейдер в файле `shader.vert` а фрагментный шейдер в с именем файл с именем `shader.frag` в этом каталоге. Шейдеры GLSL не имеют официального расширения но эти два обычно используются для их различия.

Содержимое `shader.vert` должно быть:

```
1 # версия 4 5 0  
2  
3 макет(location = 0) из vec 3 FragColor;  
4  
5 позиции vec 2 [ 3 ] = vec 2 [](  
6     vec 2 ( 0 . 0 , - 0 . 5 ),  
7     vec 2 ( 0 . 5 , 0 . 5 ),  
8     vec 2 (- 0 . 5 , 0 . 5 )  
9 );  
10  
цвета vec 3 [ 3 ]  
= vec 3 []( 11  
12     vec 3 ( 1 . 0 , 0 . 0 , 0 . 0 ),  
13     vec 3 ( 0 . 0 , 1 . 0 , 0 . 0 ),  
14     vec 3 ( 0 . 0 , 0 . 0 , 1 . 0 )  
15 );  
16  
17 void main() {  
18     gl_Position = vec 4 (позиции[gl_VertexIndex], 0 . 0 , 1 . 0 );  
19     FragColor = цвета[gl_VertexIndex];  
20 }
```

И содержимое `шейдер.fрагмент` должен

```
1 быть: # версия 4 5 0  
2  
3 расположение (location = 0) в vec 3 FragColor;  
4  
5 расположение (location = 0) вне vec 4 outColor;
```

```
6  
7     анулировать main() {  
8         outColor = vec4(FragColor, 1.0);  
9     }
```

Теперь мы собираемся скомпилировать их в байт-код SPIR-V, используя программу `glslc`

Windows

Создание файла `compile.bat` файл со следующим содержимым:

```
1 C:/VulkanSDK/x.x.x.x/Bin 3 2 /glslc.exe шейдер.vert -o верт.spv  
2 C:/VulkanSDK/x.x.x.x/Bin 3 2 /glslc.exe шейдер.фрагмент -o фрагмент.spv  
3 пауза
```

Замените путь к `glslc.exe` путем к месту установки Vulkan

SDK. Дважды щелкните файл, чтобы запустить его.

Linux

Создайте `compile.sh` файл со следующим содержимым:

```
1 /home/пользователь/VulkanSDK/x.x.x.x/x 8 6 _ 6 4 /bin/glslc шейдер.vert -o  
2 vert.spv /home/пользователь/VulkanSDK/x.x.x.x/x 8 6 _ 6 4 /bin/glslc шейдер.фрагмент  
-o frag.spv Замените путь к glslc путем к месту установки Vulkan
```

SDK. Сделайте скрипт исполняемым с помощью `chmod +x compile.sh` и запустите

его. **Конец инструкций для конкретной платформы**

Эти две команды сообщают компилятору прочитать исходный файл GLSL и вывести

файл байт-кода SPIR-V, используя `-o` флаг (вывода).

Если ваш шейдер содержит синтаксическую ошибку, то компилятор сообщит вам номер

строки и проблему, как вы и ожидали. Попробуйте опустить точку с запятой для

примера и снова запустите скрипт компиляции. Также попробуйте запустить компилятор

без каких-либо аргументов, чтобы увидеть, какие типы флагов он поддерживает. Он может,

например, также выводить байт-код в удобочитаемый формат, чтобы вы могли точно

видеть, что делает ваш шейдер, и любые оптимизации, которые были применены на этом

этапе. Компиляция шейдеров в командной строке — одна из самых простых операций,

и именно ее мы будем использовать в этом руководстве, но также возможно

компилировать шейдеры непосредственно из вашего собственного кода. Vulkan SDK

включает `lib-shaderc`, которая представляет собой библиотеку для компиляции GLSL-кода в

SPIR-V из вашей программы.

Загрузка шейдера

Теперь, когда у нас есть способ создания шейдеров SPIR-V, пришло время загрузить их в

нашу программу, чтобы в какой-то момент подключить к графическому конвейеру. Сначала

мы напишем простую вспомогательную функцию для загрузки двоичных данных из файлов.

```

1 # включить <iostream>
2
3 ...
4
5 статический std::vector<символ> ReadFile(const std::string и имя файла) {
6     std::file ifstream(имя файла, std::ios::ate | std::ios::binary);
7
8     if (!file.is_open()) {
9         выбросить std::runtime_error("не удалось открыть файл!");
10    }
11 }

```

В функцию `ReadFile` функция прочитает все байты из указанного файла и вернет их в виде массива байтов, управляемого `std::vector`. Начнем с открытия файла с двумя флагами:

`.ate`: Начните чтение с конца файла (читайте файл как двоичный файл (избегайте текстовых преобразований))

Преимущество начала чтения в конце файла заключается в том, что мы можем использовать позицию чтения для определения размера файла и выделения

```

1 буфера: size_t размер файла = (size_t) файл.tellg();
2 std::vector<символ> буфер(размер файла);

```

После этого, мы можем вернуться к началу файла и прочитать все байты сразу:

```

1 файл.seekg( 0 );
2 file.read(буфер.data(), размер файла);

```

И, наконец, закройте файл и верните байты:

```

1 file.close() файл.закрыть();
2
3 Возврат buffer;

```

Теперь мы вызовем эту функцию из одного из двух шейдеров:

createGraphicsPipeline

для загрузки байт-кода

```

1 void createGraphicsPipeline() {
2     автоматически vertShaderCode = ReadFile("шейдеры/vert.spv");
3     авто fragShaderCode = файл для чтения ("шейдеры /фрагменты.spv");
4 }

```

Убедитесь, что шейдеры загружены правильно, распечатав размер буферов и проверив, соответствуют ли они фактическому размеру файла в байтах. Обратите внимание, что код не обязательно должен завершаться нулем, поскольку это двоичный код, и позже мы уточним его размер.

Создание шейдерных модулей

Прежде чем мы сможем передать код в конвейер, мы должны обернуть его в `VkShaderModule` объект. Давайте создадим вспомогательную функцию `createShaderModule` для этого.

```
1 VkShaderModule createShaderModule(const std::vector<символ>& код) {  
2  
3 }
```

Функция примет буфер с байт-кодом в качестве параметра и создаст `VkShaderModule` из него.

Создать шейдерный модуль просто, нам нужно только указать указатель на буфер с байт-кодом и его длиной.

Эта информация указана в `VkShaderModuleCreateInfo` структура. Единственная загвоздка в том, что размер файла байт-кода указывается в байтах, но указателем байт-кода является `uint32_t` указатель, а не указатель `char`. Следовательно, нам нужно будет привести указатель с помощью `reinterpret_cast` как показано ниже. Когда вы выполняете подобное приведение, вам также необходимо убедиться, что данные удовлетворяют требованиям к выравниванию в `uint32_t`.

К счастью для нас, данные хранятся в `std::vector` где распределитель по умолчанию уже гарантирует, что данные удовлетворяют требованиям к выравниванию в наихудшем случае.

```
1 VkShaderModuleCreateInfo CreateInfo{};  
2 CreateInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;  
3 CreateInfo.CodeSize = code.size();  
4 CreateInfo.pCode = повторная интерпретация<const uint32_t*>(code.data());  
  
Модуль VkShaderModule затем может быть создан с помощью вызова vkCreateShaderModule :  
  
VkShaderModule шейдерМодуль;  
1 если (vkCreateShaderModule(устройство, &CreateInfo, nullptr,  
2  
3 &шнейдерМодуль) != VK_SUCCESS) {  
    выбросить std::runtime_error("не удалось создать модуль шейдера!");  
4 }
```

Параметры те же, что и в предыдущих функциях создания объектов:
логическое устройство, указатель на создание информационной структуры, необязательный указатель на пользовательские распределители и дескриптор выходной переменной. Буфер с кодом может быть освобожден сразу после создания шейдерного модуля. Не забудьте вернуть созданный шейдерный модуль: `возврат шейдерМодуль;`

Шейдерные модули - это просто тонкая оболочка вокруг шейдерного байт-кода, который мы ранее загрузили из файла, и определенных в нем функций. Компиляция и привязка байт-кода SPIR-V к машинному коду для выполнения графическим процессором не происходит до тех пор, пока не будет создан графический конвейер. Это означает, что нам разрешено снова уничтожить шейдерные модули, как только создание конвейера будет завершено.,

именно поэтому мы сделаем их локальными переменными в `createGraphicsPipeline` функция вместо членов класса:

```
1 void createGraphicsPipeline() {
2     auto vertShaderCode = ReadFile("шейдеры/vert.spv");
3     auto fragShaderCode = ReadFile("шейдеры/фрагмент.spv");
4
5     VkShaderModule vertShaderModule =
6         createShaderModule(vertShaderCode);
7     VkShaderModule fragShaderModule =
8         createShaderModule(фрагшадеркод);
```

Затем очистка должна произойти в конце функции путем добавления двух вызовов к `vkDestroyShaderModule`. Весь оставшийся код в этой главе будет вставлен перед этими строками.

```
1 ...
2     vkDestroyShaderModule(устройство, fragShaderModule, nullptr);
3     vkDestroyShaderModule(устройство, vertShaderModule, nullptr);
4 }
```

Создание стадии шейдера

Чтобы действительно использовать шейдеры, нам нужно назначить их определенной стадии конвейера через `VkPipelineShaderStageCreateInfo` структуры как часть фактического процесса создания конвейера.

Мы начнем с заполнения структуры вершинного шейдера, опять же в

создание графической конвейерной линии функция.

```
1 VkPipelineShaderStageCreateInfo vertShaderStageInfo{};
2 vertShaderStageInfo.тип =
3     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
4 vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
```

Первым шагом, помимо обязательного элемента `sType`, является указание Vulkan, на каком этапе конвейера будет использоваться шейдер. Для каждого из программируемых этапов, описанных в предыдущей главе, существует

```
1 значение enum.vertShaderStageInfo.module = vertShaderModule;
2 vertShaderStageInfo.pName = "главный";
```

Ближайшие два члена указать шейдеров модуль, содержащий код, и функции для вызова, известный как *точки входа*. Это означает, что можно объединить несколько фрагментных шейдеров в один шейдерный модуль и использовать разные точки входа, чтобы различать их поведение. Однако в этом случае мы будем придерживаться стандарта

Есть еще один (необязательный) участник, `pSpecializationInfo`, который мы не будем использовать здесь, но его стоит обсудить. Он позволяет вам указывать значения для констант шейдера. Вы можете использовать единый модуль шейдера, где его поведение можно настроить при создании конвейера, указав различные значения для используемых в нем констант. Это более эффективно, чем конфигурирование шейдера с использованием переменных во время рендеринга, потому

если

что компилятор может выполнять оптимизации, такие как исключение операторов, которые зависят от этих значений. Если у вас нет подобных констант, то вы можете задать члену значение, которое наша инициализация структуры выполняет автоматически. Изменить структуру в соответствии с шейдером фрагментов

```
1 несложнo: VkPipelineShaderStageCreateInfo fragShaderStageInfo{};  
2 fragShaderStageInfo.sType =  
3     VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;  
4 fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;  
5 fragShaderStageInfo.module = fragShaderModule;  
6 fragShaderStageInfo.pName = "главный";
```

Завершите определением массива, содержащего эти две структуры, которые мы позже будем использовать для ссылки на них на самом этапе создания

```
1 конвейера.VkPipelineShaderStageCreateInfo шейдерстаги[] =  
2     {vertShaderCreateInfo, fragShaderCreateInfo};
```

Это все, что нужно для описания программируемых этапов конвейера.

В следующей главе мы рассмотрим этапы с фиксированной функцией.

Код на C++ / Вершинный шейдер / Фрагментный шейдер

Исправлены функции

Старые графические API предоставляли состояние по умолчанию для большинства этапов графического конвейера. В Vulkan вы должны четко указывать большинство состояний конвейера поскольку это будет запечено в неизменяемый объект состояния конвейера. В этой главе мы введем все структуры для настройки этих операций с фиксированной функцией.

Динамическое состояние

В то время как *большинство* поскольку состояние конвейера необходимо включить в состояние конвейера, ограниченное количество состояния *фактически может* быть изменено без воссоздания конвейера во время рисования. Примерами могут служить размер области просмотра, ширина линии и константы наложения. Если вы хотите использовать динамическое состояние и не использовать эти свойства, тогда вам придется заполнить это:

структура типа

```
1 std::vector<VkDynamicState> dynamicStates = {  
2     VK_DYNAMIC_STATE_VIEWPORT,  
3     VK_DYNAMIC_STATE_SCISSOR
```

```

4     };
5
6     VkPipelineDynamicStateCreateInfo
7     dynamicState{};  динамичЕское  состояние.sType =
8         VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
9     динамичЕское  состояния.dynamicStateCount =
10        static_cast<uint32_t>(dynamicStates.size());
11     dynamicState.pDynamicStates = dynamicStates.data();

```

Это приведет к игнорированию конфигурации этих значений, и вы сможете (и обязаны будете) указать данные во время рисования. Это приводит к более гибкой настройке и является очень распространенным явлением для таких вещей, как область просмотра и состояние scissor, что привело бы к более сложной настройке при запекании в состояние конвейера.

Ввод вершин

Структура `VkPipelineVertexInputStateCreateInfo` описывает формат данных вершин, которые будут переданы вершинному шейдеру. Он описывает это примерно двумя способами:

- Привязки: интервал между данными и то, являются ли данные для каждой вершины или для каждого экземпляра (см. Создание экземпляра)
- Описания атрибутов: тип атрибутов, передаваемых вершинному шейдеру, из какой привязки их загружать и с каким смещением

Поскольку мы жестко кодируем данные вершин непосредственно в вершинном шейдере, мы заполним эту структуру, чтобы указать, что пока нет данных вершин для загрузки. Мы вернемся к этому в главе о буфере вершин.

```

1     VkPipelineVertexInputStateCreateInfo vertexInputInfo{};
2
3     vertexInputInfo.vertexInputType =
4         VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
5     vertexInputInfo.vertexBindingDescriptionCount = 0;
6     vertexInputInfo.pVertexBindingDescriptions = nullptr; // Необязательно
7     vertexInputInfo.vertexAttributeDescriptionCount = 0;
8     vertexInputInfo.pVertexAttributeDescriptions = nullptr; // Необязательно

```

`BpVertexBindingDescriptions` и `BpVertexAttributeDescriptions` элементы указывают на массив структур, описывающих вышеупомянутые детали для загрузки данных вершин. Добавьте эту структуру в функцию `createGraphicsPipeline` сразу после этапы шейдера **массив**.

Сборка ввода

Структура `VkPipelineInputAssemblyStateCreateInfo` описывает две вещи: какая геометрия будет нарисована из вершин и будет ли перезапущен примитив

должно быть включено. Первое указано в элементе `топологии` и может иметь значения типа:

- `VK_PRIMITIVE_TOPOLOGY_POINT_LIST` : точки из вершин
 - `VK_PRIMITIVE_TOPOLOGY_LINE_LIST` : линия из каждого 2 вершин без повторного использования
 - `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP` : используется конечная вершина каждой строки в качестве начальной вершины для следующей строки `TOPOLOGY_TRIANGLE_LIST` : треугольник из каждого 3 вершин без повторного использования
- * `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`: вторая и третья вершины каждого треугольника используются в качестве первых двух вершин следующего треугольника. Обычно вершины загружаются из буфера вершин по индексу в последовательном порядке, но с буфер элемента вы можете указать индексы для использования самостоятельно. Это позволяет выполнять оптимизацию, например, повторно использовать вершины. Если вы установите `primitiveRestartEnable` подпись на `VK_TRUE`, тогда можно разбивать линии и треугольники в режимах топологии `_STRIP`, используя специальный индекс `0xFFFF` или `0xFFFFFFFF`. Мы намерены рисовать треугольники на протяжении всего этого урока, поэтому будем придерживаться следующих данных для структуры:

```
1 VkPipelineInputAssemblyStateCreateInfo Вводная сборка{};  
2 Вводная сборка.sType =  
3     VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;  
4 Вводная сборка.topология = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;  
5 Вводная сборка.primitiveRestartEnable = VK_FALSE;
```

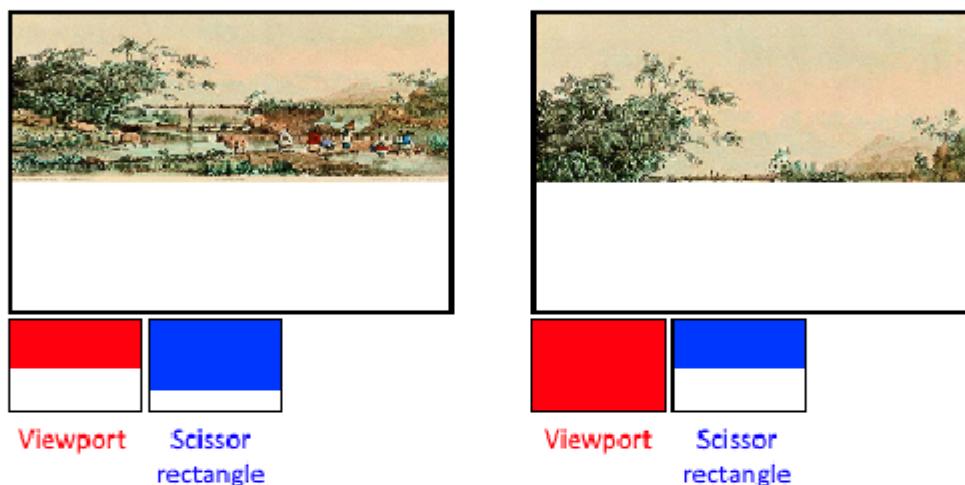
Видовые экраны и ножницы

Видовой экран в основном описывает область фреймбуфера, в которую будет отображаться вывод. Это почти всегда будет (`0, 0`) для (ширина, высота) и в этом руководстве это также будет иметь место.

```
1 Видовой экран VkViewport{};  
2 видовой экран.x = 0.0f;  
3 видовой экран.y = 0.0f;  
4 видовой экран.width = (плавающий) swapChainExtent.ширина;  
5 область просмотра.высота = (плавающий параметр)  
6 swapChainExtent.высота; viewport.minDepth = 0.0f;  
7 viewport.maxDepth = 1.0f;
```

Помните, что размер цепочки подкачки и ее изображений может отличаться от ширина и высота окна. Изображения цепочки подкачки будут использоваться в качестве фреймбуферов позже, поэтому мы должны придерживаться их размера. The `minDepth` и `maxDepth` значения определяют диапазон значений глубины для использования для фреймбуфера. Эти значения должны быть в пределах [`0.0f, 1.0f`] диапазон, но

Глубина разума может превышать максимальную глубину. Если вы не делаете ничего особенного, тогда вам следует придерживаться стандартных значений 0.0f и 1.0f. В то время как видовые экраны определяют преобразование изображения в буфер кадров, прямоугольники-ножницы определяют, в каких областях фактически будут сохранены пиксели. Любые пиксели за пределами прямоугольников-ножниц будут удалены растризатором. Они функционируют как фильтр, а не преобразование. Разница проиллюстрирована ниже. Обратите внимание, что левый прямоугольник-ножницы - это лишь одна из многих возможностей, которая приведет к получению этого изображения, при условии, что оно больше, чем область просмотра.



Итак, если бы мы хотели отрисовать весь фреймбуфер целиком, мы бы указали прямоугольник в форме ножниц, который покрывает его полностью:

```
1 VkRect2D scissor{};
2 scissor.offset = {0, 0};
3 scissor.extent = swapChainExtent;
```

Видовой экран (ы) и прямоугольник (ы)-ножницы могут быть заданы либо как статическая часть конвейера, либо как динамическое состояние, установленное в буфере команд. Хотя первое больше соответствует другим состояниям, часто бывает удобно сделать видовой экран и ножничное состояние динамическими, поскольку это дает вам гораздо больше гибкости. Это очень распространенное явление, и все реализации могут обрабатывать это динамическое состояние без потери производительности. При выборе динамических видовых экранов и прямоугольников-ножниц вам необходимо включить соответствующие динамические

СОСТОЯНИЯ ДЛЯ КОНВЕЙЕРА: std::vector<VkDynamicState> dynamicStates = {

```
1
2     VK_DYNAMIC_STATE_VIEWPORT,
3     VK_DYNAMIC_STATE_SCISSOR
4 };
5
```

```

6 VkPipelineDynamicStateCreateInfo
7 dynamicState{}; динамичЕское состояние.sType =
8     VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
9 динамичЕское состояние.dynamicStateCount =
10    static_cast<uint32_t>(dynamicStates.size());
11 dynamicState.pDynamicStates = dynamicStates.data();

```

И тогда вам нужно только указать их количество во время создания

```

1 конвейера:VkPipelineViewportStateCreateInfo viewportState{};
2 viewportState.тип =
3     VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
4 viewportState.viewportCount = 1;
5 viewportState.scissorCount = 1;

```

Фактические видовые экраны и прямоугольники-ножницы будут настроены позже во время рисования. С помощью dynamic state можно даже указывать разные видовые экраны и / или прямоугольники-ножницы в одном командном буфере. Без динамического состояния видовой экран и прямоугольник-ножницы необходимо установить в конвейере с помощью `VkPipelineViewportStateCreateInfo` структуры. Это делает область просмотра и прямоугольник-ножницы для этого конвейера неизменяемыми. Любые изменения, требуемые для этих значений, потребуют создания нового конвейера с новыми значениями.

```

1 VkPipelineViewportStateCreateInfo viewportState{};
2 Состояние видового экрана.тип =
3     VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
4 viewportState.viewportCount = 1;
5 viewportState.pViewports = &viewport;
6 viewportState.scissorCount = 1;
7 viewportState.pScissors = &scissor;

```

Независимо от того, как вы их настроите, на некоторых видеокартах возможно использовать несколько видовых экранов и прямоугольники-ножницы, таким образом, элементы структуры ссылаются на их массив. Для использования нескольких требуется включить функцию графического процессора (см.

Логическое создание устройства). **Растеризатор**

Растеризатор берет геометрию, сформированную вершинами, из вершинного шейдера и превращает ее во фрагменты, которые будут раскрашены шейдером `fragment`. Он также выполняет тестирование глубины, отбраковку граней и тест "ножницы", и его можно настроить для вывода фрагментов, заполняющих целые полигоны или только края (каркасный рендеринг). `VkPipelineRasterizationStateCreateInfo` структура.

Все это настраивается с помощью

```
1 VkPipelineRasterizationStateCreateInfo  
2 растеризатор{}; растеризатор.sType =  
    VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;  
3 растеризатор.depthClampEnable = VK_FALSE;
```

Если `depthClampable` установлено значение `VK_TRUE`, тогда фрагменты, находящиеся за пределами параметра `VK_TRUE`

ближняя и дальняя плоскости привязываются к ним, а не отбрасываются. Это

полезно в некоторых особых случаях, таких как карты теней. Для этого требуется

включить функцию графического процессора.

```
1 растеризатор.rasterizerDiscardEnable = VK_FALSE;
```

Если для параметра `rasterizerDiscardEnable` установлено значение `VK_TRUE`,

то геометрия никогда не проходит через этап растеризации. Это в основном

отключает любой вывод в буфер кадра.

```
1 растеризатор.polygonMode = VK_POLYGON_MODE_FILL;
```

Определяет способ генерации фрагментов для геометрии. В

многоугольном режиме

доступны следующие режимы:

- `VK_POLYGON_MODE_FILL` : заливка области многоугольника фрагментами
- `VK_POLYGON_MODE_LINE` : края многоугольника рисуются в виде линий

• `VK_POLYGON_MODE_POINT`: многоугольник, вершины отображаются

в виде точек, используя любой режим, кроме заполнения требует

```
1 включения в ГПУ функции. растеризатор.ширина линии = 1.0f;
```

В ширину линии элемент прост, он описывает толщину линий

в терминах количества фрагментов. Максимальная поддерживаемая ширина линии

зависит от оборудования и от толщины любой линии, превышающей `1.0f` требуется

включить функцию графического процессора `wideLines`.

```
1 растеризатор.cullMode = VK_CULL_MODE_BACK_BIT;  
2 растеризатор.FrontFace = VK_FRONT_FACE_CLOCKWISE;
```

Переменная `cullMode` определяет тип используемой выборки граней. Вы можете

отключить отбраковку, отбраковать лицевые грани, отбраковать задние грани или то, и другое. The `FrontFace`

переменная задает порядок вершин для граней, которые должны рассматриваться как обращенные вперед, и

может быть по часовой стрелке или против часовой стрелки.

```
1 растеризатор.depthBiasEnable = VK_FALSE;  
2 растеризатор.depthBiasConstantFactor = 0.0f; // Необязательно  
3 растеризатор.depthBiasClamp = 0.0f; // Необязательно  
4 растеризатор.depthBiasSlopeFactor = 0.0f; // Необязательно
```

Растеризатор может изменять значения глубины, добавляя постоянное значение или смещаю их в зависимости от наклона фрагмента. Иногда это используется для отображения теней, но мы не будем его использовать. Просто установите `depthBiasEnable` для `VK_FALSE`.

Мультисэмплинг

Структура `VkPipelineMultisampleStateCreateInfo` настраивает multisampling, который является одним из способов выполнения сглаживания. Это работает путем объединения результатов шейдера фрагментов нескольких полигонов, которые растируются до одного и того же пикселя. В основном это происходит по краям, где также возникают наиболее заметные артефакты сглаживания. Поскольку нет необходимости запускать фрагментный шейдер несколько раз, если только один полигон отображается в пиксель, это значительно дешевле, чем простой рендеринг в более высоком разрешении с последующим уменьшением масштаба. Для его включения требуется включить

```
1     функцию графического процессора VkPipelineMultisampleStateCreateInfo
2     мультисэмплинг{}; мультисэмплинг.sType =
3         VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
4     множественная выборка.sampleShadingEnable = VK_FALSE;
5     множественная выборка.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
6     множественная выборка.minSampleShading = 1.0f; // Необязательно
7     мультисэмплинг.pSampleMask = nullptr; // Необязательно
8     множественная выборка.alphaToCoverageEnable = VK_FALSE; // Необязательно
9     множественная выборка.alphaToOneEnable = VK_FALSE; // Необязательно
```

Мы вернемся к мультисэмплингу в следующей главе, а пока давайте оставим его отключенным.

Тестирование глубины и трафарета

Если вы используете буфер глубины и / или трафарета, то вам также необходимо

настроить тесты глубины и трафарета с использованием `VkPipelineDepthStencilCreateInfo`

У нас сейчас его нет, поэтому мы можем просто передать `nullptr` вместо указателя на такую структуру. Мы вернемся к этому в главе о буферизации глубины. **Смешивание цветов**

После того, как фрагментный шейдер вернул цвет, его необходимо объединить с цветом, который уже есть в фреймбуфере. Это преобразование известно как смешивание цветов, и есть два способа сделать это:

- Смешайте старое и новое значение для получения окончательного цвета
- Объедините старое и новое значение с помощью побитовой операции

Существует два типа структур для настройки смешивания цветов. Первая структура, `VkPipelineColorBlendAttachmentState` содержит конфигурацию для каждого attached framebuffer и вторая структура, `VkPipelineColorBlendStateCreateInfo` содержит глобальные настройки смешивания цветов. В нашем случае у нас есть только один кадровый буфер:

```
1     VkPipelineColorBlendAttachmentState colorBlendAttachment{};
2     colorBlendAttachment.colorWriteMask = VK_COLOR_COMPONENT_R_BIT |
3         VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT |
4         VK_COLOR_COMPONENT_A_BIT;
```

```

3 colorBlendAttachment смешиваемый = VK_FALSE;
4 Привязка ColorBlend.srcColorBlendFactor = VK_BLEND_FACTOR_ONE; // Необязательно
Привязка ColorBlend.dstColorBlendFactor = VK_BLEND_FACTOR_ZERO; // Необязательно
5 Подключение ColorBlend.colorBlendOp = VK_BLEND_OP_ADD; // Необязательно
6
colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE; // 7
7
Подключение colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO; // Необязательно
8
8
9

```

Эта структура кадрового буфера позволяет настроить первый способ наложения цветов. Операции, которые будут выполняться, лучше всего продемонстрировать с помощью

```

1 следующего псевдокода: if (Можно смешивать) {
2     finalColor.rgb = (srcColorBlendFactor * newColor.rgb)
3         <colorBlendOp> (dstColorBlendFactor * oldColor.rgb);
4     finalColor.a = (srcAlphaBlendFactor * newColor.a) <alphaBlendOp>
5         (dstAlphaBlendFactor
6     * oldColor.a); } еще {
7     finalColor = newColor;
8 }
8 finalColor = finalColor & colorWriteMask;

```

Если для параметра `blendEnable` установлено значение `VK_FALSE`, затем новый цвет из фрагмента шейдера пропускается без изменений. В противном случае выполняются две операции смешивания для вычисления нового цвета. Результирующий цвет отображается с помощью `colorWriteMask` для определения того, по каким каналам фактически осуществляется передача. Наиболее распространенный способ использования смешивания цветов - реализовать альфа-смешивание, где мы хотим, чтобы новый цвет смешивался со старым цветом на основе его непрозрачности. The `finalColor` затем следует вычислить следующим образом:

```

1 finalColor.rgb = newAlpha * newColor + (1 - newAlpha) * oldColor;
2 finalColor.a = newAlpha.a;

```

Это может быть достигнуто с помощью следующих параметров:

```

colorBlendAttachment.blendEnable = VK_TRUE;
1 colorBlendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
2 colorBlendAttachment.dstColorBlendFactor =
    VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
4 Подключение ColorBlend.colorBlendOp = VK_BLEND_OP_ADD;
5 colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
6 colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;

```

```
7 Подключение ColorBlend.alphaBlendOp = VK_BLEND_OP_ADD;
```

Вы можете найти все возможные
операции в перечислениях в спецификации.

VkBlendFactor и VkBlendOp

Вторая структура ссылается на массив структур для всех фреймбуферов
и позволяет вам устанавливать константы наложения, которые вы можете
использовать в качестве коэффициентов наложения в вышеупомянутых

```
1 Вычисления.VkPipelineColorBlendStateCreateInfo смешивание
2 цветов{}; Смешивание цветов.Тип =
3     VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
4 Смешивание цветов.logicOpEnable = VK_FALSE;
5 Смешивание цветов.logicOp = VK_LOGIC_OP_COPY; // Необязательно
6 Смешивание цветов.attachmentCount = 1;
7 Смешивание цветов.Параметры = &colorBlendAttachment;
8 Смешивание цветов.blendConstants[ 0 ] = 0.0f; // Необязательно
9 Смешивание цветов.blendConstants[ 1 ] = 0.0f; // Необязательно
10 Смешивание цветов.Константы смешивания [ 3 ] = 0.0f; // Необязательно
```

Если вы хотите использовать второй метод наложения (побитовая комбинация), то
вам следует установить `vkLogicOp`. Затем побитовая операція может быть
указана в `logicOp` поле. Обратите внимание, что это автоматически отключит
первый метод, как если бы вы установили смешиваемый для `VK_FALSE` для каждого
подключенного фреймбуфера! В этом режиме также будет использоваться функция `colorWriteMask` для
определения, какие каналы в фреймбуфере действительно будут затронуты. Также можно
отключить оба режима, как мы сделали здесь, в этом случае цвета фрагмента будут
записаны в буфер кадров без изменений.

Схема конвейера

Вы можете использовать Униформа значения в шейдерах, которые являются глобальными величинами, аналогичными динамическим
переменные состояния, которые можно изменять во время рисования, чтобы
изменить поведение ваших шейдеров без необходимости их воссоздания. Они
обычно используются для передачи матрицы преобразования в вершинный шейдер
или для создания текстурных сэмплеров в фрагментном шейдере. Эти
единобразные значения необходимо указать при создании конвейера путем создания
объекта `VkPipelineLayout`. Несмотря на то, что мы не будем использовать их до
следующей главы, нам все равно необходимо создать пустой макет конвейера.
Создайте член класса для хранения этого объекта, потому что позже мы будем
ссылаться на него из других функций:

```
1 VkPipelineLayout Конвейерное описание;
```

А затем создайте объект в

createGraphicsPipeline

функции:

```

VkPipelineLayoutCreateInfo pipelineLayoutInfo{};
pipelineLayoutInfo.sType =
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutInfo.setLayoutCount = 0; // Необязательно
3

4 pipelineLayoutInfo.pSetLayouts = nullptr; // Необязательно
5 pipelineLayoutInfo.pushConstantRangeCount = 0; // Необязательно
6 pipelineLayoutInfo.pPushConstantRanges = nullptr; // Необязательно
7
8 если (vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr,
    &pipelineLayout) != VK_SUCCESS) {
9     бросок std::runtime_error("не удалось создать макет конвейера!");
10 }

```

Структура также определяет *константы push*, которые являются еще одним способом передачи динамических значений шейдерам, о котором мы, возможно, поговорим в следующей главе. На схему конвейера будут ссылаться на протяжении всего срока службы

```

1 программы, поэтому в конце ее следует уничтожить: void cleanup() {
2     vkDestroyPipelineLayout(устройство, конвейерное
3     описание, nullptr); ...
4 }

```

Вывод

Вот и все для состояния с фиксированной функцией! Требуется много работы, чтобы настроить все это с нуля, но преимущество в том, что теперь мы почти полностью осведомлены обо всем, что происходит в графическом конвейере! Это снижает вероятность столкнуться с неожиданным поведением, поскольку состояние определенных компонентов по умолчанию - не то, что вы ожидаете. Однако, прежде чем мы сможем окончательно создать графический конвейер, необходимо создать еще один объект, и это этап рендеринга.

Код на C++ / Вершинный шейдер / Фрагментный шейдер

Этапы рендеринга

Настройка

Прежде чем мы сможем завершить создание конвейера, нам нужно сообщить Vulkan о вложениях фреймбуфера, которые будут использоваться при рендеринге. Нам нужно указать, сколько будет буферов цвета и глубины, сколько выборок использовать для каждого из них и как следует обрабатывать их содержимое на протяжении операций рендеринга. Вся эта информация заключена в объект *render pass*, для которого мы создадим новый `createRenderPass` функция. Вызовите эту функцию из `initVulkan` перед тем, как Создать графический трубопровод.

```

1 анулирование Инициализированный вулкан() {
2     CreateInstance(); setupDebugMessenger();
3     createSurface();
4     pickPhysicalDevice();
5     createLogicalDevice();
6     CreateSwapChain();
7     createImageViews();
8     createRenderPass();
9     Создайте графический конвейер();
10 }
11 ...
12 ...
13 ...
14 ...
15 void createRenderPass() {
16 }
17 }
```

Описание вложения

В нашем случае у нас будет только одно вложение с цветовым буфером, представленное одним из изображений из цепочки обмена.

```

1 void createRenderPass() {
2     VkAttachmentDescription colorAttachment{};
3     colorAttachment.format = swapChainImageFormat;
4     colorAttachment.образцы = VK_SAMPLE_COUNT_1_BIT;
5 }
```

В `format` цвет крепления должна соответствовать формату цепочки обмена изображения, и мы пока ничего не делаем с мультиспллингом, поэтому будем придерживаться 1 сэмпла.

```

1 colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
2 colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
```

The `loadOp` и `storeOp` определите, что делать с данными во вложении до рендеринга и после рендеринга. У нас есть следующие варианты для

`loadOp` :

- `VK_ATTACHMENT_LOAD_OP_LOAD`: Сохранить существующее содержимое attachment

- `VK_ATTACHMENT_LOAD_OP_CLEAR`: Очистите значения до константы в начале.

- `VK_ATTACHMENT_LOAD_OP_DONT_CARE`: Существующее содержимое не определено; нас это не волнует

В нашем случае мы собираемся использовать операцию очистки, чтобы очистить фреймбуфер до черного цвета перед рисованием нового фрейма. Есть только две возможности для

storeOp:

- **VK_ATTACHMENT_STORE_OP_STORE**: Отображаемое содержимое будет сохранено в память и может быть прочитана позже
- Содержимое фреймбуфера будет не определено после операции рендеринга

Нам интересно видеть отрисованный треугольник на экране, поэтому мы переходим

к операции сохранения здесь.

```
1 colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
2 colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
```

Параметры `loadOp` и `storeOp` применяются к данным о цвете и глубине, а также `stencilLoadOp` / `stencilStoreOp` применить к данным трафарета. Наше приложение ничего не будет делать буфером трафарета, поэтому результаты загрузки и сохранения не имеют значения.

```
1 colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
2 colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

Текстуры и фреймбуфера в Vulkan представлены с помощью `VkImage` объекты с определенным форматом пикселей, однако расположение пикселей в памяти может изменяться в зависимости от того, что вы пытаетесь сделать с изображением.

Вот некоторые из наиболее распространенных макетов:

- **VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL**: Изображения, используемые в качестве цветных-

настройка

- **VK_IMAGE_LAYOUT_PRESENT_SRC_KHR**: Изображения, которые будут представлены в swap

цепочка

- **VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL**: Изображения, которые будут использоваться в

качестве

Дестина- ции на память копию работы мы обсудим эту тему подробнее в главе текстурирования, но что важно знать, так это то, что изображения должны быть переведены в конкретные макеты

обновлением-

Начальное описание

начинается переход der. В

Окончательное описание

указывает макет для автоматического перехода-

зависит от того, когда завершится этап рендеринга. Использование

`VK_IMAGE_LAYOUT_UNDEFINED`

для

`initialLayout`

означает, что нам все равно, каким был предыдущий макет изображения

в. Предостережение от этого особого значения заключается в том, что не гарантируется сохранение

содержимого изображения, но это не имеет значения, поскольку мы все равно собираемся его

очистить . Мы хотим, чтобы изображение было готово к представлению с использованием цепочки

подкачки после рендеринга, поэтому мы используем `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` как

Окончательное описание.

Подпасы и ссылки на вложения

Один проход рендеринга может состоять из нескольких подпасов. Подпасы - это последующие операции рендеринга, которые зависят от содержимого фреймбуферов в предыдущих проходах, например, от последовательности эффектов постобработки, которые применяются один за другим. Если вы сгруппируете эти операции рендеринга в один проход рендеринга, то Vulkan сможет изменить порядок операций и сохранить пропускную способность памяти для возможного повышения производительности. Однако для нашего самого первого треугольника мы будем придерживаться одного подпуска. Каждый подпункт ссылается на одно или несколько вложений, которые мы описали, используя структуру в предыдущих разделах. Эти ссылки сами по себе

VkAttachmentReference структуры, которые выглядят следующим образом:

```
1 VkAttachmentReference colorAttachmentRef{};

2 colorAttachmentRef.attachment = 0;

3 colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

В `вложение` параметр указывает, какое приложение чтобы ссылка на ее индекс во вложении описания массива. Наш массив состоит из одного `VkAttachmentDescription`, поэтому его индекс равен `0`. В `макет` указывает, какой макет мы хотели бы иметь для вложения во время подпуска, который использует эту ссылку. Vulkan автоматически переведет вложение в этот макет при запуске подпуска. Мы намерены использовать вложение в качестве цветового буфера и `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` макет обеспечит нам наилучшую производительность, как следует из его названия.

Подпунктканал описывается с помощью Описания `VkSubpass` структура:

```
1 VkSubpassDescription subpass{};

2 подпункт.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
```

В будущем Vulkan также может поддерживать вычислительные подпасы, поэтому мы должны четко указать, что это подпас графического типа. Далее мы указываем ссылку на цветное вложение:

```
1 подпункт.colorAttachmentCount = 1;

2 подпункт.pColorAttachments = &colorAttachmentRef;
```

На индекс вложения в этом массиве есть прямая ссылка из фрагмента шейдера с помощью `layout(location = 0) out vec 4 outColor` директивы! Подпасок может ссылаться на следующие другие типы вложений.:

- `pInputAttachments`: Вложения, считываемые из шейдера
- Предварительные разрешения: Вложения, используемые для прикрепления цветов с несколькими выборками-улучшения
- Привязка `pDepthStencil`: Вложение для данных глубины и трафарета
- `pPreserveAttachments`: Вложения, которые не используются этим подпунктом, но для которых данные должны быть сохранены

Прохождение рендеринга

Теперь, когда вложение и базовый вспомогательный канал, ссылающийся на него, описаны, мы можем создать сам канал визуализации. Создайте новую переменную-член класса для хранения `VkRenderPass` объект прямо над `pipelineLayout` переменная:

```
1 VkRenderPass renderPass;
2 VkPipelineLayout Конвейерное описание;
```

Затем объект `render pass` может быть создан путем заполнения `VkRenderPassCreateInfo` структура с массивом вложений и подпусков. Ссылка `VkAttachmentReference` объекты ссылаются на вложения, используя индексы этого массива.

```
1 VkRenderPassCreateInfo renderPassInfo{};
2 renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
3 renderPassInfo.attachmentCount = 1;
4 renderPassInfo.pAttachments = &colorAttachment;
5 renderPassInfo.subpassCount = 1;
6 renderPassInfo.pSubpasses = &подпасок;
7
8 если (vkCreateRenderPass(устройство, & renderPassInfo, nullptr,
9     &renderPass) != VK_SUCCESS) {
10     выбросить std::runtime_error("не удалось создать проход рендеринга!");
11 }
```

Так же, как и схема конвейера, на проход рендеринга будут ссылаться по всей программе, поэтому его следует очистить только в конце:

```
1 void cleanup() {
2     vkDestroyPipelineLayout(устройство, конвейерное
3     описание, nullptr); vkDestroyRenderPass(устройство,
4     renderPass, nullptr); ...
5 }
```

Это была большая работа, но в следующей главе все сведется воедино, чтобы, наконец, создать объект графического конвейера!

Код на C++ / Вершинный шейдер / Фрагментный шейдер

Заключение

Теперь мы можем объединить все структуры и объекты из предыдущих глав для создания графического конвейера! Вот типы объектов, которые у нас есть сейчас, в качестве краткого обзора:

- Этапы шейдера: модули шейдера, которые определяют функциональность программируемые этапы графического конвейера
- Состояние с фиксированной функцией: все структуры, определяющие этапы конвейера с фиксированной функцией, такие как сборка ввода, растрлизатор, область просмотра и смешивание

- Схема конвейера: значения uniform и push, на которые ссылается шейдер которые могут быть обновлены во время рисования
- Этап рендеринга: вложения, на которые ссылаются этапы конвейера, и их использование - все это вместе взятое полностью определяет функциональность графического конвейера, поэтому теперь мы можем приступить к заполнению `VkGraphicsPipelineCreateInfo` структура в конце файла `createGraphicsPipeline` функция. Но перед вызовами `vkDestroyShaderModule` потому что они все еще будут использоваться при

```
1 pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
2 pipelineInfo.stageCount = 2;
4 pipelineInfo.pStages = этапы шейдера;
```

Мы начнем со ссылки на массив `VkPipelineShaderStageCreateInfo` структуры.

```
1 pipelineInfo.pVertexInputState = &vertexInputInfo;
2 pipelineInfo.pInputAssemblyState = &inputAssembly;
3 pipelineInfo.pViewportState = &viewportState;
4 pipelineInfo.pRasterizationState = &rasterizer;
5 pipelineInfo.pMultisampleState = &multisampling;
6 pipelineInfo.pDepthStencilState = nullptr; // Необязательно
7 pipelineInfo.pColorBlendState = &Смешивание цветов;
8 pipelineInfo.pDynamicState = &dynamicState;
```

Затем мы ссылаемся на все структуры, описывающие этап с фиксированной

```
1 функцией. pipelineInfo.layout = pipelineLayout;
```

После этого появляется макет конвейера, который представляет собой дескриптор Vulkan, а не указатель структуры.

```
1 pipelineInfo.renderPass = renderPass;
2 pipelineInfo.subpass = 0;
```

И, наконец, у нас есть ссылка на проход рендеринга и индекс вспомогательного прохода, где будет использоваться этот графический конвейер. Также возможно использовать другой рендеринг

проходит с этим конвейером, а не с этим конкретным экземпляром, но они должны быть совместимы с `renderPass`. Требования к

совместимости описаны здесь, но мы не будем использовать эту функцию в этом руководстве.

```
1 pipelineInfo.basePipelineHandle = VK_NULL_HANDLE; // Необязательно
2 pipelineInfo.basePipelineIndex = -1; // Необязательно
```

На самом деле есть еще два параметра:

`basePipelineHandle` и

`basePipelineIndex`. Vulkan позволяет создавать новый графический конвейер, производный от существующего конвейера. Идея производных конвейера заключается в том, что

настройка конвейеров обходится дешевле, когда они имеют много функций, общих с существующим конвейером, и переключение между конвейерами из одного и того же родительского устройства также может быть выполнено быстрее. Вы можете либо указать дескриптор `basePipelineHandle` существующего конвейера с помощью, либо сослаться на другой конвейер, который будет создан с помощью `index` с помощью `basePipelineIndex`. Прямо сейчас существует только один конвейер, поэтому мы просто укажем нулевой дескриптор и недопустимый индекс. Эти значения используются только в том случае, если `VK_PIPELINE_CREATE_DERIVATIVE_BIT` флаг также указан в поле `flags` в приложении `VkGraphicsPipelineCreateInfo`.

Теперь подготовьтесь к последнему шагу, создав член класса для хранения

```
1 VkPipeline объект:
```

```
VkPipeline graphicsPipeline;
```

И, наконец, создайте графический конвейер:

```
1 если (vkCreateGraphicsPipelines(устройство, VK_NULL_HANDLE, 1,
2     &pipelineInfo, nullptr, &graphicsPipeline) != VK_SUCCESS) {
3     выбросить std::runtime_error ("не удалось создать графический конвейер!");
4 }
```

Функция `vkCreateGraphicsPipelines`

обычные функции создания объектов в Vulkan. Она рассчитана на несколько

```
VkGraphicsPipelineCreateInfo
```

На самом деле имеет больше параметров, чем

VkPipeline

об-

запускается в одном вызове.

Второй параметр, для которого мы передали `VK_NULL_HANDLE` аргумент, ссылается на необязательный объект `VkPipelineCache`. Кэш конвейера может использоваться для хранения и повторного использования данных, относящихся к созданию конвейера, в нескольких вызовах `vkCreateGraphicsPipelines` и даже во время выполнения программы, если кэш хранится в файле. Это позволяет значительно ускорить создание конвейера позже. Мы рассмотрим это в главе о конвейерном кэше. Графический конвейер необходим для всех обычных операций рисования, поэтому его также следует уничтожать только в конце программы:

```
1 анулировать очистка() {
2     vkDestroyPipeline(устройство, графический конвейер,
3         nullptr); vkDestroyPipelineLayout(устройство,
4         конвейерное описание, nullptr); ...
5 }
```

Теперь запустите вашу программу, чтобы убедиться, что вся эта кропотливая работа привела к успешному созданию конвейера! Мы уже довольно близки к тому, чтобы увидеть, как что-то всплывает на экране. В следующих двух главах мы настроим сами фреймбуфера из изображений цепочки подкачки и подготовим команды рисования. Код на C++ / Вершинный шейдер / Фрагментный шейдер

Рисование

Фреймбуфера

Мы много говорили о фреймбуферах в последних нескольких главах, и мы настроили этап рендеринга так, чтобы ожидался единый фреймбуфер того же формата, что и изображения цепочки подкачки, но на самом деле мы еще не создали ни одного. Вложения, указанные при создании render pass, привязываются путем переноса их в объект `VkFramebuffer`. Объект `framebuffer` ссылается на все `VkImageView` объекты, представляющие вложения. В нашем случае это будет только одно: цветное вложение. Однако изображение, которое мы должны использовать для вложения, зависит от того, какое изображение возвращает цепочка обмена, когда мы извлекаем его для представления. Это означает, что мы должны создать фреймбуфер для всех изображений в цепочке подкачки и использовать тот, который соответствует полученному изображению во время рисования. С этой целью создайте другой `std::vector` член класса для хранения фреймбуферов: `std::vector<VkFramebuffer> framebuffers swapchain;`

1

Мы создадим объекты для этого массива в новой функции `createFramebuffers` вызывается из `initVulkan` сразу после создания графического конвейера:

```
1 void initVulkan() {
2     CreateInstance(); setupDebugMessenger();
3     createSurface();
4     pickPhysicalDevice();
5     createLogicalDevice();
6     CreateSwapChain();
7     createImageViews();
8     createRenderPass();
9     createGraphicsPipeline();
10    createFramebuffers();
11 }
12 ...
13 ...
14 ...
15 ...
16 пустота createFramebuffers() {
17 }
18 }
```

Начните с изменения размера контейнера, чтобы вместить все

```
1 фреймбуферы: аннулирование createFramebuffers() {
2     swapChainFramebuffers.resize(swapChainImageViews.size());
3 }
```

Затем мы пройдемся по видам изображений и создадим из них фреймбуферы:

```
1  для (size_t i = 0; i < swapChainImageViews.size(); i++) {
2      Вложения VkImageView[] = {
3          swapChainImageViews[i]
4      };
5
6      VkFramebufferCreateInfo framebufferInfo{};
7      framebufferInfo.sType =
8          VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
9      framebufferInfo.renderPass = renderPass;
10     framebufferInfo.attachmentCount = 1;
11     framebufferInfo.pAttachments = вложения;
12     framebufferInfo.width = swapChainExtent.ширина;
13     framebufferInfo.height = swapChainExtent.высота;
14     framebufferInfo.layers = 1;
15
16     если (vkCreateFramebuffer(устройство, &framebufferInfo, nullptr,
17         &swapChainFramebuffers[i]) != VK_SUCCESS) {
18         выбросить std::runtime_error ("не удалось создать фреймбуфер!");
19     }
20 }
```

Как вы можете видеть, создание фреймбуферов довольно просто. Сначала нам нужно указать, с каким именно `renderPass` фреймбуфер должен быть совместим. Вы можете использовать фреймбуфер только с теми проходами рендеринга, с которыми он совместим, что примерно означает, что они используют одинаковое количество и тип вложений.

The `attachmentCount` и `pAttachments` параметры определяют `VkImageView` объекты, которые должны быть привязаны к соответствующим описаниям вложений на этапе рендеринга. Привязка — массив.

Параметры `width` и `height` не требуют пояснений и `layers` относятся к количеству слоев в массивах изображений. Наши изображения цепочки подкачки представляют собой отдельные изображения, поэтому количество слоев равно 1. Мы должны удалить фреймбуферы перед просмотром изображения и передачей рендеринга, на которых они основаны, но только после того, как мы закончим рендеринг:

```
1 void cleanup() {
2     для (автоматического фреймбуфер : swapChainFramebuffers) {
3         vkDestroyFramebuffer(устройство, фреймбуфер, nullptr);
4     }
5
6     ...
7 }
```

Теперь мы достигли того рубежа, когда у нас есть все объекты, которые

требуется для рендеринга. В следующей главе мы собираемся написать первые настоящие команды рисования.

Код на C++ / Вершинный шейдер / Фрагментный шейдер

Буферы команд

Команды в Vulkan, такие как операции рисования и передачи памяти, не выполняются напрямую с использованием вызовов функций. Вы должны записать все операции, которые вы хотите выполнить, в объекты командного буфера. Преимущество этого в том, что когда мы готовы сообщить Vulkan, что мы хотим сделать, все команды отправляются вместе, и Vulkan может более эффективно обрабатывать команды, поскольку все они доступны вместе. Кроме того, это позволяет при желании записывать команды в несколько потоков.

Пулы команд

Мы должны создать пул команд, прежде чем сможем создавать командные буферы. Пулы команд управляют памятью, используемой для хранения буферов, и из них выделяются командные буферы. Добавьте нового члена класса для

хранения `VkCommandPool` :

```
1 VkCommandPool commandPool;
```

Затем создайте новую функцию `createCommandPool` и вызовите ее из `initVulkan` после создания фреймбуферов.

```
void initVulkan() {
    1
    2     CreateInstance();    setupDebugMessenger();
    3     createSurface();
    4     pickPhysicalDevice();
    5     createLogicalDevice();
    6     CreateSwapChain();
    7     createImageViews();
    8     createRenderPass();
    9     createGraphicsPipeline();
   10    createFramebuffers();
   11    createCommandPool();
   12
   13}
   14
   15...
   16
   17    пустота createCommandPool() {
   18
   19}
```

Для создания пула команд требуется только два параметра:

```
1 QueueFamilyIndices Справочники queueFamilyIndices =
    findQueueFamilies(физическое устройство);
2
3 VkCommandPoolCreateInfo poolInfo{};
4 poolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
5 poolInfo.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;
6 poolInfo.queueFamilyIndex =
    queueFamilyIndices.graphicsFamily.value();
```

Для пулов команд возможны два флага:

• `VK_COMMAND_POOL_CREATE_TRANSIENT_BIT`: Подсказка о том, что буферы команд очень часто перезаписываются новыми командами (это может изменить характер распределения памяти).
• `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT`: Разрешить повторную запись командных буферов по отдельности, без этого флага все они должны быть сброшены вместе

Мы будем записывать команду в буфер каждого кадра, поэтому мы хотим иметь возможность повторно-устанавливать и перезаписывать ее. Таким образом, нам нужно установить `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT` бит флага для нашего пула команд.
Буферы команд выполняются путем отправки их в одну из очередей устройств, подобно полученным нами графическим и презентационным очередям. Каждый пул команд может выделять только буферы команд, которые отправляются в очередь одного типа.
Мы собираемся записывать команды для рисования, поэтому мы выбрали семейство графических очередей.

```
1 если (vkCreateCommandPool(устройство, &poolInfo, nullptr, & commandPool) != VK_SUCCESS) {
2     бросить std::runtime_error("не удалось создать пул команд!");
3 }
```

Завершите создание пула команд с помощью функции `vkCreateCommandPool`. У него нет никаких специальных параметров. Команды будут использоваться на протяжении всей программы для рисования объектов на экране, поэтому пул следует уничтожить только в конце:

```
1 void cleanup() {
2     vkDestroyCommandPool (устройство, commandPool, nullptr);
3
4     ...
5 }
```

Выделение командного буфера

Теперь мы можем приступить к распределению командных буферов.

Создайте объект `VkCommandBuffer` в качестве члена класса. Буферы команд будут автоматически освобождены, когда их пул команд будет уничтожен, поэтому нам не нужна явная очистка.

```
1 Командный буфер VkCommandBuffer;

Теперь мы начнем работать над а createCommandBuffer функция для
выделения отдельного командного буфера из пула команд.

void initVulkan() {
1
2     CreateInstance(); setupDebugMessenger();
3     createSurface();
4     pickPhysicalDevice();
5     createLogicalDevice();
6     CreateSwapChain();
7     createImageViews();
8     createRenderPass();
9     createGraphicsPipeline();
10    createFramebuffers();
11    createCommandPool();
12    createCommandBuffer();
13}
14}
15
16...
17
18 void createCommandBuffer() {
19}
20}
```

Буферы команд выделяются с помощью функции `vkAllocateCommandBuffers`, которая принимает `VkCommandBufferAllocateInfo` struct как параметр, который определяет пул команд и количество буферов для выделения:

```
1 VkCommandBufferAllocateInfo allocInfo{};
2 allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
3 allocInfo.commandPool = commandPool;
4 allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
5 allocInfo.commandBufferCount = 1;
6
7 если (vkAllocateCommandBuffers(устройство, & allocInfo, & commandBuffer) !=
8     VK_SUCCESS) {
9     выбросить std::runtime_error("не удалось выделить командные буферы!");
10 }
```

В `уровень` параметр указывает, являются ли выделенные командные буферы первичными или вторичными командными буферами.

- `VK_COMMAND_BUFFER_LEVEL_PRIMARY`: Может быть отправлен в очередь для выполнения, но не может быть вызван из других
- `VK_COMMAND_BUFFER_LEVEL_SECONDARY`: Не может быть отправлено напрямую, но может быть вызван из основных командных буферов.

Мы не будем использовать здесь функциональность вторичного командного буфера, но вы можете представить, что полезно повторно использовать обычные операции из первичного командного буфера.

Поскольку мы выделяем только один буфер команд, параметр `commandBufferCount` всего один.

Запись в буфер команд

Теперь мы начнем работать над `recordCommandBuffer` функция, которая записывает команды, которые мы хотим выполнить, в буфер команд. The `VkCommandBuffer` используемый будет передан в качестве параметра, а также индекса текущего образа `swapchain`, в который мы хотим записать.

```
void recordCommandBuffer(Командный буфер VkCommandBuffer, uint 3 2 _t 1
                         ImageIndex) {
    ...
}
```

Мы всегда начинаем запись командного буфера с вызова `vkBeginCommandBuffer` с небольшого значения `VkCommandBufferBeginInfo` структура в качестве аргумента, который определяет некоторые подробности об использовании этого

```
1 конкретного командного буфера. VkCommandBufferBeginInfo beginInfo{};
2 beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
3 beginInfo.flags = 0; // Необязательно
4 beginInfo.pInheritanceInfo = nullptr; // Необязательно
5 если (vkBeginCommandBuffer(командный буфер и beginInfo) != VK_SUCCESS) {
6     выбросить std::runtime_error ("не удалось начать запись команды
                                     буфер!");
7 }
8 }
```

Параметр `flags` указывает, как мы собираемся использовать командный буфер. Доступны следующие значения:

- `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT`: буфер будет перезаписана, сразу после выполнения один раз.

в команда

- `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT` - команда для буфера_использования: Это второе.-

команда

обычный буфер команд, который будет полностью заполнен за один проход рендеринга.

`VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` буфер может быть отправлен

повторно, пока он также ожидает выполнения. Ни один из этих флагов не применим для нас прямо сейчас.

The `pInheritanceInfo` параметр имеет значение только для буферов вторичной команды . Он указывает, какое состояние должно наследоваться от буферов вызывающей первичной команды .

Если буфер команд уже был записан один раз, то вызов `vkBeginCommandBuffer` неявно сбросит его. Невозможно добавить команды в буфер позже.

Запуск этапа рендеринга.

Рисование начинается с начала этапа рендеринга с помощью `vkCmdBeginRenderPass`. Этап рендеринга настраивается с использованием некоторых параметров в `VkRenderPassBeginInfo` структура.

```
1 VkRenderPassBeginInfo renderPassInfo{};  
2 renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;  
3 renderPassInfo.renderPass = renderPass;  
4 renderPassInfo.framebuffer = swapChainFramebuffers[ImageIndex];
```

Первыми параметрами являются сам проход рендеринга и вложения для привязки. Мы создали фреймбуфер для каждого изображения цепочки подкачки, где оно указано как цветное вложение. Таким образом, нам нужно привязать фреймбуфер к изображению `swapchain`, к которому мы хотим отрисовать. Используя параметр `ImageIndex`, который был передан, мы можем выбрать правильный фреймбуфер для текущего изображения `swapchain`.

```
1 renderPassInfo.renderArea.offset = { 0, 0 };  
2 renderPassInfo.renderArea.extent = swapChainExtent;
```

Следующие два параметра определяют размер области рендеринга. Область рендеринга определяет, где будут выполняться загрузка и сохранение шейдеров. Пиксели за пределами этой области будут иметь неопределенные значения. Он должен соответствовать размеру вложений для достижения наилучшей производительности.

```
1 vkClearValue clearColor = {{ { 0.0f, 0.0f, 0.0f, 1.0f } }};  
2 renderPassInfo.clearValueCount = 1;  
3 renderPassInfo.pClearValues = &clearColor;
```

Последние два параметра определяют четкие значения для использования которые мы использовали в качестве операции загрузки для цветного вложения. Я определил `VK_ATTACHMENT_LOAD_OP_CLEAR`, прозрачный цвет как просто черный со 100% непрозрачностью.

```
vkCmdBeginRenderPass(commandBuffer, & renderPassInfo,  
1  
VK_SUBPASS_CONTENTS_INLINE);
```

Теперь можно начинать этап визуализации. Все функции, которые записывают команды, могут быть распознаны по их префиксу `vkCmd` . Все они возвращают результат `void`, поэтому не будет обработки ошибок до тех пор, пока мы не закончим запись. Первым параметром для каждой команды всегда является буфер команд для записи команды. Второй параметр определяет детали прохождения рендеринга

мы только что предоставили. Последний параметр определяет, как будут предоставляться команды рисования на этапе рендеринга. Он может иметь одно из двух значений:

VK_SUBPASS_CONTENTS_INLINE : Команды render pass будут помещены в сам буфер первичной команды, и никакие буферы вторичной команды не будут.

VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS : Прохождение рендеринга команды будет выполняться из дополнительных командных буферов.

Мы не будем использовать дополнительные буферы команд, поэтому остановимся на первом варианте.

Основные команды рисования

Теперь мы можем привязать графический конвейер:

```
1 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,  
                    graphicsPipeline);
```

Второй параметр указывает, является ли объект конвейера графическим или вычислительным конвейером. Теперь мы рассказали Vulkan, какие операции выполнять в графическом конвейере и какое вложение использовать в шейдере фрагментов. Как отмечалось в главе о фиксированных функциях, мы действительно указали видовой экран и ножничное состояние, чтобы этот конвейер был динамическим. Поэтому нам нужно установить их в командный буфер перед

выполнением нашей команды рисования: видовой экран VkViewport{};

```
1 видовой экран.x = 0.0f;  
2 видовой экран.y = 0.0f;  
3 видовой экран.width = static_cast<float>(swapChainExtent.ширина);  
4 окно просмотра.высота = static_cast<float>(swapChainExtent.высота);  
5 viewport.minDepth = 0.0f;  
6 viewport.maxDepth = 1.0f;  
7 vkCmdSetViewport(commandBuffer, 0, 1, &viewport);  
8  
10 VkRect2D scissor{};  
11 scissor.offset = {0, 0};  
12 scissor.extent = swapChainExtent;  
13 vkCmdSetScissor(commandBuffer, 0, 1, &scissor);
```

Теперь мы готовы выполнить команду рисования треугольника:

```
1 vkCmdDraw(commandBuffer, 3, 1, 0, 0);
```

Фактический функция vkCmdDraw немного разочаровывает, но она такая простая, потому что всю информацию мы указали заранее. Он имеет следующие параметры, помимо командного буфера:

`.vertexCount` Хотя у нас нет буфера вершин, мы технически еще есть 3 вершины,

- чтобы рисовать. `instanceCount`: Используется для рендеринга экземпляра, используйте это.
- `firstVertex` : Используется как смещение в буфер вершин, определяет значение. наименьшее значение `gl_VertexIndex`
- `firstInstance` : Используется как смещение для рендеринга наименьшее значение экземпляра, определяет `gl_InstanceIndex`.

1 если вы не выполняете

Завершение работы

Теперь этап рендеринга может быть завершен:

```
1 vkCmdEndRenderPass(командный буфер);
```

И мы закончили запись командного буфера:

```
1 если (vkEndCommandBuffer(Командный буфер) != VK_SUCCESS) {  
2     выбросить std::runtime_error ("не удалось записать командный буфер!");  
3 }
```

В следующей главе мы напишем код для основного цикла, который получит изображение из цепочки обмена, запишет и выполнит команду буфера, затем вернет готовое изображение в цепочку обмена.

Код на C++ / Вершинный шейдер / Фрагментный шейдер

Рендеринг и представление

В этой главе все будет собрано воедино. Мы собираемся написать `drawFrame` функция, которая будет вызываться из основного цикла для вывода треугольника на экран.

Давайте начнем с создания функции и вызова ее из `mainLoop`:

```
1 анулирование mainLoop() {  
2     в то время как (!glfwWindowShouldClose(окно)) {  
3         glfwPollEvents();  
4         drawFrame();  
5     }  
6 }  
7  
8 ...  
9  
10 пустота drawFrame() {  
11  
12 }
```

Схема фрейма

На высоком уровне рендеринг фрейма в Vulkan состоит из обычного набора шагов:

- Дождитесь завершения предыдущего кадра
 - Получите изображение из цепочки подкачки
 - Запишите буфер команд, который отрисовывает сцену на этом изображении
 - Отправьте записанный буфер команд
- Хотя мы расширим функцию рисования в последующих главах, сейчас это ядро нашего цикла рендеринга.

Синхронизация

Основная философия проектирования в Vulkan заключается в том, что синхронизация выполнения на графическом процессоре является явной. Порядок операций мы должны определить, используя различные примитивы синхронизации, которые сообщают драйверу порядок, в котором мы хотим, чтобы что-то выполнялось. Это означает, что многие вызовы Vulkan API, которые начинают выполнять работу на графическом процессоре, являются асинхронными, функции вернутся до завершения операции. В этой главе описан ряд событий, которые нам нужно упорядочить явно поскольку они происходят на графическом процессоре, например:

- Захват изображения с замены цепи • выполнять команды, нарисуйте на получившемся изображении • Представьте, что изображение на экране для презентации, возвращая его в `swapchain`

Каждое из этих событий приводится в действие с помощью одного вызова функции, но все они выполняются асинхронно. Вызовы функций будут возвращаться до того, как операции будут фактически завершены, и порядок выполнения также не определен. Это неудачно, потому что каждая из операций зависит от завершения предыдущей.

Таким образом, нам нужно изучить, какие примитивы мы можем использовать для достижения желаемого порядка.

Семафоры Семафор используется для добавления порядка между операциями очереди. Операции с очередью относятся к работе, которую мы отправляем в очередь либо в буфере команды, либо изнутри функции, как мы увидим позже. Примерами очередей являются графическая очередь и очередь презентаций. Семафоры используются как для упорядочивания работы внутри одной очереди, так и между разными очередями. Так получилось, что в Vulkan есть два вида семафоров: бинарные и временные. Поскольку в этом руководстве будут использоваться только двоичные семафоры, мы не будем обсуждать семафоры временной шкалы. Дальнейшее упоминание термина "семафор" относится исключительно к двоичным семафорам. Семафор может быть либо безсигнальным, либо с сигнализацией. Он начинает свою жизнь без сигнала. Способ, которым мы используем семафор для упорядочивания операций с очередью, заключается в предоставлении

тот же семафор, что и семафор 'signal' в одной операции с очередью и как семафор 'wait' в другой операции с очередью. Например, допустим, у нас есть семафор S и операции с очередями A и B, которые мы хотим выполнить по порядку. Что мы говорим Vulkan, так это то, что операция A будет "сигнализировать" семафору S, когда она завершит выполнение, вырезая, а операция B будет "ждать" семафора S, прежде чем она начнет выполняться. Когда операция A завершится, семафору S будет подан сигнал, в то время как операция B не начнется, пока не будет подан сигнал S. После начала выполнения операции B семафор S автоматически сбрасывается обратно в режим отключения сигнала, что позволяет использовать его снова. Псевдокод того, что только что было описано:

```
1 VkCommandBuffer A, B = ... // записываем буферы команд
2 VkSemaphore S = ... // создаем семафор
3 // ставим в очередь A, сигнализируем S по завершении - выполнение начинается
4 // немедленно vkQueueSubmit(work: A, signal: S, wait: None)
5 // ставим в очередь B, ждем запуска S
6 vkQueueSubmit(work: B, signal: None, wait: S)
```

Обратите внимание, что в этом фрагменте кода, как звонки `vkQueueSubmit()` вернуться Имме- немедленно - ожидание происходит только на GPU. Центральный процессор продолжает работать без блокировки. Чтобы заставить центральный процессор ждать, нам нужен другой примитив синхронизации, который мы сейчас опишем.

Ограждения Ограждение имеет аналогичное назначение, поскольку оно используется для синхронизации выполнения, но оно предназначено для упорядочивания выполнения на центральном процессоре, иначе известном как хост. Проще говоря, если хосту нужно знать, когда графический процессор что-то закончил, мы используем забор. Подобно семафорам, ограждения находятся либо в сигнализированном, либо безсигнализированном состоянии. Когда-нибудь-либо мы отправляем работу на выполнение, мы можем прикрепить ограждение к этой работе. Когда работа будет закончена, на ограждение подадут сигнал. Затем мы можем заставить хост ждать подачи сигнала на ограждение, гарантуя завершение работы до того, как хост продолжит. Конкретный пример - создание скриншота. Допустим, мы уже выполнили необходимую работу с графическим процессором. Теперь нужно перенести изображение с графического процессора на хост, а затем сохранить в памяти в файл. У нас есть командный буфер A, который выполняет передачу и блокировку F. Мы отправляем командный буфер A с блокировкой F, затем немедленно сообщаем хосту дождаться сигнала F. Это приводит к тому, что хост блокируется до тех пор, пока командный буфер A не завершит выполнение. Таким образом, мы можем позволить хосту сохранить файл на диск после завершения переноса в память. Псевдокод для того, что было описано:

```
1 VkCommandBuffer A = ... // записать командный буфер с передачей
2 VkFence F = ... // создать ограждение
3
```

```

4 // поставить в очередь A, немедленно приступить к работе, подать
5 сигнал F по завершении vkQueueSubmit(работа: A, забор: F)
6
7 vkWaitForFence(F) // блокирует выполнение до тех пор, пока A не завершит выполнение
8
9 функция save_screenshot_to_disk() // не может быть запущена до тех пор, пока передача не завершится.

ЗАКОНЧЕННЫЕ

```

В отличие от примера с семафором, этот пример *действительно* блокирует выполнение узла. Это означает, что хост не будет ничего делать, кроме как ждать завершения выполнения. В этом случае мы должны были убедиться, что передача завершена, прежде чем мы сможем сохранить снимок экрана на диск. В общем, предпочтительно не блокировать хост без необходимости. Мы хотим обеспечить графический процессор и хост полезной работой. Ожидание на ограждениях сигнала - бесполезная работа. Таким образом, мы предпочитаем семафоры или другие еще не рассмотренные примитивы синхронизации для синхронизации нашей работы. Ограждения необходимо сбросить вручную, чтобы вернуть их в состояние без сигнала. Это связано с тем, что ограждения используются для управления выполнением хоста, и поэтому хост может решать, когда сбросить ограждение. Сравните это с семафорами, которые используются для упорядочивания работы графического процессора без участия хоста.

Вкратце, семафоры используются для указания порядка выполнения операций на графическом процессоре, в то время как ограждения используются для синхронизации центрального и графического процессоров друг с другом.

Что выбрать?
 Следовательно, два места для применения синхронизации: операции с цепочкой обмена и ожидание завершения предыдущего кадра. Мы хотим использовать семафоры для операций swapchain, потому что они происходят на графическом процессоре, поэтому мы не хотим заставлять хост ждать, если мы можем ему помочь. Для ожидания завершения предыдущего кадра мы хотим использовать ограждения по противоположной причине, потому что нам нужно, чтобы ведущий подождал. Это делается для того, чтобы мы не рисовали более одного кадра одновременно. Поскольку мы перезаписываем буфер команд каждого кадра, мы не можем записывать работу следующего кадра в буфер команд до тех пор, пока не завершится выполнение текущего кадра, поскольку мы не хотим перезаписывать текущее содержимое буфера команд, пока графический процессор использует его.

Создание объектов синхронизации
 Нам понадобится один семафор, чтобы сигнализировать о том, что изображение получено из цепочки обмена и готово к рендерингу, другой, чтобы сигнализировать о том, что рендеринг завершен и можно приступить к представлению, и ограждение, чтобы убедиться, что одновременно рендерится только один кадр.

Создайте три члена класса для хранения этих объектов semaphore и объекта fence:

```
1 VkSemaphore imageAvailableSemaphore;
2 VkSemaphore renderFinishedSemaphore;
3 VkFence защита от полетов;
```

Для создания семафоров мы добавим

Создать функция для этой части

последнее руководство: `createSyncObjects` :

```
1 пустота initVulkan() {
2     CreateInstance(); setupDebugMessenger();
3     createSurface();
4     pickPhysicalDevice();
5     createLogicalDevice();
6     CreateSwapChain();
7     createImageViews();
8     createRenderPass();
9     createGraphicsPipeline();
10    createFramebuffers();
11    createCommandPool();
12    createCommandBuffer();
13    createSyncObjects();
14 }
15 }
16 ...
17 ...
18 ...
19 void createSyncObjects() {
20 }
21 }
```

Для создания семафоров требуется заполнить `VkSemaphoreCreateInfo`, но в

текущей версии API фактически нет никаких обязательных полей, кроме

`sType`:

```
1 void createSyncObjects() {
2     VkSemaphoreCreateInfo semaphoreInfo{};
3     semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
4 }
```

Будущие версии Vulkan API или расширений могут добавить

функциональность для флагов и `pNext` параметры, как и для других конструкций.

Для создания ограждения требуется заполнить поле `VkFenceCreateInfo`:

```
1 vkfencecreate_info fenceInfo{};
2 fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
```

Создание семафоров и ограждения выполняется по знакомому шаблону с помощью

`vkCreateSemaphore` & `vkCreateFence`:

```

1 если (vkCreateSemaphore(устройство, &semaphoreInfo, nullptr,
2 &imageAvailableSemaphore) != VK_SUCCESS ||
3 vkCreateSemaphore(устройство, &semaphoreInfo, nullptr,
4 &renderFinishedSemaphore) != VK_SUCCESS || vkCreateFence(device,
5 &fenceInfo, nullptr, &inFlightFence) != 3
    VK_SUCCESS) {
4 выбросить std::runtime_error("не удалось создать семафоры!");
5 }

```

Семафоры и ограждение должны быть очищены в конце программы, когда завершатся все команды и больше не потребуется синхронизация:

```

1 void cleanup() {
2     vkDestroySemaphore(устройство, imageAvailableSemaphore,
3     nullptr); vkDestroySemaphore(устройство, renderFinishedSemaphore,
4     nullptr); vkDestroyFence(устройство, защита полета, nullptr);

```

Переходим к основной функции рисования!

Ждем предыдущего кадра

В начале фрейма мы хотим дождаться завершения предыдущего фрейма, чтобы буфер команд и семафоры были доступны для использования. Для этого

мы вызываем `vkWaitForFences`:

```

1 анулирование drawFrame() {
2     vkWaitForFences(устройство, 1, &защита полета, VK_TRUE, UINT64_MAX);
3 }

```

Функция `vkWaitForFences` принимает массив ограждений и ожидает на хосте для любого или всех ограждений необходимо подать сигнал перед возвращением. The `VK_TRUE` проход здесь указывает на то, что мы хотим дождаться всех ограждений, но в случае с единственным это не имеет значения. У этой функции также есть параметр `timeout`, для которого мы устанавливаем максимальное значение 64-битного целого числа без знака, что эффективно отключает тайм-аут.

После ожидания нам нужно вручную сбросить ограждение в состояние без сигнала с помощью

`vkResetFences` вызова:

```
1 Дополнительные защиты (устройство, 1 и защита полета);
```

Прежде чем мы сможем продолжить, в нашем дизайне есть небольшая заминка. В первом кадре мы вызываем `drawFrame()`, который немедленно ожидает включения функции `flightfence` для получения сигнала. `flightfence` сигнализируется только после завершения рендеринга кадра, однако, поскольку это первый кадр, предыдущих кадров, в которых можно сигнализировать о ограждении, нет! Таким образом `vkWaitForFences()` блокирует на неопределенный срок, ожидая чего-то, чего никогда не произойдет.

Среди множества решений этой дилеммы есть хитроумный обходной путь, встроенный в API. Создайте ограждение в сигнальном состоянии, чтобы при первом вызове функции `vkWaitForFences()` возвращалось немедленно, поскольку ограждение уже сигнализировано. Для этого мы добавляем команду `VK_FENCE_CREATE_SIGNALLED_BIT` установите флагок для `VkFenceCreateInfo`

```
void createSyncObjects() {
    ...
    ...
    VkFenceCreateInfo fenceInfo{};
    fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
    fenceInfo.flags = VK_FENCE_CREATE_SIGNALLED_BIT;
    ...
    ...
}
```

Получение изображения из цепочки подкачки

Следующее, что нам нужно сделать в функции `drawFrame`, - это получить изображение

из цепочки подкачки. Напомним, что цепочка подкачки является функцией расширения, поэтому мы должны использовать функцию с `Vk * KHR` соглашением

```
об именовании:void drawFrame() {
    uint 3 2 _t ImageIndex;
    Запрашивает изображение (устройство, цепочку обмена, UINT 6 4 _MAX,
    imageAvailableSemaphore, VK_NULL_HANDLE iImageIndex);
}
```

Первые два параметра - это логическое устройство и цепочка обмена, из которой мы хотим получить изображение. Третий параметр задает время ожидания в наносекундах для того, чтобы изображение стало доступным. Использование максимального значения 64-битного целого числа без знака означает, что мы эффективно отключаем тайм-аут. Следующие два параметра определяют объекты синхронизации, которые должны быть сигнализированы, когда механизм представления закончит использовать изображение. Это момент времени, когда мы можем начать рисовать на нем. Можно указать семафор, ограждение или и то, и другое. Мы собираемся использовать наш `imageAvailableSemaphore` для этой цели здесь.

Последний параметр задает переменную для вывода индекс замены цепи изображения, которая стала доступной.

Индекс относится к `VkImage`

в нашем `swapChainImages` массив. Мы собираемся использовать этот индекс для выбора `VkFramebuffer`.

Запись командного буфера

С помощью `ImageIndex`, указывающего изображение цепочки подкачки для использования, мы можем теперь записать командный буфер. Сначала мы вызываем `vkResetCommandBuffer` в буфере команд, чтобы убедиться, что его можно записать.

```
1 vkResetCommandBuffer (commandBuffer, 0);
```

Второй параметр `vkResetCommandBuffer` это `VkCommandBufferResetFlagBits`. Отметить. Поскольку мы не хотим делать ничего особенного, мы оставляем значение равным 0. Теперь вызываем функцию `recordCommandBuffer` для записи нужных команд.

```
1 recordCommandBuffer(командный буфер, ImageIndex);
```

Теперь, имея полностью записанный командный буфер, мы можем отправить его.

Отправка командного буфера

Отправка и синхронизация очереди настраиваются с помощью параметров в

`VkSubmitInfo` структура.

```
1 VkSubmitInfo submitInfo{};
2 submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
3 VkSemaphore waitSemaphores[] = {imageAvailableSemaphore};
4 VkPipelineStageFlags waitStages[] =
5     {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
6 submitInfo.waitSemaphoreCount = 1;
7 submitInfo.pWaitSemaphores = waitSemaphores;
8 submitInfo.pWaitDstStageMask = этапы ожидания;
```

Первые три параметра определяют, какие семафоры следует ожидать до начала выполнения и на каком этапе (ах) конвейера следует ожидать. Мы хотим подождать с записью цветов к изображению, пока оно не станет доступным, поэтому мы указываем этап графического конвейера, который выполняет запись в цветовое вложение. Это означает, что теоретически реализация уже может начать выполнение нашего вершинного шейдера и тому подобное, пока изображение еще недоступно. Каждая запись в разделе этапы ожидания массив соответствует семафору с таким же индексом в `pWaitSemaphores`.

```
1 submitInfo.commandBufferCount = 1;
2 submitInfo.pCommandBuffers = &commandBuffer;
```

Следующие два параметра определяют, какие буфера команд фактически отправлять на выполнение. Мы просто отправляем единственный имеющийся у нас буфер

```
1 КОМАНД.VkSemaphore signalSemaphores[] = {renderFinishedSemaphore};
2 submitInfo.signalSemaphoreCount = 1;
3 submitInfo.pSignalSemaphores = Сигнальные семафоры;
```

Значения `signalSemaphoreCount` и `pSignalSemaphores` параметры определяют, какие семафоры будут сигнализировать после завершения выполнения буфера (ов) команд. В нашем случае мы используем для этой цели.

```

1     Завершенный рендеринг семафора
2     если (vkQueueSubmit(graphicsQueue, 1, &submitInfo, защита полета) != VK_SUCCESS)
3     {
4         throw std::runtime_error("не удалось отправить команду рисования
5
6             буфер!");
7     }

```

Теперь мы можем отправить буфер команд в графическую очередь с помощью `vkQueueSubmit`. Функция принимает массив `VkSubmitInfo` структуры в качестве аргумента эффективности при гораздо большей рабочей нагрузке. Последний параметр ссылается на необязательный параметр, который будет сигнализироваться, когда буфера команд завершат выполнение. Это позволяет нам знать, когда это безопасно для команды буфера для повторного использования, таким образом мы хотим дать ей `inFlightFence`. Теперь в следующем кадре центральный процессор будет ждать завершения выполнения этого командного буфера, прежде чем записывать в него новые команды.

Зависимости подпуска.

Помните, что подпуски на этапе рендеринга автоматически обрабатывают переходы изображения макета. Этими переходами управляют **зависимости подпусков**, которые определяют зависимости памяти и выполнения между подпасами. Прямо сейчас у нас есть только один подпасок, но операции непосредственно перед и сразу после этого подпаса также считаются неявными "подпасами". Существуют две встроенные зависимости, которые обеспечивают переход в начале прохода рендеринга и в конце прохода рендеринга, но первая не выполняется в нужное время.

Предполагается,

что переход- происходит в начале конвейера, но на этом этапе мы еще не получили изображение!

Есть два способа решить эту проблему.

Мы могли бы изменить `VkSemaphore` этапы ожидания для `to` чтобы гарантировать, что проходы рендеринга этого не делают `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` начинайте, пока изображение не будет доступно, или мы можем отложить прохождение рендеринга до этапа

`VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` Я решил использовать здесь второй вариант, потому что это хороший повод взглянуть на зависимости подпуска и на то, как они работают.

Зависимости вспомогательных каналов указаны в Структурах `VkSubpassDependency`.

Перейдите к `createRenderPass` функции и добавьте ее:

```

1 VkSubpassDependency зависимость{};
2 dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
3 dependency.dstSubpass = 0;

```

Первые два поля определяют индексы зависимости и зависимого подпрохода. Специальное значение `VK_SUBPASS_EXTERNAL` ссылается на неявный подпасок, который находится перед или после прохода визуализации, в зависимости от того, указан ли он в `srcSubpass`

или `dstSubpass`. Индекс `0` относится к нашему подпасу, который является первым и единственным. В `dstSubpass` всегда должно быть выше, чем `srcSubpass` для предотвращения циклов в графе зависимостей (если только один из подпусков не является). `зависимость.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;`

`зависимость.srcAccessMask = 0;`

`VK_SUBPASS_EXTERNAL`

В следующих двух полях указаны ожидаемые операции и этапы, на которых эти операции выполняются. Нам нужно дождаться завершения считывания цепочки подкачки с изображения, прежде чем мы сможем получить к нему доступ. Этого можно добиться ожиданием на самом

1 этапе вывода цветного вложения. `зависимость.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;`

2 `зависимость.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;`

Операции, которые должны подождать с этим, находятся на стадии прикрепления цвета и включайте в себя написание цветного приложения. Эти настройки предотвратят переход до тех пор, пока он не станет действительно необходимым (и разрешенным): когда мы захотим начать добавлять к нему цвета.

`renderPassInfo.dependencyCount = 1; renderPassInfo.pDependencies = & зависимость;`

The `VkRenderPassCreateInfo` struct есть два поля для задания массива зависимостей..

Представление

Последним этапом рисования рамки является отправка результата обратно в цепочку обмена, чтобы он в конечном итоге отображался на экране. Презентация настраивается через `VkPresentInfoKHR` структура в конце рисунка `drawFrame` функция.

```
1 VkPresentInfoKHR presentInfo{};
2 presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
3
4 presentInfo.waitSemaphoreCount = 1;
5 presentInfo.pWaitSemaphores = Сигнальные семафоры;
```

Первые два параметра определяют, какие семафоры следует ожидать перед тем, как может произойти представление, точно так же, как `VkSubmitInfo`. Поскольку мы хотим дождаться выполнения команды буфер для завершения выполнения, таким образом, рисуется наш треугольник, мы берем семафоры, о которых будет подан сигнал, и ждем их, таким образом, мы исполь

`Сигналсемафоры`

```
1 swapChains[] = {swapChain}; presentInfo.swapchainCount = 1;
2 presentInfo.pSwapchains = swapChains;
3 presentInfo.pImageIndices = &iImageIndex;
```

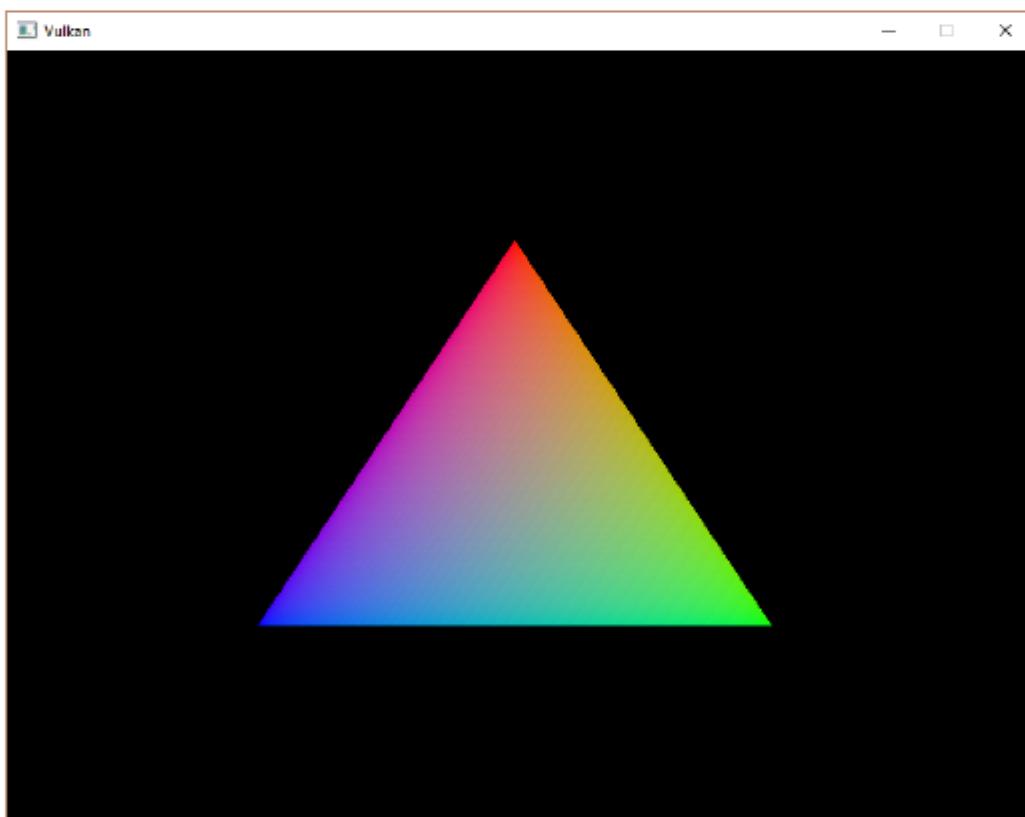
Следующие два параметра определяют цепочки подкачки для представления изображений и индекс изображения для каждой цепочки подкачки. Это почти всегда будет одна цепочка.

```
1 presentInfo.Предварительные результаты = nullptr; // Необязательно
```

Существует последний необязательный параметр, который называется массивом `VkResult` значений для проверки для каждой цепочки обмена, если изложение было успешным. В этом нет необходимости, если вы используете только одну цепочку подкачки, потому что вы можете просто использовать возвращаемое значение

```
1 текущей функции. vkQueuePresentKHR(presentQueue, &presentInfo);
```

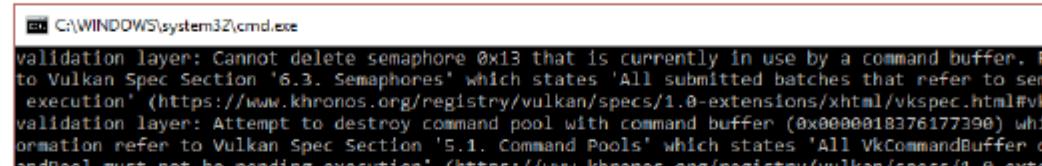
The `vkQueuePresentKHR` функция отправляет запрос на представление изображения в цепочку подкачки. Мы добавим обработку ошибок для обоих `vkAcquireNextImageKHR` и `vkQueuePresentKHR` в следующей главе, потому что их сбой не обязательно означает, что программа должна завершиться, в отличие от функций, которые мы видели до сих пор. Если до этого момента вы все делали правильно, то теперь при запуске вашей программы вы должны увидеть нечто похожее на следующее.:



Этот цветной треугольник может выглядеть немного иначе, чем тот, который вы привыкли видеть в руководствах по графике. Это потому, что этот учебник позволяет шейдеру интерполировать в линейном цветовом пространстве и преобразовывать в sRGB

цветовое пространство после этого. Смотрите
это сообщение в блоге для обсуждения разницы.

Ура! К сожалению, вы увидите, что когда уровни проверки
включены, программа вылетает, как только вы ее закрываете. Сообщения,
распечатанные на терминале от, объясняют нам, почему:



```
validation layer: Cannot delete semaphore 0x13 that is currently in use by a command buffer. Refer to Vulkan Spec Section '6.3. Semaphores' which states 'All submitted batches that refer to semaphores for execution' (https://www.khronos.org/registry/vulkan/specs/1.0-extensions/xhtml/vkspec.html#vkDestroySemaphore) validation layer: Attempt to destroy command pool with command buffer (0x0000018376177390) which refers to Vulkan Spec Section '5.1. Command Pools' which states 'All VkCommandBuffer objects in a command pool must not be pending execution' (https://www.khronos.org/registry/vulkan/specs/1.0-extensions/xhtml/vkspec.html#vkDestroyCommandPool)
```

Помните, что все операции в `drawFrame` являются асинхронными.
Это означает, что когда мы выходим из цикла в `mainLoop`, операции
рисования и презентации, возможно, все еще продолжаются.
Очистка ресурсов во время этого - плохая идея. Чтобы устранить эту
проблему, мы должны дождаться завершения работы логического
устройства перед выходом из `mainLoop` и уничтожением окна:

```
1 void mainLoop() {
2     while (true) {
3         glfwPollEvents();
4         drawFrame();
5     }
6
7     vkDeviceWaitIdle(device);
8 }
```

Вы также можете дождаться завершения операций
в определенной очереди команд с помощью функции `vkDeviceWaitIdle`.
способ выполнить синхронизацию. Вы увидите, что
программа теперь завершается без проблем при закрытии окна.

Заключение

Спустя чуть более 900 строк кода мы, наконец, добрались до стадии просмотра
ЧТО-ТО ВСПЫВАЕТ НА ЭКРАНЕ! Начальная загрузка программы Vulkan - это,
безусловно, большая работа, но вывод заключается в том, что Vulkan дает вам
огромный контроль благодаря своей четкости. Я рекомендую вам потратить
некоторое время сейчас, чтобы перечитать код и построить мысленную
модель назначения всех объектов Vulkan в программе и того, как они
сочетаются друг с другом. С этого момента мы будем использовать
полученные знания для расширения функциональности программы. В
следующей главе цикл рендеринга будет расширен для обработки нескольких
кадров в полете. Код на C++ / Вершинный шейдер / Фрагментный шейдер

Кадры в полете

Кадры в полете

Прямо сейчас у нашего цикла рендеринга есть один вопиющий недостаток. От нас требуется дождаться завершения предыдущего кадра, прежде чем мы сможем начать рендеринг следующего, что приводит к ненужному холостому ходу хоста.

Способ исправить это - разрешить использование нескольких кадров *в полете* сразу, это то есть разрешить рендеринг одного кадра, чтобы он не мешал записи следующего. Как нам это сделать? Любой ресурс, к которому обращаются и который изменяется во время рендеринга, должен дублироваться. Таким образом, нам нужно несколько буферов команд, семафоров и ограждений. В последующих главах мы также добавим несколько экземпляров других ресурсов, так что эта концепция появится снова.

Начните с добавления константы в верхней части программы, которая определяет, сколько кадров должно обрабатываться одновременно:

```
1 const int MAX_FRAMES_IN_FLIGHT = 2;
```

Мы выбираем число 2, потому что мы не хотим, чтобы центральный процессор получал *слишком* далеко впереди графического процессора. Когда в полете находится 2 кадра, центральный процессор и графический процессор могут работать над своими собственными задачами одновременно. Если процессор завершит работу раньше, он подождет, пока графический процессор завершит рендеринг, прежде чем отправлять дополнительную работу. С 3 или более кадрами в режиме ожидания., центральный процессор может опережать графический процессор, увеличивая задержку кадров. Как правило, дополнительная задержка нежелательна. Но предоставление приложению контроля над количеством кадров в полете

- еще один пример явного использования Vulkan. Каждый фрейм должен иметь свой собственный буфер команд, набор семафоров и ограждение. Переименуйте, а затем измените их на `std::vector` объекты:

```
1 std::vector<VkCommandBuffer> Командные буферы;
2 ...
3 std::vector<VkSemaphore> Доступные для изображения семафоры;
4 std::vector<VkSemaphore> Завершенные для рендеринга семафоры;
5 std::vector<VkFence> Бортовые системы;
6
7 Затем нам нужно создать несколько буферов команд. Переименуйте createCommandBuffer в . Затем нам нужно изменить размер вектора буферов команд.
```

Создайте командные буферы **размером** `MAX_FRAMES_IN_FLIGHT`, **измените** `VkCommandBufferAllocateInfo` содержать такое количество командных буферов, а затем изменить назначение на наш вектор командных буферов:

```
1 void createCommandBuffers() {
2     commandBuffers.resize(MAX_FRAMES_IN_FLIGHT);
3     ...
4     allocInfo.commandBufferCount = (uint32_t) Командные буферы.size();
5 }
```

```

6     if (vkAllocateCommandBuffers(устройство и allocInfo,
7         commandBuffers.data()) != VK_SUCCESS) {
8         выбросить std::runtime_error("не удалось выделить команду
9             буферы!");
10    }
11 }
```

функцию следует изменить, чтобы создать все объекты:

```

1 void createSyncObjects()
2 {
3     Доступные изображения.изменить размер(MAX_FRAMES_IN_FLIGHT);
4     Завершенные изображения.изменить размер(MAX_FRAMES_IN_FLIGHT);
5     Полетные ограждения.изменить размер(MAX_FRAMES_IN_FLIGHT);
6
7     VkSemaphoreCreateInfo semaphoreInfo{};
8     semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
9
10    VkFenceCreateInfo fenceInfo{};
11    fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
12    fenceInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;
13
14    для (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
15        if (vkCreateSemaphore(устройство, &semaphoreInfo, nullptr,
16            &imageAvailableSemaphores[i]) != VK_SUCCESS ||
17            vkCreateSemaphore(устройство, &semaphoreInfo, nullptr,
18            &renderFinishedSemaphores[i]) != VK_SUCCESS ||
19            vkCreateFence(устройство, &fenceInfo, nullptr,
20            &Бортовые ограждения[i]) != VK_SUCCESS) {
21                выбросить std::runtime_error("не удалось создать
22                    объекты синхронизации для фрейма!");
23            }
24        }
25    }
```

Аналогичным образом, все они также должны

```

1 быть очищены: void очистка() {
2     для (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
3         vkDestroySemaphore(устройство, renderFinishedSemaphores[i],
4             nullptr);
5         vkDestroySemaphore(устройство, доступные для изображения семафоры[i],
6             nullptr);
7         vkDestroyFence(устройство, бортовые ограждения[i], nullptr);
8     }
9     ...
10 }
```

```
9 }
```

Помните, поскольку командные буферы освобождаются для нас, когда мы освобождаем пул команд , для очистки командного буфера ничего дополнительно не нужно делать. Использовать нужные объекты в каждом кадре, мы должны следить за текущим кадром. Мы будем использовать индекс рамка

```
1 для этой цели: uint 3 2 _t свойство currentframe = 0 ;
```

В drawFrame теперь функцию можно изменить, чтобы использовать нужные

```
1 объекты: void drawFrame() {
2     vkWaitForFences(устройство, 1, и бортовые защиты [текущий кадр],
3                     VK_TRUE, UINT 6 4 _MAX);
4     Встроенные защиты (устройство, 1 и бортовые защиты [текущий кадр]);
5
6     vkAcquireNextImageKHR(устройство, цепочка обмена, UINT 6 4 _MAX,
7                           imageAvailableSemaphores[текущий кадр],
8                           VK_NULL_HANDLE, &ImageIndex);
9
10    ...
11
12    ...
13
14    submitInfo.pCommandBuffers = &commandBuffers[текущий кадр];
15
16    ...
17
18    VkSemaphore ожидает семафоры[] =
19        {доступные изображения[текущий кадр]};
20
21    ...
22
23    VkSemaphore сигналсемафоры[] =
24        {renderFinishedSemaphores[текущий кадр]};
25
26
27    если (vkQueueSubmit(graphicsQueue, 1, &submitInfo,
28                         Защита полета[текущий кадр]) != VK_SUCCESS) {
```

Конечно, мы не должны забывать каждый раз переходить к следующему кадру:

```

1 анулирование drawFrame() {
2     ...
3
4     Текущий кадр = (текущий кадр + 1) % MAX_FRAMES_IN_FLIGHT;
5 }

```

Используя оператор modulo (%), мы гарантируем, что индекс кадра повторяется после каждого поставленного в очередь кадра.

Теперь мы внедрили всю необходимую синхронизацию, чтобы гарантировать, что не более `MAX_FRAMES_IN_FLIGHT` рабочие кадры поставлены в очередь и что эти кадры не переступают друг через друга. Обратите внимание, что для других частей кода, таких как окончательная очистка, вполне допустимо полагаться на более грубую синхронизацию, например `vkDeviceWaitIdle`. Вы должны решить, какой подход использовать, исходя из требований к производительности.

Чтобы узнать больше о синхронизации на примерах, ознакомьтесь с этим подробным обзором от Khronos.

В следующей главе мы рассмотрим еще одну небольшую деталь, которая требуется для хорошо работающей программы Vulkan.

Код на C++ / Вершинный шейдер / Фрагментный шейдер

Отдых в цепочке обмена

Введение

Приложение, которое у нас сейчас есть, успешно рисует треугольник, но есть некоторые обстоятельства, с которыми оно пока не справляется должным образом. Поверхность окна может измениться таким образом, что цепочка подкачки больше не будет совместима с ней. Одной из причин, по которой это могло произойти, является изменение размера окна. Мы должны перехватить эти события и воссоздать цепочку подкачки.

Воссоздание цепочки подкачки

Создайте новую Воссоздает Wapchain функцию, которая вызывает Создает Wapchain и все из функций создания для объектов, которые зависят от цепочки обмена или размера окна.

```

1 void recreateSwapChain() {
2     vkDeviceWaitIdle(устройство);
3
4     CreateSwapChain();
5     createImageViews();
6     createFramebuffers();
7 }

```

Сначала мы вызываем `vkDeviceWaitIdle`, потому что, как и в предыдущей главе, мы не должны касаться ресурсов, которые, возможно, все еще используются. Очевидно, нам придется воссоздать саму цепочку подкачки. Представления изображений необходимо воссоздать, потому что они основаны непосредственно на изображениях цепочки подкачки. Наконец, фреймбуферы напрямую зависят от изображений цепочки подкачки и, следовательно, также должны быть воссозданы заново. Чтобы убедиться в том, что старые версии этих объектов, очищаются перед зону от- издаваться над ними, мы должны перенести часть очистки кода в функции, который мы можем вызвать из функции. Давайте назовем это `cleanupSwapChain`:

```
1 void cleanupSwapChain() {
2
3 }
4
5 пустота recreateSwapChain() {
6     vkDeviceWaitIdle(устройство);
7
8     cleanupSwapChain();
9
10    CreateSwapChain();
11    createImageViews();
12    createFramebuffers();
13 }
```

Обратите внимание, что мы не воссоздаем `renderpass` здесь для простоты. Теоретически может быть возможным изменение формата изображения цепочки подкачки в течение срока службы приложения, например, при перемещении окна из стандартного диапазона в монитор с высоким динамическим диапазоном. Это может потребовать от приложения воссоздания `renderpass`, чтобы убедиться, что изменение между динамическими диапазонами отражено должным образом. Мы переместим код очистки всех объектов, которые воссоздаются как часть подкачки обновление цепочки из очистки для Цепочка очистки:

```
1 пустота cleanupSwapChain() {
2
3     для (size_t i = 0; i < swapChainFramebuffers.size(); i++) {
4         vkDestroyFramebuffer(устройство, swapChainFramebuffers[i],
5                             nullptr);
6     }
7
8     для (size_t i = 0; i < swapChainImageViews.size(); i++) {
9         vkDestroyImageView(устройство, swapChainImageViews[i], nullptr);
10    }
11
12    vkDestroySwapchainKHR(устройство, цепочка обмена, nullptr);
13 }
14
15 анулирование очистка() {
```

```

14     cleanupSwapChain();
15
16     vkDestroyPipeline(устройство, графический конвейер, nullptr);
17     vkDestroyPipelineLayout(устройство, конвейерное описание, nullptr);
18
19     vkDestroyRenderPass(устройство, renderPass, nullptr);
20
21     для (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
22         vkDestroySemaphore(устройство, renderFinishedSemaphores[i],
23                           nullptr);
24         vkDestroySemaphore(устройство, доступные для изображения семафоры[i],
25                           nullptr);
26         vkDestroyFence(устройство, бортовые ограждения[i], nullptr);
27     }
28
29     vkDestroyCommandPool(устройство, commandPool, nullptr);
30
31     если (enableValidationLayers) {
32         destroyDebugutilsMessenger Текст(экземпляр, debugMessenger,
33                           nullptr);
34     }
35
36     vkDestroySurfaceKHR(экземпляр, поверхность, nullptr);
37     vkDestroyInstance(экземпляр, nullptr);
38
39     glfwDestroyWindow(окно);
40
41     glfwTerminate();
}

```

Обратите внимание, что Бираем Wapextent мы уже запрашиваем новое разрешение окна чтобы убедиться, что изображения цепочки подкачки имеют (новый) правильный размер, поэтому нет необходимости изменять Изменение цепочки обмена. Это все, что требуется для воссоздания цепочки подкачки! Однако, недостатком этого подхода является то, что нам нужно остановить весь рендеринг перед созданием новой цепочки подкачки . Можно создать новую цепочку подкачки, пока команды рисования на изображении из старой цепочки подкачки все еще выполняются. Вам нужно передать предыдущую цепочку подкачки в `oldSwapChain` поле в файле `VkSwapchainCreateInfoKHR` создайте и уничтожьте старую цепочку обмена, как только закончите ее использовать.

Неоптимальная или устаревшая цепочка подкачки

Теперь нам просто нужно выяснить, когда необходимо воссоздать цепочку подкачки, и вызвать нашу новую функцию. К счастью, Vulkan обычно просто сообщает нам, что цепочка обмена больше не подходит во время презентации. В `vkAcquireNextImageKHR` и `vkQueuePresentKHR` функции могут возвращать следующие специальные значения, указывающие на это.

- `VK_ERROR_OUT_OF_DATE_KHR`: Цепочка подкачки стала несовместимой с `surface` и больше не может использоваться для рендеринга. Обычно это происходит после изменения размера окна.
- `VK_SUBOPTIMAL_KHR`: Цепочку подкачки все еще можно использовать для успешного присутствует на поверхности, но свойства поверхности больше не соответствуют точно.

```
1 Результат VkResult = vkAcquireNextImageKHR(устройство, цепочка обмена,
2                                     UINT 64 _MAX, imageAvailableSemaphores[текущий фрейм],
3                                     VK_NULL_HANDLE, &ImageIndex);
4
5 если (результат == VK_ERROR_OUT_OF_DATE_KHR) {
6     recreateSwapChain();
7     Возврат;
8 } else if (результат != VK_SUCCESS && результат != VK_SUBOPTIMAL_KHR) {
9     выбросить std::runtime_error("не удалось получить изображение цепочки обмена!");
10}
```

Если при попытке получить цепочку обмена выясняется, что она устарела изображение, то его представление больше невозможно. Следовательно, мы должны немедленно воссоздать цепочку подкачки и повторить попытку в следующем вызове `drawFrame` вызов.

Вы также могли бы решить сделать это, если цепочка обмена неоптимальна, но я решил продолжить в любом случае в этом случае, потому что мы уже получили изображение. Оба

`VK_SUCCESS` и `VK_SUBOPTIMAL_KHR` считаются кодами возврата "успех".

```
1 результат = vkQueuePresentKHR(presentQueue, &presentInfo);
2
3 если (результат == VK_ERROR_OUT_OF_DATE_KHR || результат ==
4         VK_SUBOPTIMAL_KHR) {
5     воссоздает Swapchain();
6 } еще, если (результат != VK_SUCCESS) {
7     выбросить std::runtime_error("не удалось представить изображение цепочки подкачки!");
8 }
9 currentFrame = (currentFrame + 1) % MAX_FRAMES_IN_FLIGHT;
```

The `vkQueuePresentKHR` функция возвращает те же значения с тем же средним значением-
ing. В этом случае мы также воссоздадим цепочку обмена, если она неоптимальна, потому что мы хотим получить наилучший возможный результат.

Устранение тупиковой ситуации

Если мы попытаемся запустить код сейчас, то можем столкнуться с тупиковой ситуацией. Устраняя прослушивание кода, мы обнаруживаем, что приложение достигает, но никогда не переходит дальше него. Это происходит потому, что, когда `vkAcquireNextImageKHR` возвращает `VK_ERROR_OUT_OF_DATE_KHR`, мы воссоздаем цепочку обмена, а затем возвращаемся из `drawFrame`. Но прежде чем это произойдет, ожидание ограждения текущего фрейма было выполнено и сброшено. Поскольку мы возвращаемся немедленно, никакая работа не передается на выполнение и ограждении никогда не будет подан сигнал, вызывающий `vkWaitForFences` чтобы остановиться навсегда. К счастью, есть простое решение. Отложите сброс ограждения до тех пор, пока мы не узнаем наверняка, что отправим работу с ним. Таким образом, если мы вернемся раньше, забор по-прежнему сигнализируется, и `vkWaitForFences` не приведет к тупиковой ситуации при следующем использовании того же объекта fence.

Начало строки `drawFrame` теперь должно выглядеть следующим образом:

```
1     vkWaitForFences(устройство, 1, &inFlightFences[текущий кадр], VK_TRUE,
2
3         UINT 64 _MAX);
4
5     uint 32_t ImageIndex;
6
7     VkResult результат = vkAcquireNextImageKHR(устройство, цепочка обмена,
8
9         UINT 64 _MAX, imageAvailableSemaphores[текущий фрейм],
10
11        VK_NULL_HANDLE, &ImageIndex);
12
13    если (результат == VK_ERROR_OUT_OF_DATE_KHR) {
14
15        recreateSwapChain();
16
17        Возврат;
18
19    } else if (результат != VK_SUCCESS && результат != VK_SUBOPTIMAL_KHR) {
20
21        выбросить std::runtime_error("не удалось получить изображение цепочки подкачки!");
22
23    }
24
25    // Сброс настроек выполняется только в том случае, если мы отправляем работу
26    vkResetFences(device, 1, &inFlightFences[currentFrame]);
```

Обработка изменяет размеры явно

Хотя многие драйверы и платформы запускают `VK_ERROR_OUT_OF_DATE_KHR` автоматически после изменения размера окна это не гарантировано произойдет. Вот почему мы добавим дополнительный код, который также будет явно обрабатывать изменения размера. Сначала добавьте новую переменную-член, которая помечает,

```
1 что произошло изменение размера: std::vector<VkFence> Бортовые ограждения;
2
3 bool framebufferResized = false;
```

The `drawFrame` затем функцию следует изменить, чтобы она также проверяла наличие этого флага:

```

1 if (результат == VK_ERROR_OUT_OF_DATE_KHR || результат ==
2 VK_SUBOPTIMAL_KHR || Размер кадрового
3 буфера) { Размер кадрового буфера = false;
4 } еще, если (результат != VK_SUCCESS) {
5 ...
6 }

```

Важно сделать это после `vkQueuePresentKHR` чтобы гарантировать, что семафоры находятся в согласованном состоянии, в противном случае семафор с сигналом может никогда не быть должным образом обработан. Теперь, чтобы действительно определять изменения размера, мы можем использовать обратный вызов `glfwSetFramebufferSizeCallback` функция в фреймворке GLFW для настройки обратного вызова:

```

1 void initWindow() {
2     glfwInit();
3
4     glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
5
6     window = glfwCreateWindow(ШИРИНА, ВЫСОТА, "Vulkan", nullptr,
7                               nullptr);
7     glfwSetFramebufferSizeCallback(окно,
8                                   Обратный вызов framebufferResizeCallback);
9 }
10
11 staticская пустота framebufferResizeCallback(GLFWwindow * window, int ширина,
12                                               int высота) {
13 }

```

Причина, по которой мы создаем статический функция обратного вызова вызвана тем, что GLFW не знает, как правильно вызывать функцию-член с правильным `this` указатель на наш

Здравствуйте, приложение Triangle пример.

Однако мы получаем ссылку на `GLFWwindow` в обратном вызове также есть другая функция GLFW, которая позволяет вам хранить произвольный указатель

```

внутри нее:
    glfwSetWindowUserPointer window : ;
1 = glfwCreateWindow(ШИРИНА, ВЫСОТА, "Vulkan", nullptr, nullptr);
2 Указатель glfwSetWindowUserPointer(окно, это);
3 glfwSetFramebufferSizeCallback (окно, обратный вызов framebufferResizeCallback);

```

Теперь это значение можно получить из обратного вызова с помощью указатель

`glfwGetWindowUserPointer` чтобы правильно установить флаг:

```

1 static void framebufferResizeCallback(GLFWwindow * window, int ширина,
2                                     int высота) {

```

```

2     auto приложение =
3         повторная интерпретация<Здравствуйте, приложение Triangle * >(glfwGetWindowUserPointer(окно));
4     приложение-> размер фреймбуфера = true;
5 }
```

Теперь попробуйте запустить программу и изменить размер окна, чтобы увидеть, действительно ли размер фреймбуфера изменен должным образом вместе с размером окна.

Обработка минимизации

Существует еще один случай, когда цепочка подкачки может устареть, и это особый вид изменения размера окна: минимизация окна. Этот случай особенный, потому что это приведет к размеру буфера кадра 0. В этом руководстве мы справимся с этим, сделав паузу до тех пор, пока окно снова не окажется на переднем плане, расширив `recreateSwapChain`

```

void recreateSwapChain() {           функция:
1

2     int ширина = 0, высота = 0 ;
3     glfwGetFramebufferSize(окно, &ширина, &высота);
4     в то время как (ширина == 0 || высота == 0) {
5         glfwGetFramebufferSize(размер окна, ширина и
6         высота); glfwWaitEvents();
7     }
8
9     vkDeviceWaitIdle(устройство);
10
11 .. .
12 }
```

Первоначальный вызов
уже корректен и

обрабатывает случай, когда размер равен `glfwGetFramebufferSize`
ждать было бы нечего.

Поздравляем, вы успешно завершили свою самую первую программу Vulkan pro-! В следующей главе мы собираемся избавиться от жестко закодированных вершин в вершинном шейдере и фактически использовать вершинный буфер. Код на C++ / Вершинный шейдер / Фрагментный шейдер

Буфера вершин

Описание входных данных вершин

Введение

В следующих нескольких главах мы собираемся заменить жестко закодированные данные вершин в вершинном шейдере на вершинный буфер в памяти. Мы начнем с самого простого подхода создания видимого буфера процессора и использования `memcpuy` чтобы скопировать данные вершины непосредственно в нее, и после этого мы увидим, как использовать промежуточный буфер для копирования данных вершины в высокопроизводительную память.

Вершинный шейдер

Сначала измените вершинный шейдер, чтобы он больше не включал данные вершин в сам код шейдера. Вершинный шейдер принимает входные данные из вершинного буфера, используя `в`

ключевое слово.

```
1 # версия 4 5 0
2
3 макет(location = 0) в vec 2 inPosition;
4
5 макет(location = 1) в vec 3 inColor;
6
7 макет(location = 0) вне vec 3 FragColor;
8
9 анулировать main() {
10
11     gl_Position = vec 4 (inPosition, 0 . 0 , 1 . 0 );
12     FragColor = цветной;
13 }
```

Переменные `Position` и `inColor` - это *атрибуты вершин*. Это свойства

, которые задаются для каждой вершины в буфере вершин, точно так же, как мы задали вручную положение и цвет для каждой вершины, используя два массива.

Обязательно перекомпилируйте вершинный шейдер! Точно так же, как

`FragColor, макет (location = x)` аннотации присваивают индексы входным данным, которые мы можем позже использовать для ссылок на них. Важно знать, что

некоторые типы, например `dvec 3` 64-битные векторы, используют несколько слотов. Это

означает, что индекс после него должен быть как минимум на 2 выше:

```
1 макет(location = 0) в позиции dvec 3;
2 макет (location = 2) в цвете vec 3;
```

Вы можете найти более подробную информацию о квалификаторе макета в OpenGL wiki.

Данные о вершинах

Мы перемещаем данные вершин из кода шейдера в массив в коде нашей программы. Начните с включения библиотеки GLM, которая предоставляет нам типы, связанные с линейной алгеброй, такие как векторы и матрицы. Мы собираемся использовать эти типы для

```
1 указания векторов положения и цвета. # включить <glm/glm.hpp>
```

Создайте новую структуру под названием `Vertex` с двумя атрибутами, которые мы собираемся использовать для использования в вершинном шейдере внутри него:

```
1 struct Vertex {
2     glm:: vec 2 pos;
3     glm:: vec 3 цветной;
4 };
```

GLM удобно предоставляет нам типы C++, которые в точности соответствуют векторным типам, используемым в языке шейдеров.

```
1 const std::vector<Vertex> вершины = {
2     {{ 0 . 0 f, - 0 . 5 f}, { 1 . 0 f, 0 . 0 f, 0 . 0 f}},
3     {{ 0 . 5 f, 0 . 5 f}, { 0 . 0 f, 1 . 0 f, 0 . 0 f}},
4     {{- 0 . 5 f, 0 . 5 f}, { 0 . 0 f, 0 . 0 f, 1 . 0 f}}
5 };
```

Теперь используйте `Vertex` structure, чтобы указать массив данных вершин. Мы используем точно такие же значения положения и цвета, как и раньше, но теперь они объединены в один массив вершин. Это известно как *членование* атрибуты вершин.

Описания привязки

Следующим шагом будет указание Vulkan, как передать этот формат данных вершинному шейдеру после того, как он будет загружен в память графического процессора.

Существует два типа структур, необходимых для передачи этой информации.

Первая структура - это `VkVertexInputBindingDescription` и мы добавим member-функцию в структуру `Vertex`, чтобы заполнить ее нужными данными.

```
1 struct Vertex {
2     glm::vec 2 pos;
```

```

3     glm:: цвет vec 3 ;
4
5     статический VkVertexInputBindingDescription getBindingDescription() {
6         VkVertexInputBindingDescription bindingDescription{};
7
8         Возврат Описание привязки;
9     }
10 };

```

Привязка к вершинам описывает, с какой скоростью загружать данные из памяти по вершинам. Он определяет количество байтов между записями данных и то, следует ли переходить к следующей записи данных после каждой вершины или

после каждого экземпляра.

```

1 bindingDescription{}; bindingDescription.binding = 0 ;
2 bindingDescription.stride = sizeof(Вершина);
3 Описание привязки.Входной параметр = VK_VERTEX_INPUT_RATE_VERTEX;
4

```

Все наши данные для каждой вершины упакованы в один массив, поэтому у нас будет только одна привязка. В привязка параметр задает индекс привязки в массиве привязок. Параметр stride задает количество байтов от одной записи к следующей, а параметр inputRate параметр может иметь одно из следующих значений:

- **VK_VERTEX_INPUT_RATE_VERTEX**: Переход к следующей записи данных после каждой вершины
 - **VK_VERTEX_INPUT_RATE_INSTANCE**: Переход к следующему вводу данных после каждого экземпляра
- Мы не собираемся использовать рендеринг экземпляра, поэтому будем придерживаться данных для каждой вершины.

Описания атрибутов

Вторая структура, описывающая, как обрабатывать ввод вершин, - это

`VkVertexInputAttributeDescription`

. Мы собираемся добавить еще одного помощника

функция для `Vertex` для заполнения этих структур.

```

1 # включить <массив>
2
3 ...
4
5     статический std::массив<VkVertexInputAttributeDescription,
6 >
7         getAttributeDescriptions() {
8             std::array<VkVertexInputAttributeDescription, 2 >
9
10            Атрибутивные описания();
11
12        Возврат Атрибутируемые описания;
13    }

```

Как указывает прототип функции, таких структур будет две.

Структура описания атрибута описывает, как извлечь атрибут вершины из фрагмента данных вершины, полученного из описания привязки. У нас есть два атрибута, `position` и `color`, поэтому нам нужны две структуры описания атрибутов.

```
1 Атрибутивные описания[ 0 ].привязка = 0 ;
2 Атрибутивные описания[ 0 ].местоположение = 0 ;
3
4 Атрибутивные описания[ 0 ].формат = VK_FORMAT_R3_2G3_2_SFLOAT;
5
6 Атрибутивные описания[ 0 ].смещение = offsetof(вершина_точка_доступа);
```

Параметр сообщает Vulkan, из какой привязки берутся данные для каждой вершины

.Параметр Расположение ссылается на Расположение Директива ввода

Следует отметить, что в большинстве случаев в ходе экспериментов не удается достичь полного синхронизма.

есть две 32-битные компоненты. Внешне это описывает тип данных для атрибута. Немного смущает то, что форматы задаются с использованием того же перечисления, что и цветовые форматы. Следующие типы шейдеров и форматы обычно используются вместе:

```
• float:VK_FORMAT_R3_2_SFLOAT  
• vec 2:VK_FORMAT_R3_2_G3_2_SFLOAT  
• vec 3:VK_FORMAT_R3_2_G3_2_B3_2_SFLOAT  
• vec 4:VK_FORMAT_R3_2_G3_2_B3_2_A3_2_SFLOAT
```

Как вы можете видеть, вам следует использовать формат, в котором количество

цветовых каналов соответствует количеству компонентов в типе данных шейдера. Разрешено использовать больше каналов, чем количество компонентов в шейдере, но они будут автоматически отброшены. Если количество каналов меньше, чем количество компонентов, то компоненты BGA будут использовать значения по умолчанию для типа цвета (`SFLOAT` `UINT` `SINT`) и разрядность также должны соответствовать типу выходных данных шейлера. Смотрите следующие примеры:

$$\begin{pmatrix} 0 & 0 & \dots & 1 \end{pmatrix}$$

- `ivec 2 :VK_FORMAT_R3_2_G3_2_SINT`, двухкомпонентный вектор с 32-разрядным знаком
- **целые числа** •
 - `uvec 4 :VK_FORMAT_R3_2_G3_2_B3_2_A3_2_UINT`, в 4-компонентный вектор из 32-бит целые числа без знака •
 - `x` – дум, говорит о 64-битах двойной точности (64 бит) подразумевает

В параметр `format` неявно определяет размер байта данных атрибута, а параметр `offset` указывает количество байтов, прошедших с начала данных для каждой вершины, из которых нужно считывать данные. Привязка загружает по одной вершине за раз и атрибут `position` (позиция) находится со смещением в 0 байт с начала этой структуры. Это вычисляется автоматически с использованием смещение макрос.

```
Атрибутивные описания[ 1 ].привязка = 0 ;
Атрибутивные описания[ 1 ].местоположение = 1 ;
Атрибутивные описания[ 1 ].формат = VK_FORMAT_R3_2_G3_2_B3_2_SFLOAT;
```

```
4 Атрибутивные описания[ 1 ].смещение = offsetof(вершина, цвет);
```

University of California, Berkeley Department of Spanish

Ввод вершины конвейера

Теперь нам нужно настроить графический конвейер для приема данных вершин в

этом формате, обратившись к структурам в `createGraphicsPipeline`. Найдите

`vertexInputInfo` создайте структуру и измените ее, чтобы ссылаться на два

```
1 описание: auto bindingDescription = Vertex::getBindingDescription();
2 auto attributeDescriptions = Вершина::получаем attributedescriptions();
3
4 vertexInputInfo.vertexBindingDescriptionCount = 1;
5 vertexInputInfo.vertexAttributeDescriptionCount =
6     static_cast<uint32_t>(Атрибутивные описания.size());
7 vertexInputInfo.pVertexBindingDescriptions = &bindingDescription;
vertexInputInfo.pVertexAttributeDescriptions =
    attributeDescriptions.data();
```

Теперь конвейер готов принимать данные о вершинах в формате `вершины` контейнер и передаем его нашему вершинному шейдеру. Если вы запустите программу сейчас с включенными уровнями проверки, вы увидите, что она жалуется на отсутствие вершинного буфера, привязанного к привязке. Следующий шаг - создать буфер вершин и переместить в него данные вершин, чтобы графический процессор мог получить к ним доступ. Код на C++ / Вершинный шейдер / Фрагментный шейдер

Создание вершинного буфера

Введение

Буфера в Vulkan - это области памяти, используемые для хранения произвольных данных, которые могут быть считаны видеокартой. Их можно использовать для хранения данных вершин, что мы и сделаем в этой главе, но их также можно использовать для многих других целей, которые мы рассмотрим в следующих главах. В отличие от объектов Vulkan, которыми мы имели дело до сих пор, буфера не выделяют память автоматически для самих себя. Работа из предыдущих глав показала, что Vulkan API позволяет программисту контролировать практически все, и управление памятью является одной из таких вещей.

Создание буфера

Создайте новую функцию `Создайте evertexbuffer` и вызовите его из `initVulkan` прямо перед `createCommandBuffers`.

```
1 пустота Инициализирующий вулкан() {
2     CreateInstance(); setupDebugMessenger();
3     createSurface();
4     pickPhysicalDevice();
5     createLogicalDevice();
6     CreateSwapChain();
```

```
8     createImageViews();
9     createRenderPass();
10    createGraphicsPipeline();
11    createFramebuffers();
12    createCommandPool();
13    createVertexBuffer();
14    createCommandBuffers();
15    createSyncObjects();
16 }
17
18 ...
19
20 void createVertexBuffer() {
21
22 }
```

Создание буфера требует от нас заполнения `VkBufferCreateInfo` структура.

```
1 VkBufferCreateInfo BufferInfo{};
2 BufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
3 BufferInfo.size = sizeof(вершины[ 0 ]) * vertices.size();
```

Первым полем структуры является `size`, которое определяет размер буфера в байтах.

Вычислить размер данных вершины в байтах несложно с помощью `.sizeof`

```
1 BufferInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
```

Второе поле - это `usage`, который указывает, для каких целей будут использоваться данные в буфере. Можно указать несколько целей с помощью побитового или. Нашим вариантом использования будет вершинный буфер, мы рассмотрим другие типы использования в следующих

```
1 главах. BufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

Точно также, как изображения в цепочке обмена, буфера также могут принадлежать определенному семейству очередей или использоваться совместно несколькими одновременно. Буфер будет использоваться только из графической очереди, поэтому мы можем придерживаться эксклюзивного доступа. Параметр `sharingMode` используется для настройки разреженной буферной памяти, которая не является актуально прямо сейчас. Мы оставим значение по умолчанию равным 0.

Теперь мы можем создать буфер с помощью `vkCreateBuffer`. Определите член класса, который будет удерживать дескриптор буфера и вызывать его `VertexBuffer`.

```
1 VkBuffer VertexBuffer;
2
3 ...
4
5 аннулирование createVertexBuffer() {
```

```

6     VkBufferCreateInfo BufferInfo{};
7     Информация о буфере.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
8     BufferInfo.size = sizeof(vertices[0]) * vertices.size();
9     BufferInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
10    BufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
11
12    if (vkCreateBuffer(устройство, &BufferInfo, nullptr, &VertexBuffer)
13        != VK_SUCCESS) {
14        выбросить std::runtime_error("не удалось создать буфер вершин!");
15    }

```

Буфер должен быть доступен для использования в командах рендеринга до конца программы, и это не зависит от цепочки подкачки, поэтому мы очистим его в оригинале очистка функция:

```

1 анулирование очистка() {
2     cleanupSwapChain();
3
4     vkDestroyBuffer(устройство, VertexBuffer, nullptr);
5
6     ...
7 }

```

Требования к памяти

Буфер создан, но на самом деле ему еще не назначено никакой памяти

. Первым шагом выделения памяти для буфера является запрос его

требований к памяти с использованием точно названной функции Требования к памяти `vkGetBufferMemoryRequirements`

```

1 Требования к памяти VkMemoryRequirements; Требования
2 к памяти vkGetBufferMemoryRequirements(устройство, VertexBuffer,
   &memRequirements);

```

Структура `VkMemoryRequirements` состоит из трех полей:

- размер: Размер требуемого объема памяти в байтах может отличаться

от `BufferInfo.size`.

- выравнивание: Смещение в байтах, с которого начинается буфер в выделенной области

объем памяти, зависит от `BufferInfo.usage` и `BufferInfo.flags`.

- `memoryTypeBits`: Битовое поле типов памяти, подходящих для буфера. Видеокарты

могут предлагать различные типы памяти для выделения. Каждый тип памяти отличается с точки зрения разрешенных операций и характеристик производительности. Нам необходимо объединить требования к буферу и нашему собственному приложению

требования для поиска правильного типа используемой памяти. Давайте создадим новую функцию `findMemoryType` для этой цели.

```
1 uint 3 2 _t  находит memorytype(uint 3 2 _t  фильтр типов, VkMemoryPropertyFlags 1
2   свойства) {
3 }
```

Сначала нам нужно запросить информацию о доступных типах памяти с помощью

```
vkGetPhysicalDeviceMemoryProperties.  
1 VkPhysicalDeviceMemoryProperties  memProperties;  
2 vkGetPhysicalDeviceMemoryProperties(физическое устройство и memProperties);
```

Структура состоит из двух массивов `VkPhysicalDeviceMemoryProperties`

Типы памяти и Кучи памяти. Кучи памяти - это отдельные ресурсы для восстановления памяти, такие как выделенная видеопамять и место подкачки в ОЗУ на случай, если VRAM закончится. В этих кучах существуют различные типы памяти. Прямо сейчас мы будем интересоваться только типом памяти, а не кучей, из которой она берется, но вы можете себе представить, что это может повлиять на производительность.

Давайте сначала найдем тип памяти, подходящий для самого буфера:

```
1 для (uint 3 2 _t  i = 0; i < memProperties.memoryTypeCount; i++) {
2   if (фильтр шрифтов & (1 << i)) {
3     Возврат я;
4   }
5 }
6
7 выбросить std::runtime_error("не удалось найти подходящий тип памяти!");
```

The фильтр шрифтов параметр будет использоваться для указания битового поля подходящих типов памяти. Это означает, что мы можем найти индекс подходящего типа памяти, просто перебрав их и проверив, установлен ли соответствующий бит в.

Однако нас интересует не только тип памяти, подходящий для вершинного буфера. Нам также необходимо иметь возможность записывать данные наших вершин в эту память. Структура состоит из кучи и свойства каждого типа памяти. Свойства определяют специальные функции памяти, такие как возможность отображать ее, чтобы мы могли выполнять запись в нее из центрального процессора. Это свойство обозначается символом `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT`, но нам также нужно использовать `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` свойство. Мы увидим почему, когда сопоставим память.

Теперь мы можем изменить цикл, чтобы также проверить поддержку этого свойства:

```
1 для (uint 3 2 _t  i = 0; i < memProperties.memoryTypeCount; i++) {
```

```

2     if ((Фильтр шрифтов & (1 << i)) &&
3         (memProperties.memoryTypes[i].propertyFlags & properties)
4             == свойства) {
5     Возврат я;
}

```

У нас может быть более одного желаемого свойства, поэтому мы должны проверить, является ли результат побитового И не просто ненулевым, но и равным желаемому свойству битового поля. Если для буфера есть подходящий тип памяти, который также обладает всеми необходимыми нам свойствами, то мы возвращаем его индекс, в противном случае мы создаем исключение.

Выделение памяти

Теперь у нас есть способ определить правильный тип памяти, так что мы действительно можем выделить память, заполнив структуру `VkMemoryAllocateInfo`.

```

1 VkMemoryAllocateInfo allocInfo{};
2 allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
3 allocInfo.allocationSize = memRequirements.size;
4 allocInfo.memoryTypeIndex =
    findMemoryType(memRequirements.memoryTypeBits,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
    VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);

```

Выделение памяти теперь так же просто, как указание размера и типа, оба из которых являются производными от требований к памяти вершинного буфера и желаемого свойства. Создайте член класса для сохранения дескриптора в памяти и выделите его с помощью `vkAllocateMemory`.

```

1 VkBuffer VertexBuffer;
2 VkDeviceMemory vertexBufferMemory;
3
4 ...
5
6 if (vkAllocateMemory(устройство, &allocInfo, nullptr,
7     &vertexBufferMemory) != VK_SUCCESS) { выбросить std::runtime_error("не
8 удалось выделить буфер вершин память!"); }

```

Если выделение памяти прошло успешно, то теперь мы можем связать эту память с буфером, используя `vkBindBufferMemory`:

```
1 vkBindBufferMemory(устройство, VertexBuffer, vertexBufferMemory, 0);
```

Первые три параметра не требуют пояснений, а четвертый параметр - это смещение в пределах области памяти. Поскольку эта память выделена специально.

для этого вершинный буфер, смещение которого просто `0`. Если смещение ненулевое, то оно должно быть кратным на `memRequirements.alignment`. выравнивание. Конечно, точно так же, как динамическое выделение памяти в C++, память должна быть освобождена в какой-то момент. Память, привязанная к объекту `buffer`, может быть освобождена, как только буфер больше не используется,

```
поэтому давайте освободим его после уничтожения буфера:void cleanup() {  
    cleanupSwapChain();  
  
    vkDestroyBuffer(устройство, VertexBuffer, nullptr);  
    vkFreeMemory(устройство, vertexBufferMemory, nullptr);  
}
```

Заполнение буфера вершин

Теперь пришло время скопировать данные вершин в буфер. Это делается путем отображения буферной памяти в доступную для процессора память с помощью `vkMapMemory`.

```
аннулирование * данных;  
vkMapMemory(устройство, vertexBufferMemory, 0, BufferInfo.size, 0,  
иданные);
```

Эта функция позволяет нам получить доступ к области указанного ресурса памяти, определяемой смещением и размером. Смещение и размер здесь следующие `0` и `BufferInfo.size`, соответственно. Также возможно указать специальное значение `VK_WHOLE_SIZE`. Для сопоставьте всю память. Предпоследний параметр можно использовать для указания флагов, но ни один из них пока не доступен в текущем API. Для него необходимо установить значение `0`. Последний параметр задает вывод для указателя на отображенную память.

```
void * данные;  
vkMapMemory(устройство, vertexBufferMemory, 0, BufferInfo.size, 0,  
&data);  
memcpy(данные, вершины.data(), (size_t) BufferInfo.size);  
vkUnmapMemory(устройство, vertexBufferMemory);
```

Теперь вы можете просто `memcpy` данные вершин переносятся в сопоставленную память и разархивируются повторить это снова, используя `vkUnmapMemory`. К сожалению, водитель может не сразу скопировать данные в буфер памяти, например из-за кэширования. Также возможно, что записи в буфер еще не видны в отображенной памяти. Есть два способа решить эту проблему:

- Используйте связную с хостом кучу памяти, обозначенную символом `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`
 - Вызовите `vkFlushMappedMemoryRanges` после записи в отображенную память, и вызовите `vkInvalidateMappedMemoryRanges` перед чтением из отображенной памяти
- Мы выбрали первый подход, который гарантирует, что отображенная память всегда соответствует содержимому выделенной памяти. Имейте в виду, что это может

приводит к несколько худшей производительности, чем явная очистка, но мы увидим, почему это не имеет значения в следующей главе.

Очистка диапазонов памяти или использование когерентной кучи памяти означает, что драйвер будет знать о наших записях в буфер, но это не означает, что они пока действительно видны на графическом процессоре. Передача данных на графический процессор - это операция, которая происходит в фоновом режиме, и спецификация просто сообщает нам, что она гарантированно будет завершена при следующем вызове.

vkQueueSubmit

Привязка вершинного буфера

Все, что теперь остается, - это привязка вершинного буфера во время операций рендеринга.

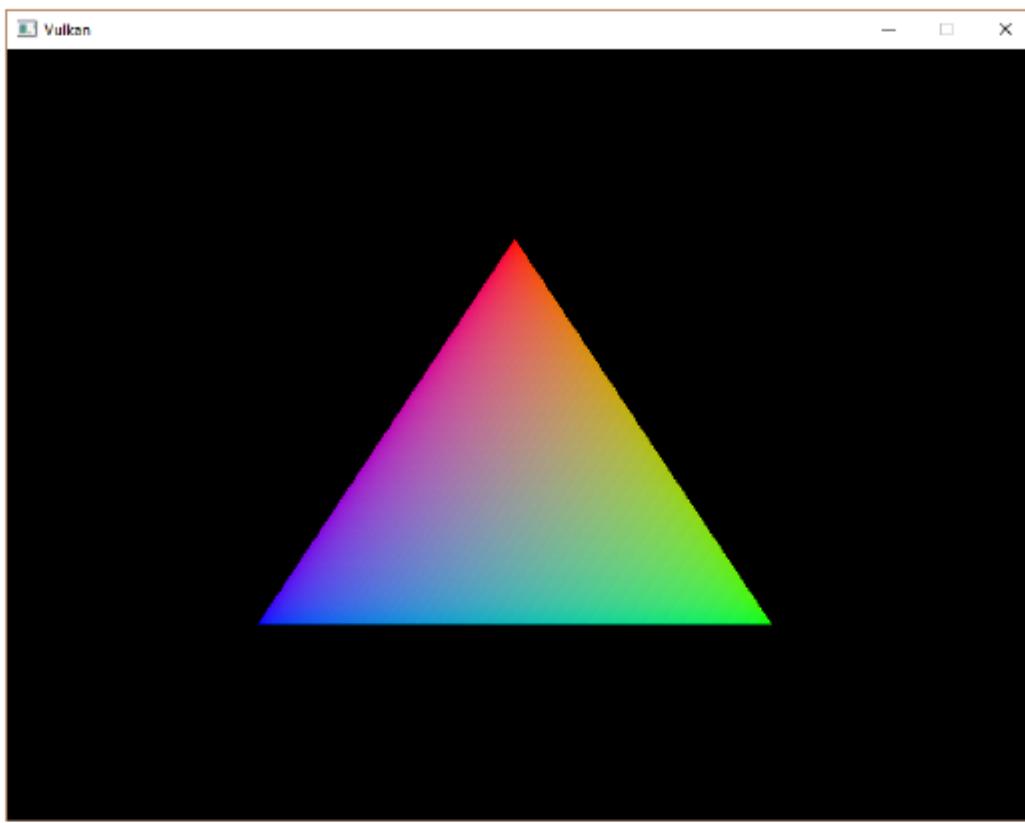
Мы собираемся расширить `recordCommandBuffer` функция для этого.

```
1 vkCmdBindPipeline(командный буфер, VK_PIPELINE_BIND_POINT_GRAPHICS,  
                    Графический конвейер);  
  
2  
3 VkBuffer vertexBuffers[] = {VertexBuffer};  
4 Смещения VkDeviceSize[] = {0};  
5 vkCmdBindVertexBuffers(командный буфер, 0, 1, vertexBuffers, смещения);  
6  
7 vkCmdDraw(командный буфер), static_cast<uint 3 2_t>(vertices.size()), 1,  
           0, 0);
```

В функции `vkCmdBindVertexBuffers` используется для привязки буферов вершин к элементам привязки, подобным той, которую мы настроили в предыдущей главе. Первые два параметра, помимо командного буфера, определяют смещение и количество привязок, для которых мы собираемся указать буферы вершин. Последние два параметра определяют массив буферов вершин для привязки и смещения байтов для начала чтения данных вершин. Вам также следует изменить вызов на `vkCmdDraw` передать количество вершин в буфере вместо жестко заданного числа.

3

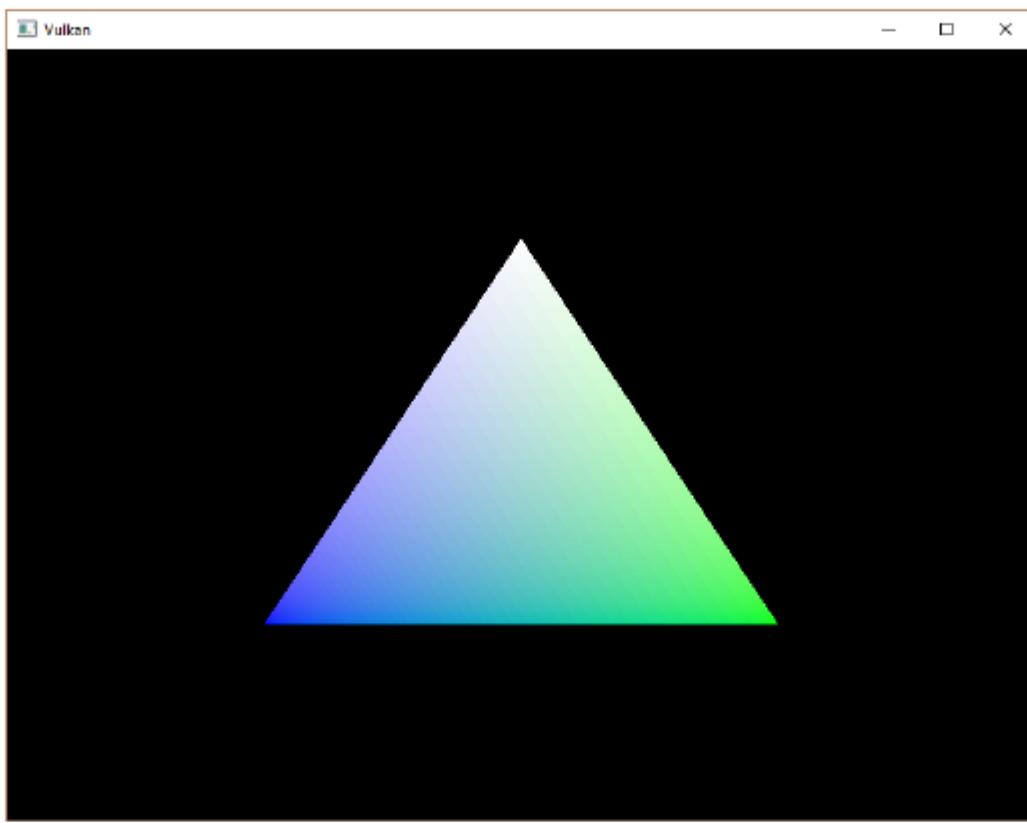
Теперь запустите программу, и вы снова увидите знакомый треугольник.:



Попробуйте изменить цвет верхней вершины на белый, изменив команду `вершины` массив:

```
1 const std::vector<Вершина> вершины = {  
2     {{ 0.0f, -0.5f}, {1.0f, 1.0f, 1.0f}},  
3     {{ 0.5f, 0.5f}, {0.0f, 1.0f, 0.0f}},  
4     {{-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}}  
5 };
```

Запустите программу еще раз, и вы должны увидеть следующее:



В следующей главе мы рассмотрим другой способ копирования вершинных данных в вершинный буфер, который повышает производительность, но требует дополнительной работы. Код на C++ / Вершинный шейдер / Фрагментный шейдер

Промежуточный буфер

Введение

Вершинный буфер, который у нас есть прямо сейчас, работает корректно, но тип памяти, который позволяет нам обращаться к нему из центрального процессора, может быть не самым оптимальным типом памяти для чтения с самой видеокарты. Наиболее оптимальной памятью является `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` флаг и обычно недоступен через центральный процессор на специализированных видеокартах. В этой главе мы собираемся создать два вершинных буфера. Один промежуточный буфер в доступной памяти процессора для загрузки данных из массива вершин в, и в конечном буфере вершин в локальной памяти устройства. Затем мы будем использовать команду копирования буфера для перемещения данных из промежуточного буфера в фактический буфер вершин.

Очередь передачи

Буфер скопировать команду требуется очередь семья, которая поддерживает передачу операций, который указывается через `VK_QUEUE_TRANSFER_BIT`. Хорошая новость заключается в том, что

любое семейство очередей с `VK_QUEUE_GRAPHICS_BIT` или `VK_QUEUE_COMPUTE_BIT` возможны
са- уже неявно поддерживаются `VK_QUEUE_TRANSFER_BIT` операции. В
реализации не требуется явно указывать это в этих случаях.

queueFlags

Если вам нравится сложная задача, вы все равно можете попробовать использовать другое
семейство очередей, специально для операций переноса. Для этого вам потребуется внести
следующие изменения в вашу программу:

- Измените `QueueFamilyIndices` и `queuefamilies` для явного поиска
 - для семейства очередей с помощью `VK_QUEUE_TRANSFER_BIT` бит, но не `VK_QUEUE_GRAPHICS_BIT`.
 - Изменить Создать логическое устройство чтобы запросить дескриптор очереди передачи
 - Создайте второй пул команд для отправляемых командных буферов в семействе очередей передачи
 - Измените `sharingMode` требуемых ресурсов `VK_SHARING_MODE_CONCURRENT` и укажите как графические объекты, так и семейства очередей передачи
 - Отправляйте любые команды передачи, такие как `vkCmdCopyBuffer` - буфер обмена (который мы будем использовать в этой главе) в очередь передачи вместо графической очереди
- Это небольшая работа, но она многому научит вас о том, как распределяются ресурсы между семействами очередей.

Абстрагирование от создания буфера

Поскольку в этой главе мы собираемся создать несколько буферов, было бы неплохо перенести создание буфера во вспомогательную функцию. Создайте новую функцию `createBuffer` и переместите код в `createVertexBuffer` (кроме отображения) в него.

```
1 анулировать createBuffer(размер VkDeviceSize, использование VkBufferUsageFlags,
2
3     vkmemorypropertyFlagает свойства, VkBuffer& buffer,
4     VkDeviceMemory& bufferMemory) {
5
6     VkBufferCreateInfo BufferInfo{};
7
8     BufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
9     BufferInfo.size = размер;
10    BufferInfo.usage = использование;
11    BufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
12
13    если (vkCreateBuffer(device, &BufferInfo, nullptr, &buffer) != VK_SUCCESS) {
14        выбросить std::runtime_error("не удалось создать буфер!");
15    }
16
17    Требования к памяти VkMemoryRequirements;
18    Требования к памяти vkGetBufferMemoryRequirements (требования к устройству, буферу и памяти);
19
20    VkMemoryAllocateInfo allocInfo{};
21
22    allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
23    allocInfo.allocationSize = memRequirements.size;
```

```

18     allocInfo.memoryTypeIndex =
19         findMemoryType(memRequirements.memoryTypeBits, свойства);
20
21     если (vkAllocateMemory(устройство, &allocInfo, nullptr, &bufferMemory)
22         != VK_SUCCESS) {
23         выбросить std::runtime_error("не удалось выделить буфер
24             памяти!");
25     }
26
27     vkBindBufferMemory(устройство, буфер, bufferMemory, 0);
28 }
```

Обязательно добавьте параметры для размера буфера, свойств и использования памяти, чтобы мы могли использовать эту функцию для создания множества различных типов буферов. Последние два параметра являются выходными переменными для записи дескрипторов. Теперь вы можете удалить код создания буфера и выделения

памяти из `createVertexBuffer` и просто вызовите `createBuffer` вместо этого:

```

1 void createVertexBuffer() {
2
3     VkDeviceSize bufferSize = размер буфера(vertices[0]) * vertices.size();
4
5     createBuffer(размер буфера, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
6
6         VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
7
7         VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, VertexBuffer,
8
8         vertexBufferMemory);
9 }
```

Запустите вашу программу, чтобы убедиться, что буфер вершин по-прежнему работает должным образом.

Использование промежуточного буфера

Теперь мы собираемся изменить `createVertexBuffer` использовать только видимый буфер хоста в качестве временного буфера и использовать локальный буфер

```

1 устройство в качестве фактического буфера вершин. void createVertexBuffer() {
2
3     VkDeviceSize bufferSize = размер буфера(вершины[0]) * vertices.size();
4
5     VkBuffer stagingBuffer;
6     VkDeviceMemory stagingBufferMemory;
7
8     createBuffer(размер буфера, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
9
9         VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
10
10        VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
11
11        промежуточный буфер, промежуточный буфер памяти);
```

```

7
8     пустота *   данные;
9     vkMapMemory(устройство,  stagingBufferMemory,  0 ,  размер  буфера,  0 ,
10    &data);  memcpy(данные,  vertices.data(),
11    (size_t)  размер  буфера);  vkUnmapMemory(устройство,
12    stagingBufferMemory);  createBuffer(размер
13    буфера,  VK_BUFFER_USAGE_TRANSFER_DST_BIT |
14    VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
15    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
16    VertexBuffer,  память  vertexBufferMemory);
17
18 }

```

Теперь мы используем новый `stagingBuffer` с `stagingBufferMemory` для отображения и копирования данных вершин. В этой главе мы собираемся использовать два новых флага использования буфера:

- `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` : Буфер может использоваться как исходный в а операция по передаче памяти.
- `VK_BUFFER_USAGE_TRANSFER_DST_BIT` : Буфер может быть использован в качестве пункта назначения в операции передачи памяти.

`B` `VertexBuffer` теперь выделяется из памяти локального устройства, что обычно означает, что мы не можем использовать . Однако мы `vkMapMemory` может копировать данные из `stagingBuffer` в `VertexBuffer`. Мы должны указать, что мы намерены это сделать, указав флаг источника передачи для `stagingBuffer` и флаг назначения передачи для `VertexBuffer`, вместе с флагом использования вершинного буфера.

Теперь мы собираемся написать функцию для копирования содержимого из одного буфера в другой, которая называется `copyBuffer`.

```

аннулирование  Буфер  копирования  (VkBuffer  srcBuffer,  VkBuffer  dstBuffer,  VkDeviceSize  1
размер)  {
2
3 }

```

Операции переноса памяти выполняются с использованием буферов команд, точно так же, как команды рисования. Поэтому сначала мы должны выделить временный буфер команд. Возможно, вы захотите создать отдельный пул команд для такого рода недолговечных буферов, поскольку реализация может иметь возможность применять оптимизации выделения памяти . Вам следует использовать `VK_COMMAND_POOL_CREATE_TRANSIENT_BIT` в этом случае установите флагок во время генерации пула команд.

```

1 void  Буфер  копирования  (VkBuffer  srcBuffer,  VkBuffer  dstBuffer,  VkDeviceSize
размер)  {  VkCommandBufferAllocateInfo
2 allocInfo{};
3 allocInfo.sType  =  VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;

```

```
4     allocInfo.level    = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
5     allocInfo.commandPool = commandPool;
6     allocInfo.commandBufferCount = 1;
7
8     VkCommandBuffer Командный буфер;
9     vkAllocateCommandBuffers(устройство, & allocInfo, &commandBuffer);
10 }
```

И сразу же начинайте записывать команду в буфер:

```
1 VkCommandBufferBeginInfo beginInfo{};
2 beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
3 beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
4
5 vkBeginCommandBuffer(командный буфер и beginInfo);
```

Мы собираемся использовать командный буфер только один раз и подождать с повторным выключением функции, пока операция копирования не завершит выполнение. Хорошей практикой является сообщать водителю о наших намерениях, используя

```
VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT .
1 VkBufferCopy copyRegion{};
2 copyRegion.srcOffset = 0; // Необязательно
3 copyRegion.dstOffset = 0; // Необязательно
4 copyRegion.size = размер; vkCmdCopyBuffer(commandBuffer,
5 srcBuffer, dstBuffer, 1 исcopyRegion);
```

Содержимое буферов передается с помощью `vkCmdCopyBuffer` команда. Она принимает исходный и конечный буферы в качестве аргументов и массив регионов для копирования. Регионы определены в `VkBufferCopy` структурирует и состоит из исходного смещения буфера, смещения буфера назначения и размера. Указать это

невозможно `VK_WHOLE_SIZE` здесь, в отличие от `vkMapMemory` команда.

`vkEndCommandBuffer(командный буфер);`
Этот командный буфер содержит только команду копирования, поэтому мы можем остановить запись сразу после этого. Теперь выполните командный буфер, чтобы

```
1 завершить передачу:VkSubmitInfo submitInfo{};
2 submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
3 submitInfo.commandBufferCount = 1;
4 submitInfo.pCommandBuffers = &commandBuffer;
5 vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
7 vkQueueWaitIdle(graphicsQueue);
```

В отличие от команд рисования, на этот раз нет событий, которых нам нужно ждать. Мы просто хотим немедленно выполнить перенос в буферах. Есть

опять же, есть два возможных способа дождаться завершения этой передачи.

Мы могли бы использовать забор и ждать с помощью `vkWaitForFences`, или
`vkQueueWaitIdle` просто подождать, пока очередь на передачу простоявает с помощью .

Ограничение позволило бы вам планировать несколько передач одновременно и ждать, пока все они завершатся, вместо выполнения по одной за раз. Это может дать драйверу больше возможностей для оптимизации.

```
1 vkFreeCommandBuffers(устройство, commandPool, 1 и commandBuffer);
```

Не забудьте очистить буфер команд, используемый для операции переноса.

Теперь мы можем вызвать `copyBuffer` из `createVertexBuffer` функция для перемещения данных вершины в локальный буфер устройства:

```
1 createBuffer(размер буфера, VK_BUFFER_USAGE_TRANSFER_DST_BIT |  
2 VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,  
3 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, VertexBuffer,  
4 vertexBufferMemory); copyBuffer(промежуточный  
5 буфер, VertexBuffer, размер буфера);  
6  
7 }
```

После копирования данных из промежуточного буфера в буфер устройства мы должны очистить их:

```
1 ...  
2  
3     Буфер копирования(stagingBuffer, VertexBuffer, bufferSize);  
4  
5     vkDestroyBuffer(устройство, stagingBuffer, nullptr);  
6     vkFreeMemory(устройство, stagingBufferMemory, nullptr);  
7 }
```

Запустите свою программу, чтобы убедиться, что вы снова видите знакомый треугольник. Улучшение может быть незаметно прямо сейчас, но данные о его вершинах теперь загружаются из высокопроизводительной памяти. Это будет важно, когда мы начнем визуализировать более сложную геометрию.

Заключение

Следует отметить, что в реальном приложении вы не должны вызывать `vkAllocateMemory` для каждого отдельного буфера.

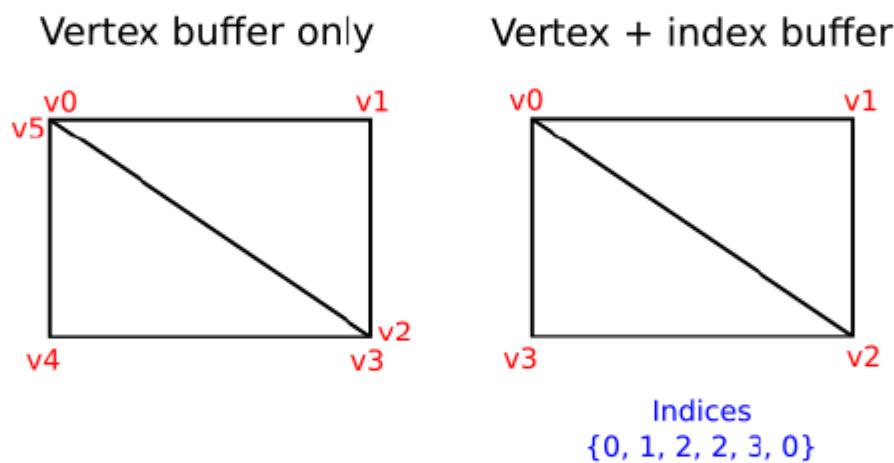
Максимальное количество одновременных выделений памяти ограничено `maxMemoryAllocationCount` ограничение на физическое устройство, которое может быть таким же низким, как 4 0 9 6 даже на высококлассном оборудовании, таком как NVIDIA GTX 1080. Правильный способ выделить память для большого количества объектов одновременно - создать пользовательский распределитель, который распределяет одно выделение между множеством различных объектов с помощью смещение параметры, которые мы видели во многих функциях.

Вы можете либо реализовать такой распределитель самостоятельно, либо использовать VulkanMem- Библиотека огуAllocator, предоставленная инициативой GPUOpen. Однако в этом руководстве можно использовать отдельное выделение для каждого ресурса, потому что мы пока не приблизимся ни к одному из этих ограничений. Код на C ++ / Вершинный шейдер / Фрагментный шейдер

Индексный буфер

Введение

3D-сетки, которые вы будете отрисовывать в реальном приложении, часто будут разделять вершины между несколькими треугольниками. Это уже происходит даже с чем-то простым, например, с рисованием прямоугольника:



Для рисования прямоугольника требуется два треугольника, что означает, что нам нужен вершинный буфер с 6 вершинами. Проблема в том, что данные двух вершин должны быть продублированы, что приводит к 50% избыточности. Ситуация становится только хуже с более сложными сетками, где вершины повторно используются в среднем в количестве 3 треугольников. Решением этой проблемы является использование *индексного буфера*. Индексный буфер - это, по сути, массив указателей на буфер вершин. Это позволяет изменять порядок данных вершин и повторно использовать существующие данные для нескольких вершин. На рисунке выше показано, как будет выглядеть индексный буфер для прямоугольника, если у нас есть вершинный буфер, содержащий каждую из четырех уникальных вершин. Первые три индекса определяют верхний правый треугольник, а последние три индекса определяют вершины нижнего левого треугольника.

Создание индексного буфера

В этой главе мы собираемся изменить данные вершины и добавить данные индекса, чтобы нарисовать прямоугольник, подобный тому, что изображен на рисунке. Измените данные вершины, чтобы они представляли четыре угла:

```
1 const std::vector<Вершина> вершины = {  
2     {{-0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}},  
3     {{0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}},  
4     {{0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}},  
5     {{-0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}}  
6};
```

Верхний левый угол - красный, верхний правый - зеленый, нижний правый - синий, а нижний левый - белый. Мы добавим новый массив `индексы` для представления содержимого индексного буфера. Он должен соответствовать индексам на рисунке, чтобы нарисовать верхний правый треугольник и

```
1 нижний левый треугольник. const std::vector<uint16_t> = {  
2     0, 1, 2, 2, 3, 0  
3};
```

Можно использовать либо `uint16_t` или `uint32_t` для вашего индексного буфера в зависимости от количества записей в `for now`, потому что мы используем менее 65535 уникальных вершин.

Как и данные о вершинах, индексы необходимо загрузить в `VkBuffer` для доступа к ним графического процессора. Определите два новых члена класса для хранения ресурсов индексного буфера:

```
VkBuffer vertexBuffer;  
1 VkDeviceMemory vertexBufferMemory;  
2  
VkBuffer indexBuffer;  
3 VkDeviceMemory indexBufferMemory;
```

Функция `createIndexBuffer`, которую мы добавим сейчас, почти идентична

```
createVertexBuffer  
  
void Инициализирующий вулкан() {  
1  
2     ...  
3     createVertexBuffer();  
4     createIndexBuffer();  
5     ...  
6 }  
7  
8 void createIndexBuffer() {  
9     VkDeviceSize bufferSize = sizeof(индексы[0]) * indexes.size();  
10  
11     VkBuffer поэтапный буфер;
```

```

12     VkDeviceMemory stagingBufferMemory;
13
14     createBuffer(размер буфера, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
15
16         VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
17
18         VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer,
19
20         stagingBufferMemory);
21
22     анулирование * данных; vkMapMemory(устройство,
23
24         stagingBufferMemory, 0, размер буфера, 0,
25
26         &data);
27
28     memcpy(данные, индексы.data(), (size_t) размер буфера);
29
30     vkUnmapMemory(устройство, stagingBufferMemory);
31
32
33     createBuffer(размер буфера, VK_BUFFER_USAGE_TRANSFER_DST_BIT |
34
35         VK_BUFFER_USAGE_INDEX_BUFFER_BIT,
36
37         VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, indexBuffer,
38
39         indexBufferMemory); copyBuffer(stagingBuffer,
40
41         indexBuffer, bufferSize);
42
43     vkDestroyBuffer(устройство, stagingBuffer, nullptr); vkFreeMemory(устройство,
44
45         stagingBufferMemory, nullptr);
46 }
```

Есть два заметных отличия. В параметр bufferSize теперь равно количеству индексов, умноженному на размер типа индекса, либо `uint 1 6 _t` или `uint 3 2 _t`. Использование `indexBuffer` должно быть `VK_BUFFER_USAGE_INDEX_BUFFER_BIT` вместо `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT`, что имеет смысл. В остальном процесс точно такой же. Мы создаем промежуточный буфер для копирования содержимого индексов в, а затем скопируйте его в локальный индексный буфер конечного устройства. Индексный буфер следует очистить в конце программы, точно так же, как вершинный буфер:

```

1 void очистка() {
2
3     cleanupSwapChain();
4
5     vkDestroyBuffer(устройство, indexBuffer, nullptr);
6     vkFreeMemory(устройство, indexBufferMemory, nullptr);
7
8     vkDestroyBuffer(устройство, VertexBuffer, nullptr);
9     vkFreeMemory(устройство, vertexBufferMemory, nullptr);
10
11 }
```

Использование индексного буфера

Использование индексного буфера для рисования требует двух изменений в

recordCommandBuffer

Сначала нам нужно привязать индексный буфер, точно так же, как мы сделали для вершинного буфера. Разница в том, что у вас может быть только один индексный буфер. К сожалению, невозможно использовать разные индексы для каждого атрибута вершины, поэтому нам все равно приходится полностью дублировать данные вершины, даже если меняется только один

```
1   атрибут.vkCmdBindVertexBuffers(commandBuffer, 0, 1, vertexBuffers, смещения);  
2  
3   vkCmdBindIndexBuffer(commandBuffer, indexBuffer, 0,  
4     VK_INDEX_TYPE_UINT 1 6 );
```

Буфер индекса связан с

vkCmdBindIndexBuffer

который имеет индекс

буфер, смещение в нем на байт и тип индексных данных в качестве параметров.

Как упоминалось ранее, возможными типами являются `VK_INDEX_TYPE_UINT 1 6` и

`VK_INDEX_TYPE_UINT 3 2`. Простая привязка

индексного буфера пока ничего не меняет, нам также нужно изменить команду

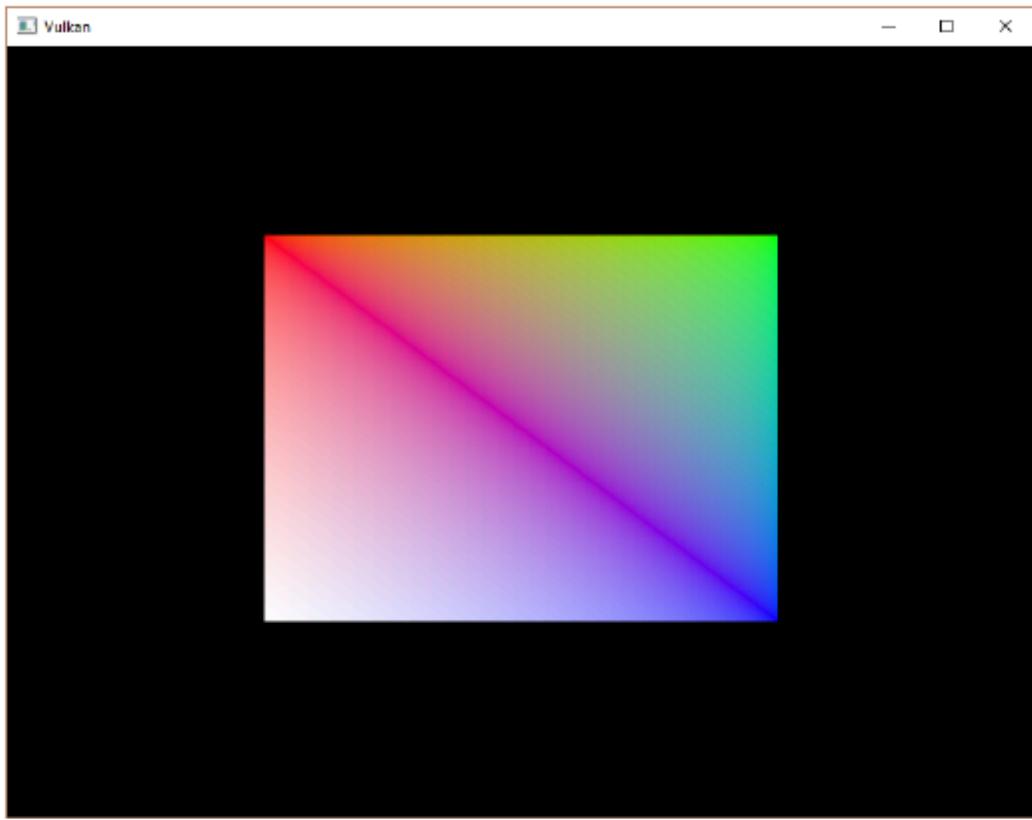
рисования, чтобы указать Vulkan использовать индексный буфер. Удалите

`vkCmdDraw` постройте и замените его на `vkCmdDrawIndexed`:

```
1   vkCmdDrawIndexed(командный буфер,  
2     static_cast ( статическая передача)  
3     <uint 3 2 _t>(индексы.размер()), 1, 0, 0, 0 );
```

Вызов этой функции очень похож на `vkCmdDraw`. Первые два параметра определяют количество индексов и количество экземпляров. Мы не используем создание экземпляров, поэтому просто укажите `1` пример. Количество индексов представляет собой количество вершин, которые будут переданы вершинному шейдеру. Следующий параметр задает смещение в индексный буфер, используя значение видеокарта для начала чтения со второго индекса. Предпоследний параметр задает смещение для добавления к индексам в индексном буфере. Последний параметр задает смещение для создания экземпляра, которое мы не используем. Теперь запустите вашу программу, и вы должны увидеть следующее:

`1` приведет к



Теперь вы знаете, как сэкономить память, повторно используя вершины с индексными буферами. Это станет особенно важным в следующей главе, где мы собираемся загружать сложные 3D-модели.

В предыдущей главе уже упоминалось, что вы должны выделить несколько повторных источников, таких как буфера, из одного выделения памяти, но на самом деле вам следует пойти на шаг дальше. Разработчики драйверов рекомендуют также хранить несколько буферов, таких как вершинный и индексный буферы, в единый и используйте смещения в `VkBuffer` (буфер обмена) таких командах, как `vkCmdBindVertexBuffers`. Преимущество заключается в том, что ваши данные в этом случае более удобны для кэширования, потому что они расположены ближе друг к другу. Даже возможно повторно использовать один и тот же фрагмент памяти для нескольких ресурсов, если они не используются во время одних и тех же операций рендеринга, при условии, конечно, что их данные обновляются. Это известно как *наложение псевдонимов* и некоторые функции Vulkan имеют явные флаги, указывающие, что вы хотите это сделать.

Код на C++ / Вершинный шейдер / Фрагментный шейдер

Унифицированные буферы

Схема описания и буфер

Введение

Теперь мы можем передавать произвольные атрибуты вершинному шейдеру для каждой версии- tex, но как насчет глобальных переменных? Начиная с этой главы, мы собираемся перейти к 3D-графике , для чего требуется матрица модель-вид-проекция. Мы могли бы включить это как данные о вершинах, но это пустая тратя памяти, и это повторно потребовало бы от нас обновлять буфер вершин всякий раз, когда изменяется преобразование. Преобразование может легко изменить каждый отдельный кадр. Правильный способ решить эту проблему в Vulkan - использовать *дескрипторы ресурсов*. Дескриптор это способ для шейдеров свободно обращаться к ресурсам, таким как буферы и изображения. Мы собираемся настроить буфер, содержащий матрицы преобразования, и предоставить вершинному шейдеру доступ к ним через дескриптор. Использование дескрипторов состоит из трех частей:

- Укажите макет дескриптора во время создания конвейера
- Выделите набор дескрипторов из пула дескрипторов
- Привяжите набор дескрипторов во время визуализации

В *Макет дескриптора* определяет типы ресурсов, которые будут обрабатываться конвейером, точно так же, как проход рендеринга определяет типы вложений, к которым будет осуществляться доступ. А *набор дескрипторов* определяет фактический буфер или повторные источники изображения, которые будут привязаны к дескрипторам, точно так же, как фреймбуфер определяет фактические виды изображений для привязки к вложениям передачи рендеринга. Затем набор дескрипторов привязывается к командам рисования точно так же, как буферы вершин и фреймбуферов. Существует много типов дескрипторов, но в этой главе мы будем работать с единообразными буферными объектами (UBO). Мы рассмотрим другие типы дескрипторов в следующих главах, но базовый процесс тот же. Допустим, у нас есть данные, которые мы хотим, чтобы вершинный шейдер

```
1    имел в структуре C, подобной этой: struct UniformBufferObject {  
2        glm:: модель mat 4 ;
```

```
3     glm:: mat 4   view;
4     glm::mat 4   proj;
5 };
```

Затем мы можем скопировать данные в а `VkBuffer` и получить к ним доступ через единый буфер дескриптора объекта из вершинного шейдера следующим образом:

```
1 макет(привязка = 0 )  единообразный UniformBufferObject {
2 модель mat 4 ;
3 представление
4 mat 4 ; mat 4
5 proj; } ubo;
6
7 аннулирует main() {
8     gl_Position = ubo.proj * ubo.просмотр * ubo.модель * vec 4 (в положении,
9         0 . 0 , 1 . 0 );
10    FragColor = нецветный;
11 }
```

Мы собираемся обновлять матрицы модели, вида и проекции в каждом кадре, чтобы заставить прямоугольник из предыдущей главы вращаться в 3D.

Вершинный шейдер

Измените вершинный шейдер, чтобы включить объект uniform buffer, как это было указано выше. Я предполагаю, что вы знакомы с преобразованиями MVP.

Если вы нет, обратитесь к ресурсу, упомянутому в первой главе.

```
1 # версия 4 5 0
2 макет (привязка = 0 )  uniform UniformBufferObject {
3
4     mat 4   модель;
5     вид mat 4 ;
6     mat 4   проект;
7 } ubo;
8
9 макет (location = 0 )  в позиции vec 2 ;
10 макет (location = 1 )  в цвете vec 3 ;
11
12 макет (location = 0 )  из vec 3   FragColor;
13
14 аннулирует main() {
15     gl_Position = ubo.проект * ubo.вид * ubo.модель * vec 4 (исходное положение,
16         0 . 0 , 1 . 0 );
17    FragColor = нецветный;
18 }
```

Обратите внимание, что порядок `uniform`, `in` и `out` объявления не имеют значения. Директива `binding` аналогична директиве `расположение` директива для атрибутов. Мы собираемся ссылаться на эту привязку в макете дескриптора. Стока с изменена, чтобы использовать преобразования для вычисления конечной позиции в координатах клипа. В отличие от 2D-треугольников, последний компонент координат клипа может отсутствовать, что приведет к разделению при преобразовании в конечное нормализованное значение координаты устройства на экране. Это используется в перспективной проекции как *разделение перспективы* и важно для того, чтобы близкие объекты выглядели крупнее, чем объекты, находящиеся дальше.

Схема набора дескрипторов

Следующий шаг - определить UBO на стороне C++ и сообщить Vulkan об этом дескрипторе в вершинном шейдере.

```
1 struct UniformBufferObject {
2     glm:: модель mat 4 ;
3     glm:: вид mat 4 ;
4     glm:: mat 4 proj;
5 };
```

Мы можем точно соответствовать определению в шейдере, используя типы данных в GLM. Данные в матрицах двоично совместимы с тем, что ожидает шейдер, поэтому позже мы можем просто `memcpy` а объект `UniformBufferObject` для `a VkBuffer`. Нам нужно предоставить подробную информацию о каждой привязке дескриптора, используемой в шейдерах для создания конвейера, точно так же, как мы должны были сделать для каждого атрибута вершины и его `расположение` Указатель. Мы настроим новую функцию для определения всей этой информации под названием `createDescriptorSetLayout`. Её следует вызывать непосредственно перед созданием конвейера, потому что он нам там понадобится.

```
1 void initVulkan() {
2     ...
3     createDescriptorSetLayout();
4     createGraphicsPipeline();
5     ...
6 }
7 ...
8 ...
9
10 пустота createDescriptorSetLayout() {
11
12 }
```

Каждая привязка должна быть описана с помощью `VkDescriptorSetLayoutBinding` `struct`.

```
1 void createDescriptorSetLayout() {
```

```
2     VkDescriptorSetLayoutBinding uboLayoutBinding{};
3     uboLayoutBinding.binding = 0;
4     uboLayoutBinding.descriptorType =
5         VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
6     uboLayoutBinding.descriptorCount = 1;
7 }
```

Первые два поля определяют привязки используемый в шейдере и тип дескриптора, который представляет собой единый буферный объект. Переменная шейдера может представлять массив однородных объектов буфера, и descriptorCount указывает количество значений в массиве. Это можно использовать, чтобы указать преобразование для каждой кости скелета, например, для скелетной анимации.

Наше преобразование MVP выполняется в одном объекте uniform buffer, поэтому мы

```
1 используем descriptorCount из 1.
```

Удаление привязки.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;

Нам также необходимо указать, на какие этапы шейдера будет ссылаться дескриптор.encoded.

Поле stageFlags может представлять собой комбинацию VkShaderStageFlagBits значения или само значение VK_SHADER_STAGE_ALL_GRAPHICS . В нашем случае мы только

ссылаемся на дескриптор из вершинного шейдера.

```
1 uboLayoutBinding.pImmutableSamplers = nullptr; // Необязательно
```

The pImmutableSamplers поле актуально только для дескрипторов, связанных с выборкой изображений, которые мы рассмотрим позже. Вы можете оставить это значение по умолчанию. Все привязки дескрипторов объединены в один VkDescriptorSetLayout

объект. Определите новый член класса выше pipelineLayout:

```
1 VkDescriptorSetLayout описатель сетевого описания;
2 VkPipelineLayout конвейерное описание;
```

Затем мы можем создать его с помощью vkCreateDescriptorSetLayout. Эта функция

включает в себя простой VkDescriptorSetLayoutCreateInfo с массивом привязок:

```
1 VkDescriptorSetLayoutCreateInfo layoutInfo{};
2 layoutInfo.sType =
3     VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
4 layoutInfo.bindingCount = 1;
5 layoutInfo.pBindings = &uboLayoutBinding;
6
7 если (vkCreateDescriptorSetLayout(устройство, &layoutInfo,
8     nullptr, &descriptorSetLayout) != VK_SUCCESS) {
9     выбросить std::runtime_error("не удалось создать набор дескрипторов"
10     "планировка!");
11 }
```

Нам нужно указать макет набора дескрипторов при создании конвейера, чтобы сообщить Vulkan, какие дескрипторы будут использовать шейдеры. Описание макеты параметров указаны в объекте компоновки конвейера.

Измените `VkPipelineLayoutCreateInfo` для ссылки на объект компоновки:

```
1 VkPipelineLayoutCreateInfo pipelineLayoutInfo{};  
2 pipelineLayoutInfo.тип =  
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;  
3 pipelineLayoutInfo.setLayoutCount = 1;  
4 pipelineLayoutInfo.pSetLayouts = &descriptorSetLayout;
```

Возможно, вам интересно, почему здесь можно указать несколько макетов набора дескрипторов, потому что один из них уже включает все привязки. Мы вернемся к этому в следующей главе, где рассмотрим пулы дескрипторов и наборы дескрипторов.

Макет дескриптора должен сохраняться, пока мы можем создавать новые графические конвейеры, т. е. до завершения работы программы:

```
1 void cleanup() {  
2     cleanupSwapChain();  
3  
4     vkDestroyDescriptorSetLayout(устройство, descriptorSetLayout,  
        nullptr);  
5  
6     ...  
7 }
```

Равномерный буфер

В следующей главе мы укажем буфер, содержащий данные UBO для шейдера, но сначала нам нужно создать этот буфер. Мы собираемся копировать новые данные в единий буфер каждый кадр, поэтому на самом деле нет никакого смысла иметь промежуточный буфер. В данном случае это просто добавило бы дополнительных накладных расходов и, вероятно, снизило бы производительность вместо того, чтобы улучшить ее. У нас должно быть несколько буферов, потому что несколько кадров могут находиться в процессе обработки одновременно, и мы не хотим обновлять буфер при подготовке следующего кадра, в то время как предыдущий все еще считывается из него! Таким образом, нам нужно иметь столько однородных буферов, сколько у нас есть кадров в полете, и записывать в однородный буфер, который в данный момент не считывается графическим процессором

С этой целью добавьте новых членов класса для `uniformBuffers` и `uniformBuffersMemory`:

```
1 VkBuffer indexBuffer;  
2 VkDeviceMemory indexBufferMemory;  
3  
4 std::vector<VkBuffer> Равномерные буферы;
```

```

5 std::vector<VkDeviceMemory> uniformBuffersMemory;
6 std::vector<void * > uniformBuffersMapped;

Аналогично, создайте
новую функцию createIndexBuffer
и выделяет буферы:
7 }

8

9 ...
10
void createUniformBuffers()
{ 11
    VkDeviceSize bufferSize = размер буфера(UniformBufferObject);
12
13
    Равномерные буферы.изменение размера(MAX_FRAMES_IN_FLIGHT);
14
    uniformBuffersMemory.resize(MAX_FRAMES_IN_FLIGHT);
15
    uniformBuffersMapped.resize(MAX_FRAMES_IN_FLIGHT);
16
17
    для (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
18        createBuffer(размер буфера, VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
19                     VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
20                     VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, uniformBuffers[i],
21                     uniformBuffersMemory[i]); vkMapMemory(устройство,
22                     uniformBuffersMemory[i], 0, размер буфера,
23                     0, &uniformBuffersMapped[i]);
}
}

```

ЭТО ВЫЗЫВАЕТСЯ ПОСЛЕ

Мы сопоставляем буфер сразу после создания, используя `vkMapMemory` для получения указателя в который мы можем записать данные позже. Буфер остается сопоставленным с этим указателем на протяжении всего срока службы приложения. Этот метод называется "**постоянное сопоставление**". и работает во всех реализациях Vulkan. Отсутствие необходимости отображать буфер каждый раз, когда нам нужно его обновить, повышает производительность, поскольку отображение не бесплатное.

Унифицированные данные будут использоваться для всех вызовов draw, поэтому буфер, содержащий их, должен быть уничтожен только тогда, когда мы остановим рендеринг.

```

void cleanup() {
1
2     ...
3
}

```

```

4     для  (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
5         vkDestroyBuffer(устройство, uniformBuffers[i], nullptr);
6         vkFreeMemory(устройство, uniformBuffersMemory[i], nullptr);
7     }
8
9     vkDestroyDescriptorSetLayout(устройство, descriptorsetLayout,
10                                nullptr);
11
12     ...
13 }
```

Обновление унифицированных данных

<p>Создание новой функции</p> <pre> drawFrame функция перед отправкой следующего кадра: аннулирование drawFrame() { ... updateUniformBuffer(текущий кадр); ... } VkSubmitInfo submitInfo{}; submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO; ... } ... } пустота updateUniformBuffer(uint32_t currentImage) {</pre>	<p>updateUniformBuffer</p> <p>и добавьте к нему вызов из</p>
--	--

Эта функция будет генерировать новое преобразование каждого кадр, чтобы заставить геометрию вращаться по кругу.

Нам нужно включить два новых заголовка для реализации этой

функциональности: #define GLM_FORCE_RADIANS

```

1 # включить <glm/glm.hpp>
2 # включить <glm/gtc/matrix_transform.hpp>
3
4
5 # включить <хронограф>
```

В `glm / gtc /matrix_transform.hpp` заголовок предоставляет функции, которые могут быть использованы для генерации преобразований модели, таких как `glm::rotate`, просмотр преобразований, подобных, и проекционных преобразований, подобных `glm::perspective`. The `GLM_FORCE_RADIANS` определение необходимо, чтобы убедиться, что такие функции, как `glm::rotate` используют радианы в качестве аргументов, чтобы избежать любой возможной путаницы. The `chrono` заголовок стандартной библиотеки предоставляет функции для точного хронометражирования. Мы будем использовать это, чтобы убедиться, что геометрия поворачивается на 90 градусов в секунду независимо от частоты кадров.

```
1  пустота updateUniformBuffer(uint 3 2 _t  текущее  изображение) {
2
3      статический  автоматический  режим  Время  запуска  =
4          std::chrono::high_resolution_clock::сейчас  ();
5
6      автоматический  currentTime  =  std::chrono::high_resolution_clock::now();
7      с  плавающей  точкой  time  =  std::chrono::duration<с  плавающей  точкой,
8
9          std::хронограф::секунды::период>(текущее
10         время  -  время  начала).count();
11 }
```

Функция `updateUniformBuffer` начнет с некоторой логики для вычисления времени в секундах с момента начала рендеринга с точностью до плавающей запятой.

Теперь мы определим преобразования модели, вида и проекции в объекте `uniform buffer`. Вращение модели будет простым вращением вокруг оси `Z` с использованием времени переменная:

```
1 UniformBufferObject  ubo{};  ubo.model
2 =  glm::поворот  (glm::mat 4 ( 1 . 0 f),  время  *  glm::радианы( 9 0 . 0 f),
3     glm::  vec 3 ( 0 . 0 f,  0 . 0 f,  1 . 0 f));
```

The `glm::поворот` функция принимает в качестве параметров существующее преобразование, угол поворота и ось вращения. Конструктор `glm::mat 4 (1 . 0 f)` возвращает матрицу идентификаторов. Использование угла поворота, равного `время * glm::радианы (9 0 , 0 f)` соответствует цели поворота на 90 градусов в секунду.

```
1 ubo.view =  glm::LookAt(glm::vec 3 ( 2 . 0 f,  2 . 0 f,  2 . 0 f),  glm::vec 3 ( 0 . 0 f,
2     0 . 0 f,  0 . 0 f),  glm::vec 3 ( 0 . 0 f,  0 . 0 f,  1 . 0 f));
```

Для преобразования вида я решил посмотреть на геометрию сверху под углом 45 градусов. The `glm::LookAt` функция принимает положение глаза, центральное положение и верхнюю ось в качестве параметров.

```
1 ubo.proj =  glm::перспектива (glm ::  радианы ( 4 5 . 0 f)),
2 swapChainExtent.ширина  /  (с  плавающей  точкой)  swapChainExtent.высота,  0 , 1  f,
3 1 0 , 0 f);
```

Я решил использовать перспективную проекцию с вертикальным полем зрения 45 градусов. Другими параметрами являются соотношение сторон, ближняя и дальняя плоскости обзора. IT

важно использовать текущий экстент цепочки подкачки для расчета соотношения сторон, чтобы учесть новую ширину и высоту окна после изменения размера.

```
1 ubo.proj[ 1 ][ 1 ] *= - 1 ;
```

GLM изначально был разработан для OpenGL, где координата Y в координатах клипа инвертирована. Самый простой способ компенсировать это - перевернуть знак на коэффициенте масштабирования оси Y в матрице проекции. Если вы этого не сделаете, то изображение будет отображаться вверх ногами. Теперь все преобразования определены, поэтому мы можем скопировать данные из объекта uniform buffer в текущий uniform buffer . Это происходит точно так же, как мы делали для вершинных буферов, за исключением того, что промежуточный буфер отсутствует. Как отмечалось ранее, мы сопоставляем единый буфер только один раз, поэтому можем выполнять прямую запись в него без необходимости повторного сопоставления:

```
1 memcpys(uniformBuffersMapped[Текущее изображение], &ubo, размер (убо));
```

Использование UBO таким образом - не самый эффективный способ передачи часто меняющихся значений шейдеру. Более эффективным способом передачи небольшого буфера данных в шейдеры являются *нажимные константы*. Мы можем рассмотреть их в следующей главе.

В следующей главе мы рассмотрим наборы дескрипторов, которые фактически будут связывать

```
VkBuffer  
      S к дескрипторам унифицированного буфера, чтобы шейдер мог получить доступ к этому данным преобразования.
```

Код на C++ / Вершинный шейдер / Фрагментный шейдер

Пул дескрипторов и наборы

Введение

Схема дескрипторов из предыдущей главы описывает тип дескрипторов, которые могут быть связаны. В этой главе мы собираемся создать набор дескрипторов для каждого ресурса, чтобы привязать его к единому дескриптору буфера.

Пул дескрипторов

Наборы дескрипторов нельзя создать напрямую, они должны быть выделены из пула, такого как буфера команд. Эквивалент наборов дескрипторов, что неудивительно, называется *пул дескрипторов*. Мы напишем новую функцию `createDescriptorPool` для

```
ее настройки.
```

```
1 void initVulkan() {  
2     ...  
3     createUniformBuffers();  
4     createDescriptorPool();  
5     ...  
6 }  
7
```

```
8 ...
9
10 аннулирование createDescriptorPool() {
11
12 }
```

Сначала нам нужно описать, какие типы дескрипторов будут содержать наши наборы дескрипторов и сколько их, используя `VkDescriptorPoolSize` structures.

```
1 VkDescriptorPoolSize poolSize{};
2 poolSize.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
3 poolSize.descriptorCount =
    static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);
```

Мы выделим один из этих дескрипторов для каждого кадра. На эту структуру размера пула ссылается `main`'s `VkDescriptorPoolCreateInfo`:

```
1 VkDescriptorPoolCreateInfo poolInfo{};
2 poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
3 poolInfo.poolSizeCount = 1;
4 poolInfo.pPoolSizes = &poolSize;
```

Помимо максимального количества доступных отдельных дескрипторов, нам также необходимо указать максимальное количество наборов дескрипторов, которые могут быть выделены:

```
1 poolInfo.maxSets = static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);
```

Структура имеет необязательный флаг, аналогичный пулам команд, который определяет, можно ли освободить отдельные наборы дескрипторов или нет: мы не собираемся трогать набор дескрипторов после его создания, поэтому нам не

нужен этот флаг. Вы можете оставить `flags` равным значению по умолчанию 0.

```
1 VkDescriptorPool Описательный пул;
2 ...
3 ...
4 если (vkCreateDescriptorPool(устройство, &poolInfo, nullptr,
5     &DescriptorPool) != VK_SUCCESS) {
6     выбросить std::runtime_error("не удалось создать пул дескрипторов!");
7 }
```

Добавьте новый член класса для хранения пула дескрипторов и вызовите `vkCreateDescriptorPool` для его создания.

Набор дескрипторов

Теперь мы можем выделить сами наборы дескрипторов. Добавьте функцию `a createDescriptorSets` для этой цели:

```

1 анулирование Инициализированный вулкан() {
2     ...
3     createDescriptorPool();
4     createDescriptorSets();
5     ...
6 }
7 ...
8 ...
9
10 void createDescriptorSets() {
11
12 }
```

Распределение набора дескрипторов описывается с помощью `VkDescriptorSetAllocateInfo` структуры. Вам необходимо указать пул дескрипторов для выделения, количество выделяемых наборов дескрипторов и макет дескриптора, на котором они будут основаны:

```

1 стандартный формат::векторные макеты<VkDescriptorSetLayout>(MAX_FRAMES_IN_FLIGHT,
2     descriptorsetLayout);
3 VkDescriptorSetAllocateInfo allocInfo{};
4 allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
5 allocInfo.DescriptorPool = DescriptorPool;
6 allocInfo.descriptorSetCount =
7     static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);
6 allocInfo.pSetLayouts = layouts.data();
```

В нашем случае мы создадим один набор дескрипторов для каждого кадра в полете, все с одинаковым расположением. К сожалению, нам действительно нужны все копии макета, потому что следующая функция ожидает массив, соответствующий количеству наборов. Добавьте член класса для хранения

дескрипторов набора и выделите их с помощью `vkAllocateDescriptorSets` :

```

1 VkDescriptorPool Дескрипторпул;
2 std::vector<VkDescriptorSet> дескрипторы;
3 ...
4 ...
5 дескрипторы.resize(MAX_FRAMES_IN_FLIGHT);
6 if (vkAllocateDescriptorSets(устройство и allocInfo,
7
8     Наборы описаний.data()) != VK_SUCCESS) {
9     выбросить std::runtime_error("не удалось выделить наборы описаний!");
9 }
```

Вам не нужно явно очищать наборы дескрипторов, потому что они будут автоматически освобождены при уничтожении пула дескрипторов. Для `vkAllocateDescriptorSets` выделит наборы дескрипторов, каждый с одним унифицированным дескриптором буфера.

Вызов

```
1 анулирование очистка() {
2     ...
3     vkDestroyDescriptorPool(устройство, дескрипторпул, nullptr);
4
5     vkDestroyDescriptorSetLayout(устройство, descriptorsetLayout,
6                                 nullptr);
7 }
```

Наборы дескрипторов уже распределены, но дескрипторы внутри все еще нуждаются в настройке. Теперь мы добавим цикл для заполнения

```
1 каждого дескриптора: для (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
2
3 }
```

Дескрипторы, которые ссылаются на буферы, такие как наш унифицированный дескриптор буфера, сконфигурированы с помощью `VkDescriptorBufferInfo` struct . Эта структура определяет буфер и область внутри него, содержащую данные для

```
1 дескриптора. для (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
2
3     VkDescriptorBufferInfo BufferInfo{};
4
5     BufferInfo.buffer = Равномерные
6     буферы[i]; BufferInfo.offset = 0;
7     BufferInfo.range = sizeof(UniformBufferObject);
8 }
```

Если вы перезаписываете весь буфер, как мы делаем в этом случае, то также можно использовать параметр `VK_WHOLE_SIZE` значение для диапазона. Конфигурация дескрипторов обновляется с помощью `vkUpdateDescriptorSets` функция, которая принимает массив `VkWriteDescriptorSet` структуры в качестве параметра.

```
1 VkWriteDescriptorSet descriptorWrite{};
2 descriptorWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
3 descriptorWrite.dstSet = Наборы описаний[i];
4 descriptorWrite.dstBinding = 0;
5 descriptorWrite.dstArrayElement = 0;
```

Первые два поля определяют дескриптор, установленный для обновления, и привязку. Мы указали наш единый индекс привязки буфера . Помните, что дескрипторами могут быть массивы, поэтому нам также нужно указать первый индекс в массиве, который мы хотим обновить. Мы не используем массив, поэтому индекс просто 0 .

```
1 descriptorWrite.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
2 descriptorWrite.descriptorCount = 1;
```

Нам нужно снова указать тип дескриптора. Можно обновлять сразу несколько нескольких дескрипторов в массиве, начиная с индекса `dstArrayElement`. В поле `descriptorCount` указывается, сколько элементов массива вы хотите обновить.

```
1 Запись описания.pBufferInfo = &BufferInfo;
2 Запись с помощью дескриптора.pImageInfo = nullptr; // Необязательно
3 descriptorWrite.pTexelBufferView = nullptr; // Необязательно
```

Последнее поле ссылается на массив с descriptorCount структурами, которые на самом деле настройте дескрипторы. В зависимости от типа дескриптора, какой из трех вам действительно нужно использовать. Поле используется для дескрипторов, которые **ссылаются на данные буфера**, **pImageInfo** используется для дескрипторов, которые ссылаются на данные изображения, **И pTexelBufferView** используется для дескрипторов, которые ссылаются на представления буфера. Наш дескриптор основан на буферах, поэтому мы используем **pBufferInfo**.

```
1 vkUpdateDescriptorSets(device, 1, &descriptorWrite, 0, nullptr);
```

Обновления применяются с помощью **vkUpdateDescriptorSets**. Он принимает массивы двух типов в качестве параметров: массив из **VkWriteDescriptorSet** и массив из **VkCopyDescriptorSet**. Последний может использоваться для копирования дескрипторов друг в друга, как следует из его названия.

Использование наборов дескрипторов

Теперь нам нужно обновить функцию **recordCommandBuffer**, чтобы фактически привязать правильный набор дескрипторов для каждого кадра к дескрипторам в шейдере с помощью **vkCmdBindDescriptorSets**. Это должно быть сделано до

vkCmdDrawIndexed звоните:

```
1 vkCmdBindDescriptorSets(командный буфер,
VK_PIPELINE_BIND_POINT_GRAPHICS, конвейерное описание,
0, 1, &наборы описаний[currentFrame], 0, nullptr);
2 vkCmdDrawIndexed(командный буфер,
static_cast<uint32_t>(индексы.размер()), 1, 0, 0, 0);
```

В отличие от вершинных и индексных буферов, наборы дескрипторов не являются уникальными для графических конвейеров. Поэтому нам нужно указать, хотим ли мы привязать наборы дескрипторов к графическому или вычислительному конвейеру. Следующий параметр - это макет, на котором основаны дескрипторы. Следующие три параметра определяют индекс первого набора дескрипторов, количество наборов для привязки и массив наборов для привязки. Мы вернемся к этому чуть позже. Последние два параметра определяют массив смещений, которые используются для динамических дескрипторов. Мы рассмотрим их в следующей главе.

Если вы запустите свою программу сейчас, то заметите, что, к сожалению, ничего не видно. Проблема в том, что из-за Y-образного поворота, который мы сделали в матрице проекции, вершины теперь рисуются против часовой стрелки, а не по часовой. Это приводит к отбору обратной поверхности и предотвращает

отрисовку любой геометрии. Перейдите к **createGraphicsPipelineFrontFace** чтобы исправить это:

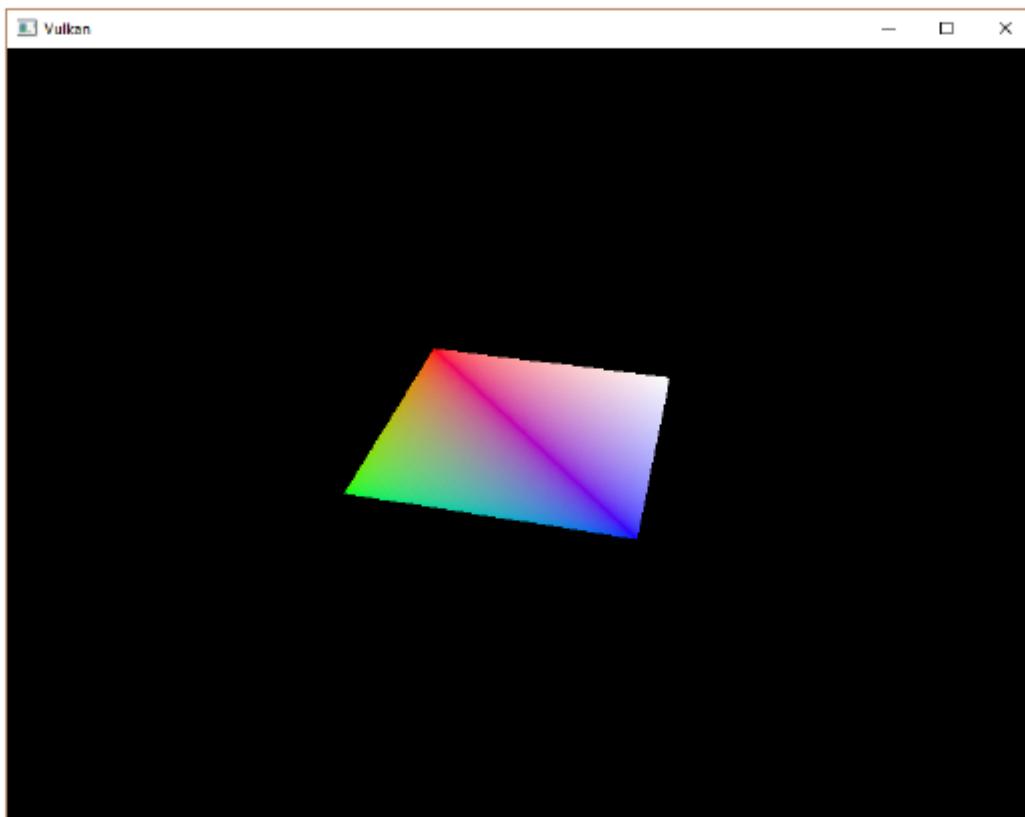
выполните и измените функцию

```
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
```

```
1
```

```
2 растеризатор.FrontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
```

Запустите свою программу еще раз, и теперь вы должны увидеть следующее:



Прямоугольник превратился в квадрат, потому что матрица проекции теперь соответствует соотношению сторон. The `updateUniformBuffer` заботится об изменении размера экрана, поэтому нам не нужно воссоздавать набор дескрипторов в `recreateSwapChain`.

Требования к выравниванию

Одна вещь, которую мы до сих пор упускали из виду, - это то, как именно данные в структуре C++ должны соответствовать единообразному определению в шейдере. Кажется достаточно очевидным просто использовать

```
1 одни и те же типы в обоих: struct UniformBufferObject {
2     glm:: модель mat 4 ;
3     glm:: вид mat 4 ;
4     glm:: mat 4  proj;
5 };
6
7 макет (привязка = 0 ) единый UniformBufferObject {
8     модель mat 4 ;
9     вид mat 4 ;
```

```
10     mat 4  
11 } ubo;
```

Однако это еще не все, что нужно сделать. Например, попробуйте изменить структуру и шейдер, чтобы они выглядели следующим образом:

```
1 struct UniformBufferObject {  
2     glm:: vec 2 foo;  
3     glm:: mat 4 модель;  
4     glm:: mat 4 вид;  
5     glm::mat 4 проект;  
6 };  
7  
8 макет(привязка = 0) единообразный UniformBufferObject {  
9     vec 2 foo;  
10    модель mat 4 ;  
11    вид mat 4 ;  
12    mat 4 proj;  
13 } ubo;
```

Перекомпилируйте свой шейдер и свою программу, запустите их, и вы обнаружите, что красочный квадрат, над которым вы работали до сих пор, исчез! Это потому, что мы не приняли во внимание *требования к выравниванию*. Vulkan ожидает, что данные в вашей структуре будут выровнены в памяти определенным способом, например:

- Скаляры должны быть выровнены по N (= 4 байта с учетом 32-битных значений с плавающей запятой).
- A `vec 2` должно быть выровнено на 2N (= 8 байт).
- A `vec 3` или `vec 4` должно быть выровнено на 4N (= 16 байт).
- Вложенная структура должна быть выровнена по базовому выравниванию ее элементов округлено до значения, кратного 16.
- A `mat 4` матрица должна иметь такое же выравнивание, как и `a vec 4`.

Полный список требований к выравниванию вы можете найти в спецификации.

Наш оригинальный шейдер всего с тремя шейдерами `mat 4` поля уже соответствуют выравниванию г-требования. Поскольку каждый из них `mat 4` имеет размер $4 \times 4 \times 4 = 64$ байта, модель имеет смещение 0, имеет смещение 64 и `proj` имеет смещение 128. Все это кратно 16, и именно поэтому все работало нормально.

Новая структура начинается с `a vec 2`, который имеет размер всего 8 байт и, следовательно, отбрасывает все смещения. Теперь модель имеет смещение в 8, вид смещение в 7 2 и проекция смещение на 1 3 6, ни один из которых не кратен 16. Чтобы решить эту проблему, мы можем использовать спецификатор `alignas`, представленный в C++ 11:

```
структура UniformBufferObject {  
1  
2     glm::vec 2 foo;  
3     выравнивание ( 1 6 ) glm::модель mat 4 ;
```

```
4     glm:: mat 4 view;
5     glm::mat 4 proj;
6 };
```

Если вы сейчас скомпилируете и запустите свою программу снова, вы должны увидеть, что шейдер снова правильно получает свои матричные значения.

К счастью, есть способ не думать об этих требованиях к выравниванию **большую часть времени**. Мы можем определить `GLM_FORCE_DEFAULT_ALIGNED_GENTYPES` (значения по умолчанию) правильно

перед включением GLM:

```
1 # определить GLM_FORCE_RADIANS
2 # определить GLM_FORCE_DEFAULT_ALIGNED_GENTYPES
3 # включить <glm/glm.hpp>
```

Это вынудит GLM использовать версии `vec 2` и `mat 4` с выравниванием, требования к которому уже указаны для нас. Если вы добавите это определение, вы сможете удалить спецификатор `alignas`, и ваша программа все равно должна работать.

К сожалению, этот метод может выйти из строя, если вы начнете использовать вложенные структуры. Рассмотрим следующее определение в коде C++:

```
1 struct Foo {
2     glm::vec 2 v;
3 };
4
5 структура UniformBufferObject {
6     Foo f 1;
7     Foo f 2;
8 };
```

И следующее определение

```
1 шейдера:struct Foo {
2     vec 2 b;
3 };
4
5 layout(привязка = 0) uniform UniformBufferObject {
6     Foo f 1;
7     Foo f 2;
8 } ubo;
```

В этом случае `f 2` будет иметь смещение, равное `8`, принимая во внимание, что он должен иметь смещение в `1 6` поскольку это вложенная структура. В этом случае вы должны указать выравнивание самостоятельно:

```
1 struct UniformBufferObject {
2     Foo f 1;
3     выравнивание( 1 6 ) Foo f 2;
4 };
```

Эти ошибки - хорошая причина всегда четко указывать на выравнивание.

Так вы не будете застигнуты врасплох странными симптомами ошибок

```
1 выравнивания.структура UniformBufferObject {
2     alignas( 16 ) glm:: модель mat 4 ;
3     alignas( 16 ) glm:: вид mat 4 ;
4     alignas( 16 ) glm:: мат 4 proj;
5 };
```

Не забудьте перекомпилировать свой шейдер после удаления поля `foo`.

Несколько наборов дескрипторов

Как указывалось в некоторых структурах и вызовах функций, на самом деле возможно связать несколько наборов дескрипторов одновременно. Вам необходимо указать дескриптор макет для каждого набора дескрипторов при создании макета конвейера. Шейдеры могут затем ссылаться на конкретные наборы дескрипторов следующим образом:

```
1 макет(сет = 0, привязка = 0) единообразный UniformBufferObject { ... }
```

Вы можете использовать эту функцию для размещения дескрипторов, которые различаются для каждого объекта, и дескрипторов, которые являются общими, в отдельные наборы дескрипторов. В этом случае вы избегаете повторной привязки большинства дескрипторов при вызовах `draw`, что потенциально более эффективно. Код на C++ / Вершинный шейдер / Фрагментный шейдер

Отображение текстур

Изображения

Введение

До сих пор геометрия была раскрашена с использованием цветов для каждой вершины, что является довольно ограниченным подходом. В этой части урока мы собираемся реализовать отображение текстур, чтобы геометрия выглядела более интересной. Это также позволит нам загружать и отрисовывать базовые 3D-модели в следующей главе. Добавление текстуры в наше приложение будет включать следующие шаги:

- Создайте объект изображения, сохраненный в памяти устройства • Заполните его пикселями из файла изображения
- Создайте сэмплер изображения • Добавьте комбинированный дескриптор `image sampler` для выборки цветов из текстуры

Мы уже работали с объектами изображений раньше, но они были автоматически созданы расширением `swap chain`. На этот раз нам придется создать его самим. Создание изображения и заполнение его данными аналогично созданию вершинного буфера. Мы начнем с создания промежуточного ресурса и заполнения его пиксельными данными, а затем скопируем это в конечный объект изображения, который мы будем использовать для рендеринга. Хотя для этой цели можно создать промежуточное изображение, Vulkan также позволяет копировать пиксели из `VkBuffer` к изображению и API для этого на самом деле быстрее на некотором оборудовании. Сначала мы создадим этот буфер и заполним его значениями пикселей, а затем создадим изображение для копирования пикселей. Создание изображения не сильно отличается от создания буферов. Это включает в себя запрос требований к памяти, выделение памяти устройства и ее привязку, точно так же, как мы видели раньше.

Однако, есть еще кое-что, что мы должны заботиться о При работе с изображениями. Образы могут иметь разные *макеты* которые влияют на то, как пиксели организованы в памяти. Например, из-за того, как работает графическое оборудование, простое сохранение пикселей строка за строкой может не привести к наилучшей производительности. При выполнении какой-либо операции с изображениями вы должны убедиться, что они имеют

макет, оптимальный для использования в этой операции. На самом деле мы уже видели некоторые из этих макетов, когда указывали этап рендеринга:

- VK_IMAGE_LAYOUT_PRESENT_SRC_KHR : Оптимальный для презентации
 - VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL : Оптимально в качестве вложения
- для записи цветов из фрагментного шейдера
- VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL : Оптимально в качестве источника в преобразовании-операция fer, подобная `vkCmdCopyImageToBuffer`
 - VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL : Оптимально в качестве пункта назначения в а
- операция переноса, например `vkCmdCopyBufferToImage`
- VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL: Оптимально для выборки из шейдера

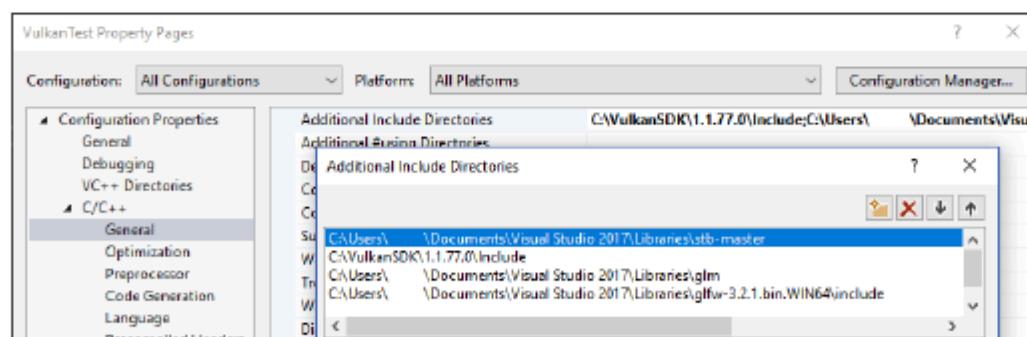
Одним из наиболее распространенных способов перехода макета изображения является барьер для трубопровода. Конвейерные барьеры в основном используются для синхронизации доступа к повторным источникам, например, для проверки того, что изображение было записано до его чтения, но они также могут использоваться для перехода к макетам. В этой главе мы увидим, как для этой цели используются конвейерные барьеры. Барьеры могут дополнительно использоваться для передачи владения семейством очередей при использовании `VK_SHARING_MODE_EXCLUSIVE`.

Библиотека изображений

Доступно множество библиотек для загрузки изображений, и вы даже можете написать свой собственный код для загрузки простых форматов, таких как BMP и PPM. В этом руководстве мы будем использовать библиотеку stb_image из коллекции stb. Преимущество этого в том, что весь код находится в одном файле, поэтому для него не требуется какой-либо сложной сборки конфигурации. Скачать `stb_image.h` и сохраните его в удобном месте, например, в каталоге, в котором вы сохранили GLFW и GLM. Добавьте местоположение в свой включаемый путь.

Visual Studio

Добавьте каталог C `stb_image.h` в нем в Дополнительные включаемые каталоги пути.



Makefile

Добавьте каталог C `stb_image.h` в каталоги включения для GCC:

```
1 VULKAN_SDK_PATH = /home/пользователь/VulkanSDK/x.x.x.x/x 8 _ 6 _ 4
2 STB_INCLUDE_PATH = /home/пользователь/ библиотеки/stb
3
4 ...
5
6 CFLAGS = -std=c++17 -I$(VULKAN_SDK_PATH)/включить
    -I$(STB_INCLUDE_PATH)
```

Загрузка изображения

Включите библиотеку изображений следующим

```
1 образом: # определите STB_IMAGE_IMPLEMENTATION
2 # включите <stb_image.h>
```

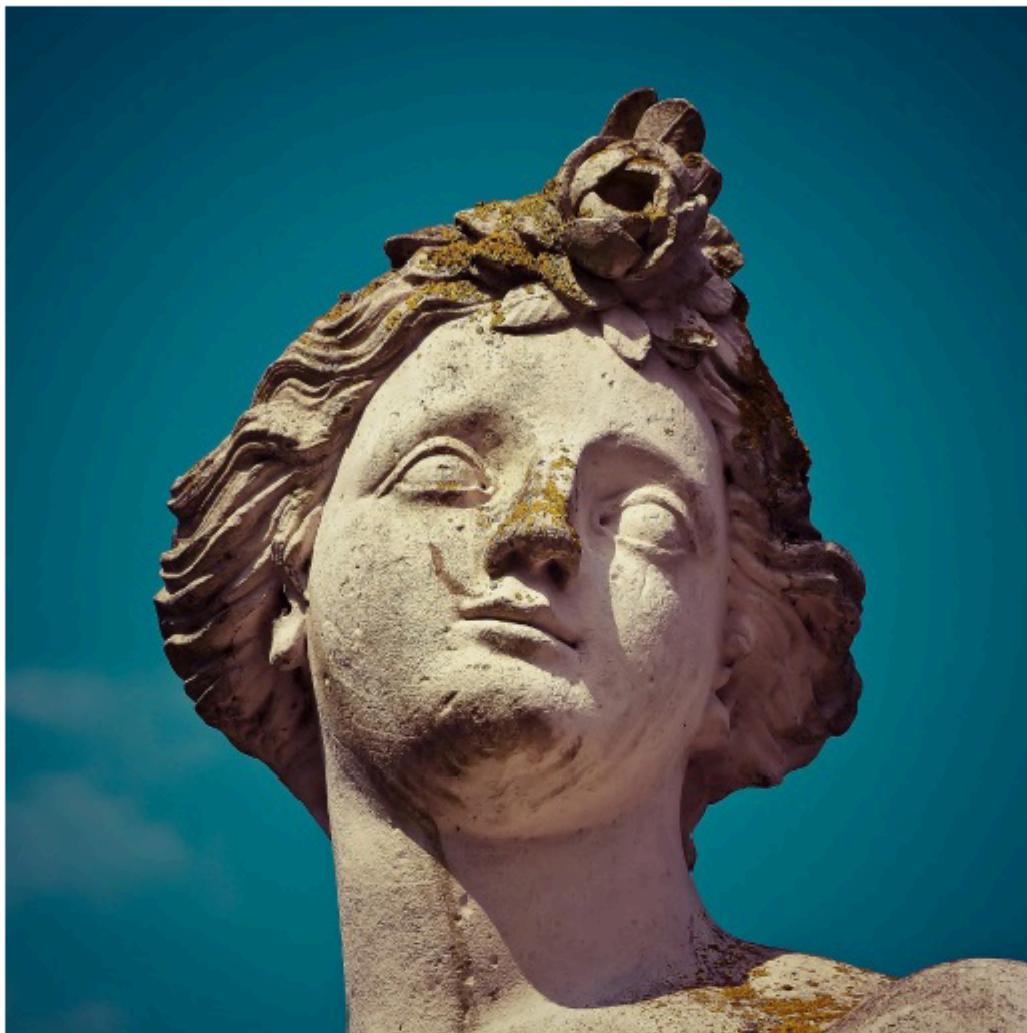
Заголовок определяет только прототипы функций по умолчанию. Один файл code должен включать заголовок `STB_IMAGE_IMPLEMENTATION` определение должно включать тела функций, иначе мы получим ошибки связывания.

```
1 void initVulkan() {
2     ...
3     createCommandPool();
4     createTextureImage();
5     createVertexBuffer(); ...
6 }
7 ...
8 ...
9 ...
10 void createTextureImage()
11 {
12 }
```

Создайте новую функцию

Создадим текстовое изображение где мы загрузим изображение и загрузите его в объект изображения Vulkan. Мы собираемся использовать командные буферы, поэтому его следует вызывать после `createCommandPool`.

Создайте новый каталог Текстуры рядом с каталогом шейдеры для хранения текстур изображений. Мы собираемся загрузить изображение под названием `texture.jpg` из этого каталога. Я решил использовать следующее лицензионное изображение CC0 с размером 512x512 пикселей, но не стесняйтесь выбирать любое изображение, которое вы хотите. Библиотека поддерживает наиболее распространенные форматы файлов изображений, такие как JPEG, PNG, BMP и GIF.



Загрузить изображение с помощью этой библиотеки

```
1  действительно просто:void createTextureImage() {
2      int texWidth, texHeight, текстовые каналы;
3      stbi_uc * пиксели = stbi_load("textures/texture.jpg", &texWidth,
4          &texHeight, &texChannels, STBI_rgb_alpha);
5      VkDeviceSize ImageSize = texWidth * texHeight * 4;
6
7      if (!пикселей) {
8          выбросить std::runtime_error ("не удалось загрузить изображение текстуры!");
9      }
}
```

stbi_load функция принимает путь к файлу и количество каналов для загрузки как
Аргументы. В STBI_rgb_alpha значение принудительно загружает изображение с помощью
Альфа-канал, даже если у него его нет, что хорошо для согласованности с другими текстурами
в будущем. Средние три параметра являются выходными данными для

ширина, высота и фактическое количество каналов в изображении.
Возвращаемый указатель является первым элементом в массиве значений
пикселей. Пиксели располагаются строка за строкой по 4 байта на пиксель в
случае, если в общей сложности $\text{texWidth} * \text{texHeight} * 4$ значения.

STBI_rgb_alpha

Промежуточный буфер

Теперь мы собираемся создать буфер в видимой памяти хоста, чтобы мы могли
использовать `vkMapMemory` и скопировать в него пиксели. Добавьте
переменные для этого временного буфера в функцию `createTextureImage` :

```
1 VkBuffer stagingBuffer;
2 VkDeviceMemory stagingBufferMemory;
```

Буфер должен находиться в видимой памяти хоста, чтобы мы могли отобразить его, и он
должен использоваться в качестве источника передачи, чтобы мы могли скопировать его в изображение

```
1 Позже: Создайте буфер (ImageSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
2     VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
3     VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
4     промежуточный буфер, промежуточный буфер памяти);
```

Затем мы можем напрямую скопировать значения пикселей, полученные из
библиотеки загрузки изображений, в буфер:

```
1 void* данные; vkMapMemory(устройство,
2     stagingBufferMemory, 0, размер изображения, 0, &данные);
3     memcopy(данные, пиксели), static_cast<size_t>(ImageSize));
4     vkUnmapMemory(устройство, stagingBufferMemory);
```

Не забудьте очистить исходный массив пикселей

```
1 прямо сейчас: stbi_image_free(пиксели);
```

Текстурное изображение

Хотя мы могли бы настроить шейдер для доступа к значениям пикселей в
буфере, для этой цели лучше использовать объекты изображений в Vulkan.
Объекты изображения упростят и ускорят получение цветов, позволяя нам
использовать, например, 2D-координаты. Пиксели внутри объекта изображения
называются текселями, и с этого момента мы будем использовать это название.

Добавьте следующие новые члены класса: `VkImage TextureImage;`

```
1 VkDeviceMemory textureImageMemory;
```

Параметры для изображения задаются в а

VkImageCreateInfo

struct:

```
1 VkImageCreateInfo ImageInfo{};
2 ImageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
3 ImageInfo.imageType = VK_IMAGE_TYPE_2D;
4 ImageInfo.extent.width = static_cast<uint32_t>(текстовая ширина);
5 ImageInfo.extent.height = static_cast<uint32_t>(texHeight);
6 ImageInfo.extent.depth = 1;
7 ImageInfo.mipLevels = 1;
8 ImageInfo.arrayLayers = 1;
```

Тип изображения, указанный в поле `ImageType`, сообщает Vulkan, с помощью какой системы координат будут адресованы текстелы на изображении. Возможно создавать одномерные, 2D и 3D изображения. Одномерные изображения можно использовать для хранения массива данных или градиента., двумерные изображения используются в основном для текстуры и трехмерные изображения можно использовать для хранения воксельной объемы, например. В `степени` поле задает размеры изображения, в основном, сколько текстелов на каждой оси. Вот почему глубина должна быть 1 вместо того, чтобы 0 . Наша текстура не будет массивом, и мы не будем сейчас использовать mipmapping.

```
1 ImageInfo.format = VK_FORMAT_R8G8B8A8_SRGB;
```

Vulkan поддерживает множество возможных форматов изображений, но мы должны использовать один и тот же форматируйте текстелы как пиксели в буфере, иначе операция копирования завершится неудачей. `ImageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;`

```
1
```

Поле `tiling` может иметь одно из двух значений:

- `VK_IMAGE_TILING_LINEAR`: Текселы расположены в порядке следования строк, как у нас
 - `VK_IMAGE_TILING_OPTIMAL`: Текселы расположены в определенном для реализации порядке для оптимального доступа, В отличие от макета изображения, режим разбиения на листы не может быть изменен позже . Если вы хотите иметь возможность прямого доступа к текстелям в памяти изображения, то вы должны использовать `VK_IMAGE_TILING_LINEAR` буфер вместо промежуточного изображения, так что в этом не будет необходимости. Мы будем использовать `VK_IMAGE_TILING_OPTIMAL` для эффективного доступа из шейдера.
- ```
1 ImageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
```

Существует только два возможных значения для параметра `initialLayout` изображения:

- `VK_IMAGE_LAYOUT_UNDEFINED`: Не пригодна для использования GPU и самый первый переход будет отказаться от текстелей.
- `VK_IMAGE_LAYOUT_PREINITIALIZED`: Не пригодна для использования графического процессора, но первый переход позволит сохранить текстелей.

Существует несколько ситуаций, когда необходимо сохранить тексели во время первого перехода. Одним из примеров, однако, было бы, если бы вы хотели использовать изображение в качестве промежуточного изображения в сочетании с макетом `VK_IMAGE_TILING_LINEAR`. В этом случае вы хотели бы загрузить в него данные `texel`, а затем преобразовать изображение в источник передачи без потери данных. Однако в нашем случае, мы сначала собираемся преобразовать изображение в пункт назначения для передачи, а затем скопировать в него данные `texel` из объекта `buffer`, поэтому нам не нужно это свойство, и мы можем безопасно использовать `VK_IMAGE_LAYOUT_UNDEFINED`.

```
1 ImageInfo.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT |
2 VK_IMAGE_USAGE_SAMPLED_BIT;
```

Поле `использование` имеет ту же семантику, что и при создании буфера. Изображение будет использоваться в качестве места назначения для копии в буфер, поэтому его следует настроить в качестве места назначения для передачи. Мы также хотим иметь доступ к изображению из шейдера для раскрашивания нашей сетки, поэтому использование должно включать `VK_IMAGE_USAGE_SAMPLED_BIT`.

```
1 ImageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

Изображение будет использоваться только одним семейством очередей: тем, которое поддерживает графические (и, следовательно, также) операции

```
1 ImageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
2 ImageInfo.flags = 0; // Необязательно
```

В `Образцы` флаг связан с мультисплайнингом. Это относится только к изображениям которые будут использоваться в качестве вложений, поэтому придерживайтесь одного образца. Есть несколько необязательных флагов для изображений, которые относятся к разреженным изображениям. Разреженные изображения - это изображения, в которых только определенные области фактически поддерживаются памятью. Если бы вы использовали, например, 3D-текстуру для воксельной местности, то вы могли бы использовать это, чтобы избежать выделения памяти для хранения больших объемов значений "воздуха". Мы не будем использовать его в этом руководстве, поэтому оставьте значение по умолчанию равным `0`.

```
1 если (vkCreateImage(device, &ImageInfo, nullptr, &TextureImage) !=
2 VK_SUCCESS) {
3 бросить std::runtime_error("не удалось создать изображение!");
4 }
```

Изображение создано с помощью `vkCreateImage`, у которого нет какого-либо конкретного `VkImageCreateInfo` заслуживающие внимания параметры. Возможно, что функция `VK_FORMAT_R8G8B8A8_SRGB` для-мат не поддерживается графическим оборудованием. У вас должен быть список приемлемых альтернатив, и выберите лучшую из поддерживаемых. Однако поддержка этого конкретного формата настолько широко распространена, что мы пропустим этот шаг. Использование различных форматов также потребовало бы неприятных преобразований. Мы вернемся к этому в главе о буфере глубины, где мы будем внедрять такую систему. `VkMemoryRequirements` Требования к памяти;

```

2 vkGetImageMemoryRequirements(требования к устройству, текстурному изображению и памяти);
3
4 VkMemoryAllocateInfo allocInfo{};
5 allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
6 allocInfo.allocationSize = memRequirements.size;
7 allocInfo.memoryTypeIndex =
8
9 Найдите Memorytype(memRequirements.memoryTypeBits,
10 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
11
12 если (vkAllocateMemory(устройство, &allocInfo, nullptr,
13 &textureImageMemory) != VK_SUCCESS) {
14 выбросить std::runtime_error("не удалось выделить память изображения!");
15 }
16
17 vkBindImageMemory(устройство, TextureImage, textureImageMemory, 0);

```

Выделение памяти для изображения работает точно так же, как и выделение памяти для буфера. Используйте `vkGetImageMemoryRequirements` вместо `vkGetBufferMemoryRequirements`, и используйте `vkBindImageMemory` вместо `vkBindBufferMemory`.

Эта функция уже становится довольно большой, и в последующих главах потребуется создать больше изображений, поэтому мы должны абстрагировать создание изображений от `createImage` функции, как мы делали для буферов. Создайте функцию и переместите создание объекта изображения и выделение ему памяти:

```

1 анулирование createImage(uint32_t ширина, uint32_t высота, формат VkFormat),
2
3 Разбиение на листы VkImageTiling, использование
4 VkImageUsageFlags, свойства VkMemoryPropertyFlags,
5 VkImage& image, VkDeviceMemory& imageMemory) {
6
7 VkImageCreateInfo ImageInfo{};
8
9 ImageInfo.sType =
10 VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO; ImageInfo.ImageType = VK_IMAGE_TYPE_2D;
11
12 ImageInfo.extent.width = ширина;
13
14 ImageInfo.extent.height = высота;
15
16 ImageInfo.extent.depth = 1;
17
18 ImageInfo.mipLevels = 1;
19
20 ImageInfo.arrayLayers = 1;
21
22 ImageInfo.format = формат;
23
24 ImageInfo.tiling = разбиение на листы;
25
26 ImageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
27
28 ImageInfo.usage = использование;
29
30 ImageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
31
32 ImageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
33
34
35
36
37 если (vkCreateImage(device, &ImageInfo, nullptr, &image) !=
38 VK_SUCCESS) {

```

```
18 выбросить std::runtime_error ("не удалось создать образ!");
19 }
20
21 Требования к памяти VkMemoryRequirements;
22 Требования к памяти vkGetImageMemoryRequirements (требования к устройству, изображению и памяти);
23
24 VkMemoryAllocateInfo allocInfo{};
25 allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
26 allocInfo.allocationSize = memRequirements.size;
27 allocInfo.memoryTypeIndex =
28 findMemoryType(memRequirements.memoryTypeBits, свойства);
29
30 если (vkAllocateMemory(устройство, &allocInfo, nullptr, &imageMemory)
31 != VK_SUCCESS) {
32 выбросить std::runtime_error("не удалось выделить память изображения!");
33 }
34
35 vkBindImageMemory(устройство, изображение, imageMemory, 0);
36 }
```

Я задал параметры width, height, format, tiling mode, usage и memory properties, потому что все они будут отличаться в зависимости от изображений, которые мы будем создавать на протяжении всего этого урока.

Функция `createTextureImage` теперь может быть упрощена до:

```
1 void createTextureImage() {
2
3 int texWidth, texHeight, texChannels;
4
5 stbi_uc * pixels = stbi_load("textures/texture.jpg", &texWidth,
6
7 &texHeight, &texChannels, STBI_rgb_alpha);
8
9 VkDeviceSize ImageSize = texWidth * texHeight * 4;
10
11
12 if (!pixels) {
13
14 throw std::runtime_error ("не удалось загрузить изображение текстуры!");
15
16 }
17
18
19 VkBuffer stagingBuffer;
20
21 VkDeviceMemory stagingBufferMemory;
22
23 Создайте буфер (ImageSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
24
25 VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
26
27 VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer,
28
29 stagingBufferMemory);
30
31
32 недействительность * данные;
33
34
35 vkMapMemory(устройство, stagingBufferMemory, 0, размер изображения, 0, и данные);
36
37 memcpy (данные, пиксели), static_cast<size_t>(ImageSize));
38
39 vkUnmapMemory(устройство, stagingBufferMemory);
```

```

18
19 stbi_image_free(в пикселях);
20
21 Создайте изображение(texWidth, texHeight, VK_FORMAT_R8G8B8A8_SRGB,
22 VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_TRANSFER_ST_BIT
23 | VK_IMAGE_USAGE_SAMPLED_BIT,
24 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, TextureImage,
25 textureImageMemory);
26
27 }

```

## Переходы между макетами

Функция, которую мы собираемся написать сейчас, включает в себя запись и повторное выполнение командного буфера, так что сейчас самое время перенести эту логику во вспомогательную функцию или две:

```

1 VkCommandBuffer beginSingleTimeCommands() {
2
3 VkCommandBufferAllocateInfo allocInfo{};
4
5 allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
6 allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
7 allocInfo.commandPool = commandPool;
8 allocInfo.commandBufferCount = 1;
9
10
11 VkCommandBuffer commandBuffer;
12 vkAllocateCommandBuffers(устройство, & allocInfo, & commandBuffer);
13
14
15 VkCommandBufferBeginInfo beginInfo{};
16
17 beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
18 beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
19
20 vkBeginCommandBuffer(командный буфер и beginInfo);
21
22
23 Возрат Командный буфер;
24
25 }
26
27
28 пустота endSingleTimeCommands(VkCommandBuffer commandBuffer) {
29
30 vkEndCommandBuffer(командный буфер);
31
32
33 VkSubmitInfo submitInfo{};
34
35 submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
36 submitInfo.commandBufferCount = 1;
37 submitInfo.pCommandBuffers = &commandBuffer;
38
39
40 vkQueueSubmit(graphicsQueue, 1, &submitInfo,
41 VK_NULL_HANDLE); vkQueueWaitIdle(graphicsQueue);
42
43 }

```

```
31 vkFreeCommandBuffers(устройство, commandPool, 1 и commandBuffer);
32 }
```

Код для этих функций основан на существующем  
коде в теперь можно упростить эту функцию до:

.ВыcopyBuffer

```
1 void Буфер копирования (VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize
2 size) { VkCommandBuffer
3 commandBuffer = beginSingleTimeCommands();
4 VkBufferCopy copyRegion{};
5 copyRegion.size = размер;
6 vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, 1,
7 ©Region);
8 endSingleTimeCommands(командный буфер);
9 }
```

Если бы мы все еще использовали буферы, то теперь мы могли бы написать  
функцию для записи и выполнения изображения, для завершения работы, но для этой команды требуется  
чтобы оно сначала было в нужном макете. Создайте новую функцию для  
обработки макета переходы:

```
1 void transitionImageLayout(изображение VkImage, формат VkFormat),
2 VkImageLayout Старое описание, VkImageLayout новое
3 описание) { VkCommandBuffer commandBuffer = beginSingleTimeCommands();
4 endSingleTimeCommands(commandBuffer);
5 }
```

Одним из наиболее распространенных способов выполнения переходов между макетами является использование *изображения барьера памяти*. Подобный конвейерный барьер обычно используется для синхронизации  
доступа к ресурсам, например, для обеспечения завершения записи в буфер перед чтением из него  
, но его также можно использовать для переноса макетов изображений и передачи владения семейством  
очередей, когда VK\_SHARING\_MODE\_EXCLUSIVE используется. Существует  
эквивалентный *барьер буферной памяти* чтобы сделать это для буферов.

```
1 Барьер VkImageMemoryBarrier{};
2 барьер.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
3 барьер.oldLayout = oldLayout;
4 барьер.newLayout = newLayout;
```

Первые два поля определяют переход макета.  
использовать VK\_IMAGE\_LAYOUT\_UNDEFINED как oldLayout ,  
если вас не волнует существующее содержимое изображения.

Можно

```
1 барьер.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
2 барьер.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
```

Если вы используете барьер для передачи владения семейством очередей, то эти два поля должны быть индексами семейств очередей. Для них должно быть установлено

значение `VK_QUEUE_FAMILY_IGNORED`

```
1 барьер.image = изображение; если вы не хотите этого делать (не значение по умолчанию).
2
3 барьер.Вложенный ресурс.Аспектная
4 маска = VK_IMAGE_ASPECT_COLOR_BIT; барьер.Подресурсный диапазон.Базовый уровень = 0; барьер.Подресурсный диапазон.levelCount = 1; барьер.Подресурсный диапазон.levelOffset = 0;
5 Значения image и subresourceRange указывают
```

изображение, на которое оказывается воздействие, и конкретную часть изображения. Наше изображение не является массивом и не имеет уровней пинга `mipmap`, поэтому указаны только один уровень и `layer`.

```
barrier.srcAccessMask = 0; // TODO
```

```
барьер.dstAccessMask = 0; // TODO
```

```
1 Барьера в основном используются для целей синхронизации, поэтому необходимо указать,
2 какие типы операций, связанных с ресурсом, должны выполняться до барьера
, а какие операции, связанные с ресурсом, должны ожидаться на барьере.
```

Нам нужно сделать это, несмотря на то, что мы уже используем `vkQueueWaitIdle` для синхронизации вручную `chronize`. Правильные значения зависят от старого и нового макета, поэтому мы вернемся к этому, как только разберемся, какие переходы мы собираемся

использовать.

```
vkCmdPipelineBarrier(
```

```
1 Командный буфер,
2 0 /* TODO */ *, 0 /* TODO */ *,
3 0,
4 0, nullptr,
5 0, nullptr,
6 1, &барьер
7);
8);
```

Все типы барьеров конвейера передаются с использованием одной и той же функции.

Первый параметр после буфера команд указывает, на каком этапе конвейера выполняются

операции, которые должны выполняться перед барьером. Второй параметр

определяет этап конвейера, на котором операции будут ожидаться на барьере.

Этапы конвейера, которые вам разрешено указывать до и после барьера, зависят

от того, как вы используете ресурс до и после барьера. Допустимые значения

перечислены в этой таблице спецификации. Например, если вы собираетесь читать из

униформы после барьера, вы должны указать использование `VK_ACCESS_UNIFORM_READ_BIT`

и самый ранний шейдер, который будет считываться из uniform как этап конвейера, например,

```
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT. Не имело бы смысла
```

указывать этап конвейера без шейдеров для этого типа использования, и уровни проверки

предупредят вас, когда вы укажете этап конвейера, который не соответствует

типу использования.

Третим параметром является либо `0` или `VK_DEPENDENCY_BY_REGION_BIT`. Последнее превращает барьер в условие для каждого региона. Это означает, что реализации разрешено уже начинать чтение, например, с тех частей ресурса, которые были написаны к настоящему времени.

Последние три пары параметров ссылаются на массивы конвейерных барьераов трех доступных типов: барьеры памяти, барьеры буферной памяти и барьеры памяти изображений, подобные тому, который мы используем здесь. Обратите внимание, что мы не используем раздел "Буфер глубины", параметра пока нет, но мы будем использовать его для специальных переходов в

## Копирование буфера в изображение

Прежде чем мы вернемся к `createTextureImage`, мы собираемся написать еще одну вспомогательную функцию: `copyBufferToImage`:

```
1 void copyBufferToImage(буфер VkBuffer, изображение VkImage, uint 3 2 _t
2
3 ширина, uint 3 2 _t высота) { VkCommandBuffer
4
5 commandBuffer = beginSingleTimeCommands();
6
7 endSingleTimeCommands(commandBuffer);
8 }
```

Также, как и в случае с копиями из буфера, вам нужно указать, какая часть буфера в какую часть изображения будет скопирована. Это происходит через

`VkBufferImageCopy` структуры:

```
1 VkBufferImageCopy region{};
2
3 region.bufferOffset = 0;
4 region.bufferRowLength = 0;
5 region.bufferImageHeight = 0;
6
7 region.imageSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
8 region.imageSubresource.mipLevel = 0;
9 region.imageSubresource.baseArrayLayer = 0;
10 region.imageSubresource.Количество слоев = 1;
11
12 region.imageOffset = { 0, 0, 0 };
13
14 регион.imageExtent = {
15
16 ширина,
17 высота,
18 1
19 };
```

Большинство этих полей не требуют пояснений. В

смещение в буфере, с которого начинаются значения

пикселей. Значение `bufferImageHeight` поля определяют, как пиксели расположены в памяти. Для

задает байт `bufferOffset`  
`bufferRowLength` и

например, у вас может быть несколько байтов заполнения между строками изображения.

Спецификация- если требуется `o for both` указывает на то, что пиксели просто плотно упакованы, как они в нашем случае. The , и `imageExtent` поля

указывают, в какую часть изображения мы хотим скопировать пиксели.

Операции копирования буфера в изображение ставятся в очередь с помощью `vkCmdCopyBufferToImage` функция:

```
1 vkCmdCopyBufferToImage(
2 Командный
3 буфер, буфер,
4 изображение,
5 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
6 1 ,
7 ®ion
8);
```

Четвертый параметр указывает, какой формат изображения используется в данный момент. Я предполагаю здесь, что изображение уже переведено в формат, который оптимальен для копирования пикселей. Прямо сейчас мы копируем только один фрагмент пикселей во все изображение, но можно указать массив `vkBufferImageCopy` для выполнения множества различных копий из этого буфера в изображение за одну операцию.

## Подготовка текстурного изображения

Теперь у нас есть все инструменты, необходимые для завершения настройки изображения текстуры, поэтому мы возвращаемся к функции `createTextureImage` . Последнее, что мы сделали, это создали изображение текстуры. Следующий шаг - скопировать промежуточный буфер в изображение текстуры. Это включает в себя два шага:

- Переместите изображение текстуры в `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`
- Выполните операцию копирования буфера в изображение

Это легко сделать с помощью функций, которые мы только что создали:

```
1 transitionImageLayout(Текстурное изображение, VK_FORMAT_R8G8B8A8_SRGB,
 VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL);
2 copyBufferToImage(stagingBuffer, TextureImage,
 static_cast<uint32_t>(ширина текста),
 static_cast<uint32_t>(высота текста));
```

Изображение было создано с использованием `VK_IMAGE_LAYOUT_UNDEFINED` layout, так что при переходе следует указать старый layout `TextureImage`. Повторите участнику, что мы можем это сделать, потому что мы не заботимся о его содержимом перед выполнением операции копирования. Чтобы иметь возможность начать выборку из изображения текстуры в шейдере, нам нужен один последний переход, чтобы подготовить его к доступу шейдера:

```
1 transitionImageLayout(Текстурное изображение, VK_FORMAT_R8G8B8A8_SRGB,
2 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
3 VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL);
```

## Маски барьеров перехода

Если вы запустите свое приложение с включенными уровнями проверки сейчас, то вы увидите, что оно жалуется на маски доступа и этапы конвейера в `transitionImageLayout` недействителен. Нам все еще нужно настроить их на основе макетов в переходе.

Нам нужно обработать два перехода:

- Не определено → пункт назначения передачи: при передаче записываются записи, которые не

нужно ждать

на что угодно

- Пункт назначения передачи → чтение шейдера: чтение с шейдера должно ожидаться при передаче записи с шейдера, в частности, чтение шейдером фрагмента, потому что именно там мы со

```
VkPipelineStageFlags
1 sourceStage; VkPipelineStageFlags этап назначения;
2
3
4 if (Старое описание == VK_IMAGE_LAYOUT_UNDEFINED && Новое описание ==
5 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL)
6 {
7 барьер.srcAccessMask = 0;
8 барьер.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
9
10 Исходная сцена = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
11 Целевая сцена = VK_PIPELINE_STAGE_TRANSFER_BIT;
12
13 } иначе, если (Старое описание == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL &&
14 newLayout == VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL) {
15 barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
16 barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
17 sourceStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
18 destinationStage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
19
20 } еще {
21 выбросить std::invalid_argument("неподдерживаемый переход макета!");
22 }
23
24 vkCmdPipelineBarrier(
25 commandBuffer,
26 sourceStage,
27 destinationStage, 0,
28 0, nullptr,
```

```
25 0, nullptr,
26 1, &барьер
27);
```

Как вы можете видеть в вышеупомянутой таблице, записи о переносе должны выполняться на этапе конвейерной передачи. Поскольку записи не нужно ничего ждать, вы можете указать пустую маску доступа и как можно более раннюю стадию конвейера для операций перед барьером. Это должно быть `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT`.  
следует отметить, что `VK_PIPELINE_STAGE_TRANSFER_BIT` не является *реальный* этап в графических и вычислительных конвейерах. Это скорее псевдостационар, на котором происходят передачи. Смотрите документацию для получения дополнительной информации и других примеров псевдостадий.

Изображение будет записано на том же этапе конвейера и впоследствии прочитано шейдером фрагментов, поэтому мы указываем доступ к чтению шейдера на этапе конвейера шейдеров фрагментов.

Если в будущем нам потребуется выполнить больше переходов, мы расширим функцию. Теперь приложение должно успешно запуститься, хотя, конечно, пока нет визуальных изменений.

Следует отметить, что отправка командного буфера приводит к неявному `VK_ACCESS_HOST_WRITE_BIT` синхронизации в начале. Начиная с `transitionImageLayout` функция выполняет буфер команд только с одной командой вы могли бы использовать эту неявную синхронизацию и установить `srcAccessMask`. Для 0 если вам когда-нибудь понадобится `VK_ACCESS_HOST_WRITE_BIT` зависимость от перехода к макету. Вам решать, хотите ли вы говорить об этом явно или нет, но лично я не сторонник полагаться на эти "скрытые" операции, подобные OpenGL. На самом деле существует специальный тип макета изображения, который поддерживает все операции, `VK_IMAGE_LAYOUT_GENERAL`. Проблема с ним, конечно, заключается в том, что он не обязательно обеспечивает наилучшую производительность для любой операции. Это требуется для некоторых особых случаев, таких как использование изображения как для ввода, так и для вывода, или для чтения изображения после того, как оно покинуло предварительно инициализированный макет.

Все вспомогательные функции, которые на данный момент отправляют команды, были настроены на выполняются синхронно, ожидая, пока очередь не перейдет в режим ожидания. Для практических приложений рекомендуется объединить эти операции в одном командном буфере и выполнять их асинхронно для повышения пропускной способности, особенно переходы и копирование в `createTextureImage` функция. Попробуйте поэкспериментировать с этим, создав `setupCommandBuffer`, в который вспомогательные функции записывают команды, и добавьте `flushSetupCommands` для выполнения команд, которые были записаны на данный момент. Лучше всего это делать после того, как сработает отображение текстур чтобы проверить, правильно ли по-прежнему настроены ресурсы текстур.

## Очистка

createTextureImage Завершите функция путем очистки промежуточного буфера и его память в конце:

```
1 transitionImageLayout(Текстурное изображение, VK_FORMAT_R8G8B8A8_SRGB,
2
3 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
4
5 VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL); vkDestroyBuffer(устройство,
6
7 промежуточный буфер, nullptr); vkFreeMemory(устройство,
8
9 промежуточный буфер памяти, nullptr);
10 }
```

Изображение основной текстуры используется до завершения

```
1 работы программы: аннулирование очистка() {
2
3 cleanupSwapChain();
4
5 vkDestroyImage(устройство, TextureImage, nullptr);
6 vkFreeMemory(устройство, textureImageMemory, nullptr);
7
8 }
```

Изображение теперь содержит текстуру, но нам все еще нужен способ получить к ней доступ из графического конвейера. Мы поработаем над этим в следующей главе. Код на C++ / Вершинный шейдер / Фрагментный шейдер

## Просмотр изображения и сэмплер

В этой главе мы собираемся создать еще два ресурса, которые необходимы для графического конвейера для сэмплирования изображения. Первый ресурс - это тот, который мы уже видели раньше при работе с изображениями цепочки подкачки, но второй является новым - он связан с тем, как шейдер будет считывать texsels из изображения.

### Просмотр изображений текстур

Мы уже видели ранее, с изображениями цепочки подкачки и фреймбуфером, что доступ к изображениям осуществляется через просмотр изображений, а не напрямую. Нам также нужно будет создать такой вид изображения для изображения текстуры. Добавьте элемент класса для хранения `VkImageView` для изображения текстуры и создайте новую функцию `createTextureImageView` где мы ее создадим:

```
1 VkImageView textureImageView;
2
3 ...
```

```

4
5 пустота Инициирующий вулкан() {
6
7 ...
8
9 createTextureImage();
10
11 createTextureImageView(); createVertexBuffer(); ...
12
13 ...
14
15 пустота createTextureImageView() {
16
17 }

```

Код для этой функции может быть основан непосредственно на `createImageViews`.

Вам нужно внести только два изменения: формат и этот образ:

```

1 VkImageViewCreateInfo viewInfo{};
2 viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
3 viewInfo.image = TextureImage;
4 viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
5 viewInfo.format = VK_FORMAT_R8G8B8A8_SRGB;
6 viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
7 viewInfo.subresourceRange.baseMipLevel = 0;
8 viewInfo. Субресурсный диапазон.levelCount = 1;
9 viewInfo.subresourceRange.baseArrayLayer = 0;
10 viewInfo.subresourceRange.layerCount = 1;

```

Я опустил явный параметр `viewInfo.components` инициализации, потому что `VK_COMPONENT_SWIZZLE_IDENTITY` определяется как 0 в любом случае. Завершите создание представления изображения, вызвав команду `vkCreateImageView`:

```

1 если (vkCreateImageView(устройство, & viewInfo, nullptr, & textureImageView)
2 != VK_SUCCESS) {
3 бросить std::runtime_error("не удалось создать вид изображения текстуры!");
4 }

```

Поскольку большая часть логики дублируется из хотите преобразовать его в новый `createImageView` функция:

`createImageViews`, Вы можете

```

1 VkImageView createImageView (изображение VkImage, формат VkFormat) {
2
3 VkImageViewCreateInfo viewInfo{};
4
5 viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
6 viewInfo.image = изображение;
7 viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
8 viewInfo.format = формат;

```

```

7 viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
8 viewInfo.subresourceRange.baseMipLevel = 0;
9 viewInfo.subresourceRange.levelCount = 1;
10 viewInfo.subresourceRange.baseArrayLayer = 0;
11 viewInfo.subresourceRange.layerCount = 1;
12
13 VkImageView Просмотр изображений;
14 если (vkCreateImageView(устройство, &viewInfo, nullptr, &imageView) !=
15 VK_SUCCESS) {
16 выбросить std::runtime_error("не удалось создать изображение текстуры
17 смотрите!");
18 }
19

```

The `createTextureImageView` теперь функцию можно упростить

```

1 ДО: анулировать createTextureImageView() {
2 textureImageView = создать изображение (TextureImage,
3 VK_FORMAT_R8G8B8A8_SRGB);
4 }

```

И `createImageViews` может быть упрощен до:

```

недействительно createImageViews()
{
1 swapChainImageViews.resize(swapChainImages.size());
2
3 для (uint32_t i = 0; i < swapChainImages.size(); i++) {
4 swapChainImageViews[i] = createImageView(swapChainImages[i],
5 swapChainImageFormat);
6 }
7 }

```

Убедитесь, что вы уничтожили представление изображения в конце программы,

непосредственно перед уничтожением самого изображения:

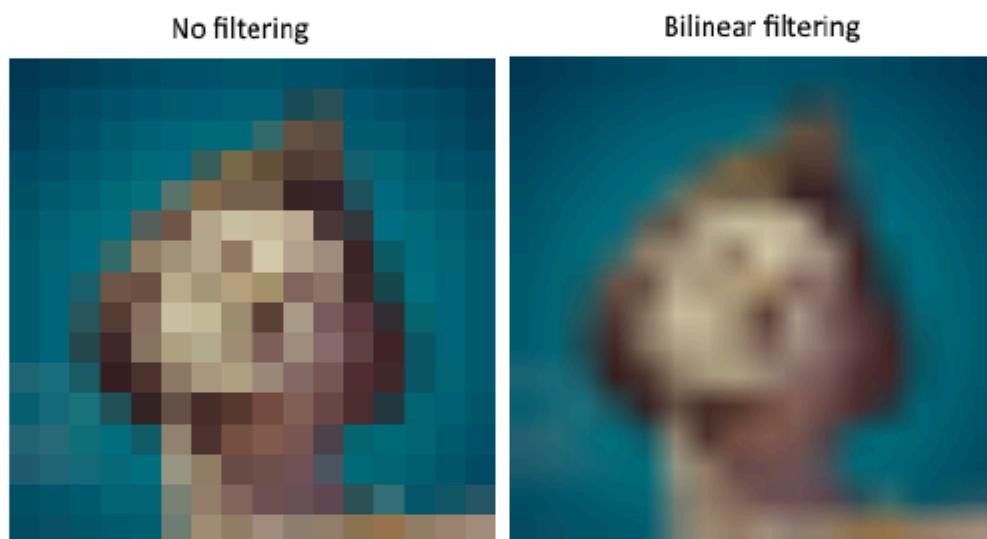
```

1 анулирование очистка () {
2 cleanupSwapChain();
3
4 vkDestroyImageView(устройство, textureImageView, nullptr);
5
6 vkDestroyImage(устройство, TextureImage, nullptr);
7 vkFreeMemory(устройство, textureImageMemory, nullptr);

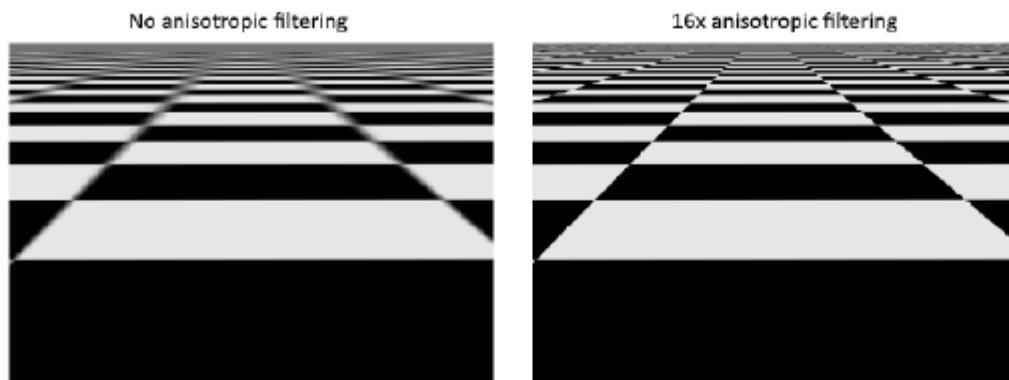
```

## Сэмплеры

Шейдеры могут считывать текстели непосредственно с изображений, но это не очень распространено, когда они используются в качестве текстур. Доступ к текстурам обычно осуществляется через сэмплеры, которые применяют фильтрацию и преобразования для вычисления конечного полученного цвета. Эти фильтры полезны для решения таких проблем, как передискретизация. Рассмотрим текстуру, отображенную в геометрию с большим количеством фрагментов, чем текстелов. Если бы вы просто взяли ближайший текстель для координаты текстуры в каждом фрагменте, то вы получили бы результат, подобный первому изображению:



Если бы вы объединили 4 ближайших текстеля с помощью линейной интерполяции, то получили бы более плавный результат, подобный показанному справа. Конечно, ваше приложение может предъявлять требования к художественному стилю, которые больше соответствуют левому стилю (например, Minecraft), но в обычных графических приложениях предпочтительнее правый. Объект `sampler` автоматически применяет эту фильтрацию для вас при считывании цвета из текстуры. Недостаточная дискретизация - противоположная проблема, когда у вас больше текстелов, чем фрагментов. Улучшения. Это приведет к появлению артефактов при выборке высокочастотных шаблонов, таких как текстура шахматной доски под острым углом:



Как показано на левом изображении, текстура на расстоянии превращается в размытое месиво. Решением этой проблемы является анизотропная фильтрация, которая также может быть применена автоматически с помощью сэмплера. Помимо этих фильтров, сэмплер также может выполнять преобразования. Он определяет, что произойдет, когда вы попытаетесь прочитать текстели за пределами изображения с помощью его режима адресации. Изображение ниже отображает некоторые из возможностей:



Теперь мы создадим функцию `createTextureSampler` для настройки такого объекта `sampler`. Мы будем использовать этот сэмплер для считывания цветов из текстуры в шейдере позже.

```

1 void initVulkan() {
2 ...
3 createTextureImage();
4 createTextureImageView();
5 createTextureSampler(); ...
6 }
7 ...
8 ...
9 ...
10
11 void createTextureSampler()
12 {
13 }
```

Пробоотборники настраиваются с помощью `VkSamplerCreateInfo` структура, которая определяет все фильтры и преобразования, которые она должна применять.

```
1 VkSamplerCreateInfo samplerInfo{};
2 samplerInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
3 samplerInfo.magFilter = VK_FILTER_LINEAR;
4 samplerInfo.minFilter = VK_FILTER_LINEAR;
```

Поля `magFilter` и `minFilter` определяют способ интерполяции текселов, которые увеличены или уменьшены в размерах. Увеличение связано с проблемой избыточной дискретизации, описанной выше, а уменьшение связано с недостаточной дискретизацией.

Варианты следующие `VK_FILTER_NAREST` и `VK_FILTER_LINEAR`, соответствует режимам, продемонстрированным на изображениях выше.

```
1 samplerInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
2 samplerInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
3 samplerInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
```

Режим адресации можно задать для каждой оси с помощью полей `addressMode`.

Доступные значения перечислены ниже. Большинство из них показано на

изображении выше. Обратите внимание, что оси называются U, V и W вместо X, Y и Z.

Это соглашение для координат текстурного пространства.

- `VK_SAMPLER_ADDRESS_MODE_REPEAT` : Повторите текстуру при переходе на следующий уровень.- измените размеры изображения.
- `VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT`: Похоже на повтор, но инвертирует координаты для зеркального отображения при выходе за пределы размеров.
- `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`: Выберите цвет края , ближайший к координате за пределами размеров изображения.
- `VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE`: Похоже на привязку к краю, но вместо этого использует край, противоположный ближайшему краю. • `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER`: Возвращает сплошной цвет при выборке за пределы размеров изображения.

На самом деле не имеет значения, какой режим адресации мы здесь используем, потому что в этом руководстве мы не собираемся использовать выборку за пределами изображения. Однако режим повтора, вероятно, является наиболее распространенным

режимом, поскольку его можно использовать для облицовки текстур, таких как полы и

```
1 стены. samplerInfo.anisotropyEnable = VK_TRUE;
2 samplerInfo.maxAnisotropy = ???;
```

Эти два поля указывают, следует ли использовать анизотропную фильтрацию. Нет причин не использовать это, если не беспокоит производительность. В `maxAnisotropy` поле ограничивает количество образцов texel, которые могут быть использованы для расчета конечного цвета. Меньшее значение приводит к повышению производительности, но к более низкому качеству результатов. Чтобы выяснить, какое значение мы можем использовать, нам нужно получить свойства физического устройства следующим образом:

```
1 Свойства VkPhysicalDeviceProperties{};
2 vkGetPhysicalDeviceProperties(Физическое устройство и свойства);
```

Если вы посмотрите документацию для `VkPhysicalDeviceProperties` structure, вы увидите, что она содержит `VkPhysicalDeviceLimits` элемент с именем `limits`. У этой структуры, в свою очередь, есть элемент с именем `maxSamplerAnisotropy` и это максимальное значение, которое мы можем указать для Максимальная анизотропия. Если мы хотим добиться максимального качества, мы можем просто использовать это значение напрямую:

```
1 samplerInfo.maxAnisotropy = свойства.пределы.Максимальная выборка анизотропии;

Вы можете либо запросить свойства в начале вашей программы и
передать их функциям, которым они нужны, либо запросить их в
createTextureSampler сама функция.
```

```
1 samplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK;

The borderColor поле указывает, какой цвет возвращается при выборке за пределами
изображение с режимом адресации "зажим в рамку". Можно вернуть черный,
белый или прозрачный в форматах float или int. Вы не можете указать произвольный цвет.
```

```
1 samplerInfo.Ненормализованные координаты = VK_FALSE;

1 The Ненормализованные координаты поле указывает, какую систему координат вы
хотите использовать для обозначения текстелов на изображении. Если это поле имеет значение VK_TRUE,
то вы можете просто использовать координаты в [0, texWidth) и [0, texHeight)
диапазон. Если это так, то текстели адресуются с помощью [0, 1] дальность действия
по всем осям. В реальных приложениях почти всегда используются нормализованные координаты,
потому что тогда можно использовать текстуры разного разрешения с точно
одинаковыми координатами.
```

```
1 samplerInfo.compareEnable = VK_FALSE;
2 samplerInfo.CompareOp = VK_COMPARE_OP_WAYS;

Если функция сравнения включена, то сначала текстели будут сравниваться со значением,
и результат этого сравнения используется в операциях фильтрации. В основном это
используется для фильтрации теневых карт с процентным приближением. Мы рассмотрим это в
следующей главе.
```

```
1 samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
2 samplerInfo.mipLodBias = 0.0f;
3 samplerInfo.minLod = 0.0f;
4 samplerInfo.maxLod = 0.0f;
```

Все эти поля применимы к `mipmapping`. Мы рассмотрим `mipmapping` в следующей  
главе, но в основном это другой тип фильтра, который можно применить.  
Функционирование сэмплера теперь полностью определено. Добавьте член класса для удержания  
дескриптора объекта `sampler` и создайте сэмплер с помощью `vkCreateSampler`:

```

1 VkImageView textureImageView;
2 VkSampler textureSampler;
3
4 ...
5
6 пустота createTextureSampler() {
7 ...
8
9 if (vkCreateSampler(устройство, &samplerInfo, nullptr,
10 &textureSampler) != VK_SUCCESS) {
11 выбросить std::runtime_error("не удалось создать текстуру
12 пробоотборник!");
13 }
14 }
```

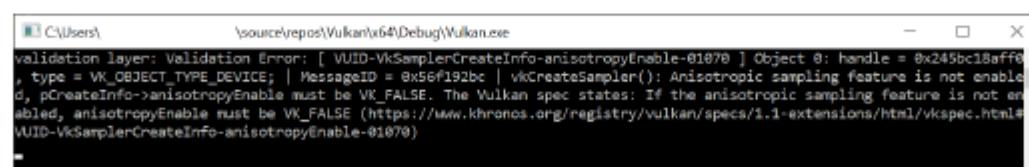
Обратите внимание, что сэмплер нигде не ссылается на `VkImage`. Сэмплер представляет собой объект `distinct`, который предоставляет интерфейс для извлечения цветов из текстуры. Его можно применить к любому изображению, которое вы хотите, будь то 1D, 2D или 3D. Это отличается от многих старых API, которые объединяли изображения текстур и фильтрацию в единое состояние. Уничтожьте пробоотборник в конце программы, когда у нас больше не будет доступа к изображению:

```

1 аннулирование очистка () {
2 cleanupSwapChain();
3
4 vkDestroySampler(устройство, textureSampler, nullptr);
5 vkDestroyImageView(устройство, textureImageView, nullptr);
6
7 ...
8 }
```

## Функция устройства с анизотропией

Если вы запустите свою программу прямо сейчас, вы увидите сообщение уровня проверки, подобное этому:



Это потому, что анизотропная фильтрация на самом деле является дополнительной функцией устройства. Нам нужно обновить `createLogicalDevice` функцию

```

1 для запроса: VkPhysicalDeviceFeatures deviceFeatures{};
2 deviceFeatures.samplerAnisotropy = VK_TRUE;
```

И хотя маловероятно, что современная видеокарта не будет поддерживать это, нам следует обновить `isDeviceSuitable` чтобы проверить, доступно ли оно:

```
1 bool isDeviceSuitable(физическое устройство VkPhysicalDevice) {
2 ...
3
4 Поддерживаемые функции VkPhysicalDeviceFeatures;
5 vkGetPhysicalDeviceFeatures(устройство и поддерживаемые функции);
6
7 Возврат индексы.IsComplete() && поддерживаются расширения &&
8 swapChainAdequate && поддерживаемые функции.Анизотропия выборки;
}
```

В `vkGetPhysicalDeviceFeatures` изменяет назначение `VkPhysicalDeviceFeatures` создайте структуру, указывающую, какие функции поддерживаются, а не запрашиваются, путем установки логических значений. Вместо того, чтобы обеспечивать доступность анизотропной фильтрации, также можно просто не использовать ее, установив условно:

```
1 samplerInfo.anisotropyEnable = VK_FALSE;
2 samplerInfo.maxAnisotropy = 1, 0 f;
```

В следующей главе мы подвернем объекты изображения и сэмплера воздействию шейдеров, чтобы нарисовать текстуру на квадрате.

Код на C++ / Вершинный шейдер / Фрагментный шейдер

## Комбинированный сэмплер изображений

### Введение

Впервые мы рассмотрели дескрипторы в разделе "Единообразные буферы" Учебник. В этой главе мы рассмотрим новый тип дескриптора: *комбинированный сэмплер изображений*. Этот дескриптор позволяет шейдерам получать доступ к возрастному ресурсу через объект `sampler`, подобный тому, который мы создали в предыдущей главе. Мы начнем с изменения макета дескриптора, пул дескрипторов и набора дескрипторов, чтобы включить такой комбинированный дескриптор для выборки изображений. После этого мы собираемся добавить координаты текстуры и модифицировать шейдер фрагмента, чтобы считывать цвета текстуры вместо простой интерполяции цветов вершин.

### Обновление дескрипторов

Перейдите к функции `createDescriptorSetLayout` и добавьте привязку `vkDescriptorSetLayout` для комбинированного дескриптора средства выборки изображений. Мы просто поместим его в привязку после однородного буфера:

```

1 VkDescriptorSetLayoutBinding samplerLayoutBinding{};
2 samplerLayoutBinding.binding = 1;
3 samplerLayoutBinding.descriptorCount = 1;
4 samplerLayoutBinding.descriptorType =
 VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
5 Привязка к образцу.pImmutableSamplers = nullptr;
6 Пример привязки.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
7
8 std:: массив<VkDescriptorSetLayoutBinding, 2>
привязки =
 {uboLayoutBinding, samplerLayoutBinding};
9 VkDescriptorSetLayoutCreateInfo layoutInfo{}; layoutInfo.Тип
10 =
 VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
11 layoutInfo.bindingCount = static_cast<uint 3 2 _t>(привязки.size());
12 layoutInfo.pBindings = привязки.data();

```

Обязательно установите `stageFlags`, чтобы указать, что мы намерены использовать объединенный дескриптор семплера изображений в шейдере фрагментов. Именно там будет определен цвет фрагмента. Можно использовать выборку текстуры в вершинном шейдере, например, для динамической деформации сетки вершин с помощью карты высот. Мы также должны создать больший дескриптора бассейн, чтобы освободить место для размещения-ного комбинированного изображения сэмплеров, добавив еще один `VkPoolSize` тип

`VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` в файл `VkDescriptorPoolCreateInfo`.

Перейдите к функции `createDescriptorPool` и измените ее, чтобы включить

`VkDescriptorPoolSize` для этого дескриптора:

```

1 std:: массив<VkDescriptorPoolSize, 2> poolSizes{};
2 poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
3 poolSizes[0].descriptorCount =
 static_cast<uint 3 2 _t>(MAX_FRAMES_IN_FLIGHT);
4 poolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
5 poolSizes[1].descriptorCount =
 static_cast<uint 3 2 _t>(MAX_FRAMES_IN_FLIGHT);
6
7 VkDescriptorPoolCreateInfo poolInfo{};
8 poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
9 poolInfo.poolSizeCount = static_cast<uint 3 2 _t>(poolSizes.size());
10 poolInfo.pPoolSizes = poolSizes.data();
11 poolInfo.maxSets = static_cast<uint 3 2 _t>(MAX_FRAMES_IN_FLIGHT);

```

Неадекватные пулы дескрипторов являются хорошим примером проблемы, которую не улавливают уровни проверки: Начиная с версии Vulkan 1.1, `vkAllocateDescriptorSets` может произойти сбой с кодом ошибки `VK_ERROR_POOL_OUT_OF_MEMORY` если пул не достаточно большой, но драйвер также может попытаться решить проблему внутренне. Это означает, что иногда (в зависимости от оборудования, размера пула и распределения

размер) драйвер позволит нам обойтись без выделения, превышающего пределы нашего пула дескрипторов. В других случаях, `vkAllocateDescriptorSets` потерпит `VK_ERROR_POOL_OUT_OF_MEMORY`.  
 Распределение выполняется успешно на некоторых машинах, но не выполняется на других.  
 поскольку Vulkan перекладывает ответственность за распределение на драйвер,  
 больше нет строгого требования выделять только столько дескрипторов  
 определенного типа (и т.д.). Сколько указано  
~~для~~  
 соответствующими участниками `descriptorCount` для создания  
 пула дескрипторов. Тем не менее, наилучшей практикой остается делать это и  
 в будущем, `VK_LAYER_KHRONOS_validation` предупредит об этом типе  
 проблемы, если вы включите проверку наилучших практик. Последний шаг  
 - привязать фактические ресурсы изображения и сэмплера к дескрипторам из  
 набора дескрипторов. Перейдите к разделу `createDescriptorSets` функция.

```

1 для (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
2
3 VkDescriptorBufferInfo BufferInfo{};
4
5 BufferInfo.buffer = Равномерные
6 буферы[i]; BufferInfo.offset = 0;
7 BufferInfo.range = sizeof(UniformBufferObject);
8
9 VkDescriptorImageInfo ImageInfo{};
10
11 ImageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
12 ImageInfo.ImageView = textureImageView;
13 ImageInfo.sampler = textureSampler;
14
15 ...
16
17 }
```

Ресурсы для структуры комбинированного сэмплера изображений должны быть  
 указаны в `VkDescriptorImageInfo` struct, точно так же, как ресурс буфера для  
 однородного буфера дескриптор указан в `a VkDescriptorBufferInfo` struct.  
 Здесь объекты из предыдущей главы собираются вместе.

```

1 std::array<VkWriteDescriptorSet, 2> descriptorWrites{};
2
3 descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
4 descriptorWrites[0].dstSet = descriptorSets[i];
5 descriptorWrites[0].dstBinding = 0;
6 descriptorWrites[0].dstArrayElement = 0;
7 descriptorWrites[0].descriptorType =
8 VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
9 descriptorWrites[0].descriptorCount = 1;
10 descriptorWrites[0].pBufferInfo = &BufferInfo;
11
12 descriptorWrites[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
13 descriptorWrites[1].dstSet = descriptorSets[i];
```

```

13 descriptorWrites[1].dstBinding = 1;
14 descriptorWrites[1].dstArrayElement = 0;
15 descriptorWrites[1].descriptorType =
16 VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
16 descriptorWrites[1].descriptorCount = 1;
17 descriptorWrites[1].pImageInfo = &ImageInfo;
18
19 Обновленные наборы скрипторов (устройство,
20 static_cast<uint 3 2 _t>(descriptorWrites.size()),
21 descriptorWrites.data(), 0, nullptr);

```

Дескрипторы должны быть обновлены этой информацией об изображении, так же как и буфер. На этот раз мы используем массив `pImageInfo` ВМЕСТО `pBufferInfo`. Дескрипторы теперь готовы к использованию шейдерами!

## Координаты текстуры

Существует один важный компонент для отображения текстур, которого все еще не хватает, и это фактические координаты для каждой вершины. Координаты определяют, как изображение фактически сопоставляется с геометрией.

```

1 структура Вершина {
2
3 glm:: vec 2 pos;
4
5 glm:: vec 3 color;
6
7 glm:: vec 2 texCoord;
8
9
10 статический VkVertexInputBindingDescription getBindingDescription() {
11
12 VkVertexInputBindingDescription bindingDescription{};
13
14 bindingDescription.binding = 0;
15 bindingDescription.stride = sizeof(Вершина);
16
17 Описание привязки.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
18
19
20 Возврат Описание привязки;
21
22 }
23
24
25 статический std::массив<VkVertexInputAttributeDescription, 3 >
26
27 getAttributeDescriptions() {
28
29 std::array<VkVertexInputAttributeDescription, 3 >
30
31 Атрибутивные описания{};
32
33
34 Атрибутивные описания[0].привязка = 0;
35
36 Атрибутивные описания[0].местоположение = 0;
37
38 Атрибутивные описания[0].format = VK_FORMAT_R3_2_G3_2_SFLOAT;
39
40 Атрибутивные описания[0].offset = offsetof(Вершина, точка доступа);
41
42
43 Атрибутивные описания[1].привязка = 0;

```

```

24 Атрибутивные описания[1].местоположение = 1 ;
25 Атрибутивные описания[1].format = VK_FORMAT_R3_2_G3_2_B3_2_SFLOAT;
26 Атрибутивные описания[1].offset = смещение (вершина, цвет);
27
28 attributeDescriptions[2].привязка = 0 ;
29 attributeDescriptions[2].местоположение = 2 ;
30 Атрибутивные описания[2].format = VK_FORMAT_R3_2_G3_2_SFLOAT;
31 Атрибутивные описания[2].offset = offsetof(Вершина, texCoord);
32
33 Возврат attributeDescriptions;
34 }
35 };

```

Измените структуру `Vertex`, чтобы включить `a_vec2` для координат текстуры.

Убедитесь, что также добавьте `a_vkVertexInputAttributeDescription` чтобы мы могли использовать `access` координаты текстуры в качестве входных данных в вершинном шейдере. Это необходимо для того, чтобы иметь возможность передавать их в шейдер фрагментов для интерполяции по поверхности

```

1 квадрата.const std::вектор<Вершина> vertices = {
2 {{-0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}, {1.0f, 0.0f}},
3 {{0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}, {0.0f, 0.0f}},
4 {{0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}, {0.0f, 1.0f}},
5 {{-0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}, {1.0f, 1.0f}}
6 };

```

В этом уроке я просто заполнил квадрат текстурой, используя координаты из `0`, `0` в левом верхнем углу, чтобы `1`, `1` в правом нижнем углу. Не стесняйтесь экспериментировать с разными координатами. Попробуйте использовать координаты, приведенные ниже `0` или выше `1` чтобы увидеть режимы адресации в действии!

## Шейдеры

Последним шагом является изменение шейдеров для выборки цветов из текстуры. Нам сначала нужно изменить вершинный шейдер, чтобы передать координаты текстуры в фрагментный

```

1 шейдер:макет(location = 0) в vec2 inPosition;
2 макет(location = 1) в vec3 inColor;
3 макет(location = 2) в vec2 inTexCoord;
4
5 layout(location = 0) выводит vec3 FragColor;
6 layout(location = 1) выводит vec2 fragTexCoord;
7
8 аннулирует main() {
9 gl_Position = ubo.проект * ubo.вид * ubo.модель * vec4 (исходное положение,
0 . 0 , 1 . 0);

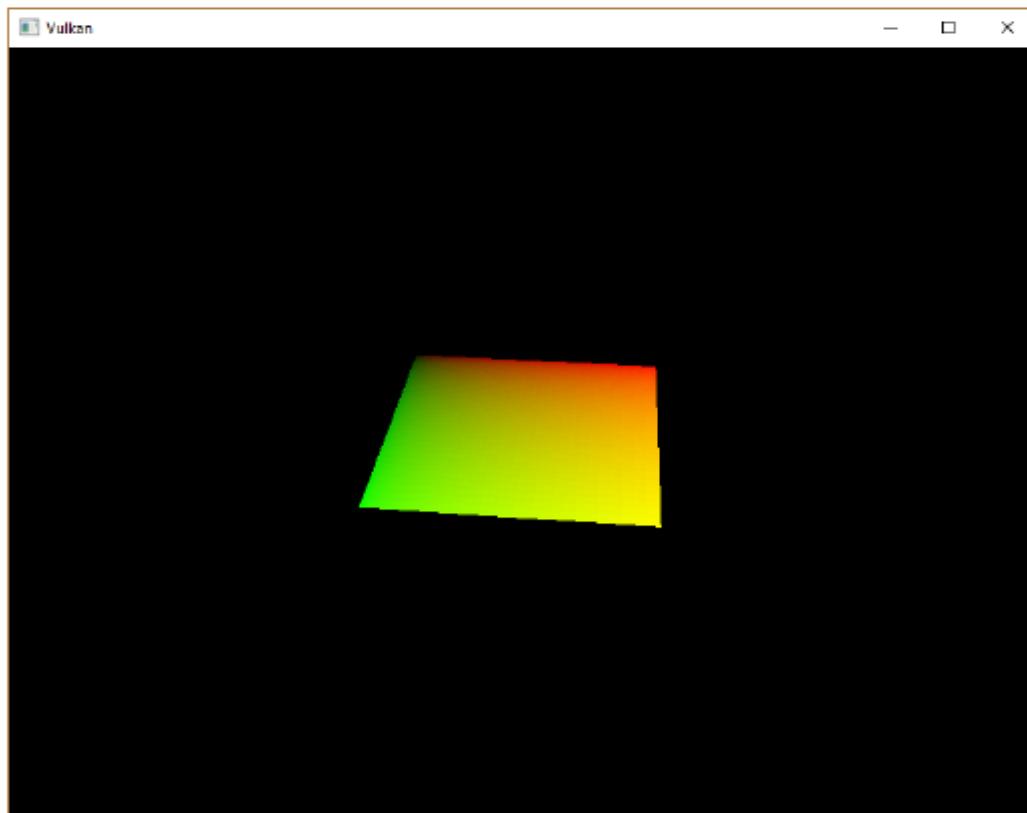
```

```
10 FragColor = нецветный;
11 fragTexCoord = intTexCoord;
12 }
```

Точно так же, как и цвета для каждой вершины, `fragTexCoord` растеризатор будет плавно чередовать значения по площади квадрата. Мы можем визуализировать это, заставив фрагментный шейдер выводить координаты

```
1 текстуры в виде цветов: # версия 4 5 0
2
3 макет (location = 0) в vec 3 FragColor;
4 макет (location = 1) в vec 2 fragTexCoord;
5
6 макет (location = 0) вне vec 4 outColor;
7
8 анулирует main() {
9 Внешний цвет = vec 4 (fragTexCoord, 0.0, 1.0);
10 }
```

Вы должны увидеть что-то вроде изображения ниже. Не забудьте перекомпилировать шейдеры!



Зеленый канал представляет горизонтальные координаты, а красный канал вертикальные координаты. Черный и желтый углы подтверждают, что текст-

координаты тюра корректно интерполируются из `0 , 0` для `1 , 1` поперек квадрата.

Визуализация данных с использованием цветов - эквивалент шейдерного программирования для `printf` отладка, за неимением лучшего варианта!

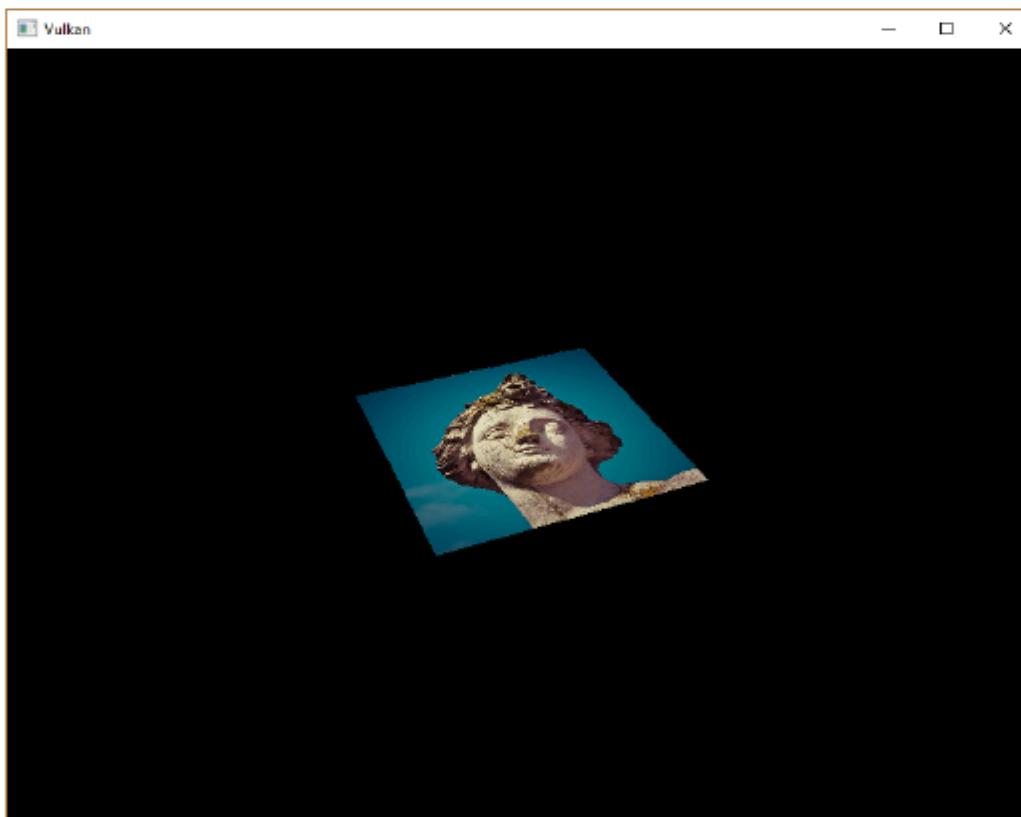
Комбинированный дескриптор сэмплера изображений представлен в GLSL с помощью сэмплера `uniform`. Добавьте ссылку на него в шейдере фрагмента:

```
1 макет(привязка = 1) uniform sampler 2D texSampler;
```

Существуют эквивалентные типы `sampler 1D` и `sampler 3D` для других типов изображений. Убедитесь, что здесь используется правильная привязка.

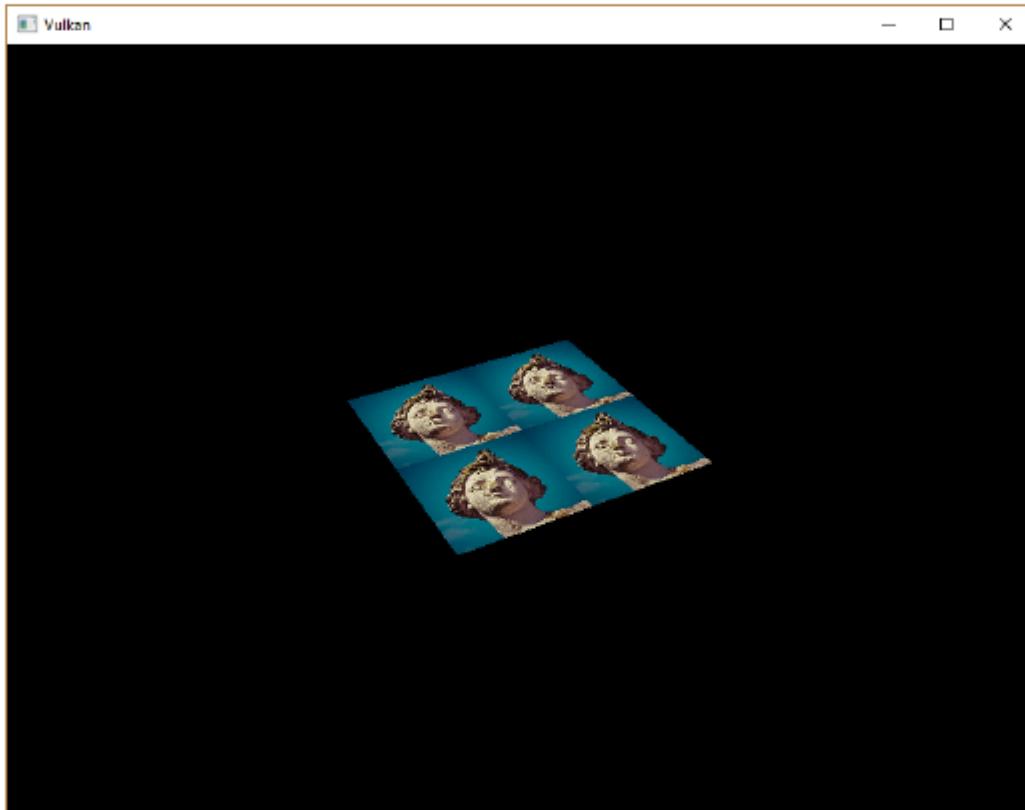
```
1 void main() {
2 outColor = текстура(texSampler, fragTexCoord);
3 }
```

Текстуры отбираются с помощью встроенной функции `texture`. В качестве аргументов используется `sampler` и `coordinate`. Сэмплер автоматически выполняет фильтрацию и преобразования в фоновом режиме. Теперь вы должны видеть текстуру на квадрате при запуске приложения.:



Попробуйте поэкспериментировать с режимами адресации, масштабируя координаты текстуры до значений, превышающих `1`. Например, следующий фрагментный шейдер выдает результат на изображении ниже при использовании `VK_SAMPLER_ADDRESS_MODE_REPEAT`:

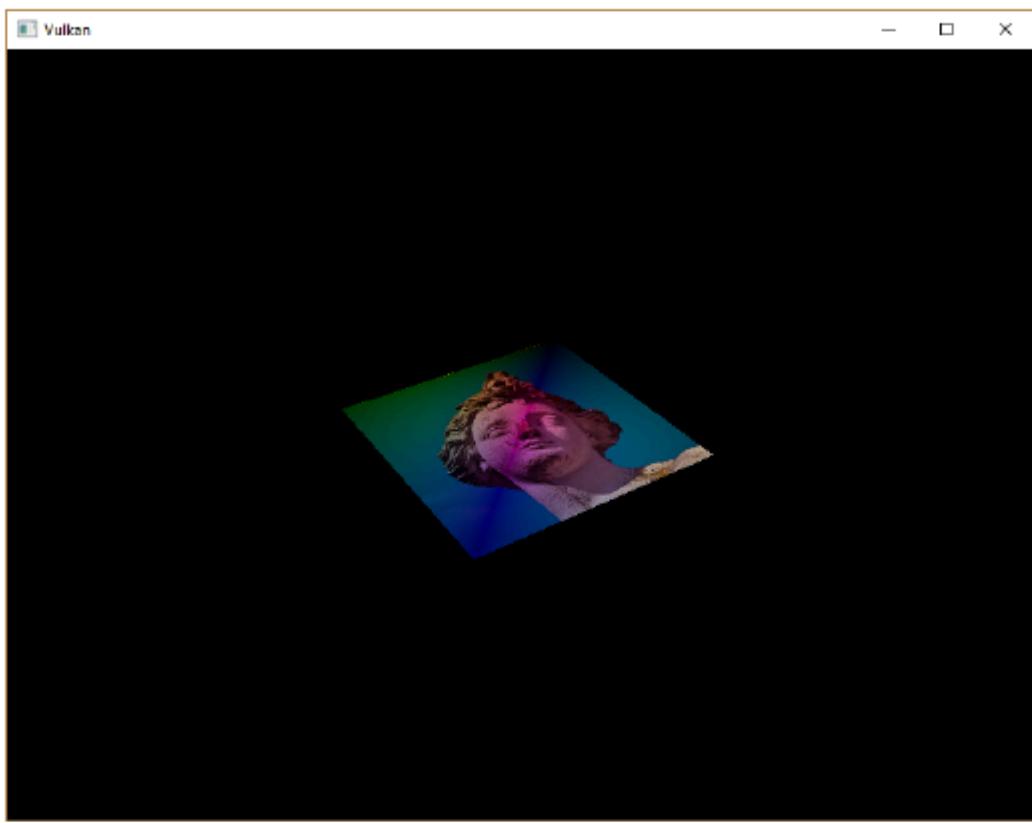
```
1 void main() {
2 outColor = текстура(texSampler, fragTexCoord * 2.0);
3 }
```



Вы также можете манипулировать цветами текстуры, используя

```
1 цвета вершин: void main() {
2 outColor = vec4(фрагКолор * текстура(texSampler,
3 fragTexCoord).rgb, 1.0);
4 }
```

Я разделил здесь RGB- и альфа-каналы, чтобы не масштабировать альфа-канал.



Теперь вы знаете, как получить доступ к изображениям в шейдерах! Это очень мощный метод в сочетании с изображениями, которые также записываются в фреймбуферы. Вы можете использовать эти изображения в качестве входных данных для реализации интересных эффектов, таких как постобработка и отображение камеры в 3D-мире. Код на C++ / Вершинный шейдер / Фрагментный шейдер

# Буферизация глубины

## Введение

Геометрия, с которой мы работали до сих пор, проецируется в 3D, но она по-прежнему остается полностью плоской. В этой главе мы собираемся добавить координату Z к расположению, чтобы подготовиться к созданию 3D-сеток. Мы будем использовать эту третью координату для размещения квадрата поверх текущего квадрата, чтобы увидеть проблему, возникающую, когда геометрия не сортируется по глубине.

## ТРЕХМЕРНАЯ геометрия

```
struct для использования 3D-вектора для позиции и обновления
Вершины
ВСООТВЕТСТВУЮЩЕМ ОПИСАНИИ VkVertexInputAttributeDescription :Измените

1 struct Vertex {
2 glm:: vec 3 pos;
3 glm:: vec 3 color;
4 glm:: vec 2 texCoord;
5
6 ...
7
8 статический std::массив<VkVertexInputAttributeDescription, 3>
9 getAttributeDescriptions() {
10 std::array<VkVertexInputAttributeDescription, 3>
11 Атрибутивные описания{};
12
13 Атрибутивные описания[0].привязка = 0 ;
14 Атрибутивные описания[0].местоположение = 0 ;
15 Атрибутивные описания[0].format = VK_FORMAT_R3_2_G3_2_B3_2_SFLOAT;
16 Атрибутивные описания[0].offset = смещение (вершина, точка доступа);
17
18 ...
19 }
20 };
21 }
```

Затем обновите вершинный шейдер, чтобы он принимал и преобразовывал

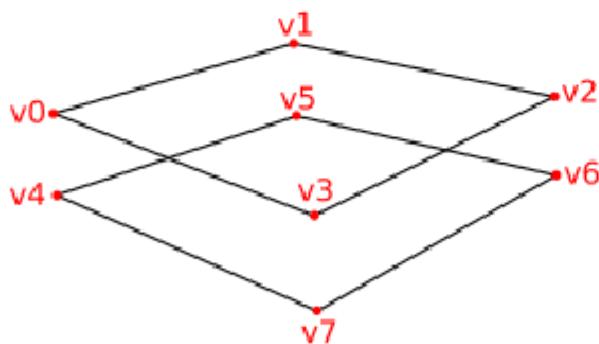
3D-координаты в качестве входных данных. Не забудьте потом

```
1 перекомпилировать его! layout(location = 0) in vec 3 inPosition;
2 ...
3 void main() {
4 gl_Position = ubo.проект * ubo.вид * ubo.модель * vec 4 (исходное положение,
5 1.0);
6 фрагКолор = inColor;
7 фрагТекСоординаты = inTexCoord;
8 }
9 }
```

Наконец, обновите контейнер `vertices`, чтобы включить

```
1 координаты Z: const std::vector<Вершина> вершины = {
2 {{-0.5f, -0.5f, 0.0f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f, 0.0f}},
3 {{0.5f, -0.5f, 0.0f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f, 0.0f}},
4 {{0.5f, 0.5f, 0.0f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f, 0.0f}},
5 {{-0.5f, 0.5f, 0.0f}, {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f, 1.0f}}
6 };
```

Если вы запустите свое приложение сейчас, то увидите точно такой же результат, как раньше. Пришло время, чтобы добавить некоторые дополнительные геометрии, чтобы сделать сцену более интересной, и, чтобы продемонстрировать проблему, которую мы собираемся решить в этой главе. Дублируйте вершины, чтобы определить позиции для квадрата прямо под текущим таким образом:



Используйте координаты Z из  $-0.5f$  и добавьте соответствующие индексы для дополнительного квадрата:

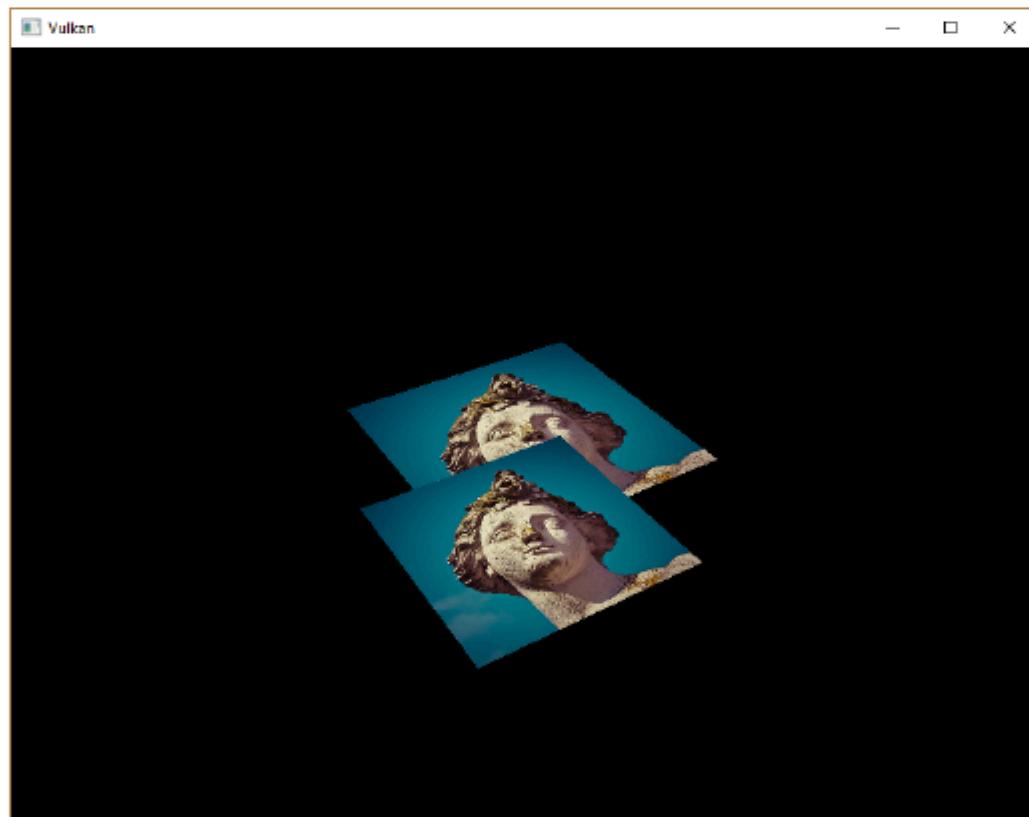
```
1 const std::vector<Вершина> vertices = {
2 {{-0.5f, -0.5f, 0.0f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f, 0.0f}},
```

```

3 {{ 0 . 5 f, - 0 . 5 f, 0 . 0 f}, { 0 . 0 f, 1 . 0 f, 0 . 0 f}, { 1 . 0 f, 0 . 0 f}},
4 {{ 0 . 5 f, 0 . 5 f, 0 . 0 f}, { 0 . 0 f, 0 . 0 f, 1 . 0 f}, { 1 . 0 f, 1 . 0 f}},
5 {{- 0 . 5 f, 0 . 5 f, 0 . 0 f}, { 1 . 0 f, 1 . 0 f, 1 . 0 f}, { 0 . 0 f, 1 . 0 f}},
6
7 {{- 0 . 5 f, - 0 . 5 f, - 0 . 5 f}, { 1 . 0 f, 0 . 0 f, 0 . 0 f}, { 0 . 0 f, 0 . 0 f}},
8 {{ 0 . 5 f, - 0 . 5 f, - 0 . 5 f}, { 0 . 0 f, 1 . 0 f, 0 . 0 f}, { 1 . 0 f, 0 . 0 f}},
9 {{ 0 . 5 f, 0 . 5 f, - 0 . 5 f}, { 0 . 0 f, 0 . 0 f, 1 . 0 f}, { 1 . 0 f, 1 . 0 f}},
10 {{- 0 . 5 f, 0 . 5 f, - 0 . 5 f}, { 1 . 0 f, 1 . 0 f, 1 . 0 f}, { 0 . 0 f, 1 . 0 f}}
11 };
12
13 постоянный std::векторные индексы<uint 1 6 _t> = {
14 0 , 1 , 2 , 2 , 3 , 0 ,
15 4 , 5 , 6 , 6 , 7 , 4
16 };

```

Запустите сейчас свою программу, и вы увидите нечто, напоминающее иллюстрацию Эшера:



Проблема в том, что фрагменты нижнего квадрата рисуются поверх фрагментов верхнего квадрата просто потому, что он появляется позже в индексном массиве. Есть два способа решить эту проблему:

- Отсортируйте все вызовы draw по глубине от начала к началу

• Используйте глубинное тестирование с буфером глубины

Первый подход обычно используется для рисования прозрачных объектов, потому что прозрачность, не зависящая от порядка, является сложной задачей для решения. Однако проблема упорядочивания фрагментов по глубине гораздо чаще решается с использованием буфера глубины. Буфер глубины - это дополнительное вложение, которое сохраняет глубину для каждой позиции, точно так же, как вложение цвета хранит цвет каждой позиции. Каждый раз, когда растеризатор создает фрагмент, тест глубины проверяет, находится ли новый фрагмент ближе, чем предыдущий. Если это не так, то новый фрагмент отбрасывается. Фрагмент, прошедший проверку глубины, записывает свою собственную глубину в буфер глубины. Этим значением можно манипулировать с помощью шейдера фрагментов, точно так же, как вы можете манипулировать выводом цвета.

```
1 # define GLM_FORCE_RADIANS
2 # определить GLM_FORCE_DEPTH_ZERO_TO_ONE
3 # включить <glm/glm.hpp>
4 # включить <glm/gtc/matrix_transform.hpp>
```

Матрица перспективной проекции, сгенерированная GLM, будет использовать OpenGL диапазон глубины - `1 . 0` Для `1 . 0` по умолчанию. Нам нужно настроить его на использование диапазона Vulkan `0 . 0` Для `1 . 0` используя определение `GLM_FORCE_DEPTH_ZERO_TO_ONE`.

## Изображение глубины и вид

Вложение глубины основано на изображении, точно так же, как вложение цвета. Разница в том, что цепочка обмена не будет автоматически создавать изображения глубины для нас. Нам нужно только одно изображение глубины, потому что одновременно выполняется только одна операция рисования. Для создания изображения глубины снова потребуется три вида

```
1 ресурсов: изображение, память и вид изображения. VkImage DepthImage;
2 VkDeviceMemory depthImageMemory;
3 VkImageView depthImageView;
```

Создайте новую функцию `createDepthResources` для настройки этих

```
1 ресурсов: аннулирование Инициализированный вулкан() {
2 ...
3 createCommandPool();
4 createDepthResources();
5 createTextureImage(); ...
6 }
7 ...
8
9
10 void createDepthResources()
{ 11 }
```

```
12
13 }
```

Создание изображения глубины довольно просто. Оно должно иметь то же разрешение, что и цветовое вложение, определяемое экстентом цепочки подкачки, использованием изображения, подходящим для вложения глубины, оптимальной разметкой и локальной памятью устройства. Единственный вопрос: какой формат подходит для изображения глубины? Формат должен содержать компонент глубины, обозначаемый символом `_D??_` в формате `VK_FORMAT_`. В отличие от изображения текстуры, нам не обязательно нужен определенный формат, потому что мы не будем получать прямой доступ к текстурам из программы. Он просто должен иметь разумную точность, по крайней мере, 24 бита является обычным явлением в реальных приложениях. Существует несколько форматов, которые соответствуют этому требованию:

- `VK_FORMAT_D 3 2 _SFLOAT`: 32-разрядный параметр с плавающей точкой для определения глубины
  - `VK_FORMAT_D 3 2 _SFLOAT_S 8 _UINT`: 32-разрядный параметр с плавающей точкой со знаком для определения глубины и 8-разрядный компонент трафарета
  - `VK_FORMAT_D 2 4 _UNORM_S 8 _UINT`: 24-битный параметр с плавающей точкой для определения глубины и 8-битный компонент трафарета
- Компонент трафарета используется для тестов трафарета, который является дополнительным тестом, который можно комбинировать с тестированием глубины. Мы рассмотрим это в
- `VK_FORMAT_D 3 2 _SFLOAT` форматируем,
- потому что поддержка для него чрезвычайно распространена (см. Базу данных оборудования), но приятно добавить дополнительную гибкость в наше приложение, где это возможно. Мы собираемся написать функцию `findSupportedFormat` для этого берется список возможных форматов в порядке от наиболее желательного к наименее желательному и проверяется, какой из них является первым, который поддерживается:

```
VkFormat находит поддержанный формат(const std::vector<VkFormat>& 1
 кандидаты, плитка VkImageTiling, функции VkFormatFeatureFlags)
{
2
3 }
```

Поддержка формата зависит от режима тайлинга и использования, поэтому мы также должны включить их в качестве параметров. Поддержку формата можно запросить с помощью `vkGetPhysicalDeviceFormatProperties` для (формат `VkFormat` : кандидаты) { 1функция:

```
2 VkFormatProperties реквизит;
3 vkGetPhysicalDeviceFormatProperties(физическое устройство, формат,
 и реквизиты);
4 }
```

Свойства `VkFormat` struct содержит три поля:

- `linearTilingFeatures`: Варианты использования, поддерживаемые линейным разбиением на листы

```

•optimalTilingFeatures: Варианты использования, поддерживаемые с помощью
optimal
 разбиение на листы
•: Варианты использования, поддерживаемые для буферов
Здесь уместны только первые два, а тот, который мы проверяем, зависит от разбиения на
листы параметр
ФУНКЦИИ: if (tiling == VK_IMAGE_TILING_LINEAR && (props.linearTilingFeatures
 & особенности) ==
 особенности) { Возврат формат;
} еще, если (tiling == VK_IMAGE_TILING_OPTIMAL &&
 (props.optimalTilingFeatures & features) ==
 особенности) { Возврат формат;
}
}

```

Если ни один из возможных форматов не поддерживает желаемое использование,  
то мы можем либо вернуть специальное значение, либо просто выдать исключение:

```

1 VkFormat находит поддержанный формат(const std::vector<VkFormat>&
 кандидаты, плитка VkImageTiling, функции
 VkFormatFeatureFlags) {
2 для (формат VkFormat : кандидаты) {
3 VkFormatProperties props;
4 vkGetPhysicalDeviceFormatProperties(физическое устройство, формат,
 и реквизиты);
5
6 if (tiling == VK_IMAGE_TILING_LINEAR &&
 (реквизиты.linearTilingFeatures & features) ==
 функции) { Возврат формат;
7 } еще, если (tiling == VK_IMAGE_TILING_OPTIMAL &&
 (props.optimalTilingFeatures & features) ==
 особенности) { Возврат формат;
8 }
9 }
10 }
11 }
12
13 выбросить std::runtime_error("не удалось найти поддерживаемый формат!");
14 }

```

Теперь мы будем использовать эту функцию для создания findDepthFormat вспомогательной функции для просмотра-выберите формат с компонентом глубины, который поддерживает

```

1 Использование в качестве вложения глубины: VkFormat findDepthFormat() {
2 Возврат Найти поддержанный формат(
3 {VK_FORMAT_D_3_2_SFLOAT, VK_FORMAT_D_3_2_SFLOAT_S_8_UINT,
 VK_FORMAT_D_2_4_UNORM_S_8_UINT},
4 VK_IMAGE_TILING_OPTIMAL,
5 VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT - FORMAT_ФУНКЦИИ_DEPTH_STENCIL_ATTACHMENT_BIT
6);

```

```
7 }
```

Обязательно используйте флаг `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT` вместо `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT` в данном случае. Все эти возможные форматы содержат компонент глубины, но последние два также содержат компонент трафарета. Мы пока не будем использовать это, но нам нужно учитывать это при выполнении переходов макета на изображениях в этих форматах. Добавьте простую вспомогательную функцию, которая сообщает нам, содержит ли выбранный формат глубины компонент трафарета:

```
1 bool Имеет_stencilcomponent(формат VkFormat) {
2 Возврат format == VK_FORMAT_D_3_2_SFLOAT_S_8_UINT || формат ==
3 VK_FORMAT_D_2_4_UNORM_S_8_UINT;
4 }
```

Вызовите функцию, чтобы найти формат глубины из `createDepthResources`:

```
1 VkFormat depthFormat = найти_depthformat();
```

Теперь у нас есть вся необходимая информация для вызова наших `createImage` и `createImageView` вспомогательных функций:

```
1 Создайте_изображение(swapChainExtent.ширина, swapChainExtent.высота,
 Формат_глубины, VK_IMAGE_TILING_OPTIMAL,
 VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
 DepthImage, depthImageMemory);
2 depthImageView = createImageView(изображение_глубины, формат_глубины);
```

Однако параметр `createImageView` в настоящее время функция предполагает, что вспомогательным источником всегда является `VK_IMAGE_ASPECT_COLOR_BIT`, поэтому нам нужно будет превратить это поле параметра `into` в:

```
1 VkImageView createImageView(изображение VkImage, формат VkFormat,
 VkImageAspectFlags (Аспектные
 флаги) { ...
2 viewInfo.subresourceRange.aspectMask =
4 aspectFlags; ...
5 }
```

Обновите все вызовы этой функции, чтобы использовать правильный аспект:

```
1 swapChainImageViews[i] = createImageView(swapChainImages[i],
 Формат_файла_swapChainImageFormat, VK_IMAGE_ASPECT_COLOR_BIT);
2 ...
3 depthImageView = createImageView(изображение_глубины, формат_глубины,
 VK_IMAGE_ASPECT_DEPTH_BIT);
4 ...
5 textureImageView = createImageView(Текстурное_изображение,
 VK_FORMAT_R_8_G_8_B_8_A_8_SRGB, VK_IMAGE_ASPECT_COLOR_BIT);
```

Это все для создания изображения глубины. Нам не нужно сопоставлять его или копировать другое изображение, потому что мы собираемся очистить его в начале этапа рендеринга, как вложение с цветом.

## Явный переход изображения глубины

Нам не нужно явно изменять компоновку изображения с учетом глубины, поскольку мы позаботимся об этом на этапе рендеринга. Однако для полноты я все же опишу процесс в этом разделе. Вы можете пропустить его, если хотите. Позвоните по адресу `transitionImageLayout` в конце файла `createDepthResources` функционирует следующим образом:

```
1 transitionImageLayout(изображение глубины, формат глубины,
2 VK_IMAGE_LAYOUT_UNDEFINED,
3 VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL);
```

Неопределенный макет можно использовать в качестве исходного макета, потому что в нем нет содержимого изображения глубины, которое имело бы значение. Нам нужно обновить часть логики в `transitionImageLayout` чтобы использовать правильный аспект вложенного

```
1 ресурса: if (newLayout == VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL) {
2 барьер.Субресурсный диапазон.Аспектная маска = VK_IMAGE_ASPECT_DEPTH_BIT;
3
4 если (hasStencilComponent(format)) {
5 барьер.Субресурсный диапазон.Аспектная маска |=
6 VK_IMAGE_ASPECT_STENCIL_BIT;
7 }
8 } еще {
9 барьер.Субресурсный диапазон.Аспектная маска = VK_IMAGE_ASPECT_COLOR_BIT;
```

Хотя мы не используем компонент трафарета, нам нужно включить его в переходы макета изображения глубины.

Наконец, добавьте правильные маски доступа и этапы конвейера:

```
1 если (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout ==
2 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL)
3 { барьер.srcAccessMask = 0;
4 барьер.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
5
6 Исходная сцена = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
7 Целевая сцена = VK_PIPELINE_STAGE_TRANSFER_BIT;
8 } иначе, если (Старое описание == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL &&
9 newLayout ==
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL) { барьер.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
```

```

9 барьер.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
10
11 Исходная сцена = VK_PIPELINE_STAGE_TRANSFER_BIT;
12 destinationStage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT; }
13 иначе, если (Старое описание == VK_IMAGE_LAYOUT_UNDEFINED && новое описание ==
14 VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL)
15 { барьер.srcAccessMask = 0;
16 барьер.dstAccessMask =
17 VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT | VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
18 Исходная
19 страница = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
20 destinationStage = VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
21 } ещё {
22 выбросить std::invalid_argument ("неподдерживаемый переход макета!");
23}

```

Буфер глубины будет считан для выполнения тестов глубины, чтобы увидеть, виден ли фрагмент, и будет записан в него при отрисовке нового фрагмента. Чтение происходит на этапе, а запись в разделе `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT`. Вы должны выбрать самую раннюю стадию конвейера, соответствующую указанным операциям, чтобы она была готова к использованию в качестве глубинного вложения, когда это необходимо.

## Прохождение рендеринга

Теперь мы собираемся изменить `VkAttachmentDescription`, чтобы включить вложение глубины. Сначала укажите параметр `VkAttachmentDescription`:

```

1 VkAttachmentDescription depthAttachment{};
2 depthAttachment.format = findDepthFormat();
3 depthAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
4 depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
5 depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
6 depthAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
7 depthAttachment.stencilStoreOp =
8 VK_ATTACHMENT_STORE_OP_DONT_CARE; depthAttachment.initialLayout =
9 VK_IMAGE_LAYOUT_UNDEFINED; depthAttachment.finalLayout =
VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

```

Значение `format` должно совпадать с самим изображением глубины. На этот раз мы этого не делаем, позаботьтесь о сохранении данных о глубине (`storeOp`), поскольку они не будут использоваться после завершения черчения `storeOp`. Это может позволить оборудованию выполнять дополнительные операции. Как и для цветового буфера, нам не важна предыдущая глубина содержимого, поэтому мы можем использовать `VK_IMAGE_LAYOUT_UNDEFINED` как `initialLayout`.

```
1 VkAttachmentReference depthAttachmentRef{};

2 depthAttachmentRef.вложение = 1;

3 depthAttachmentRef.макет =
 VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

Добавьте ссылку на вложение для первого (и единственного) подпуска:

```
1 Вспомогательный канал VkSubpassDescription{}; вспомогательный
2 канал.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
3 вспомогательный канал.colorAttachmentCount = 1; вспомогательный
4 канал.pColorAttachments = &colorAttachmentRef;
5 подпасок.pDepthStencilAttachment = &depthAttachmentRef;
```

В отличие от цветных вложений, вспомогательный канал может использовать только одну настройку глубины (+ трафарет). На самом деле не имело бы никакого смысла проводить тесты глубины для нескольких

```
1 буферов.std::array<VkAttachmentDescription, 2> вложения =
 {colorAttachment, depthAttachment};
2 VkRenderPassCreateInfo renderPassCreateInfo{};
3 renderPassCreateInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
4 renderPassCreateInfo.attachmentCount =
 static_cast<uint32_t>(attachments.size());
5 renderPassCreateInfo.Вложения = attachments.data();
6 renderPassCreateInfo.subpassCount = 1;
7 renderPassCreateInfo.pSubpasses = &подпасок;
8 renderPassCreateInfo.dependencyCount = 1;
9 renderPassCreateInfo.pDependencies = &зависимость;
```

Затем обновите структуру `VkSubpassDependency`, чтобы

```
1 ссылаться на оба вложения. зависимость.srcStageMask =
 VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT |
 VK_PIPELINE_STAGE_ARYL_FRAGMENT_TESTS_BIT;
2 зависимость.dstStageMask =
 VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
 | VK_PIPELINE_STAGE_ARYL_FRAGMENT_TESTS_BIT;
3 зависимость.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
 VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT / ДОСТУП_ДЕТАЛИ_;
```

Наконец, нам нужно расширить наши зависимости от подпуска, чтобы убедиться, что нет конфликта между переходом изображения глубины и его очисткой в рамках операции загрузки. Доступ к изображению глубины осуществляется впервые на ранней стадии конвейера тестирования фрагмента, и поскольку у нас есть операция загрузки, которая очищает, мы должны указать маску доступа для записи.

## Фреймбуфер

Следующий шаг - изменить создание фреймбуфера, чтобы привязать изображение глубины к вложению глубины. Перейдите на страницу [createFramebuffers](#) и укажите глубину просмотра изображения в качестве второго вложения:

```
1 std::array<VkImageView, 2> вложения = {
2 swapChainImageViews[i],
3 просмотр depthImageView
4 };
5
6 VkFramebufferCreateInfo framebufferInfo{};
7 framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
8 framebufferInfo.renderPass = renderPass;
9 framebufferInfo.attachmentCount =
10 static_cast<uint32_t>(вложения.size());
11 Информация о фреймбуфере.pAttachments =
12 вложения.data(); framebufferInfo.width =
13 swapChainExtent.ширина; framebufferInfo.height = swapChainExtent.высота;
14 framebufferInfo.layers = 1;
```

Цвет вложение отличается для каждой цепочки буферов изображения, но такой же глубины изображения могут использоваться во всех из них, потому что только один subpass работает в то же время благодаря нашему семафору.

Вам также потребуется переместить вызов на [createFramebuffers](#), чтобы убедиться, что он вызывается после фактического создания изображения

```
1 глубины void Инициализирующий вулкан() {
2 ...
3 createDepthResources();
4 createFramebuffers();
5 ...
6 }
```

## Очистить значения

Поскольку теперь у нас есть несколько вложений с `VK_ATTACHMENT_LOAD_OP_CLEAR`, нам также нужно указать несколько чистых значений. Перейдите к [recordCommandBuffer](#) и создайте массив из структур `VkClearValue`:

```
1 std::vector<VkClearValue, 2> clearValues{};
2 clearValues[0].color = {{0.0f, 0.0f, 0.0f, 1.0f}};
3 clearValues[1].depthStencil = {1.0f, 0};
4
5 renderPassInfo.clearValueCount =
 static_cast<uint32_t>(clearValues.size());
```

```
6 renderPassInfo.pClearValues = Очистить значения.data();
```

Диапазон глубин в буфере глубины находится **0 . 0** Для **1 . 0** в Вулкане, где **1 . 0** лежит в плоскости дальнего обзора и **0 . 0** на плоскости ближнего обзора. Начальное значение в каждой точке в буфере глубины должно быть максимально возможной глубины, что **. 1 . 0** является Обратите внимание, что порядок `clearValues` должно совпадать с порядком, указанным вами. вложения.

## Глубина и состояние трафарета.

Вложение глубины готово к использованию, но проверка глубины все еще должна быть включена в графическом конвейере. Он настраивается через

`VkPipelineDepthStencilStateCreateInfo` структура:

```
1 VkPipelineDepthStencilStateCreateInfo depthStencil{};
2 depthStencil.тип =
3 VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
4 depthStencil.depthTestEnable = VK_TRUE;
5 depthStencil.depthWriteEnable = VK_TRUE;
```

В поле `depthCompareOp` поле указывает, следует ли сравнивать глубину новых фрагментов с буфером глубины, чтобы определить, следует ли их отбросить. Поле `depthWriteEnable` поле указывает, действительно ли новая глубина фрагментов, прошедших тест глубины, должна быть записана в буфер глубины.

```
1 depthStencil.depthCompareOp = VK_COMPARE_OP_LESS;
```

В поле `depthCompareOp` указывается сравнение, которое выполняется для сохранения значения или отбрасываем фрагменты. Мы придерживаемся соглашения о меньшей глубине = ближе, поэтому глубина новых фрагментов должна быть **Меньше**.

```
1 depthStencil.depthBoundsTestEnable = VK_FALSE;
2 depthStencil.minDepthBounds = 0.0f; // Необязательно
3 Ограничение глубины.maxDepthBounds = 1.0f; // Необязательно
```

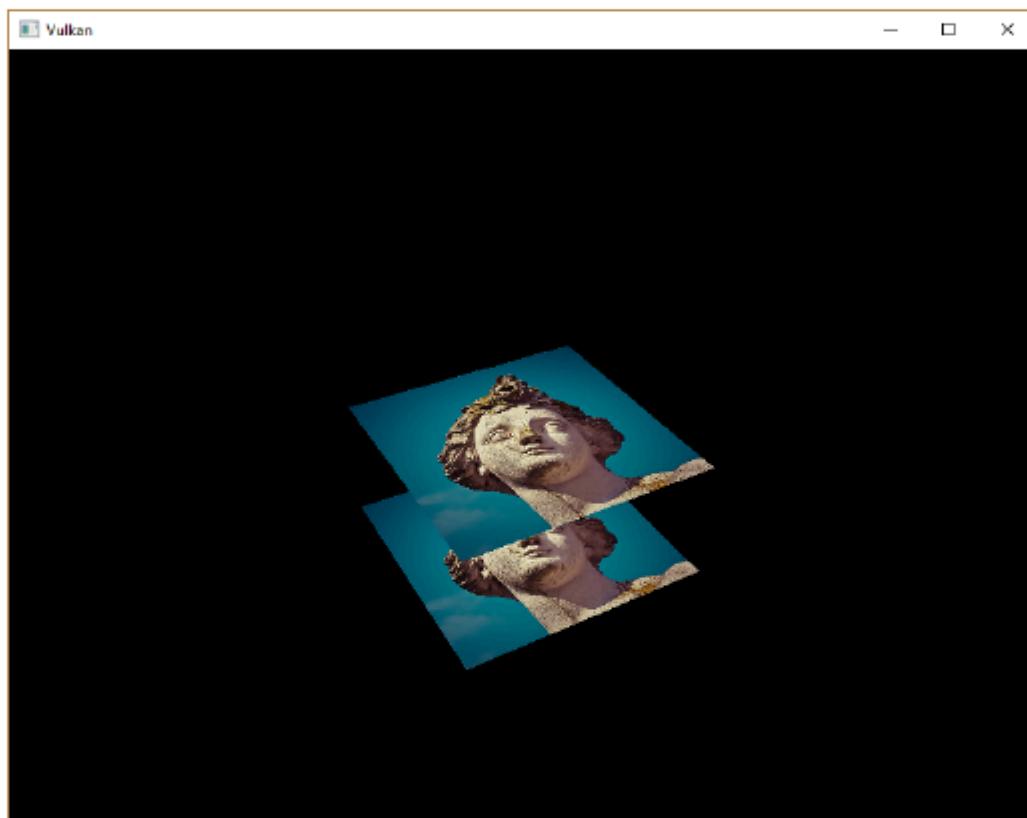
Поля `Проверяемые значения глубины`, `minDepthBounds` и `maxDepthBounds` являются используется для дополнительного теста с привязкой к глубине. По сути, это позволяет вам сохранять только фрагменты, которые попадают в указанный диапазон глубины. Мы не будем использовать эту функциональность.

```
1 depthStencil.stencilTestEnable = VK_FALSE;
2 depthStencil.front = {}; // Необязательно
3 depthStencil.back = {}; // Необязательно
```

Последние три поля настраивают операции с буфером трафарета, которые мы также не будем использовать в этом руководстве. Если вы хотите использовать эти операции, то вам нужно будет убедиться, что формат изображения глубины / трафарета содержит компонент трафарета.

```
1 pipelineInfo.pDepthStencilState = &depthStencil;
```

Обновите структуру `VkGraphicsPipelineCreateInfo`, чтобы ссылаться на глубину `stencilState`, которое мы только что заполнили. Состояние шаблона глубины всегда должно быть указано, если проход рендеринга содержит вложение шаблона глубины. Если вы запустите свою программу сейчас, то увидите, что фрагменты геометрии теперь упорядочены правильно:



## Изменение размера окна обработки

Разрешение буфера глубины должно изменяться при изменении размера окна, чтобы оно соответствовало новому разрешению цветового вложения. Расширьте `recreateSwapChain` функция для воссоздания ресурсов глубины в этом случае:

```
1 void recreateSwapChain() {
2 int ширина = 0, высота = 0;
3 в то время как (width == 0 || height == 0) {
4 glfwGetFramebufferSize(окно, &ширина,
5 &высота); glfwWaitEvents();
6 }
7
8 vkDeviceWaitIdle(устройство);
```

```
9 cleanupSwapChain();
10
11
12 CreateSwapChain(); createImageViews();
13 createDepthResources();
14
15 createFramebuffers();
16 }
```

Операции очистки должны выполняться в функции очистки цепочки обмена:

```
1 void cleanupSwapChain() {
2
3 vkDestroyImageView(устройство, depthImageView, nullptr);
4 vkDestroyImage(устройство, DepthImage, nullptr);
5 vkFreeMemory(устройство, depthImageMemory, nullptr);
6
7 ...
8 }
```

Поздравляем, теперь ваше приложение, наконец, готово к рендерингу произвольной 3D геометрии и приданю ей правильного вида. Мы собираемся опробовать это в следующей главе, нарисовав текстурированную модель! Код на C++ / Вершинный шейдер / Фрагментный шейдер

# Загружаемые модели

## Введение

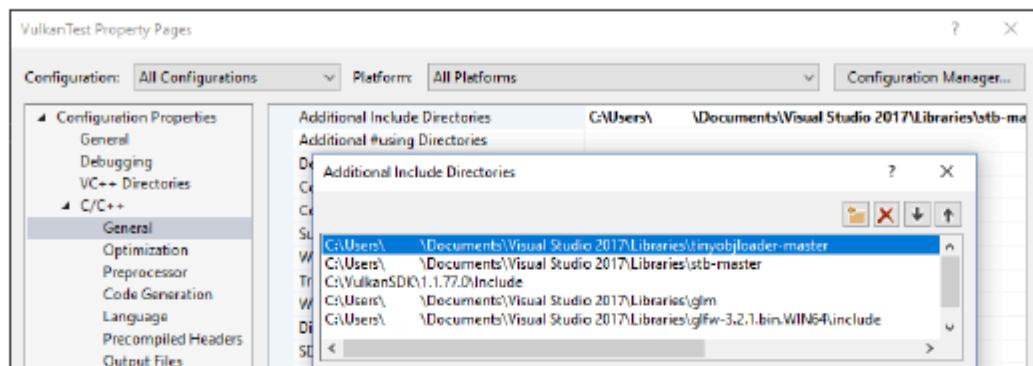
Теперь ваша программа готова к визуализации текстурированных 3D-сеток, но текущая геометрия в вершинах и индексах массивы пока не очень интересны. В этой главе мы собираемся расширить программу для загрузки вершин и индексов из файла реальной модели, чтобы видеокарта действительно выполняла некоторую работу. Во многих руководствах по графическому API читатель может написать свой собственный загрузчик OBJ в главе, подобной этой. Проблема заключается в том, что любому отдаленно интересному 3D приложению вскоре потребуются функции, которые не поддерживаются этим форматом файла, такие как скелетная анимация. Мы будем загружать данные сетки из OBJ-модели в этой главе, но мы больше сосредоточимся на интеграции данных сетки с самой программой, а не на деталях их загрузки из файла.

## Библиотека

Мы будем использовать библиотеку tinyobjloader для загрузки вершин и граней из OBJ-файла. Это быстро и легко интегрируется, потому что это единая файловая библиотека, такая как stb\_image. Перейдите в репозиторий, указанный выше, и загрузите файл `tiny_obj_loader.h` файл в папку в каталоге вашей библиотеки.

### Visual Studio

Добавьте каталог с `tiny_obj_loader.h` в него в дополнительные включают каталоги пути.



### Makefile

Добавьте каталог с помощью tiny\_obj\_loader.h во включаемые каталоги для GCC:

```

1 VULKAN_SDK_PATH = /home/пользователь/VulkanSDK/x.x.x.x/x8_6_6_4
2 STB_INCLUDE_PATH = /home/пользователь/ библиотеки /stb
3 TINYOBJ_INCLUDE_PATH = /home/пользователь/ библиотеки/tinyobjloader
4
5 ...
6
7 CFLAGS = -std= c++ 1 7 -I$(VULKAN_SDK_PATH)/включить
 -I$(STB_INCLUDE_PATH) -I $(TINYOBJ_INCLUDE_PATH)

```

### Пример сетки

В этой главе мы пока не будем включать освещение, поэтому полезно использовать образец модели, в которой освещение встроено в текстуру.

Самый простой способ найти такие модели - поискать 3D-сканы на Sketchfab. Многие модели на этом сайте доступны в формате OBJ с разрешительной лицензией. Для этого урока я решил использовать модель Viking room от nigelgoh (CC BY 4.0). Я изменил размер и ориентацию модели, чтобы использовать ее в качестве замены текущей геометрии:

- viking\_room.obj
- viking\_room.png

Не стесняйтесь использовать свою собственную модель, но убедитесь, что она состоит только из одного материала и имеет размеры примерно 1,5 x 1,5 x 1,5 единицы. Если оно больше этого значения, то вам придется изменить матрицу вида. Поместите файл модели в новый Модели каталог рядом с шейдерами И Текстуры, и поместите текстурное изображение в Текстуры каталог.

Введите в вашу программу две новые переменные конфигурации, чтобы определить пути к модели и текстуре:

```
1 const uint32_t ШИРИНА = 800;
```

```
2 const uint 3 2 _t ВЫСОТА = 6 0 0 ;
3
4 постоянный std::string MODEL_PATH = "модели/viking_room.obj";
5 const стандартный параметр::string TEXTURE_PATH = "текстуры /viking_room.png";
```

И обновите `createTextureImage` чтобы использовать эту переменную `path`:

```
1 stbi_uc * пиксели = stbi_load(TEXTURE_PATH.c_str(), &texWidth,
&texHeight, &текстовые каналы, STBI_rgb_alpha);
```

## Загрузка вершин и индексов

Сейчас мы собираемся загрузить вершины и индексы из файла модели, поэтому вам следует удалить глобальные значения `вершины` и `индексы` теперь массивы. Замените их неконстантными контейнерами в качестве членов класса:

```
1 std::vector<Вершина> вершины;
2 std::vector<uint 3 2 _t> индексы;
3 VkBuffer VertexBuffer;
4 VkDeviceMemory vertexBufferMemory;
```

Вам следует изменить тип индексов `uint 1 6 _t` Для `uint 3 2 _t`, потому что вершин будет намного больше, чем 65535. Не забудьте также изменить параметр `vkCmdBindIndexBuffer` :

```
1 vkCmdBindIndexBuffer(командный буфер, индексный буфер, 0 ,
VK_INDEX_TYPE_UINT 3 2);
```

Библиотека `tinyobjloader` включена таким же образом, как и библиотеки `STB`. Включите в `tiny_obj_loader.h` файл и убедитесь, что вы определили в одном исходном файле, чтобы включить тела функций и избежать ошибок компоновщика: `# define TINYOBJLOADER_IMPLEMENTATION`

```
1 # включить <tinyobj_loader.h>
```

Теперь мы собираемся написать `loadModel` функция, которая использует эту библиотеку для заполнения контейнеров с вершинами и индексами данными вершин из сетки. Это должно быть вызвано где-то перед созданием

```
1 вершинного и индексного буферов: void initVulkan() {
2 ...
3 loadModel();
4 createVertexBuffer();
5 createIndexBuffer();
6 ...
7 }
```

```
9 ...
10
11 пустота loadModel()
12 {
13 }
```

Модель загружается в структуры данных библиотеки путем вызова `tinyobj::LoadObj` функция:

```
1
2 tinyobj::attrib_t атрибут;
3 std::vector<tinyobj::shape_t> фигуры;
4 std::vector<tinyobj::material_t> материалы;
5 std::string предупреждает, ошибка;
6
7 если (!tinyobj::LoadObj(и атрибут, и формы, и материалы, и предупреждение, и ошибка,
8 MODEL_PATH.c_str())) {
9 выбросить std::runtime_error(предупреждение + ошибка);
10 }
11 }
```

OBJ-файл состоит из положений, нормалей, текстурных координат и граней. Границы состоят из произвольного количества вершин, где каждая вершина ссылается на позицию, нормальную и / или текстурную координату по индексу. Это позволяет не только повторно использовать целые вершины, но и отдельные атрибуты.

The `attrib` контейнер содержит все позиции, нормали и координаты текстуры в своих атрибутах. вершинах, атрибутах. нормальях и атрибутах. текстовых координатах векторах.

Контейнер `shapes` содержит все отдельные объекты и их грани. Каждая грань

состоит из массива вершин, и каждая вершина содержит индексы атрибутов положения, нормали и текстурных координат. OBJ-модели также могут определять материал и текстуру для каждой грани, но мы будем их игнорировать.

Строка `err` содержит ошибки, а строка `warn` строка содержит предупреждения, которые появлялись при загрузке файла, например, отсутствующее определение материала. Загрузка только действительно завершилась неудачно, если значение `err` возвращает функцию `LoadObj`.

мы в OBJ-файлах на самом деле могут содержать произвольное количество вершин, тогда как наше приложение может отображать только треугольники. К счастью, параметр `LoadObj` имеет необязательный параметр

параметр для автоматической триангуляции таких граней, который включен по умолчанию.

Мы собираемся объединить все грани в файле в единую модель, поэтому просто выполните итерацию по всем фигурам:

```
1 для (const auto& shape : фигуры) {
2
3 }
```

Функция триангуляции уже убедилась в наличии трех вершин на грани, поэтому теперь мы можем напрямую перебирать вершины и выгружать их прямо в наш вектор:

```
1 для (const auto& shape : фигуры) {
2 для (const auto& index : форма.сетка.индексы) {
3 Вершина, вершина{};
4
5 вершины.push_back(вершина);
6 индексы.push_back(индексы.размер());
7 }
8 }
```

Для простоты мы предположим, что каждая вершина на данный момент уникальна, отсюда и простые индексы автоматического увеличения. ✓ The type auto-increment indexes. The указатель переменная имеет тип `tinyobj::index_t`, который содержит `vertex_index, normal_index` и `texcoord_index` элементы.

. Нам нужно использовать эти индексы для поиска фактических атрибутов вершин в

```
attrib массивы:
vertex.pos = {
1
2 атрибут.вершины[3 * index.vertex_index + 0],
3 атрибут.вершины[3 * index.vertex_index + 1],
4 атрибут.вершины[3 * index.vertex_index + 2]
5 };
6
7 вершина.texCoord = {
8 attrib.texcoords[2 * index.texcoord_index + 0],
9 attrib.texcoords[2 * index.texcoord_index + 1]
10};
11
12 vertex.color = { 1.0f, 1.0f, 1.0f};
```

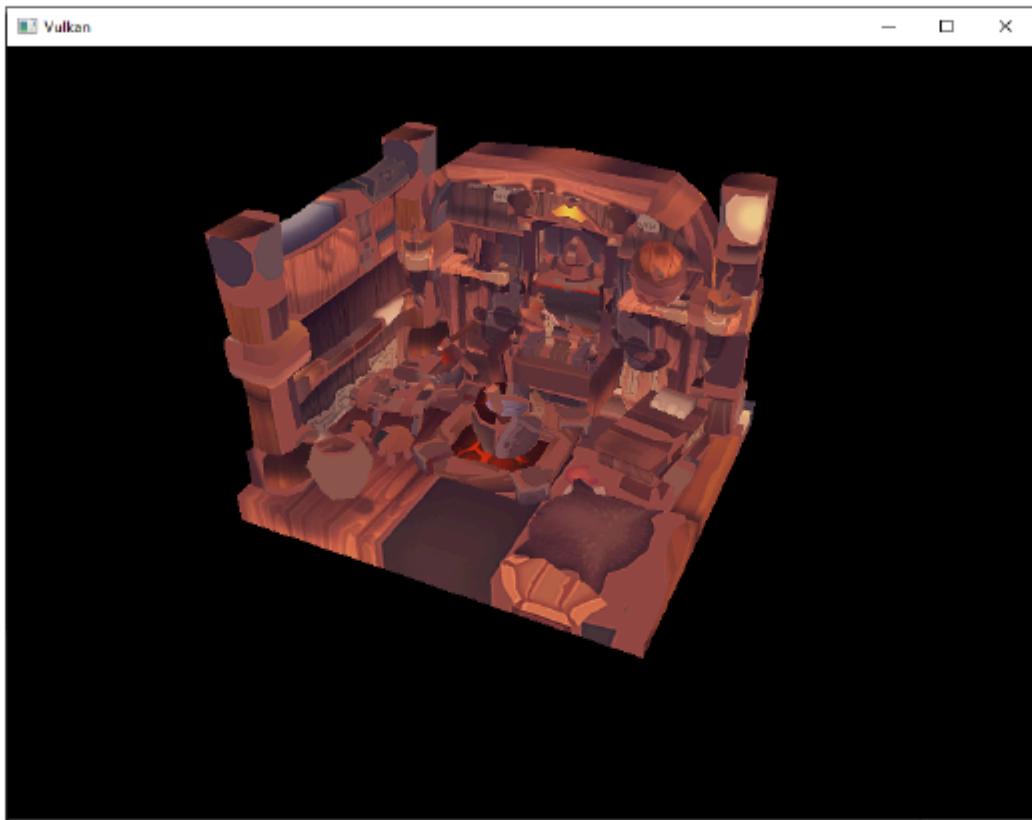
К сожалению, массив `attrib.vertices` представляет собой массив из `float` значения вместо чего-то вроде `glm::vec 3`, поэтому вам нужно умножить индекс на . Аналогично,

3

для каждой записи есть два компонента текстурных координат. Смещения , и `_2` используются для доступа к компонентам X, Y и Z или к компонентам U и V.

0 1

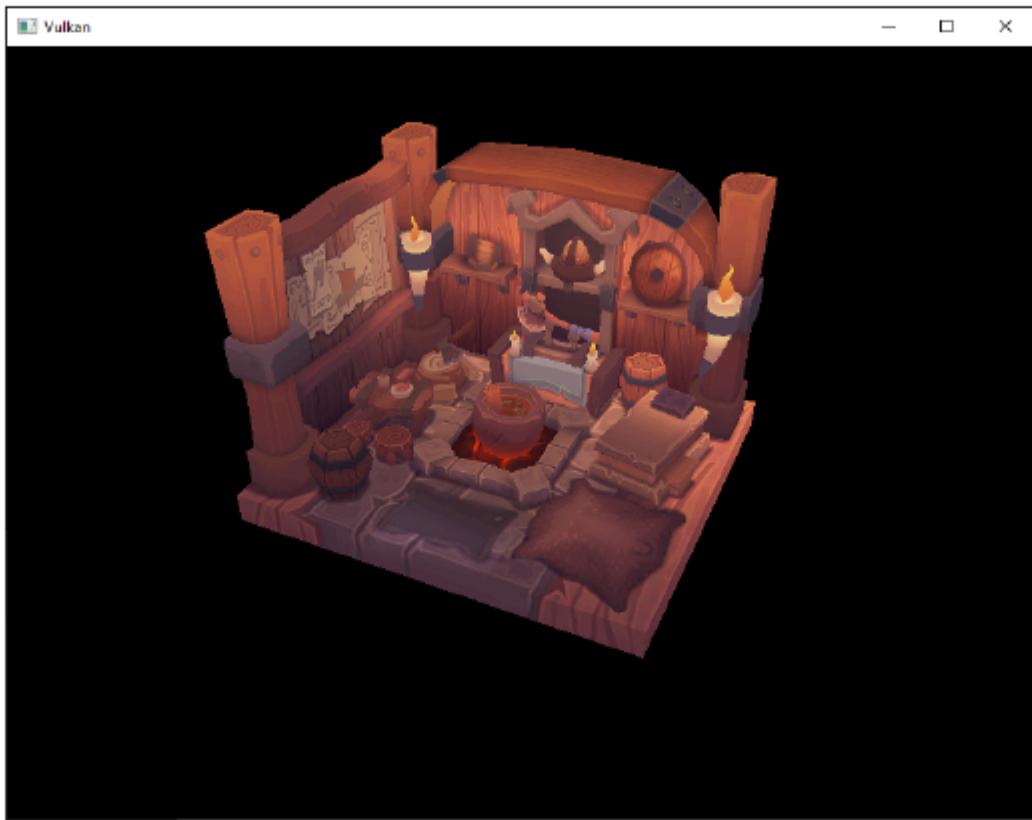
в случае текстурных координат. Запустите вашу программу сейчас с включенной оптимизацией (например, Режим Release в Visual Studio и с флагом компилятора `-O3` для GCC'). Это необходимо, потому что в противном случае загрузка модели будет очень медленной. Вы должны увидеть что-то вроде следующего:



Отлично, геометрия выглядит правильной, но что происходит с текстурой? Формат OBJ предполагает систему координат, в которой вертикальная координата `0` означает нижнюю часть изображения, однако мы загрузили наше изображение в Vulkan в ориентации сверху вниз, где `0` означает верхнюю часть изображения. Решите это, перевернув вертикальную составляющую координат текстуры:

```
1 vertex.texCoord = {
2 атрибут.texcoords[2 * index.texcoord_index + 0],
3 1.0f - атрибут.texcoords[2 * index.texcoord_index + 1]
4 };
```

Когда вы снова запустите свою программу, теперь вы должны увидеть правильный результат:



Вся эта тяжелая работа, наконец, начинает приносить плоды с такой демонстрацией, как эта!

По мере вращения модели вы можете заметить, что задняя часть стенок выглядит немного забавно. Это нормально и просто потому, что модель на самом деле не предназначена для просмотра с этой стороны.

## Дедупликация вершин

К сожалению, мы пока не используем преимущества индексного буфера. Вектор вершины содержит много дублированных данных о вершинах, потому что многие вершины включены в несколько треугольников. Мы должны сохранять только уникальные вершины и использовать индексный буфер для их повторного использования всякий раз, когда они появляются.

Простой способ реализовать это - использовать карту или `unordered_map` чтобы отслеживать

```
1 уникальные вершины и соответствующие индексы:
2
3 # включить <unordered_map>
4 ...
5
6 std::unordered_map<Вершина, uint32_t> Уникальные_вершины{};
7 для (const auto и shape : формы) {
8 для (const auto& index : форма.сетка.индексы) {
```

```

9 Вершина вершина{};
10
11 ...
12
13 if (Количество уникальных вершин.count(vertex) == 0) {
14 Уникальные вершины[vertex] =
15 static_cast<uint 3 2_t>(вершины.size());
16 вершины.push_back(вершина);
17 }
18 индексы.push_back(уникальные вершины[vertex]);
19 }
20 }
```

Каждый раз, когда мы считываем вершину из OBJ-файла, мы проверяем, видели ли мы уже вершину с точно таким же положением и координатами текстуры раньше. Если нет, мы добавляем ее в `вершины` и сохраняем ее индекс в контейнере `uniqueVertices`. После этого мы добавляем индекс новой вершины к индексам. Если мы видели точно такую же вершину раньше, то мы ищем ее индекс в `uniqueVertices` и сохраняем этот индекс в `indexes`. Программа не сможет скомпилироваться прямо сейчас, потому что использование пользовательского типа, такого как наша `struct`, в качестве ключа в хэш-таблице требует от нас реализации двух функций: проверки равенства и вычисления хэша. Первое легко реализовать путем переопределения оператора `==` в `Vertex struct`:

```

1 оператор bool==(const Вершина и другие) const {
2 Возврат pos == другое.pos && color == другое.color && texCoord ==
2 другое.texCoord;
3 }
```

Хэш-функция для `Vertex` реализуется путем указания специального шаблона-изделия для `std::hash<T>`. Хэш-функции - сложная тема, но cppreference.com рекомендует следующий подход, объединяющий поля структуры для создания хэш-функции достойного качества:

```

1 пространство имен std {
2 шаблон<> структура хэш<Вершина> {
3 size_t оператор()(Вершина const& вершина) const {
4 Возврат ((хэш<glm::vec 3>()(vertex.pos) ^
5 (хэш<glm::vec 3>()(вершина.color) << 1)) >> 1) ^
6 (хэш<glm::vec 2>()(вершина.texCoord) << 1);
7 }
8 };
9 }
```

Этот код должен быть размещен за пределами структуры `Vertex`. Хэш-функции для типов GLM должны быть включены с помощью следующего заголовка:

```
1 # определить GLM_ENABLE_EXPERIMENTAL
2 # включить <glm/gtx/hash.hpp>
```

Хэш-функции, определенные в `gtx` папка, что означает, что это технология-  
формально все еще является экспериментальным расширением GLM. Поэтому вам  
нужно определить `GLM_ENABLE_EXPERIMENTAL` для его использования. Это означает, что API может измениться с  
будущем появится новая версия GLM, но на практике API очень стабилен. Теперь  
вы должны быть в состоянии успешно скомпилировать и запустить свою программу.  
Если вы проверите размер `вершин`, тогда вы увидите, что оно уменьшилось с 1  
500 000 до 265 645! Это означает, что каждая вершина используется повторно в  
среднем количестве ~6 треугольников. Это определенно экономит нам много памяти  
графического процессора. Код на C++ / Вершинный шейдер / Фрагментный шейдер

# Генерация Mip-карт

## Введение

Теперь наша программа может загружать и отрисовывать 3D-модели. В этой главе мы добавим еще одну функцию - генерацию mipmap. Mip-карты широко используются в играх и программном обеспечении для рендеринга, и Vulkan дает нам полный контроль над тем, как они создаются.

Mip-карты - это предварительно рассчитанные уменьшенные версии изображения. Каждое новое изображение имеет половину ширины и высоты предыдущего. Mip-карты используются как форма *Уровень детализации* или *LOD*. Объекты, находящиеся далеко от камеры, будут отбирать пробу их текстуры отличаются от изображений mip меньшего размера. Использование изображений меньшего размера увеличивает скорость рендеринга и позволяет избежать артефактов, таких как муаровые узоры. Пример того, как выглядят mip-карты.:



## Создание изображения

В Вулкан для каждого МИП изображений, хранящихся в различных уровнях *Mip* о

.vkImage

Уровень Mip 0 - это исходное изображение, и уровни m<sub>i</sub>p после уровня 0 обычно называются цепочкой m<sub>i</sub>p.

Количество уровней m<sub>i</sub>p указывается, когда vkImage создан. До этого момента мы всегда устанавливали это значение равным единице. Нам нужно рассчитать количество уровней m<sub>i</sub>p на основе размеров изображения.

Сначала добавьте член класса для хранения этого номера:

```
1 ...
2 uint 3 2 _t mipLevels;
3 VkImage TextureImage;
4 ...

Значение для mipLevels можно найти, как только мы загрузим текстуру в
createTextureImage:

инт texWidth, texHeight, texChannels;
1 stbi_uc * пиксели = stbi_load(TEXTURE_PATH.c_str(), &texWidth,
2
3 &texHeight, &texChannels, STBI_rgb_alpha);
4 ...
5 mipLevels =
6 static_cast<uint 3 2 _t>(std::floor(std::log 2 (std::max(texWidth,
7 texHeight))) + 1;
```

При этом вычисляется количество уровней в цепочке m<sub>i</sub>p. Функция max

выбирает самое большое измерение. Функция вычисляет, во сколько раз больше, чем

размер можно разделить на 2.

этаж

Функция обрабатывает случаи, когда

наибольшее измерение не является

1 добавляется так, чтобы исходное изображение имело

степенью 2. уровень m<sub>i</sub>p.

Чтобы использовать это значение, нам нужно изменить createImage, createImageView,

и transitionImageLayout функции, позволяющие нам указывать количество уровней

m<sub>i</sub>p. Добавьте параметр mipLevels к функциям:

```
1 void createImage(uint 3 2 _t ширина, uint 3 2 _t высота, uint 3 2 _t
2
3 Уровни mipLevels, формат VkFormat, тайлинг
4 VkImageTiling, использование VkImageUsageFlags,
5 свойства VkMemoryPropertyFlags, VkImage& image,
6 VkDeviceMemory& imageMemory) { ...
7
8 ImageInfo.mipLevels = mipLevels;
9
10 ...
11 }
```

```
1 VkImageView createImageView(изображение VkImage, формат VkFormat,
2
3 VkImageAspectFlags, aspectFlags, uint 3 2 _t
4 mipLevels) { ...
```

```

3 viewInfo.Вспомогательный ресурс.levelCount =
4 mipLevels; ...
1 недействительный transitionImageLayout(изображение VkImage, формат VkFormat,
VkImageLayout Старое описание, VkImageLayout новое описание,
uint 3 2 _t mipLevels) {
2 ...
3 ...
4 барьер.диапазон вложенных ресурсов.levelCount = mipLevels;
...
Обновите все вызовы этих функций, чтобы использовать правильные
1 значения: createImage(swapChainExtent.ширина, swapChainExtent.высота, 1,
Формат глубины, VK_IMAGE_TILING_OPTIMAL,
VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
DepthImage, depthImageMemory);
2 ...
3 Создайте изображение (texWidth, texHeight, mipLevels, VK_FORMAT_R 8 G 8 B 8 A 8 _SRGB,
VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_TRANSFER_ST_BIT |
VK_IMAGE_USAGE_SAMPLED_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
TextureImage, textureImageMemory);
1 swapChainImageViews[i] = createImageView(swapChainImages[i],
Формат файла swapChainImageFormat, VK_IMAGE_ASPECT_COLOR_BIT, 1);
2 ...
3 depthImageView = createImageView(изображение глубины, формат глубины,
VK_IMAGE_ASPECT_DEPTH_BIT, 1);
4 ...
5 textureImageView = createImageView(текстурное изображение,
VK_FORMAT_R 8 G 8 B 8 A 8 _SRGB, VK_IMAGE_ASPECT_COLOR_BIT, mipLevels);

1 transitionImageLayout(изображение глубины, формат глубины,
VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
1); transitionImageLayout(текстурное
2 ...
3 изображение, VK_FORMAT_R 8 G 8 B 8 A 8 _SRGB,
VK_IMAGE_LAYOUT_UNDEFINED,
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, минимальные уровни);

```

## Создание Mip-карт

Наше текстурное изображение теперь имеет несколько уровней mip, но промежуточный буфер может использоваться только для заполнения уровня mip 0. Остальные уровни по-прежнему не определены. Чтобы заполнить эти уровни, нам нужно сгенерировать данные с единственного уровня, который у нас есть. Мы будем

используйте команду `vkCmdBlitImage`. Эта команда выполняет операции копирования, масштабирования, и фильтрации. Мы вызовем ее несколько раз, чтобы отобразить данные для каждого уровня нашего текстурного изображения.

`vkCmdBlitImage` это считается операцией перевода, поэтому мы должны проинформировать Vulkan мы намерены использовать изображение текстуры как в качестве источника, так и в качестве пункта назначения для передачи. Добавить `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` к флагам использования изображения текстуры в `createTextureImage` :

```
1 ... Создайте
2 изображение (texWidth, texHeight, mipLevels, VK_FORMAT_R8G8B8A8_SRGB,
 VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_TRANSFER_SRC_BIT |
 VK_IMAGE_USAGE_TRANSFER_DST_BIT |
 VK_IMAGE_USAGE_SAMPLED_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
 TextureImage, textureImageMemory);
3 ...
```

Как и другие операции с изображениями, `vkCmdBlitImage` зависит от макета

изображения, с которым он работает. Мы могли бы перевести все изображение в, но `VK_IMAGE_LAYOUT_GENERAL` это, скорее всего, будет медленным. Для оптимального per-

для этого исходное изображение должно быть в формате `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` и целевое изображение должно быть в. Vulkan позволяет

нам переходить на каждый уровень `mip` изображения независимо. Каждый `blit` будет иметь дело только с двумя уровнями `mip` одновременно, поэтому мы можем перевести каждый уровень в оптимальное расположение между командами `blits`.

`transitionImageLayout` выполняет переходы макета только для всего изображения, поэтому нам нужно будет написать еще несколько команд конвейерного барьера. Удалите существующий переход на `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` в

Создайте текстовое изображение:

```
1 ...
2 transitionImageLayout(текстурное изображение, VK_FORMAT_R8G8B8A8_SRGB,
 VK_IMAGE_LAYOUT_UNDEFINED,
 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, mipLevels);
3 copyBufferToImage(stagingBuffer, TextureImage,
 static_cast<uint32_t>(ширина текста),
 static_cast<uint32_t>(высота текста));
4 //переведен в режим
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL при
генерации mip-карт
5 ...
```

При этом каждый уровень изображения текстуры останется в режиме `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`.

Каждый уровень будет переведен в `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` после завершения чтения из него команды `blit`.

Теперь мы собираемся написать функцию, которая генерирует `mipmaps`:

```

1 анулирование Создание карточек (изображение VkImage, текстовая ширина int 3 2 _t, int 3 2 _t
2
3
4
5
6
7
8
9
10
11
12
13
14
15
}

```

Мы собираемся сделать несколько переходов, поэтому будем

использовать это повторно Указанные выше поля останутся неизменными

VkImageMemoryBarrier'subresourceRange.mipLevel

для всех барьеров.'oldLayout newLayout srcAccessMask , и

dstAccessMask будет изменен

для каждого перехода.

```

1 int 3 2 _t mipWidth = texWidth;
2 int 3 2 _t mipHeight = texHeight;
3
4 для (uint 3 2 _t i = 1; i < mipLevels; i++) {
5
6 }

```

В этом цикле будет записана каждая из команд vkCmdBlitImage . Обратите

внимание, что переменная цикла начинается с 1, а не с 0.

```

1 barrier.subresourceRange.baseMipLevel = i - 1;
2 barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
3 barrier.newLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
4 barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
5 barrier.dstAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
6
7 vkCmdPipelineBarrier(командный буфер,
8 VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_TRANSFER_BIT,
9 0, 0,
10 nullptr, 0,
11 nullptr,
12 1, и барьер);

```

Первый, мы переходный уровень  $i - 1$  Для `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL`.

Этот переход будет ожидать уровня  $i - 1$  подлежащий заполнению, либо из предыдущего

команда `blit` или из `vkCmdCopyBufferToImage`. Текущая команда `blit` будет ждать этого перехода.

```
1 VkImageBlit blit{};

2 blit.srcOffsets[0] = { 0, 0, 0 };
3 blit.srcOffsets[1] = { mipWidth, mipHeight, 1 };
4 blit.srcSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
5 blit.srcSubresource.mipLevel = i - 1;
6 blit.srcSubresource.baseArrayLayer = 0;
7 blit.srcSubresource.layerCount = 1;

8 blit.dstOffsets[0] = { 0, 0, 0 };
9 blit.dstOffsets[1] = { mipWidth > 1 ? mipWidth / 2 : 1, mipHeight >
10 1 ? Средняя высота / 2 : 1, 1 };

11 blit.dstSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
12 blit.dstSubresource.mipLevel = i;
13 blit.dstSubresource.baseArrayLayer = 0;
14 blit.dstSubresource.Количество слоев = 1;
```

Далее мы указываем регионы, которые будут использоваться в операции `blit`. Исходный уровень `mip` равен . Два элемента массива `srcOffsets` определяют ТРЕХМЕРНУЮ область, из которой будут выгружаться данные. Определение `dstOffsets` определяет регион, в который будут перенесены данные. Символы X и Y размеры `dstOffsets[ 1 ]` делятся на два, поскольку каждый уровень `mip` вдвое меньше предыдущего уровня. Размерность Z в `srcOffsets[ 1 ]` и `dstOffsets[ 1 ]` должно быть равно 1, поскольку 2D-изображение имеет глубину 1. `vkCmdBlitImage(commandBuffer,`

```
1
2 изображение,
3 VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL, изображение,
4 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, &blit,
5 VK_FILTER_LINEAR);
```

Теперь мы записываем команду `blit`. Обратите внимание, что `TextureImage` используется для обоих параметров. Это потому, что мы переключаемся между разными уровнями одного и того же изображения. Исходный уровень `mip` был только что перенесен в `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` и уровень назначения по-прежнему остается `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`. От `Создайте текстовое изображение`.

Будьте осторожны, если вы используете выделенную очередь передачи (как предлагается в буферах `Vertex`): `vkCmdBlitImage` должен быть отправлен в очередь с графическими возможностями.

Последний параметр позволяет нам указать `VkFilter` для использования в `blit`. Здесь у нас есть

те же параметры фильтрации, что и при создании `VkSampler`. Мы

используем `VK_FILTER_LINEAR` чтобы включить интерполяцию.

```
1 барьер.Старое описание = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
2 барьер.Новое описание = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
```

```

3 барьер.srcAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
4 барьер.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
5
6 vkCmdPipelineBarrier(командный буфер,
7 VK_PIPELINE_STAGE_TRANSFER_BIT (ВКОНТАКТЕ),
8 VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
9 0, 0, nullptr,
10 0, nullptr,
11 1, &барьер);

```

Этот барьер переходит на уровень mip  $i - 1$  Для `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` (только для оптимального чтения). Этот переход ожидает завершения текущей команды blit. Все операции выборки будут ожидать завершения этого перехода.

```

1 ...
2 if (mipWidth > 1) mipWidth /= 2;
3 если (mipHeight > 1) mipHeight /= 2;
4 }

```

В конце цикла мы делим текущие размеры mip на два. Мы проверяем каждое измерение перед разделением, чтобы убедиться, что размерность никогда не становится равной 0. Это обрабатывает случаи, когда изображение не квадратное, поскольку одно из измерений mip достигло бы 1 раньше другого измерения. Когда это произойдет, это измерение должно оставаться равным 1 для всех оставшихся уровней.

```

1 barrier.subresourceRange.baseMipLevel = mipLevels - 1;
2 barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
3 барьер.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
4 барьер.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
5 барьер.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
6
7 vkCmdPipelineBarrier(командный буфер),
8 VK_PIPELINE_STAGE_TRANSFER_BIT,
9 VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
10 0, 0, nullptr,
11 0, nullptr,
12 1, &барьер);
13
14 endSingleTimeCommands(командный буфер);
}

```

Прежде чем мы завершим командный буфер, мы вставим еще один барьер конвейера. Этот барьер-rier переводит последний уровень mip из `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` Для `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` (только для оптимального чтения). Это не было обработано циклом, поскольку последний уровень mip никогда не удаляется из. Наконец, добавьте вызов `generateMipmaps` в Создайте текстовое изображение:

```

1 transitionImageLayout(Текстурное изображение, VK_FORMAT_R8G8B8A8_SRGB,
2 VK_IMAGE_LAYOUT_UNDEFINED,
3 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, mipLevels);
4
5 copyBufferToImage(stagingBuffer, TextureImage,
6 static_cast<uint32_t>(ширина текста),
7 static_cast<uint32_t>(высота текста));
8
9 // переведен в режим VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
10 при
11 генерации mip-карт
12 ...
13 Генерация
14 изображений (TextureImage, texWidth, texHeight, mipLevels);

```

MIP-карты наших текстурных изображений теперь полностью заполнены.

## Поддержка линейной фильтрации

Очень удобно использовать встроенную функцию, такую как `vkCmdBlitImage` для генерации всех уровней mip, но, к сожалению, не гарантируется, что он будет поддерживаться на всех платформах. Для этого требуется формат изображения текстуры, который мы используем для поддержки линейной фильтрации, который можно проверить с помощью функции `vkGetPhysicalDeviceFormatProperties`. Для этого мы добавим в функцию проверку.

Сначала добавьте дополнительный параметр, определяющий формат изображения:

```

1 void createTextureImage() {
2
3 ...
4
5 Генерировать карты памяти(TextureImage, VK_FORMAT_R8G8B8A8_SRGB, texWidth,
6 texHeight, mipLevels);
7
8 }
9
10 ...
11
12 анулирование Сгенерировать карты памяти (VkImage
13 image, VkFormat ImageFormat, int32_t
14 texWidth, int32_t texHeight, uint32_t mipLevels) {
15
16 ...
17 }

```

В функции `generateMipmaps` используйте `vkGetPhysicalDeviceFormatProperties` чтобы запросить свойства формата изображения текстуры:

```

1 анулировать generateMipmaps(VkImage image, VkFormat ImageFormat, int32_t 1
2
3 texWidth, int32_t texHeight, uint32_t mipLevels) {
4
5
6 // Проверьте, поддерживает ли формат изображения
7 // линейное растушевывание VkFormatProperties Свойства формата;
8
9 vkGetPhysicalDeviceFormatProperties(Физическое устройство, формат изображения,
10 и свойства формата);
11
12 }

```

7 ...

Свойства `VkFormatProperties` в структуре содержит три поля `optimalTilingFeatures`, `linearTilingFeatures` и `bufferFeatures`, что каждый из них описывает, как созданный нами формат может использоваться в зависимости от способа его использования. Текстурное изображение с оптимальным форматом разбиения на листы, поэтому нам нужно проверить, поддержка функции линейной фильтрации может быть оптимальная характеристика материала's проверено с помощью:

```
if (!(formatProperties.optimalTilingFeatures & VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT))
 выбросить std::runtime_error ("Формат текстурного изображения не поддерживается линейный блинтинг!");
```

В этом случае есть две альтернативы. Вы могли бы реализовать функцию, которая ищет распространенные форматы изображений текстур в поисках того, который выполняет поддерживает линейный блинтинг, или вы могли бы реализовать генерацию mipmap программно с помощью такой библиотеки, как `stb_image_resize`. Затем каждый уровень mip может быть загружен в изображение тем же способом, которым вы загрузили исходное изображение. Следует отметить, что на практике редко удается сгенерировать уровни mipmap во время выполнения в любом случае. Обычно они предварительно генерируются и сохраняются в файле текстуры рядом с базовым уровнем для повышения скорости загрузки. Реализация изменения размера в программном обеспечении и загрузка нескольких уровней из файла оставлены читателю в качестве упражнения.

## Sampler

Пока `VkImage` хранит данные mipmap, `VkSampler` управляет тем, как эти данные считываются во время рендеринга. Vulkan позволяет нам указывать и `mipmapMode` ("Lod" означает "Уровень детализации"). При выборке текстуры, программа выборки выбирает уровень mip в соответствии со следующим псевдокодом:

```
1lod = getLodLevelFromScreenSize(); // меньше, когда объект находится близко, может быть отрицательным
2lod = зажим (lod + mipLodBias, minLod, maxLod);
3
4level = зажим (floor(lod), 0, texture.mipLevels - 1); // зажат в количество уровней mip в текстуре
5
6if (mipmapMode == VK_SAMPLER_MIPMAP_MODE_NEAREST) {
7 цвет = образец(уровень);
8} еще {
9 цвет = смешивание (образец (уровень), проба(уровень + 1));
10}
```

If `samplerInfo.mipmapMode` is `VK_SAMPLER_MIPMAP_MODE_NEAREST`, `lod` выбирает уровень `mip` для выборки. Если режим `mipmap` `VK_SAMPLER_MIPMAP_MODE_LINEAR`, используется для выбора двух уровней `mip` для выборки. Эти уровни будут выбраны `lod` и результаты линейно смешиваются.

На работу с образцом также влияют `lod` :

```
1 if (lod <= 0) {
2 color =
3 readTexture(уф, магФильтр); } еще {
4 color = readTexture (уф, минифильтр);
5 }
```

Если объект находится близко к камере, в качестве фильтра используется `magFilter`. Если объект находится дальше от камеры, используется `minFilter`. Обычно, `lod` неотрицателен, и равен только 0 при закрытии камеры. `mipLodBias` позволяет нам принудительно использовать Vulkan lower `lod` и уровень, чем обычно используется.

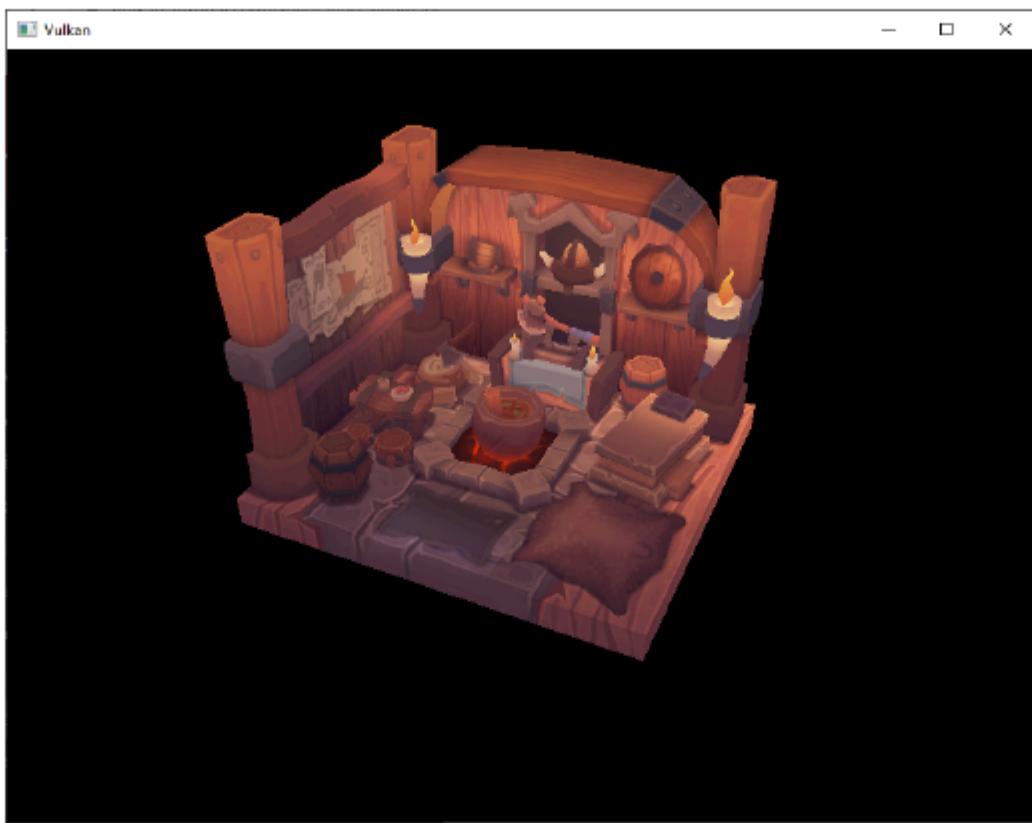
Чтобы увидеть результаты, описанные в этой главе, нам нужно выбрать значения для нашего

`textureSampler`. Мы уже установили `minFilter` и `magFilter` для использования `VK_FILTER_LINEAR`. Нам просто нужно выбрать значения для `minLod` `maxLod`, `mipLodBias`, и `mipmapMode`.

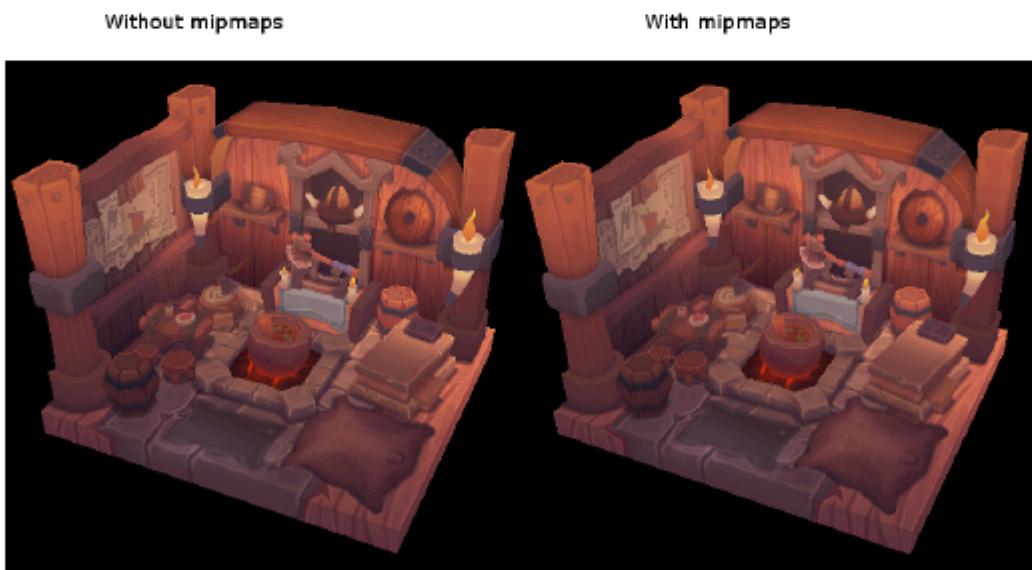
```
1 анулирование createTextureSampler() {
2 ...
3 samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
4 samplerInfo.minLod = 0.0f; // Необязательно
5 samplerInfo.maxLod = static_cast<float>(mipLevels);
6 samplerInfo.mipLodBias = 0.0f; // Необязательно
7 ...
8 }
```

Чтобы можно было использовать полный диапазон уровней `mip`, мы установили `minLod` на `0.0f` и `maxLod` к количеству уровней `mip`. У нас нет причин изменять значение `lod`, поэтому мы установили `mipLodBias` до 0,0 фунта стерлингов.

Теперь запустите вашу программу, и вы должны увидеть следующее:



Разница несущественна, поскольку наша сцена очень проста. Если присмотреться, то есть тонкие различия.



Наиболее заметным отличием является надпись на бумаге. С mipmaps надпись сглажена. Без mipmaps надпись имеет резкие края и

пробелы от муаровых артефактов.

Вы можете поиграть с настройками сэмплера, чтобы увидеть, как они влияют на mipmap-ping. Например, изменив `minLod`, вы можете заставить сэмплер не использовать самые низкие уровни mip:

```
1 samplerInfo.minLod = static_cast<плават>(mipLevels / 2);
```

Эти настройки приведут к созданию такого изображения:



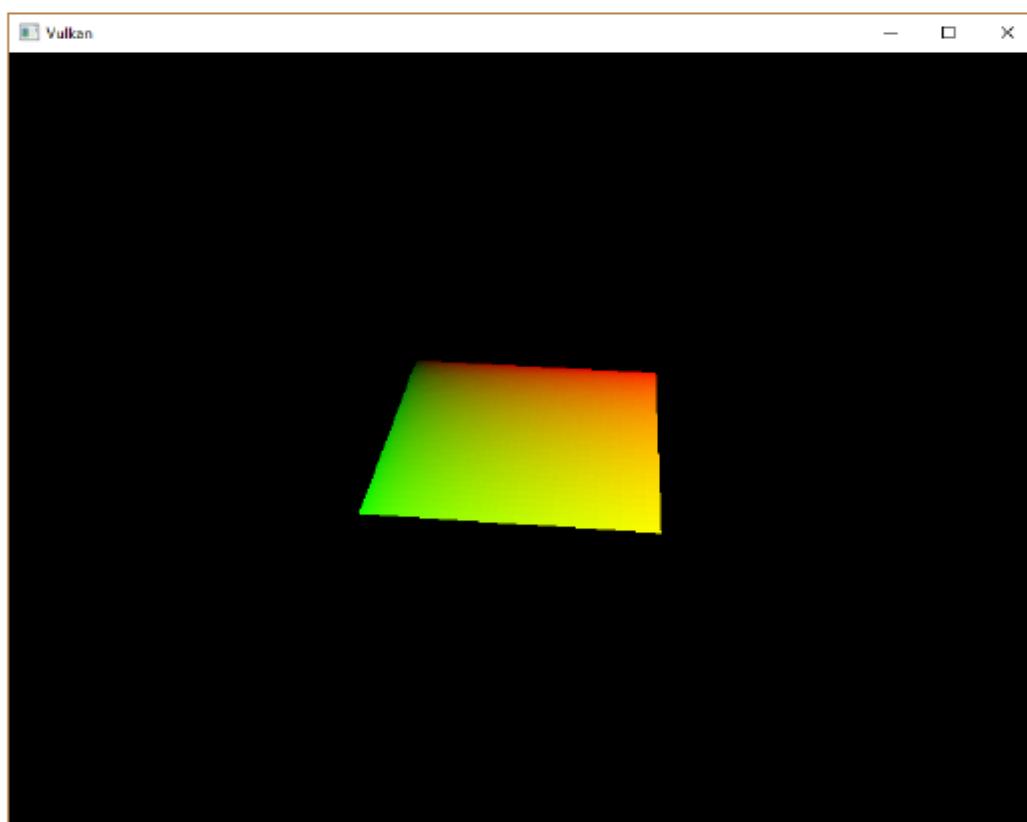
Вот как будут использоваться более высокие уровни mip, когда объекты находятся дальше от камеры.

Код на C++ / Вершинный шейдер / Фрагментный шейдер

# Множественная дискретизация

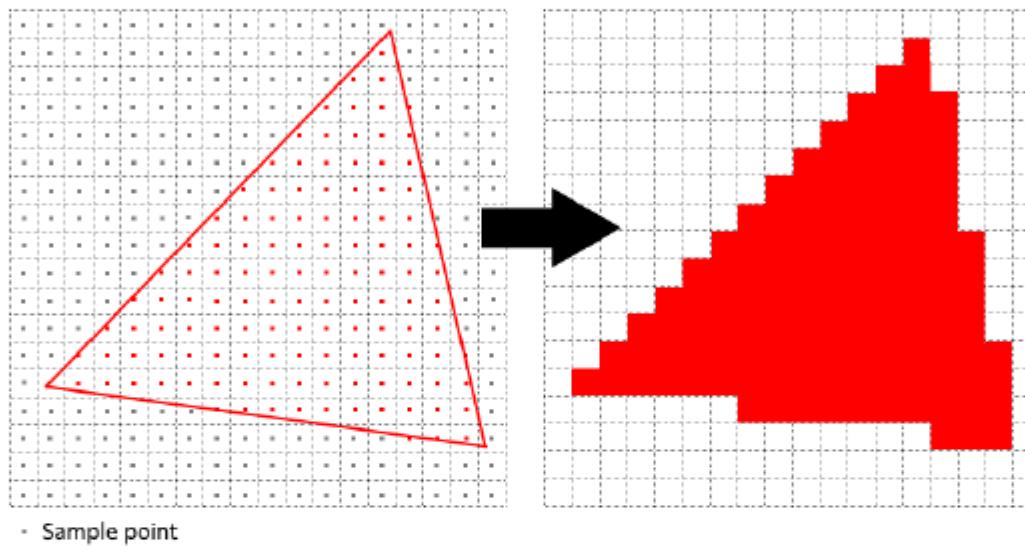
## Введение

Наша программа теперь может загружать несколько уровней детализации текстур, что исправляет артефакты при рендеринге объектов вдали от зрителя. Изображение теперь намного более гладкое, однако при ближайшем рассмотрении вы заметите неровные, похожие на пилу узоры по краям нарисованных геометрических фигур. Это особенно заметно в одной из наших ранних программ, когда мы визуализировали четырехъядерный:

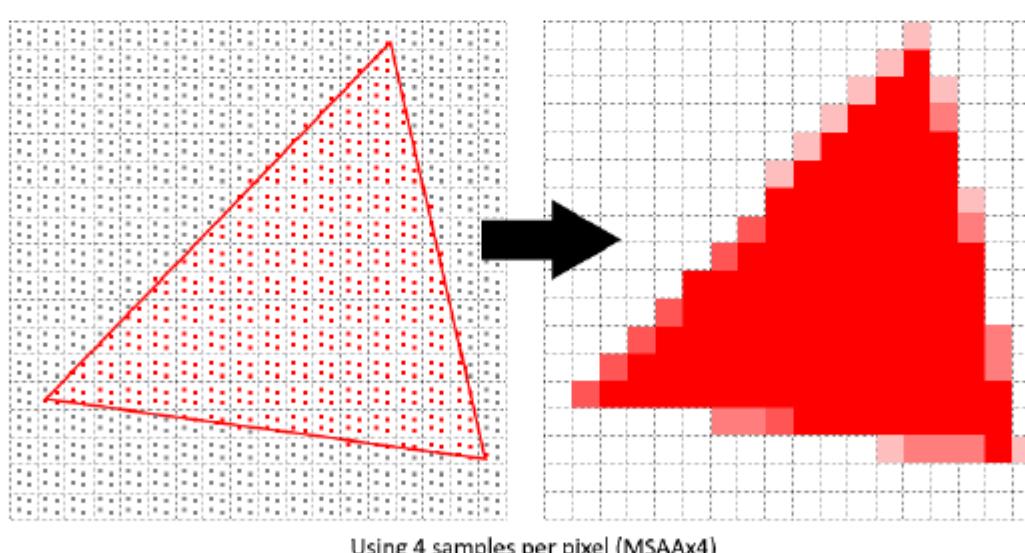


Этот нежелательный эффект называется "сглаживанием" и является результатом ограниченного количества пикселей, доступных для рендеринга. Поскольку там нет дисплеев.

при неограниченном разрешении он всегда будет в какой-то степени виден. Существует несколько способов исправить это, и в этой главе мы сосредоточимся на одном из наиболее популярных: многовариантном сглаживании (MSAA). При обычном рендеринге цвет пикселя определяется на основе одного образца точка, которая в большинстве случаев является центром целевого пикселя на экране. Если часть нарисованной линии проходит через определенный пиксель, но не покрывает точку образца , этот пиксель останется пустым, что приведет к эффекту неровной "лестницы".



Что делает MSAA, так это использует несколько точек выборки на пиксель (отсюда и название) для определения его окончательного цвета. Как и следовало ожидать, большее количество выборок приводит к лучшим результатам, однако это также требует больших вычислительных затрат.



В нашей реализации мы сосредоточимся на использовании максимально доступной выборки

считай. В зависимости от вашего приложения это не всегда может быть лучшим подходом, и, возможно, было бы лучше использовать меньше образцов для повышения производительности, если конечный результат соответствует вашим требованиям к качеству.

## Определение количества доступных образцов

Давайте начнем с определения того, сколько образцов может использовать наше оборудование.

Большинство современных графических процессоров поддерживают как минимум 8 сэмплов, но это число не гарантировано, что оно везде будет одинаковым. Мы будем следить за этим, добавив

```
1 нового члена класса: ...
2 VkSampleCountFlagBits msaaSamples = VK_SAMPLE_COUNT_1_BIT;
3 ...
```

По умолчанию мы будем использовать только одну выборку на пиксель, что эквивалентно отсутствию множественной дискретизации, и в этом случае конечное изображение останется неизменным. Точное максимальное количество выборок можно получить из `VkPhysicalDeviceProperties` связанных с выбранным нами физическим устройством. Мы используем буфер глубины, поэтому мы должны учитывать количество выборок как для цвета, так и для глубины. Максимальное количество выборок, поддерживаемое обоими (&), будет максимальным, которое мы можем поддерживать. Добавьте функцию, которая будет извлекать эту информацию для нас:

```
1 VkSampleCountFlagBits получает maxusablesamplecount() {
2
3 VkPhysicalDeviceProperties Физические свойства устройства;
4 vkGetPhysicalDeviceProperties(физическое устройство,
5
6 Свойства физических устройств);
7
8 Количество VkSampleCountFlags =
9
10 physicalDeviceProperties.limits.framebufferColorSampleCounts &
11 physicalDeviceProperties.limits.framebufferDepthSampleCounts;
12
13 if (количество отсчетов и VK_SAMPLE_COUNT_6_4_BIT) { Возврат 6
14 VK_SAMPLE_COUNT_6_4_BIT; }
15 если (количество отсчетов и VK_SAMPLE_COUNT_3_2_BIT) { Возврат
16
17 VK_SAMPLE_COUNT_3_2_BIT; }
18 если (количество отсчетов и VK_SAMPLE_COUNT_1_6_BIT) { Возврат
19
20 VK_SAMPLE_COUNT_1_6_BIT; }
21 если (количество отсчетов и VK_SAMPLE_COUNT_8_BIT) { Возврат
22
23 VK_SAMPLE_COUNT_8_BIT; }
24 если (количество отсчетов и VK_SAMPLE_COUNT_4_BIT) { Возврат
25
26 VK_SAMPLE_COUNT_4_BIT; }
27 если (количество отсчетов и VK_SAMPLE_COUNT_2_BIT) { Возврат
28
29 VK_SAMPLE_COUNT_2_BIT; }
30
31 Возврат VK_SAMPLE_COUNT_1_BIT;
32 }
```

Теперь мы будем использовать эту функцию для установки переменной `msaaSamples` во время процесса выбора физического устройства. Для этого нам нужно немного изменить `pickPhysicalDevice`

недействительно `pickPhysicalDevice()` функция:

```
1
2 ...
3 для (const auto& device : устройства) {
4 if (isDeviceSuitable(устройство)) {
5 физическое устройство = устройство;
6 msaaSamples = getMaxUsableSampleCount();
7 разбить;
8 }
9 }
10 ...
11 }
```

## Настройка цели рендеринга

В MSAA каждый пиксель отбирается в закадровом буфере, который затем выводится на экран. Этот новый буфер немного отличается от обычных изображений, которые мы рендерили - они должны иметь возможность хранить более одного образца на пиксель. Как только буфер с несколькими выборками создан, он должен быть разрешен до значения по умолчанию `framebuffer` (который хранит только одну выборку на пиксель). Вот почему мы должны создать дополнительную цель рендеринга и изменить наш текущий процесс рисования. Нам нужна только одна цель рендеринга, поскольку одновременно активна только одна операция рисования, как и в случае с буфером

```
1 глубины. Добавьте следующих членов класса: ...
2 VkImage colorImage;
3 VkDeviceMemory colorImageMemory;
4 VkImageView colorImageView;
5 ...
```

Это новое изображение должно будет хранить желаемое количество выборок на пиксель, поэтому нам нужно передать это число в `VkImageCreateInfo` в процессе создания изображения. Измените функцию `createImage`, добавив параметр `numSamples`:

```
void createImage(uint32_t ширина, uint32_t высота, uint32_t
1
2
3
4
```

На данный момент обновите все вызовы этой функции с помощью `VK_SAMPLE_COUNT_1_BIT` - мы будем заменять это соответствующими значениями по мере продвижения к реализации:

```
1 createImage(swapChainExtent.width, swapChainExtent.height, 1,
2 VK_SAMPLE_COUNT_1_BIT, // формат глубины, VK_IMAGE_TILING_OPTIMAL,
3 VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
4 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, DepthImage,
5 depthImageMemory); createImage(texWidth,
6 ...,
7 texHeight, mipLevels, VK_SAMPLE_COUNT_1_BIT,
8 VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_TILING_OPTIMAL,
9 VK_IMAGE_USAGE_TRANSFER_SRC_BIT | VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT,
10 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, TextureImage,
11 textureImageMemory);
```

Теперь мы создадим цветовой буфер с несколькими выборками. Добавьте функцию `createColorResources` и обратите внимание, что мы используем `msaaSamples` здесь в качестве параметра функции

для `createImage`. Мы также используем только один уровень mip, поскольку это обеспечивается с помощью спецификации Vulkan в случае изображений с более чем одной выборкой на пиксель.

Кроме того, для этого цветового буфера не нужны `mipmaps`, поскольку он не будет использоваться как текстура:

```
void createColorResources() {
1
2 VkFormat colorFormat = swapChainImageFormat;
3
4 Создайте изображение (swapChainExtent.ширина, swapChainExtent.высота, 1,
5 msaaSamples, colorFormat, VK_IMAGE_TILING_OPTIMAL,
6 VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT | VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT,
7 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, colorImage,
8 colorImageMemory); colorImageView
9 = createImageView(colorImage, colorFormat,
10 VK_IMAGE_ASPECT_COLOR_BIT, 1);
11 }
```

Для обеспечения согласованности вызовите функцию

Созданные глубинные ресурсы :

```
1 непосредственно перед void initVulkan() {
2 ...
3 createColorResources();
4 createDepthResources();
5 ...
6 }
```

Теперь, когда у нас есть цветовой буфер с несколькими выборками, пришло время позаботиться о глубине. Изменить `createDepthResources` и обновите количество выборок, используемых буфером глубины:

```
1 анулирование createDepthResources() {
2 ...
3 createImage(swapChainExtent.ширина, swapChainExtent.высота, 1,
4 msaaSamples, depthFormat,
5 VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
6 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
7 DepthImage, depthImageMemory);
8 ...
9 }
```

Сейчас мы создали пару новых ресурсов Vulkan, так что давайте не забудем освобождать их при необходимости:

```
1 void cleanupSwapChain() {
2 vkDestroyImageView(устройство, colorImageView,
3 nullptr); vkDestroyImage(устройство, colorImage,
4 nullptr); vkFreeMemory(устройство,
5 colorImageMemory, nullptr); ...
6 }
```

И обновите `recreateSwapChain` чтобы новое цветное изображение можно было воссоздать в правильном разрешении при изменении размера окна:

```
1 анулировать Воссоздать wapchain() {
2 ...
3 createImageViews();
4 createColorResources();
5 createDepthResources(); ...
6 }
7 }
```

Мы прошли начальную настройку MSAA, теперь нам нужно начать использовать этот новый ресурс в нашем графическом конвейере, фреймбуфере, прохождении рендеринга и увидеть результаты!

## Добавление новых вложений

Давайте сначала разберемся с этапом рендеринга.

`createRenderPass` создать

и обновите

Измените информационные структуры для создания

```
1 вложения по цвету и глубине: void createRenderPass() {
2 ...
3 colorAttachment.samples = msaaSamples;
4 colorAttachment.finalLayout =
5 VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
6 ...
7 depthAttachment.samples =
8 msaaSamples; ...
```

Вы заметите, что мы изменили финальное описание с `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` для `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`. Это связано с тем, что многопараметрические изображения не могут быть представлены напрямую. Сначала нам нужно преобразовать их в обычное изображение. Это требование не распространяется на буфер глубины, поскольку он не будет представлен ни в какой точке. Поэтому нам нужно будет добавить только одно новое вложение для color, которое является так называемым вложением resolve:

```
1 ...
2 VkAttachmentDescription colorAttachmentResolve{};
3 colorAttachmentResolve.format = swapChainImageFormat;
4 colorAttachmentResolve.samples = VK_SAMPLE_COUNT_1_BIT;
5 colorAttachmentResolve.loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
6 colorAttachmentResolve.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
7 colorAttachmentResolve.stencilLoadOp =
8 VK_ATTACHMENT_LOAD_OP_DONT_CARE;
9 colorAttachmentResolve.stencilStoreOp =
10 VK_ATTACHMENT_STORE_OP_DONT_CARE;
11 colorAttachmentResolve.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
12 colorAttachmentResolve.finalLayout =
13 VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
14 ...
```

Теперь необходимо указать этапу рендеринга преобразовать цветное изображение с несколькими выборками в обычное вложение. Создайте новую ссылку на вложение, которая будет указывать на цветовой буфер, который будет служить целью разрешения.:

```
1 ...
2 VkAttachmentReference colorAttachmentResolveRef{};
3 colorAttachmentResolveRef.attachment = 2;
4 colorAttachmentResolveRef.layout =
5 VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
6 ...
```

Установите элемент `elements` в структуре `subpass` таким образом, чтобы он указывал на недавно созданные приложения созданную ссылку на вложение. Этого достаточно, чтобы позволить этапу рендеринга определить операцию разрешения нескольких выборок, которая позволит нам вывести изображение на экран:

```
1 ...
2 вспомогательный канал.pResolveAttachments =
3 &colorAttachmentResolveRef; ...
```

Теперь обновите информационную структуру прохождения рендеринга новым цветным вложением:

```
1 ...
2 std::vector<VkAttachmentDescription, 3> вложения =
3 {colorAttachment, depthAttachment, colorAttachmentResolve};
4 ...
```

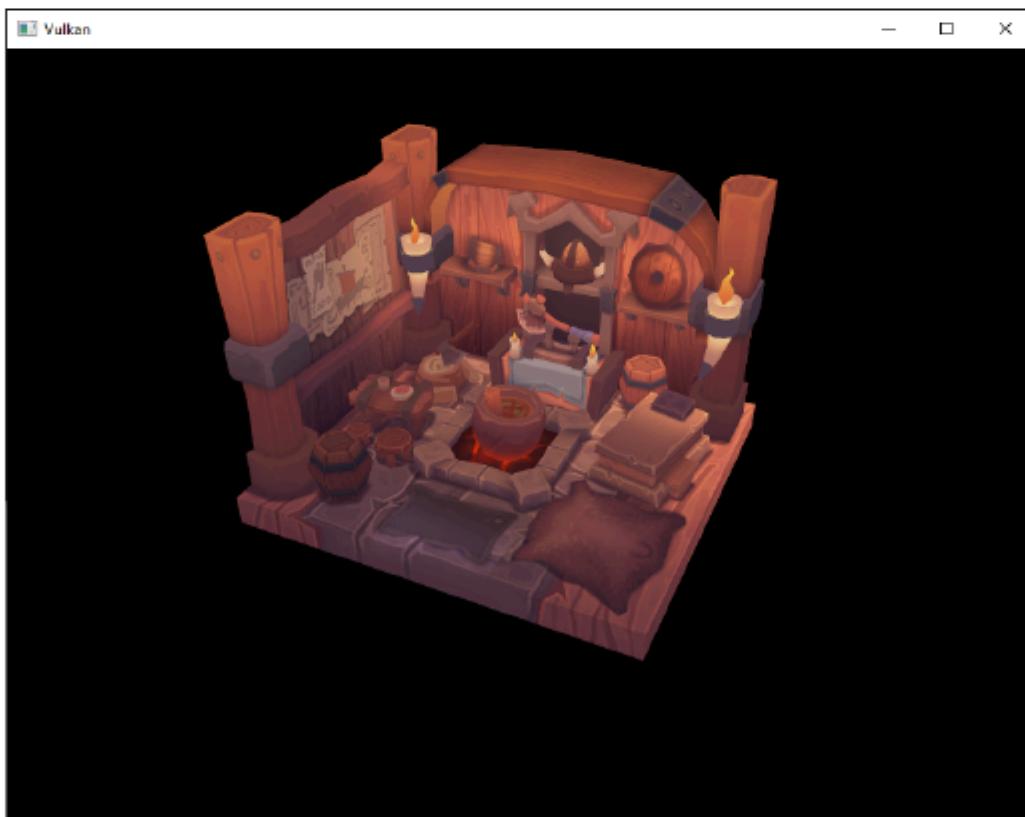
Выполнив этап рендеринга, измените `createFramebuffers` и добавьте новый вид изображения в список:

```
1 анулировать createFramebuffers() {
2 ...
3 стандартный формат::массив<VkImageView, 3> вложения = {
4 colorImageView,
5 depthImageView,
6 swapChainImageViews[i]
7 };
8 ...
9 }
```

Наконец, укажите вновь созданному конвейеру использовать более одного образца, изменив `createGraphicsPipeline`:

```
1 пустота createGraphicsPipeline() {
2 ...
3 мультисэмплинг.rasterizationSamples =
4 msaaSamples; ...
5 }
```

Теперь запустите вашу программу, и вы должны увидеть следующее:



Как и в случае с mipmaping, разница может быть заметна не сразу.

При ближайшем рассмотрении вы заметите, что края больше не такие зазубренные, и все изображение кажется немного более гладким по сравнению с оригиналом.



Разница более заметна при взгляде вблизи на один из краев изображения.:



## Улучшение качества.

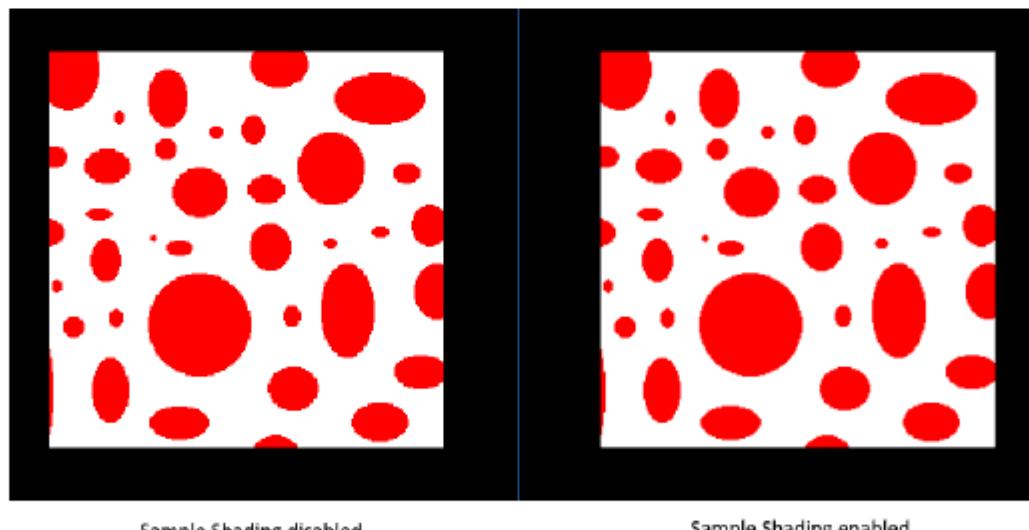
Существуют определенные ограничения нашей текущей реализации MSAA, которые могут повлиять на качество выходного изображения в более детализированных сценах. Например, в настоящее время мы не решаем потенциальные проблемы, вызванные сглаживанием шейдеров,

т.е. MSAA сглаживает только края геометрии, но не внутреннее наполнение.

Это может привести к ситуации, когда вы получаете гладкий многоугольник, отображаемый на экране, но примененная текстура все равно будет выглядеть сглаженной, если она содержит высококонтрастные цвета. Один из способов решения этой проблемы - включить затенение образца, что еще больше улучшит качество изображения, хотя и с дополнительными затратами на производительность:

```
void Создайте логическое устройство() {
1
2 ...
3 Характеристики устройства.sampleRateShading = VK_TRUE; // включить выборку
4 функция затенения для устройства
5 ...
6
7 пустота createGraphicsPipeline() {
8 ...
9 множественная выборка.sampleShadingEnable = VK_TRUE; // включить
10 выборку
11 затенение в конвейере
12 множественная выборка.minSampleShading = .2f; // минимальная доля для выборки
13
14 затенение; ближе к
15 одному - более плавное ...
16 }
```

В этом примере мы оставим затенение образца отключенным, но в определенных сценариях улучшение качества может быть заметным:



## Заключение

Потребовалось много работы, чтобы добраться до этого момента, но теперь у вас, наконец, есть хорошая база для программы Vulkan. Знание основных принципов работы Vulkan

того, чем вы сейчас обладаете, должно быть достаточно, чтобы начать изучать больше функций, таких как:

- Push-константы • Рендеринг с возможностью создания экземпляра
- Динамическая униформа • Отдельные изображения и дескрипторы сэмплера • Конвейерный кэш • Генерация многопоточного буфера команд • Несколько подпасов
- Вычислительные шейдеры

Текущая программа может быть расширена многими способами, такими как добавление блинно-фонового освещения, эффектов постобработки и отображения теней. Вы должны быть в состоянии узнать, как работают эти эффекты, из руководств по другим API, потому что, несмотря на понятность Vulkan, многие концепции по-прежнему работают одинаково. Код на C++ / Вершинный шейдер / Фрагментный шейдер

# Вычислительный Шейдер

## Введение

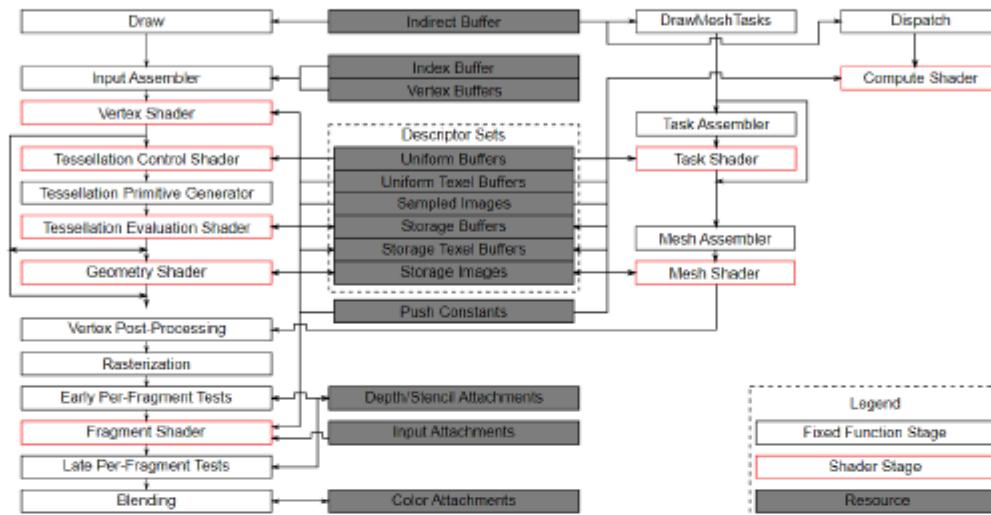
В этой дополнительной главе мы рассмотрим вычислительные шейдеры. До сих пор все предыдущие главы были посвящены традиционной графической части конвейера Vulkan . Но в отличие от старых API, таких как OpenGL, поддержка вычислительных шейдеров в Vulkan обязательна. Это означает, что вы можете использовать вычислительные шейдеры во всех доступных реализациях Vulkan , независимо от того, является ли это высокопроизводительным графическим процессором для настольных компьютеров или маломощным встроенным устройством. Это открывает мир вычислений общего назначения на графических процессорах (GPGPU), независимо от того, где запущено ваше приложение. GPGPU означает, что вы можете выполнять общие вычисления на своем графическом процессоре, то, что традиционно было областью центральных процессоров. Но поскольку графические процессоры становятся все более и более мощными и гибкими, многие рабочие нагрузки, для выполнения которых потребовались бы возможности центрального процессора общего назначения, теперь можно выполнять на графическом процессоре в режиме реального времени. Несколько примеров того, где могут быть использованы вычислительные возможности графического процессора, - это манипулирование изображениями, тестирование видимости, постобработка, расширенные расчеты освещения, анимации, физика (например, для системы частиц) и многое другое. И даже возможно использовать compute только для невизуальной вычислительной работы, которая не требует какого-либо графического вывода, например, вычисления чисел или вещей, связанных с искусственным интеллектом. Это называется "безголовыми вычислениями".

## Преимущества

Выполнение дорогостоящих вычислений на графическом процессоре имеет несколько преимуществ. . Наиболее очевидным из них является разгрузка работы центрального процессора. Другой способ не требует перемещения данных между основной памятью процессора и памятью графического процессора. Все данные могут храниться на графическом процессоре, не дожидаясь медленной передачи из основной памяти. Помимо этого, графические процессоры в значительной степени распараллелены, и некоторые из них содержат десятки тысяч небольших вычислительных блоков. Это часто делает их более подходящими для высокопараллельных рабочих процессов, чем центральный процессор с несколькими большими вычислительными блоками.

## Трубопровод "Вулкан"

Важно знать, что compute полностью отделен от графической части конвейера. Это видно на следующей блок-схеме конвейера Vulkan из официальной спецификации.:



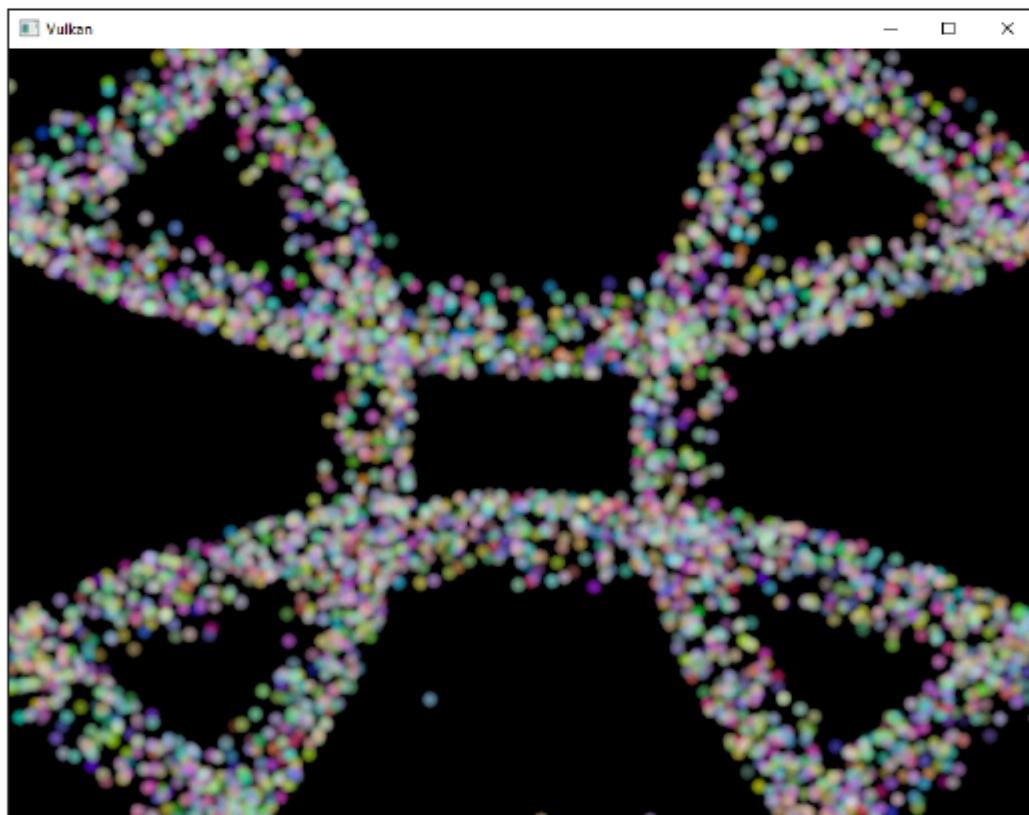
На этой диаграмме мы можем видеть традиционную графическую часть конвейера слева и несколько этапов справа, которые не являются частью этого графического конвейера, включая вычислительный шейдер (stage). Поскольку этап вычислительного шейдера отделен от графического конвейера, мы сможем использовать его везде, где сочтем нужным. Это сильно отличается, например, от фрагментного шейдера, который всегда применяется к преобразованному выходному результату вершинного шейдера. Центр диаграммы также показывает, что, например, наборы дескрипторов также используются compute, поэтому все, что мы узнали о макетах дескрипторов, наборах дескрипторов и дескрипторах, также применимо здесь.

## Пример

Простой для понимания пример, который мы реализуем в этой главе, - это система частиц на базе графического процессора. Такие системы используются во многих играх и часто состоят из тысяч частиц, которые необходимо обновлять с интерактивной частотой кадров . Для рендеринга такой системы требуются 2 основных компонента: вершины, передаваемые как буферы вершин, и способ их обновления на основе некоторого уравнения. "Классическая" система частиц на базе ЦП будет хранить данные о частицах в системной основной памяти системы, а затем использовать ЦП для их обновления. После обновления вершины необходимо снова перенести в память графического процессора, чтобы он мог повторно воспроизвести обновленные частицы в следующем кадре. Наиболее простым способом было бы воссоздание вершинного буфера с новыми данными для каждого кадра. Это , очевидно, очень дорого. В зависимости от вашей реализации существуют другие

такие опции, как отображение памяти графического процессора, чтобы она могла быть записана процессором (называется "панель изменения размера" в настольных системах или унифицированная память на встроенных графических процессорах) или просто с использованием локального буфера хоста (что было бы самым медленным методом из-за пропускной способности PCI-E). Но независимо от того, какой метод обновления буфера вы выберете, вам всегда требуется "обходной путь" к центральному процессору для обновления частиц. С системой частиц на базе графического процессора этот переход туда и обратно больше не требуется. Версии загружаются в графический процессор только при запуске, а все обновления выполняются в памяти графического процессора с использованием вычислительных шейдеров. Одной из основных причин, почему это быстрее, является гораздо более высокая пропускная способность между графическим процессором и его локальной памятью. В сценарии, основанном на процессоре, вы были бы ограничены основной памятью и пропускной способностью PCI-express, которая часто составляет лишь часть пропускной способности памяти графического процессора. При выполнении этого на графическом процессоре с выделенной очередью вычислений вы можете обновлять частицы параллельно с частью рендеринга графического конвейера. Это называется "асинхронные вычисления" и является расширенной темой, не рассматриваемой в этом руководстве.

Вот скриншот из кода этой главы. Показанные здесь частицы обновляются вычислительным шейдером непосредственно на графическом процессоре, без какого-либо взаимодействия с процессором:



## Манипулирование данными

В этом руководстве мы уже узнали о различных типах буферов, таких как вершинный и индексный буферы для передачи примитивов и унифицированные буферы для передачи данных в шейдер. И мы также использовали изображения для отображения текстур. Но до сих пор мы всегда записывали данные с помощью центрального процессора и выполняли только чтение на графическом процессоре. Важной концепцией, представленной в вычислительных шейдерах, является возможность произвольного чтения из буферов **и записи в буферы**. Для этого Vulkan предлагает два выделенных типа хранилища.

### Объекты буфера хранения шейдеров (SSBO)

Буфер хранения шейдеров (SSBO) позволяет шейдерам считывать данные из буфера и записывать их в буфер. Их использование аналогично использованию объектов унифицированного буфера. Самые большие различия заключаются в том, что вы можете присваивать SSBO псевдонимы других типов буферов и что они могут быть сколь угодно большими. Возвращаясь к системе частиц на базе графического процессора, теперь вы можете задаться вопросом, как работать с вершинами, обновляемыми (записываемыми) вычислительным шейдером и считываемыми (рисуемыми) вершинным шейдером, поскольку для обоих вариантов использования, по-видимому, потребуются разные типы буферов. Но это не так. В Vulkan вы можете указать несколько способов использования буферов и изображений. Итак, чтобы буфер вершин частиц использовался как буфер вершин (в графическом проходе) и как буфер хранения (в вычислительном проходе), вы просто создаете буфер с этими двумя флагами использования:

```
1 VkBufferCreateInfo BufferInfo{};
2 ...
3 BufferInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT |
4 VK_BUFFER_USAGE_STORAGE_BUFFER_BIT
5 | VK_BUFFER_USAGE_TRANSFER_DST_BIT;
6 ...
7 if (vkCreateBuffer(устройство, &BufferInfo, nullptr,
8 &shaderStorageBuffers[i]) != VK_SUCCESS) {
9 выбросить std::runtime_error("не удалось создать буфер вершин!");
10 }
```

Два флага `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT` и `VK_BUFFER_USAGE_STORAGE_BUFFER_BIT` устанавливаются с помощью `BufferInfo.usage`. Сообщите реализации, что мы хотим использовать этот буфер для двух разных сценариев: в качестве вершинного буфера в вершинном шейдере и в качестве буфера хранения. Обратите внимание, что мы также добавили флаг `VK_BUFFER_USAGE_TRANSFER_DST_BIT` здесь, чтобы мы могли передавать данные с хоста на графический процессор. Это важно, поскольку мы хотим, чтобы буфер хранения шейдеров оставался только в памяти графического процессора (), который нам нужен для передачи данных из `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` хост для этого буфера.

Вот тот же код, использующий использование `createBuffer` вспомогательная функция:

```
1 createBuffer(размер буфера, VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
2 VK_BUFFER_USAGE_VERTEX_BUFFER_BIT |
3 VK_BUFFER_USAGE_TRANSFER_DST_BIT,
4 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, shaderStorageBuffers[i],
5 shaderStorageBuffersMemory[i]);
```

Объявление GLSL-шейдера для доступа к такому буферу выглядит

```
следующим образом: struct Particle {
1 положение vec 2 ;
2 скорость vec 2 ;
3 цвет vec 4 ;
4 };
5
6
7 макет (std 1 4 0 , привязка = 1) буфер без частиц, доступный только для чтения {
8 Частица particlesIn[];
9 };
10
компоновка(std 1 4 0 , привязка =
2) буфера ParticleSSBOout { 11
12 Выделение частиц[];
13 };
```

В этом примере у нас есть типизированный SSBO, в котором каждая частица имеет положение и значение скорости (см. Структуру `Particle`). Таким образом, SSBO содержит несвязанное количество частиц, указанное `[ ]`. Отсутствие необходимости указывать количество элементов в SSBO является одним из преимуществ, например, по сравнению с однородными буферами. `std 1 4 0` является определителем компоновки памяти, который определяет, как элементы-члены буфера хранения шейдеров выровнены в памяти. Это дает нам определенные гарантии, необходимые для сопоставления буферов между хостом и графическим процессором.

Запись в такой объект `storage buffer` в вычислительном шейдере проста

и аналогична записи в буфер на стороне C++:

```
1 particlesOut[индекс].position = частица в[индексе].позиция +
2 particlesIn[индекс].velocity.xy * ubo.deltaTime;
```

## Изображения для хранения

*Обратите внимание, что в этой главе мы не будем манипулировать изображениями. Этот параграф приведен для того, чтобы ознакомить читателей с тем, что вычислительные шейдеры также можно использовать для манипулирования изображениями.*

Образ хранилища позволяет выполнять чтение из образа и запись в него. Типичные варианты использования применяете эффекты изображения к текстурам, выполняете постобработку (которая, в свою очередь, очень похожа) или создаете mip-карты. Аналогично для изображений:

```

1 VkImageCreateInfo ImageInfo
2 {};
3 ImageInfo.usage = VK_IMAGE_USAGE_SAMPLED_BIT |
4 VK_IMAGE_USAGE_STORAGE_BIT;
5 ...
6
7 если (vkCreateImage(device, &ImageInfo, nullptr, &TextureImage) !=
8 VK_SUCCESS) {
9 выбросить std::runtime_error("не удалось создать изображение!");
10 }

```

Два флага `VK_IMAGE_USAGE_SAMPLED_BIT` и `VK_IMAGE_USAGE_STORAGE_BIT` устанавливаются с помощью `ImageInfo.usage`. Сообщите реализации, что мы хотим использовать это изображение для двух разных сценариев: как изображение, выбранное в шейдере фрагментов и как изображение хранилища в компьютерном шейдере;

Объявление шейдера GLSL для изображения хранилища выглядит аналогично изображениям с выборками, используемым, например, в шейдере фрагментов:

```

1 макет (привязка = 0, rgba 8) единообразного входного изображения image 2D только для чтения; макет
2 (привязка = 1, rgba 8) единообразного выходного изображения image 2D только для записи;

```

Несколько отличиями здесь являются дополнительные атрибуты, такие как изображение, то только для чтения и только для определители, сообщающие реализации, что мы будем читать только из входного изображения и записывать в выходное изображение. И последнее, но не менее важное, нам нужно использовать `image 2D` введите, чтобы объявить образ хранилища. Затем выполняется чтение и запись в образы хранилища в вычислительном шейдере с помощью `imageLoad` и `imageStore`:

```

1 пиксель vec 3 = загрузка изображения (inputImage,
2 ivec2 (gl_GlobalInvocationID.xy)).rgb;
2 Хранилище изображений(outputImage, ivec2 (gl_GlobalInvocationID.xy), пиксель);

```

## Семейства вычислительных очередей

В главе "Физические устройства и семейства очередей" мы уже узнали о семействах очередей и о том, как выбрать семейство графических очередей. При вычислении используется бит флага свойств семейства `queue VK_QUEUE_COMPUTE_BIT`. Итак, если мы хотим выполнить вычислительную работу, нам нужно получить очередь из семейства очередей, поддерживающего вычисления. Обратите внимание, что вулкан требует реализации, которая поддерживает графические операции, чтобы иметь по крайней мере одну очередь семья, которая поддерживает оба графики и вычислительных операций, но также возможно, что реализаций предлагают выделенные компоненты очереди. Эта выделенная очередь вычислений (в которой нет графического бита) намекает на асинхронную очередь вычислений. Чтобы сохранить это руководство для начинающих.

для удобства мы будем использовать очередь, которая может выполнять как графические, так и вычислительные операции. Это также избавит нас от необходимости иметь дело с несколькими продвинутыми механизмами синхронизации. Для нашего примера вычислений нам нужно немного изменить код

```
1 создание устройства: uint 3 _t queueFamilyCount = 0; vkGetPhysicalDeviceQueueFamilyProperties(устройство,
2 &queueFamilyCount,
3 nullptr);
4
5 std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
6 vkGetPhysicalDeviceQueueFamilyProperties(устройство, &queueFamilyCount,
7 Семейства очередей.данные());
8
9 int i = 0;
10 для (const auto& queueFamily : семьи очередей) {
11 if ((Семья очередей.queueFlags & VK_QUEUE_GRAPHICS_BIT) &&
12 (Семейство очередей.queueFlags &
13 VK_QUEUE_COMPUTE_BIT)) { индексы.graphicsAndComputeFamily = i;
14 }
15
16 i++;
17 }
```

Измененный код выбора индекса семейства очередей теперь будет пытаться найти семейство очередей, поддерживающее как графику, так и вычисления. Затем мы можем получить очередь вычислений из этого семейства очередей в [Создание логического устройства](#):

```
1 vkGetDeviceQueue(устройство, индексы.Графика и вычислительное семейство.значение(),
2 0, &computeQueue);
```

## Этап вычисления шейдера

В графических примерах мы использовали различные этапы конвейера для загрузки шейдеров и доступа к дескрипторам. Доступ к вычислительным шейдерам осуществляется аналогичным образом, используя конвейер `VK_SHADER_STAGE_COMPUTE_BIT`. Таким образом, загрузка вычислительного шейдера - это то же самое, что загрузка вершинного шейдера, но с другим этапом шейдирования. Мы подробно поговорим об этом в следующих параграфах. Compute также вводит новый тип точки привязки для дескрипторов и конвейеров с именем `VK_PIPELINE_BIND_POINT_COMPUTE` который нам придется использовать позже.

## Загрузка вычислительных шейдеров

Загрузка вычислительных шейдеров в нашем приложении такая же, как загрузка любых других шейдеров. Единственное различие заключается в том, что нам нужно будет использовать `VK_SHADER_STAGE_COMPUTE_BIT` упомянуто выше.

```

1 авто computeShaderCode = ReadFile("шейдеры/compute.spv");
2
3 vkShaderModule computeShaderModule =
4 Создайте vkshadermodule(computeShaderCode);
5
6 VkPipelineShaderStageCreateInfo
7 computeShaderStageInfo{}; computeShaderStageInfo.Тип =
8 VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
9 computeShaderStageInfo.stage = VK_SHADER_STAGE_COMPUTE_BIT;
10 computeShaderStageInfo.module = computeShaderModule;
11 computeShaderStageInfo.pName = "главный";
12 ...

```

## Подготовка буферов хранилища шейдеров

Ранее мы узнали, что мы можем использовать буферы хранилища шейдеров для передачи произвольных данных для вычисления шейдеров. В этом примере мы загрузим массив частиц в графический процессор, чтобы мы могли манипулировать им непосредственно в памяти графического процессора. В главе "Кадры в полете" мы говорили о дублировании ресурсов для каждого кадра в полете, чтобы мы могли сохранять загруженность центрального и графического процессоров. Сначала мы объявляем вектор для объекта buffer и резервной

```

1 копии его памяти устройства: std::vector<VkBuffer> shaderStorageBuffers;
2
3 std::vector<VkDeviceMemory> shaderStorageBuffersMemory;

```

В разделе `createShaderStorageBuffers` затем мы изменяем размер этих векторов, чтобы они соответствовали максимальному количеству кадров в цикле:

```

1 shaderStorageBuffers.resize(MAX_FRAMES_IN_FLIGHT);
2 shaderStorageBuffersMemory.resize(MAX_FRAMES_IN_FLIGHT);

```

Установив эту настройку, мы можем начать перемещать исходную информацию о частицах в графический процессор. Сначала мы инициализируем вектор частиц на стороне хоста:

```

1 // Инициализировать частицы
2 стандартный параметр::default_random_engine rndEngine((без знака)time(nullptr));
3 std::uniform_real_distribution<float> rndDist(0.0f, 1.0f);
4
5 // Начальные положения частиц на окружности
6 std::vector<Частица> particles(PARTICLE_COUNT);
7 для (auto& particle : частицы) {
8
8 float r = 0.25f * sqrt(rndDist(rndEngine));
9 с плавающей точкой тета = rndDist(rndEngine) * 2 *
10 3.14159265358979323846;
11
12 значение с плавающей точкой x = r * cos (тета) * ВЫСОТА
13 / ШИРИНА; значение с плавающей точкой y = r * sin (тета);

```

```

12 частица.положение = glm::vec \langle 2 \rangle (x, y);
13 частица.скорость = glm::нормализовать (glm::vec \langle 2 \rangle (x, y)) *
14 \langle 0, 0 0 0 2 5 \rangle f;
15 particle.color = glm::vec \langle 4 \rangle (rndDist(rndEngine),
16 rndDist(rndEngine), rndDist(rndEngine), 1.0 f);
17 }
```

Затем мы создаем промежуточный буфер в памяти хоста для хранения начальной частицы свойства:

```

1 VkDeviceSize bufferSize = sizeof(частица) * PARTICLE_COUNT;
2
3 VkBuffer stagingBuffer;
4 VkDeviceMemory stagingBufferMemory;
5 createBuffer(размер буфера, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
6 VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
7 VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer,
8 stagingBufferMemory);
9 анулирование * данных; vkMapMemory(устройство,
10 stagingBufferMemory, 0, размер буфера, 0,
11 &data);
12 memcpy(данные, particles.data(), (size_t)размер буфера);
13 vkUnmapMemory(устройство, stagingBufferMemory);
```

Используя этот промежуточный буфер в качестве источника, мы затем создаем хранилище шейдеров для каждого кадра буферы и копируем свойства частиц из промежуточного буфера в каждый из них:

```

1 для (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
2 createBuffer(размер буфера, VK_BUFFER_USAGE_STORAGE_BUFFER_BIT
3 | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT |
4 VK_BUFFER_USAGE_TRANSFER_DST_BIT,
5 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
6 shaderStorageBuffers[i], shaderStorageBuffersMemory[i]);
7 // Скопируйте данные из промежуточного буфера (хоста) в
8 // шейдер
9 // буфер хранения (GPU)
10 Буфер копирования(stagingBuffer, shaderStorageBuffers[i],
11 Размер буфера);
12 }
13 }
```

## Дескрипторы

Настройка дескрипторов для вычислений практически идентична настройке графики. Единственное отличие заключается в том, что дескрипторы должны иметь VK\_SHADER\_STAGE\_COMPUTE\_BIT установлено так, чтобы сделать их доступными на этапе вычисления:

```

1 std::vector<VkDescriptorSetLayoutBinding,
2 3> layoutBindings{}; layoutBindings[0].binding
3 = 0; layoutBindings[0].descriptorCount = 1;
4 layoutBindings[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
5 layoutBindings[0].pImmutableSamplers
6 = nullptr; layoutBindings[0].stageFlags
7 = VK_SHADER_STAGE_COMPUTE_BIT; ...

```

Обратите внимание, что здесь вы можете комбинировать этапы шейдера, поэтому, если вы хотите, чтобы дескриптор был доступен из вершины и этапа вычисления, например, для однородного буфера с общими для них параметрами, вы просто устанавливаете биты для

```

1 обоих этапов: привязки к макету[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT |
2 VK_SHADER_STAGE_COMPUTE_BIT;

```

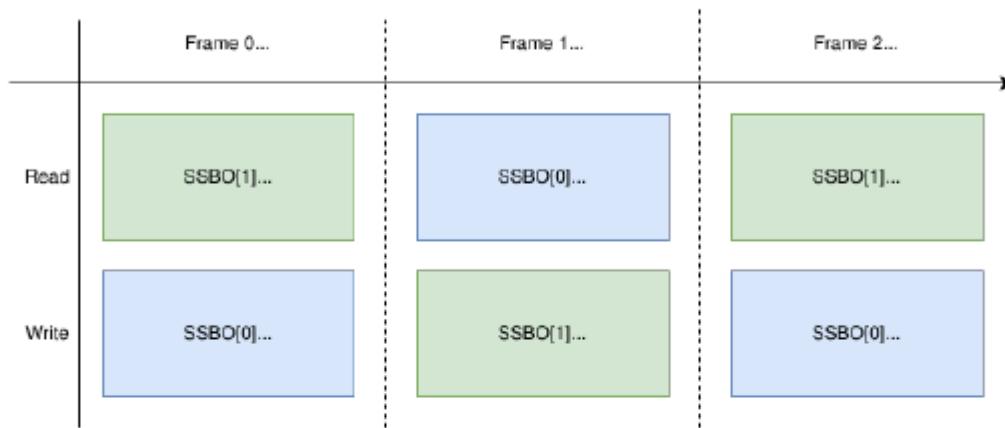
Вот настройка дескриптора для нашего примера. Макет

```

1 выглядит примерно так: std::vector<VkDescriptorSetLayoutBinding,
2 3> layoutBindings{}; layoutBindings[0].binding = 0;
3 layoutBindings[0].descriptorCount = 1;
4 layoutBindings[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
5 layoutBindings[0].pImmutableSamplers = nullptr;
6 layoutBindings[0].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
7
8 layoutBindings[1].привязка = 1;
9 layoutBindings[1].descriptorCount = 1;
10 layoutBindings[1].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
11 layoutBindings[1].pImmutableSamplers = nullptr;
12 layoutBindings[1].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
13
14 layoutBindings[2].привязка = 2;
15 layoutBindings[2].descriptorCount = 1;
16 layoutBindings[2].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
17 layoutBindings[2].pImmutableSamplers = nullptr;
18 layoutBindings[2].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
19
20 VkDescriptorSetLayoutCreateInfo
21 layoutInfo{}; layoutInfo.тип =
22 VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
23 layoutInfo.bindingCount = 3;
24 layoutInfo.pBindings = layoutBindings.data();
25
26 если (vkCreateDescriptorSetLayout(устройство, &layoutInfo,
27 nullptr, &computeDescriptorSetLayout) != VK_SUCCESS) {
28 выбросить std::runtime_error ("не удалось создать вычислительный дескриптор
29
30 установите макет!");
31 }

```

Глядя на эту настройку, вы можете задаться вопросом, почему у нас есть две привязки макета для объектов буфера хранения шейдеров, хотя мы будем рендерить только одну систему particle . Это происходит потому, что положения частиц обновляются кадр за кадром на основе дельта-времени. Это означает, что каждый кадр должен знать о положениях частиц в последних кадрах, чтобы он мог обновить их новым временем разности и записать их в свой собственный SSBO:



Для этого вычислительный шейдер должен иметь доступ к SSBO последнего и текущего фреймов . Это делается путем передачи обоих в compute в нашей настройке дескриптора. Смотрите storageBufferInfoLastFrame

И storageBufferInfoCurrentFrame:

```

1 для (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
2
3 VkDescriptorBufferInfo uniformBufferInfo{};
4 uniformBufferInfo.buffer = uniformBuffers[i];
5 uniformBufferInfo.offset = 0;
6 uniformBufferInfo.range = sizeof(UniformBufferObject);
7
8 std::array<VkWriteDescriptorSet, 3>
9 descriptorWrites{}; ...
10
11 VkDescriptorBufferInfo storageBufferInfoLastFrame{};
12 storageBufferInfoLastFrame.buffer = shaderStorageBuffers[(i - 1) %
13 MAX_FRAMES_IN_FLIGHT];
14 storageBufferInfoLastFrame.cмещение = 0;
15 storageBufferInfoLastFrame.range = sizeof(частица) *
16 PARTICLE_COUNT;
17
18 Описание[1].sType =
19 VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
20 descriptorWrites[1].dstSet = computeDescriptorSets[i];
21 descriptorWrites[1].dstBinding = 1;
22 descriptorWrites[1].dstArrayElement = 0;
23 descriptorWrites[1].descriptorType =

```

```

20 VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
21 descriptorWrites[1].descriptorCount = 1;
22 descriptorWrites[1].pBufferInfo = &storageBufferInfoLastFrame;
23
24 VkDescriptorBufferInfo storageBufferInfoCurrentFrame{};
25 storageBufferInfoCurrentFrame.buffer = shaderStorageBuffers[i];
26 storageBufferInfoCurrentFrame.offset = 0;
27 storageBufferInfoCurrentFrame.range = sizeof
28 (частица) * PARTICLE_COUNT;
29
30 Описатель записывает[2].sType =
31 VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET / НАВОР ТИПОВ СТРУКТУРЫ;
32 descriptorWrites[2].dstSet = computeDescriptorSets[i];
33 descriptorWrites[2].dstBinding = 2;
34 descriptorWrites[2].dstArrayElement = 0;
35 descriptorWrites[2].descriptorType =
36 VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
37 descriptorWrites[2].descriptorCount = 1;
38 descriptorWrites[2].pBufferInfo = &storageBufferInfoCurrentFrame;
39
40 vkUpdateDescriptorSets(устройство, 3, descriptorWrites.data(), 0,
41 nullptr);
42 }

```

Помните, что мы также должны запросить типы дескрипторов для SSBO из нашего пула дескрипторов:

```

1 std::vector<VkDescriptorPoolSize, 2> poolSizes{};
2 ...
3
4 poolSizes[1].type = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
5 poolSizes[1].descriptorCount =
6 static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT) * 2;

```

Нам нужно удвоить количество **VK\_DESCRIPTOR\_TYPE\_STORAGE\_BUFFER** типов, запрошен из пула двумя, потому что наши наборы ссылаются на SSBO последнего и текущего кадра.

## Конвейеры вычислений

Поскольку compute не является частью графического конвейера, мы не можем использовать `vkCreateGraphicsPipelines`. Вместо этого нам нужно создать выделенный вычислительный конвейер с помощью `vkCreateComputePipelines` для выполнения наших вычислительных команд. Поскольку вычислительный конвейер не затрагивает ни одно из состояний растеризации, его состояние намного меньше, чем у графического конвейера:

```
1 VkComputePipelineCreateInfo pipelineInfo{};
```

```

2 Конвейерная информация.sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
3 pipelineInfo.layout = computePipelineLayout;
4 pipelineInfo.stage = computeShaderStageInfo;
5
6 if (vkCreateComputePipelines(устройство, VK_NULL_HANDLE, 1,
 &pipelineInfo, nullptr, &computePipeline) != VK_SUCCESS) {
7 выбросить std::runtime_error("не удалось создать вычислительный конвейер!");
8 }

```

Настройка намного проще, так как нам требуется только один этап шейдинга и конвейер компоновки. Компоновка конвейера работает так же, как и с

```

1 графическим конвейером: VkPipelineLayoutCreateInfo
2 pipelineLayoutInfo{}; pipelineLayoutInfo.тип =
3 VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
4 pipelineLayoutInfo.setLayoutCount = 1;
5 pipelineLayoutInfo.pSetLayouts = &computeDescriptorsetLayout;
6
7 если (vkCreatePipelineLayout(устройство, &pipelineLayoutInfo,
 nullptr, &computePipelineLayout) != VK_SUCCESS) {
8 выбросить std::runtime_error("не удалось создать вычислительный конвейер"
 планировка!");
}

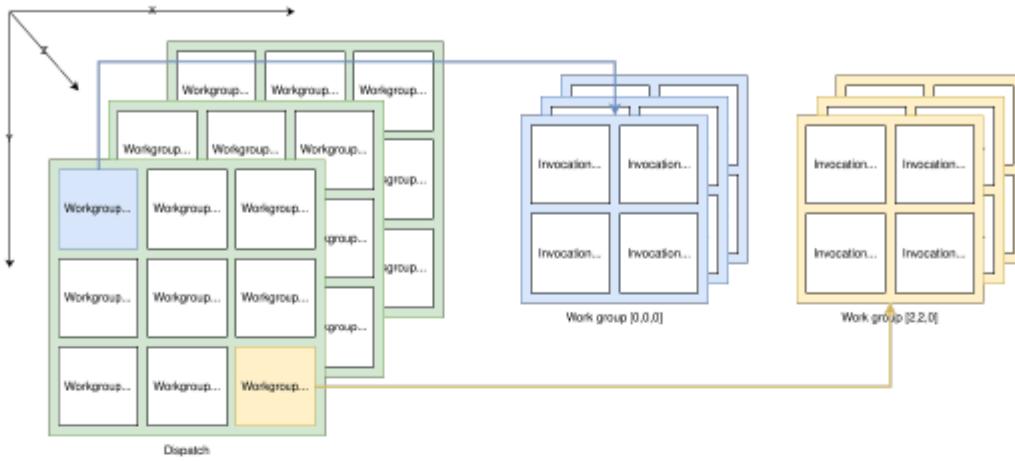
```

## Вычислительное пространство

Прежде чем мы перейдем к тому, как работает вычислительный шейдер и как мы передаем вычислительные нагрузки на графический процессор, нам нужно поговорить о двух важных концепциях вычислений: **рабочие группы и призывы**. Они определяют абстрактную модель выполнения для того, как вычислительные нагрузки обрабатываются вычислительным оборудованием графического процессора в трех измерениях (x, y и z).

**Рабочие группы** определят, как вычислительные нагрузки формируются и обрабатываются вычислительным оборудованием графического процессора. Вы можете думать о них как о рабочих элементах, с которыми должен работать графический процессор. Размеры рабочей группы устанавливаются приложением во время командного буфера с помощью команды отправки. И тогда каждая рабочая группа представляет собой набор **вызовов** которые выполняют один и тот же вычислительный шейдер. Вызовы потенциально могут выполняться параллельно, и их размеры устанавливаются в вычислительном шейдере. Вызовы внутри одной рабочей группы имеют доступ к общей памяти.

Это изображение показывает связь между этими двумя в трех измерениях:



Количество измерений для рабочих групп (определяется вызовы зависят (определяются локальными размерами в вычислительном шейдере) от того, как структурированы входные данные. Например, если вы работаете с одномерным массивом, как мы делаем в этой главе, вам нужно указать только размер x для обоих. В качестве примера: если мы отправим количество рабочих групп [64, 1, 1] с локальным размером вычислительного шейдера [32, 32, 1], наш вычислительный шейдер будет вызван  $64 \times 32 \times 32 = 65\,536$  раз. Обратите внимание, что максимальное количество рабочих групп и локальных размеров отличается от реализации к реализации, поэтому вам всегда следует проверять значение, связанное с вычислениями `maxComputeWorkGroupCount`, `maxComputeWorkGroupInvocations` и `maxComputeWorkGroupSize` ограничения в `VkPhysicalDeviceLimits`.

## Вычислительные шейдеры

Теперь, когда мы узнали обо всех деталях, необходимых для настройки вычислительного конвейера шейдеров, пришло время взглянуть на вычислительные шейдеры. Все, что мы узнали об использовании GLSL-шейдеров, например, для вершинных и фрагментных шейдеров, также применимо к вычислительным шейдерам. Синтаксис тот же, и многие концепции, такие как передача данных между приложением и шейдером, одинаковы. Но есть некоторые важные различия. Самый простой

вычислительный шейдер для обновления линейного массива частиц может

```

1 выглядеть следующим образом: # версия 4 5 0
2
3 макет (привязка = 0) uniform ParameterUBO {
4
5 float
6 deltaTime; } ubo;
7
8 структурная частица {
9 позиция vec 2;
```

```

9 скорость vec 2 ;
10 цвет vec 4 ;
11 };
12
13 компоновка (std 1 4 0 , привязка = 1) буфер без частиц только для чтения {
14 particlesIn[];
15 };
16
17 компоновка (std 1 4 0 , привязка = 2) буфер без частиц {
18 Particle particlesOut[];
19 };
20
21 расположение (local_size_x = 2 5 6 , local_size_y = 1 , local_size_z = 1) в;
22
23 void
24 main() {
25 uint index = gl_GlobalInvocationID.x;
26
27 Particle particleIn = particlesIn[индекс];
28
29 particlesOut[индекс].position = particleIn.position +
30 Частица.скорость.xy * убо.Скорость;
31 particlesOut[индекс].velocity =
32 particleIn.velocity; ...
33 }

```

Верхняя часть шейдера содержит объявления для входных данных шейдера. Первый - это объект uniform buffer с привязкой 0, о чём мы уже узнали в этом руководстве. Ниже мы объявляем нашу структуру частиц, которая соответствует объявлению в коде C++. Затем привязка 1 ссылается на объект shader storage buffer с данными частиц из последнего кадра (см. Настройку дескриптора), а привязка 2 указывает на SSBO для текущего кадра, который мы будем обновлять с помощью этого шейдера. Интересной вещью является это объявление только для вычислений, относящееся к вычислительному пространству:

```
1 макет (local_size_x = 2 5 6 , local_size_y = 1 , local_size_z = 1) в;
```

Это определяет количество вызовов этого вычислительного шейдера в текущей рабочей группе. Как отмечалось ранее, это локальная часть вычислительного пространства. Отсюда местные новости\_ префикс. Поскольку мы работаем с линейным одномерным массивом частиц, нам нужно только указать число для измерения x в local\_size\_x. The Главная затем функция считывает данные из SSBO последнего кадра и записывает обновленную позицию частицы в SSBO для текущего кадра. Подобно другим типам шейдеров, вычислительные шейдеры имеют свой собственный набор встроенных входных переменных. Встроенные

всегда имеют префикс `gl_`. Одним из таких встроенных является `gl_GlobalInvocationID`, переменная, которая однозначно идентифицирует текущий вызов вычислительного шейдера в текущей отправке. Мы используем это для индексации в нашем массиве частиц.

## Выполнение команд вычисления

### Отправка

Теперь пришло время фактически приказать графическому процессору выполнить некоторые вычисления. Это делается путем вызова внутри буфера команд. Хотя это и не совсем верно, а отправка предназначена для вычислений в виде вызова рисования, например, `vkCmdDraw` для графики. Это отправляет заданное количество вычислительных рабочих элементов с

```
1 максимальным интервалом, три измерения. VkCommandBufferBeginInfo beginInfo{};
2 beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
3
4 если (vkBeginCommandBuffer(commandBuffer, &beginInfo) != VK_SUCCESS) {
5 выбросить std::runtime_error("не удалось начать запись команды
6 буфера!");
7 }
8 ...
9
10 vkCmdBindPipeline(командный буфер, VK_PIPELINE_BIND_POINT_COMPUTE,
11 computePipeline);
12
13 vkCmdBindDescriptorSets(командный буфер,
14 VK_PIPELINE_BIND_POINT_COMPUTE, computePipelineLayout, 0, 1,
15 &computeDescriptorSets[i], 0, 0);
16
17 vkCmdDispatch(computeCommandBuffer, PARTICLE_COUNT / 256, 1, 1);
18
19 }
```

В `vkCmdDispatch` отправит `PARTICLE_COUNT / 256` локальные рабочие группы в измерение `x`. Поскольку наш массив частиц линейный, мы оставляем два других измерения равными единице, что приводит к одномерной отправке..... Но почему мы делим количество частиц (в нашем массиве) на 256? Это потому, что в предыдущем параграфе мы определили, что каждый вычислительный шейдер в рабочей группе будет выполнять 256 вызовов. Итак, если бы у нас было 4096 частиц, мы бы отправили 16 рабочих групп, причем каждая рабочая группа выполняла бы 256 вызовов вычислительных шейдеров. Получение правильных двух чисел обычно требует некоторой доработки и профилирования, в зависимости от вашей рабочей нагрузки и оборудования, на котором вы работаете. Если размер частиц

был бы динамическим и не всегда может быть разделен, например, на 256, вы всегда можете использовать `gl_GlobalInvocationID` в начале вашего вычислительного шейдера и возвращайтесь из него, если глобальный индекс вызова больше количества ваших частиц. И точно так же, как это было в случае с вычислительным конвейером, буфер вычислительных команд содержит намного меньше состояний, чем графический буфер команд. Нет необходимости запускать прохождение рендеринга или устанавливать видовой экран.

## Отправка работы

Поскольку в нашем примере выполняются как вычислительные, так и графические операции, мы будем выполнять две отправки как в графическую, так и в вычислительную очередь для каждого кадра (см. Функцию `drawFrame`):

```
1 if (vkQueueSubmit(computeQueue, 1, &submitInfo, nullptr) !=
2 VK_SUCCESS) {
3 выбросить std::runtime_error("не удалось отправить команду вычисления
4 буфера!");
5 }
6 if (vkQueueSubmit(graphicsQueue, 1, &submitInfo,
7 inFlightFences[Текущий фрейм]) !=
8 VK_SUCCESS) { выбросить std::runtime_error("не
 удалось отправить команду рисования 7
 буфера!");
9 }
```

Первая отправка в очередь вычислений обновляет позиции частиц с помощью вычислительного шейдера, а вторая отправка затем использует эти обновленные данные для построения системы частиц.

## Синхронизация графики и вычислений

Синхронизация - важная часть Vulkan, особенно при выполнении вычислений в сочетании с графикой. Неправильная или отсутствующая синхронизация может привести к тому, что вершинный этап начнет отрисовывать (=считывать) частицы, в то время как вычислительный шейдер не завершил их обновление (=записывать) (опасность чтения после записи), или же вычислительный конечный шейдер может начать обновлять частицы, которые все еще используются вершинной частью конвейера (опасность записи после чтения). Поэтому мы должны убедиться, что таких случаев не произойдет, правильно синхронизировав графику и вычислительную нагрузку. Существуют разные способы сделать это, в зависимости от того, как вы отправляете свою вычислительную нагрузку, но в нашем случае с двумя отдельными отправками мы будем использовать семафоры и ограждения, чтобы гарантировать, что ver-текс-шейдер не начнет извлекать вершины, пока вычислительный шейдер не завершит их обновление.

Это необходимо, поскольку, несмотря на то, что две отправки упорядочены одна за другой, нет гарантии, что они выполняются на графическом процессоре в этом порядке.

Добавление семафоров ожидания и сигнала обеспечивает такой порядок выполнения. Итак, сначала мы добавляем новый набор примитивов синхронизации для вычислительной работы. Вычислительные барьеры, так же как и графические барьеры, являются создано в сигнализированном состоянии, потому что в противном случае время первого розыгрыша истекло бы в ожидании подачи сигнала на ограждения, как описано здесь:

```

1 std::vector<VkFence> computeInFlightFences;
2 std::vector<VkSemaphore> Вычисленные семафоры;
3 ...
4 computeInFlightFences.resize(MAX_FRAMES_IN_FLIGHT);
5 Вычисленные семафоры.изменение размера (MAX_FRAMES_IN_FLIGHT);
6
7 VkSemaphoreCreateInfo semaphoreInfo{};
8 semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
9
10 VkFenceCreateInfo fenceInfo{};
11 fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
12 fenceInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;
13
14 для (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
15 ...
16 если (vkCreateSemaphore(устройство, &semaphoreInfo, nullptr,
17 &computeFinishedSemaphores[i]) != VK_SUCCESS ||
18 vkCreateFence(устройство, &fenceInfo, nullptr,
19 &computeInFlightFences[i]) != VK_SUCCESS) {
20 выбросить std::runtime_error("не удалось создать compute
21 объекты синхронизации для фрейма!");
22 }
23 }
```

Затем мы используем их для синхронизации отправки в вычислительный буфер с отправкой в graph-ics:

```

1 // Отправка вычислений
2 vkWaitForFences(устройство, 1, &computeInFlightFences [текущий фрейм],
3 VK_TRUE, UINT64_MAX);
4 updateUniformBuffer(текущий кадр);
5
6 Встроенные защиты (устройства, 1 и computeInFlightFences [текущий кадр]);
7
8 vkResetCommandBuffer(computeCommandBuffers[текущий кадр],
9 /* VkCommandBufferResetFlagBits */ 0);
9 recordComputeCommandBuffer(вычислительные командные буфера[текущий кадр]);
```

```

10 submitInfo.commandBufferCount = 1;
11 submitInfo.pCommandBuffers = &computeCommandBuffers[текущий
12 кадр]; submitInfo.signalSemaphoreCount = 1;
13 submitInfo.pSignalSemaphores =
14 &Вычисленные семафоры[Текущий кадр];
15
16 if (vkQueueSubmit(computeQueue, 1, &submitInfo,
17 computeInFlightFences[Текущий фрейм])
18 != VK_SUCCESS) { выбросить std::runtime_error("не
19 удалось отправить команду вычисления буфер!");
20 }
21
22 // Представление графики
23 vkWaitForFences(устройство, 1, и бортовые защиты [текущий кадр], VK_TRUE,
24 UINT 6 4 _MAX);
25
26 ...
27 Встроенные защиты (устройство, 1 и бортовые защиты [текущий кадр]);
28
29 vkResetCommandBuffer(Командные буферы[текущий кадр],
30 /* VkCommandBufferResetFlagBits */ 0);
31 recordCommandBuffer(Командные буферы[текущий кадр], ImageIndex);
32
33 VkSemaphore waitSemaphores[] = {
34 computeFinishedSemaphores[текущий кадр],
35 доступные изображения [текущий кадр] };
36 VkPipelineStageFlags waitStages[] = {
37 VK_PIPELINE_STAGE_VERTEX_INPUT_BIT,
38 VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT };
39 submitInfo = {};
40 submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
41 submitInfo.waitSemaphoreCount = 2;
42 submitInfo.pWaitSemaphores = waitSemaphores;
43 submitInfo.pWaitStageMask = waitsstages;
44 submitInfo.commandBufferCount = 1;
45
46 submitInfo.pCommandBuffers = &commandBuffers[текущий
47 кадр]; submitInfo.signalSemaphoreCount = 1;
48 submitInfo.pSignalSemaphores =
49 &renderFinishedSemaphores[Текущий кадр];
50
51 if (vkQueueSubmit(graphicsQueue, 1, &submitInfo,
52 Летные ограждения [текущий кадр]) != VK_SUCCESS) { выбросить std::runtime_error("не
53 удалось отправить команду рисования");

```

```
 буфер! ");
45 }
```

Подобно образцу из главы о семафорах, эта настройка немедленно запустит вычислительный шейдер, поскольку мы не указали никаких семафоров ожидания. Это нормально, поскольку мы ожидаем завершения выполнения буфера команды вычисления текущего кадра перед отправкой вычисления с помощью `vkWaitForFences` команды. С другой стороны, отправке графики необходимо дождаться завершения работы по вычислению, чтобы не начинать выборку вершин, пока вычислительный буфер все еще обновляет их.

Итак, мы ждем на `computeFinishedSemaphores` текущего кадра, а отправка графики ожидает на `VK_PIPELINE_STAGE_VERTEX_INPUT_BIT` этап, на котором используются вершины. Но ему также необходимо дождаться представления, чтобы фрагментный шейдер не выводил данные в цветные вложения до тех пор, пока изображение не будет представлено. Итак, мы также ждем доступные изображения-семафоры на текущем кадре на этапе `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`.

## Рисование системы частиц

Ранее мы узнали, что буфера в Vulkan могут иметь несколько вариантов использования, и поэтому мы создали буфер хранения шейдеров, который содержит наши частицы, используя как бит буфера хранения шейдеров, так и бит вершинного буфера. Это означает, что мы можем использовать буфер хранения шейдеров для рисования точно так же, как мы использовали "чистые" вершинные буфера в предыдущих главах. Сначала мы настраиваем входное состояние

```
1 вершины в соответствии с нашей структурой частиц: struct Particle {
2
3
4 статический std::массив<VkVertexInputAttributeDescription, 2>
5 getAttributeDescriptions() {
6 std::array<VkVertexInputAttributeDescription, 2>
7 Атрибутивные описания{};
8
8 Атрибутивные описания[0].привязка = 0 ;
9 Атрибутивные описания[0].местоположение = 0 ;
10 Атрибутивные описания[0].формат = VK_FORMAT_R3_2_G3_2_SFLOAT;
11 Атрибутивные описания[0].смещение(частица,
12 должность);
13
12 attributeDescriptions[1].привязка = 0 ;
13 attributeDescriptions[1].местоположение = 1 ;
```

```

14 Атрибутивные описания[1].формат =
15 VK_FORMAT_R3_2_G3_2_B3_2_A3_2_SFLOAT;
16 Атрибутивные описания[1].смещение = смещение (частица, цвет);
17 Возврат attributeDescriptions;
18 }
19 };

```

Обратите внимание, что мы не добавляем `velocity` к входным атрибутам вершины, поскольку это используется только вычислительным шейдером.

Затем мы связываем и рисуем его, как мы делали бы с любым вершинным

```

1 буфером: vkCmdBindVertexBuffers(commandBuffer, 0, 1,
2 &shaderStorageBuffer[текущий кадр], смещения);
3 vkCmdDraw(commandBuffer, PARTICLE_COUNT, 1, 0, 0);

```

## Заключение

В этой главе мы узнали, как использовать вычислительные шейдеры для переноса работы с центрального процессора на графический. Без вычислительных шейдеров многие эффекты в современных играх и приложениях были бы либо невозможны, либо работали бы намного медленнее. Но даже больше, чем графика, `compute` имеет множество вариантов использования, и эта глава дает вам лишь краткое представление о том, что возможно. Итак, теперь, когда вы знаете, как использовать вычислительные шейдеры, возможно, вы захотите ознакомиться с некоторыми расширенными разделами вычислений, такими как:

- Общая память
- Асинхронные вычисления • Атомарные операции • Подгруппы

Вы можете найти несколько примеров расширенных вычислений в официальном репозитории примеров Khronos Vulkan.  
[Код на C++ / Вершинный шейдер / Фрагментный шейдер / Compute shader](#)

## **Вопросы и ответы**

На этой странице перечислены решения распространенных проблем, с которыми вы можете столкнуться при разработке приложений Vulkan.

### **Я получаю сообщение об ошибке нарушения доступа на уровне проверки ядра.**

Убедитесь, что сервер статистики MSI Afterburner / RivaTuner не запущен, поскольку у него есть некоторые проблемы с совместимостью с Vulkan.

### **Я не вижу никаких сообщений от уровней проверки - уровни / Уровни проверки недоступны**

Сначала убедитесь, что уровни проверки имеют возможность печатать ошибки, сохранив терминал открытым после завершения работы вашей программы. Вы можете сделать это из Visual Studio, запустив свою программу с помощью Ctrl-F5 вместо F5, а в Linux - выполнив свою программу из окна терминала. Если сообщений по-прежнему нет и вы уверены, что уровни проверки включены, то вам следует убедиться, что ваш Vulkan SDK установлен правильно, следуя инструкциям "Проверьте установку" на этой странице. Также убедитесь, что версия вашего SDK не ниже 1.1.106.0 для поддержки слоя.

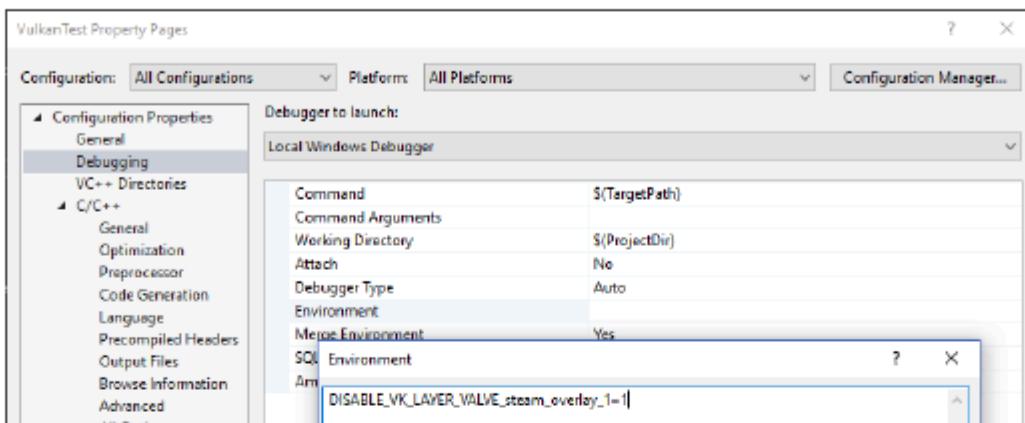
### **vkCreateSwapchainKHR вызывает ошибку в SteamOverlayVulkanLayer64.dll**

Похоже, это проблема совместимости в бета-версии клиента Steam. Там вот несколько возможных обходных путей: \* Отказаться от бета-тестирования Steam. Установите для переменной окружения значение

1 \* Удалите запись Steam overlay Vulkan layer в реестре в разделе

HKEY\_LOCAL\_MACHINE\SOFTWARE\Khronos\Vulkan\ImplicitLayers

Пример:



## Ошибка vkCreateInstance с помощью VK\_ERROR\_INCOMPATIBLE\_DRIVER

Если вы используете macOS с последней версией MoltenVK SDK, то `vkCreateInstance` может возвращать ошибку. Это связано с тем, что для Vulkan SDK версии 1.3.216 или новее требуется включить `VK_KHR_PORTABILITY_subset` расширение для использования MoltenVK, поскольку оно в настоящее время не полностью соответствует требованиям. Необходимо добавить `VK_INSTANCE_CREATE_ENUMERATE_PORTABILITY_BIT_KHR` установите флагок для вашего `VkInstanceCreateInfo` и добавьте `VK_KHR_PORTABILITY_ENUMERATION_EXTENSION_NAME` в список расширений вашего экземпляра.

Пример кода:

```

1 ...
2 ...
3 std::vector<постоянный символ * > Требуемые расширения;
4
5 для(uint32_t i = 0; i < glfwExtensionCount; i++) {
6 requiredExtensions.emplace_back(glfwExtensions[i]);
7 }
8
9 Требуемые расширения.emplace_back(VK_KHR_PORTABILITY_ENUMERATION_EXTENSION_NAME);
10
11 CreateInfo.flags |= VK_INSTANCE_CREATE_ENUMERATE_PORTABILITY_BIT_KHR;
12
13 CreateInfo.enabledExtensionCount = (uint32_t)
14 Требуемые расширения.size();
15
16 если (vkCreateInstance(&CreateInfo, nullptr, &instance) != VK_SUCCESS)
17 {
18 выбросить std::runtime_error ("не удалось создать экземпляр!");
19 }
```