Содержание

Об этой книге

Это книга посвящена Vulkan. Vulkan – это программный интерфейс (API) для управления такими устройствами, как графические процессоры (GPU). Хотя Vulkan является логическим преемником OpenGL, он очень сильно от него отличается. Одним из таких отличий, которое стразу заметят опытные программисты, является его избыточность. Вам нужно будет написать очень много кода, чтобы Vulkan сделал что-то полезное, не говоря уже о чем-то заметном. Большинство из того, что раньше делал драйвер OpenGL, теперь является обязанностью программиста. Это включает в себя синхронизацию, планирование, управление памятью и т. п. В результате вы найдете, что большая часть этой книги посвящена подобным темам, даже хотя они являются более общими темами, нежели сам Vulkan.

Эта книга ориентирована на опытных программистов, которые уже знакомы с другими графическими и вычислительными API. Соответственно, многие темы, связанные с графикой, рассматриваются без глубокого введения, есть некоторые отсылки вперед и примеры кода не закончены или иллюстративны, а не являются законченными программами, которые вы могли бы набрать. Весь исходный код, доступный на сайте книги, полон и протестирован и может служить хорошей отправной точкой.

Vulkan предназначается для использования в качестве интерфейса между большими, сложными графическими и вычислительными приложениями и GPU. Большинство возможностей, ранее реализовываемых драйверами, реализующими графические API вроде OpenGL, теперь является ответственностью приложения. Сложные игровые движки, большие пакеты для рендеринга и сложное программное обеспечение хорошо подходит для этой задачи; у них больше информации о своем поведении, чем может быть у любого драйвера. Vulkan плохо подходит для простых примеров или для обучения основам графики.

В первой главе этой книги мы представляем Vulkan и некоторые его фундаментальные понятия. По мере продвижения через Vulkan мы будем рассматривать более сложные темы, постепенно строя систему рендеринга, которая будет показывать некоторые уникальные стороны Vulkan и демонстрировать его возможности.

В главе 1 «Обзор Vulkan» мы дадим краткое введение в Vulkan и понятия, которые образуют его основу. Мы рассмотрим создание объектов Vulkan и покажем основы начала работы с Vulkan.

В главе 2 «Память и ресурсы» мы познакомимся с системой памяти в Vulkan, пожалуй, наиболее важной частью этого интерфейса. Мы покажем, как выделять память для использования устройствами Vulkan и драйвером Vulkan, а также различными компонентами внутри вашего приложения.

В главе 3 «Очереди и команды» мы рассмотрим командные буферы и *очереди*, в которые они помещаются. Мы покажем, как работают процессы Vulkan и как

ваше приложение может строить пакеты команд для отправки на GPU для выполнения.

В главе 4 «Перемещение данных» мы рассмотрим несколько наших первых команд Vulkan, все из которых основаны на перемещении данных. Мы будем использовать понятия, впервые рассмотренные в главе 3, для построения командных буферов, которые могут копировать и организовывать данные, хранящиеся в ресурсах и памяти и введенные в главе 2.

В главе 5 «Показ» мы расскажем, как показать полученные вашим приложением изображения на экране. Показ (presentation) – это термин, используемый для взаимодействия с оконной системой, зависящей от платформы, поэтому в этой главе рассматриваются некоторые зависящие от платформы аспекты.

В главе 6 «Шейдеры и конвейеры» мы представим SPIR-V, бинарный язык шейдеров, используемый Vulkan. Также мы введем объект-конвейер, покажем, как его можно построить из шейдеров на SPIR-V, и далее рассмотрим вычислительные конвейеры, которые могут быть использованы для выполнения расчетов с использованием Vulkan.

В главе 7 «Графические конвейеры» мы на основе материала из главы 6 введем *графический конвейер*, включающий в себя всю необходимую конфигурацию для рендеринга графических примитивов при помощи Vulkan.

В главе 8 «Рендеринг» мы рассмотрим различные команды для рендеринга, имеющиеся в Vulkan, включая индексированный и неиндексированный рендеринг, дублирование геометрии (instancing) и непрямые (indirect) команды. Мы покажем, как передать данные в графический конвейер и как выводить более сложную геометрию, чем та, что была рассмотрена в главе 7.

В главе 9 «Обработка геометрии» мы глубже рассмотрим первую половину графического конвейера Vulkan и тесселляционные и геометрические шейдеры. Мы покажем более продвинутые возможности, которые эти шейдеры могут выполнять, и рассмотрим весь конвейер вплоть до растеризации.

В главе 10 «Обработка фрагментов» мы продолжим рассмотрение, начатое в главе 9, и рассмотрим все, что происходит во время и после растеризации для превращения вашей геометрии в поток пикселов, которые могут быть показаны пользователю.

В главе 11 «Синхронизация» мы рассмотрим различные примитивы для синхронизации, доступные в приложении на Vulkan, включая *барьеры* (fence), *события* (event) и *семафоры*. Вместе они образуют основу любого приложения, эффективно использующего параллельную природу Vulkan.

В главе 12 «Получение данных назад» мы обратим направление взаимодействия из предыдущих глав и рассмотрим детали получения данных из Vulkan в ваше приложение. Мы покажем, как замерять затраченное GPU на выполнение операции время, как собирать статистику по операциям устройств и получать данные от Vulkan обратно в ваше приложение.

Наконец, в главе 13 «Многопроходный рендеринг» мы снова вернемся к некоторым рассмотренным ранее темам, соединяя их вместе для получения более

сложного приложения – приложения для отложенного рендеринга с использованием многопроходной архитектуры и нескольких очередей для обработки.

Приложение к этой книге содержит таблицу функций для построения командных буферов с краткой информацией по используемым ими атрибутам.

Vulkan – это большая, сложная и новая система. Очень сложно рассмотреть каждый ее аспект в такой книге. Мы рекомендуем читателю, кроме этой книги, внимательно ознакомиться со спецификациями Vulkan, а также другими книгами по разнородным вычислительным системам и компьютерной графике на основе иных API. Подобные материалы содержат хорошую математическую базу и описания других используемых в книге понятий.

Об исходном коде

Исходный код к этой книге может быть скачан с нашего сайта (http://www.vulkan-programmingguide.com). Одна из особенностей Vulkan, которую сразу же заметят опытные программисты на других графических API, – это его избыточность (многословность). В первую очередь это связано с тем, что многие задачи, которые традиционно выполнялись драйверами, теперь переданы вашему приложению. Во многих случаях, однако, небольшой базовый код вполне справится с этой задачей. Поэтому мы подготовили простую библиотеку, занимающуюся задачами, общими для всех примеров и многих настоящих приложений. Это не значит, что данная книга является учебником по использованию данной библиотеки. Она служит для ясности и краткости примеров кода.

Конечно, по мере рассмотрения той или иной функциональности Vulkan на протяжении всей книги мы будем приводить примеры кода, многие из которых на самом деле принадлежат именно нашей библиотеке, а не тому или иному примеру. Некоторые возможности, рассматриваемые в этой книге, не содержат соответствующих примеров кода. Это в первую очередь касается продвинутых возможностей, относящихся в основном к полномасштабным серьезным приложениям. Не бывает коротких и простых примеров на Vulkan. Во многих случаях простой пример показывает использование различных возможностей. Возможности, используемые тем или иным примером, перечислены в readme-файле к этому примеру. Но при этом нет соответствия 1:1 между примерами и листингами в книге и конкретными примерами в исходном коде. Считается, что любой, кто требует списка соответствий 1:1 между примерами и главами, просто не прочел этого параграфа. Соответствующие запросы будут просто закрываться со ссылкой на этот параграф.

Исходный код рассчитан на сборку с последним Vulkan SDK от LunarG, который можно скачать по адресу http://lunarg.com/vulkan-sdk. На момент написания последней версией SDK была 1.0.22. Более новые версии SDK должны быть обратно совместимы со старыми версиями, поэтому мы советуем взять последнюю версию SDK перед компиляцией и выполнением примеров. SDK также содержит некоторое количество своих примеров, и мы советуем запустить их, для того чтобы убедиться в том, что драйвер и SDK установлены корректно.

Кроме Vulkan SDK, вам также понадобится работающий CMake для создания проектов для сборки примеров. Также вам нужен современный компилятор. Примеры используют некоторые особенности языка C++ 11 и стандартные библиотеки C++ для таких вещей, как работа с нитями и синхронизация. Эти возможности содержали ошибки в ранних версиях компиляторов, так что, пожалуйста, используйте современный компилятор. Мы проверили все на Microsoft Visual Studio 2015 на Windows и GCC 5.3 на Linux. Примеры были проверены на 64-битных Windows 7, Windows 10 и Ubuntu 10.16 с последними драйверами от AMD, NVidia и Intel.

Необходимо заметить, что Vulkan является кросс-платформенным и кросс-вендорным. Многие из этих примеров *должны* работать на Android и других мобильных платформах. Мы надеемся портировать примеры на как можно большее число платформ в будущем и очень оценим вашу (читателя) помощь и вклад.

Ошибки

Vulkan является новой технологией. На момент написания книги спецификации были в доступе на протяжении всего нескольких недель. Хотя авторы и работали над созданием спецификаций Vulkan, он очень большой, и над ним работало большое число людей. Что-то в этой книге не полностью протестировано, и хотя мы верим в правильность, могут быть ошибки. В процессе сборки всех примеров вместе доступные реализации Vulkan все еще имели ошибки, слои валидации все еще не ловили всех ошибок, как следовало бы, и спецификации содержали пропуски и неясные места. Так же как и читатели, мы сами все еще изучаем Vulkan, поэтому, хотя текст и был проверен на точность, мы надеемся на помощь читателей через наш сайт: http://www.vulkanprogrammingguide.com.

Об авторе

Грехем Селлерс – архитектор программных систем в AMD, занимающийся разработкой драйверов OpenGL и Vulkan для продуктов AMD Radeon и FirePro. Его страсть к компьютерам и технологиям началась в раннем возрасте с BBC Micro, за которым последовала большая цепочка 8- и 16-битовых домашних компьютеров, которые ему все еще нравятся. Он получил степень магистра в университете Саутхэмптона, Англия, и теперь живет в Орландо, штат Флорида, со своей женой и двумя детьми.

Глава 1

Обзор Vulkan

Что вы узнаете в этой главе:
- что такое Vulkan и что лежит в его основе;
- как создать минимальное приложение, использующее Vulkan;
- терминология и понятия, используемые на протяжении всей книги.

В этой главе мы представим Vulkan и объясним, что это такое. Мы представим некоторые базовые понятия данного API, включая инициализацию, время жизни объектов, экземпляр (instance) Vulkan и логическое и физическое устройства. К концу главы мы создадим простое приложение на Vulkan, которое инициализирует систему Vulkan, находит доступные устройства, показывает их свойства и возможности и, наконец, аккуратно завершает работу.

Введение

Vulkan – это программный интерфейс для графических и вычислительных устройств. Устройство Vulkan обычно состоит из процессора и некоторого количества блоков с зашитой функциональностью для ускорения операций, используемых в графических и вычислительных задачах. Процессор в таком устройстве обычно является устройством, поддерживающим очень большое число нитей, поэтому вычислительная модель в Vulkan базируется на параллельных вычислениях. Устройство Vulkan также имеет доступ к памяти, которая может быть (а может и не быть) общей с процессором, на котором выполняется ваше приложение.

Vulkan – это явный API. Это значит, что практически все является вашей обязанностью. Драйвер – это программа, которая берет команды и данные, образующие API, и переводит их во что-то, что аппаратура может понять. В старых API, таких как OpenGL, драйвер отслеживает состояние кучи объектов, управляет для вас памятью и синхронизацией, проверяет на ошибки во время выполнения вашего приложения. Это очень здорово для программистов, но это потребляет драгоценное время CPU, после того как вы закончили отладку и ваше приложение работает верно. Vulkan борется с этим, передавая практически все отслеживание состояния, синхронизацию и управление памятью в руки программиста и делегируя проверки правильности *слоям*, которые необходимо явно включать. Они не участвуют в работе вашего приложения в нормальных условиях.

По этим причинам Vulkan довольно многословен и в чем-то хрупок. Вам нужно выполнить огромный объем работы, для того чтобы Vulkan работал хорошо, и неверное использование API часто может привести к порче графики или даже падению программы в тех случаях, когда более старые API просто выдали бы полезное сообщение об ошибке. В обмен на это Vulkan предоставляет больше контроля над устройством, ясную многонитевую модель и гораздо более высокое быстродействие, чем те API, которые он заменяет.

Также Vulkan был разработан для того, чтобы быть больше, чем просто *графический* API.Он может использоваться для разнородных устройств, таких как графические процессоры (GPU), DSP и оборудование, использующее заранее заданную функциональность. Функциональность разделяется на несколько больших пересекающихся между собой категорий. Текущая версия Vulkan определяет категорию переноса, которая используется для копирования данных; вычислительную категорию, которая используется для выполнения вычислительных шейдеров; и графическую категорию, включающую растеризацию, сборку примитивов, смешивание, тест глубины и трафарета и другую функциональность, знакомую программистам графики.

До определенной степени поддержка каждой из этих категорий необязательна, может быть устройство Vulkan, которое вообще не поддерживает графику. Как следствие даже API, предназначенный для вывода картинки на экран (называемый *представлением*), не просто является необязательным, но и предоставлен как *расширение* Vulkan, а не его базовая часть.

Экземпляры, устройства и очереди

Vulkan включает в себя иерархию различной функциональности, начинающуюся сверху с *экземпляра* (instance), собирающего все поддерживающие Vulkan устройства вместе. Каждое устройство предоставляет одну или несколько *очередей*. Именно через эти очереди и выполняется вся запрашиваемая вашим приложением работа.

Экземпляр Vulkan – это программная конструкция, которая логически отделяет состояние вашего приложения от других приложений или от библиотек, выполняемых в контексте вашего приложения. Физические устройства в системе представлены как члены экземпляра, каждое из них имеет различные возможности, включая набор поддерживаемых очередей.

Физическое устройство обычно представлено отдельным устройством или набором связанных между собой устройств. В любой заданной системе существует фиксированное и конечное число устройств, конечно, если не поддерживает переконфигурацию прямо на ходу. Логическое устройство, создаваемое экземпляром, – это программная конструкция, оборачивающая физическое устройство и представляющая зарезервированный набор ресурсов, связанных с конкретным физическим устройством. Оно включает в себя возможное подмножество доступных очередей на физическом устройстве. Можно создать несколько различных логических устройств, представляющих одно и то же физическое устройство,

и именно с логическим устройством ваше приложение и будет работать большую часть времени.

На рис. 1.1 изображена эта иерархия. На рисунке приложение создало два экземпляра Vulkan. В системе есть три физических устройства, доступных обоим экземплярам. Приложение создает одно логическое устройство на первом физическом устройстве, два логических устройства на втором физическом устройстве и еще одно устройство на третьем. Каждое логическое устройство задействует определенное подмножество очередей соответствующего физического устройства. На самом деле многие приложения, использующие Vulkan, не будут так сложны и просто создадут одно логическое устройство для одного из физических устройств, используя всего один экземпляр Vulkan. Рисунок 1.1 служит просто демонстрацией гибкости системы Vulkan.

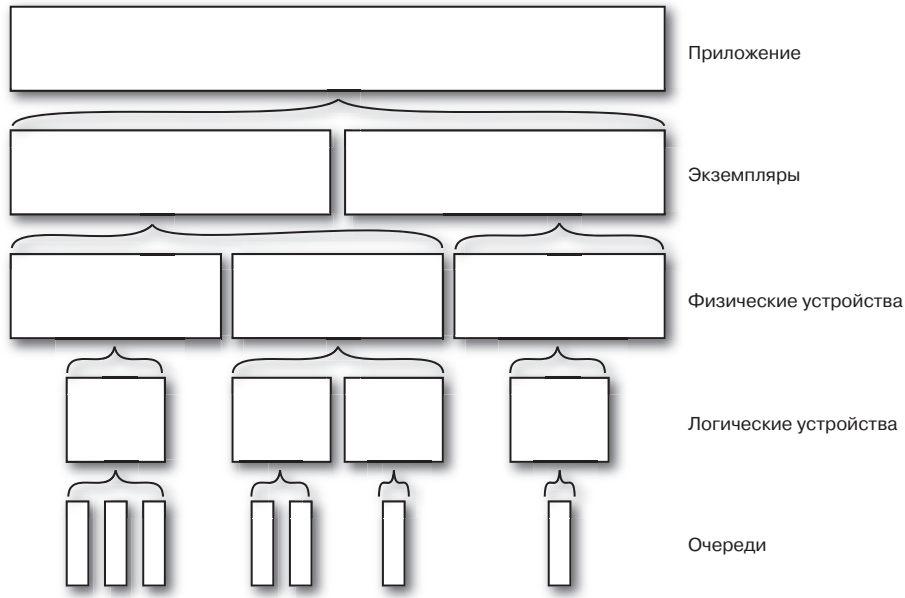

Рис. 1.1 ❖ Иерархия Vulkan из экземпляров, устройств и очередей

Следующие разделы рассмотрят, как создать экземпляр Vulkan, получить доступные физические устройства в системе, подсоединить логическое устройство к одному из них и, наконец, получить дескрипторы очередей, предоставляемых устройством.

Экземпляр Vulkan

Vulkan может рассматриваться как подсистема вашего приложения. После того как ваше приложение подключает и инициализирует библиотеки Vulkan, оно отслеживает определенное состояние. Поскольку Vulkan не предоставляет никакого

глобального состояния вашему приложению, все отслеживаемое состояние должно храниться в предоставляемом вами объекте. Этот объект и является *экземпляром*, и он представлен как экземпляр `VkInstance`. Для создания такого объекта мы позовем нашу первую функцию Vulkan, `vkCreateInstance`, прототип которой приведен ниже.

```
VkResult vkCreateInstance (
   const VkInstanceCreateInfo*  pCreateInfo,
   const VkAllocationCallbacks* pAllocator,
   VkInstance*                  pInstance);
```

Это описание типично для функций Vulkan. Когда нужно передать много аргументов функции, часто используются указатели на структуры. Здесь `pCreateInfo` является указателем на структуру `VkInstanceCreateInfo`, содержащую параметры, описывающие экземпляр. Ниже приводится определение структуры `VkInstanceCreateInfo`.

```
typedef struct VkInstanceCreateInfo {
  VkStructureType           sType;
  const void*               pNext;
  VkInstanceCreateFlags     flags;
  const VkApplicationInfo*  pApplicationInfo;
  uint32_t                  enabledLayerCount;
  const char* const*        ppEnabledLayerNames;
  uint32_t                  enabledExtensionCount;
  const char* const*        ppEnabledExtensionNames;
} VkInstanceCreateInfo;
```

Первым параметром практически любой структуры Vulkan, используемым для передачи параметров API, является поле `sType`, которое сообщает Vulkan тип данной структуры. Каждой такой структуре в базовом API и в расширениях выделен свой тег. Проверяя этот тег, инструменты Vulkan, слои и драйверы могут определить тип структуры для проверки (валидации) и использования в расширениях. Далее, поле `pNext` позволяет передать функции *связанный список* структур. Это дает возможность расширять множество параметров без изменения соответствующей структуры. Поскольку мы используем структуру для создания экземпляра, то мы передаем `VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO` как `sType` и `nullptr` в качестве `pNext`.

Поле `flags` структуры `VkInstanceCreateInfo` зарезервировано для дальнейшего использования и должно быть равным нулю. Следующее поле, `pApplicationInfo`, является необязательным указателем на другую структуру, описывающую ваше приложение. Вы можете использовать для него значение `nullptr`, но хорошие приложения должны передать через него что-то полезное. `pApplicationInfo` указывает на структуру `VkApplicationInfo`, описание которой приводится ниже.

```
typedef struct VkApplicationInfo {
  VkStructureType sType;
  const void*     pNext;
```

```
  const char*     pApplicationName;
  uint32_t        applicationVersion;
  const char*     pEngineName;
  uint32_t        engineVersion;
  uint32_t        apiVersion;
} VkApplicationInfo;
```

Опять мы видим поля `sType` и `pNext` в этой структуре. Поле `sType` должно быть равно `VK_STRUCTURE_TYPE_APPLICATION_INFO`, а в качестве `pNext` можно использовать `nullptr`. Поле `pApplicationName` является указателем на строку, завершенную нулевым байтом, содержащую имя вашего приложения, а `applicationVersion` – это версия вашего приложения. Это позволяет инструментам и драйверам принимать решение о том, как нужно относиться к вашему приложению без необходимости гадать[1], что за приложение выполняется. Аналогично `pEngineName` и `engineVersion` содержат имя и версию движка или библиотеки, используемой вашим приложением.

Наконец, `apiVersion` содержит версию Vulkan API, на которую рассчитывает ваше приложение. Оно должно быть равно *минимальной* версии Vulkan, которая требуется вашему приложению для работы, а не версии заголовка, которую вы установили. Это позволяет выполнять ваше приложение на наибольшем ассортименте устройств и платформ, даже если для них не доступны обновления Vulkan.

Возвращаясь к структуре `VkInstanceCreateInfo`, мы видим поля `enabledLayerCount` и `ppEnabledLayerNames`. Они содержат число *слоев экземпляра* (instance layer), которое вы хотите разрешить, и их имена соответственно. Слои используются для перехвата вызовов Vulkan API для логгирования, профилировки, отладки или других дополнительных возможностей. Аналогично `enabledExtensionCount` задает число расширений, которые вы хотите включить[2], и `ppEnabledExtensionNames` является списком их имен. Если вы не используете никаких расширений, то можете установить эти поля равными нулю и `nullptr` соответственно.

Наконец, возвращаясь к функции `vkCreateInstance()`, параметр `pAllocator` указывает на аллокатор памяти CPU, который ваше приложение может передать для

---

[1] То, что является наилучшим для одного приложения, может отличаться от наилучшего для другого приложения. Кроме того, приложения пишут люди, а люди пишут код с ошибками. Для полной оптимизации или обхода ошибок приложения драйвера должны иногда использовать имена выполнимых фалов или даже поведение приложения, чтобы понять, что именно выполняется, и вести себя соответственно. Хотя подобное решение неидеально, оно убирает такое угадывание.

[2] Как и OpenGL, Vulkan поддерживает расширения как базовую часть API. Однако в OpenGL мы создаем контекст, запрашиваем поддерживаемые расширения и затем начинаем их использовать. Это означает, что драйвера должны предполагать, что ваше приложение в любой момент может внезапно начать использовать расширение, и должны быть к этому готовы. Более того, драйвер не может заранее определить, какие именно расширения вам нужны, что еще больше затрудняет этот процесс. В Vulkan приложения обязаны запросить расширения и явно их разрешить. Это позволяет драйверам выключать неиспользующиеся расширения и делает для приложения сложнее случайно начать использовать функциональность, являющуюся частью расширения, которое не планировалось включать.

управления используемой Vulkan памятью. При передаче `nullptr` Vulkan будет использовать свой собственный внутренний аллокатор, что мы и будем делать далее. Управление памятью CPU будет рассмотрено в главе 2 «Память и ресурсы».

В случае успеха функция `vkCreateInstance()` возвращает `VK_SUCCESS` и помещает *дескриптор* нового экземпляра в переменную, на которую указывает параметр `pInstance`. Дескриптор – это значение, при помощи которого мы ссылаемся на объекты. Все дескрипторы Vulkan всегда являются 64-битовыми, независимо от битности вашей операционной системы. После того как мы получили дескриптор нашего экземпляра, мы можем использовать его для вызова функций, связанных с экземпляром.

Физические устройства Vulkan

После того как у нас есть экземпляр Vulkan, мы можем найти все совместимые с Vulkan устройства в системе. В Vulkan есть два типа устройств – физические и логические. Физические устройства – это обычные части системы – графические карты, ускорители, DSP или другие компоненты. Есть фиксированное количество физических устройств в системе, и каждое из них имеет свой набор возможностей.

Логическое устройство – это программная абстракция физического устройства, сконфигурированная в соответствии с тем, как задано приложением. Почти все свое время ваше приложение будет иметь дело именно с логическим устройством, но, прежде чем мы создадим логическое устройство, нам необходимо определить подключенные физические устройства. Для этого мы вызовем функцию `vkEnumeratePhysicalDevices()`, прототип которой приводится ниже.

```
VkResult vkEnumeratePhysicalDevices (
  VkInstance        instance,
  uint32_t*         pPhysicalDeviceCount,
  VkPhysicalDevice* pPhysicalDevices);
```

Первый параметр функции `vkEnumeratePhysicalDevices()`, `instance`, – это экземпляр, который мы создали ранее. Далее, параметр `pPhysicalDeviceCount` указывает на беззнаковую целочисленную переменную, которая является как входным, так и выходным параметром. На выходе Vulkan записывает в эту переменную число физических устройств в системе. На входе там должно содержаться максимальное число устройств, которое ваше приложение может обработать. Параметр `pPhysicalDevices` – это указатель на массив из дескрипторов `VkPhysicalDevice`.

Если вы хотите узнать, сколько всего устройств в системе, то задайте в качестве значения для `pPhysicalDevices` `nullptr`, тогда Vulkan проигнорирует начальное значение параметра `pPhysicalDeviceCount` и просто запишет в него число поддерживаемых устройств. Вы можете динамически управлять размером массива из `VlPhysicalDevice` за счет двухкратного вызова `vkEnumeraterPhysicalDevices()`, вначале вы передаете в качестве `pPhysicalDevices` `nullptr` (при этом `pPhysicalDeviceCount` должен быть валидным указателем), и для второго раза `pPhysicalDevices` равен указателю на массив соответствующего размера (возвращенного первым вызовом).

В случае, ели нет никаких ошибок, вызов `vkEnumeratePhysicalDevices()` вернет значение `VK_SUCCESS` и запишет число «узнанных» устройств в `pPhysicalDeviceCount` и их дескрипторы в `pPhyiscalDevices`. В листинге 1.1 показаны создание структур `VkApplicationInfo` и `VkInstanceCreateInfo`, создание экземпляра Vulkan, получение числа поддерживаемых устройств и, наконец, получение дескрипторов самих этих устройств. Это является упрощенной версией `vkapp:init` из нашей библиотеки.

Листинг 1.1 ❖ Создание экземпляра Vulkan

```
VkResult vkapp::init()
{
  VkResult result = VK_SUCCESS;
  VkApplicationInfo appInfo = { };
  VkInstanceCreateInfo instanceCreateInfo = { };

  // Общая структура информации о приложении
  appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
  appInfo.pApplicationName = "Application";
  appInfo.applicationVersion = 1;
  appInfo.apiVersion = VK_MAKE_VERSION(1, 0, 0);

  // Создание экземпляра
  instanceCreateInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
  instanceCreateInfo.pApplicationInfo = &appInfo;

  result = vkCreateInstance(&instanceCreateInfo, nullptr, &m_instance);

  if (result == VK_SUCCESS)
  {
    // Сперва определим, сколько в системе устройств
    uint32_t physicalDeviceCount = 0;
    vkEnumeratePhysicalDevices(m_instance, &physicalDeviceCount, nullptr);

    if (result == VK_SUCCESS)
    {
      // Выделим место в массиве и получим дескрипторы
      // физических устройств
      m_physicalDevices.resize(physicalDeviceCount);
      vkEnumeratePhysicalDevices(m_instance,
                                 &physicalDeviceCount,
                                 &m_physicalDevices[0]);
    }
  }

  return result;
}
```

Дескриптор физического устройства используется для получения от устройства информации о его возможностях и создания логического устройства. Для получения информации от устройства мы воспользуемся функцией `vkGetPhysicalDeviceProperties()`, которая заполняет структуру, описывающую все свойства физического устройства. Ее описание приводится ниже.

```
void vkGetPhysicalDeviceProperties (
  VkPhysicalDevice                physicalDevice,
  VkPhysicalDeviceProperties*     pProperties);
```

При вызове `vkGetPhysicalDeviceProperties()` передайте один из дескрипторов, полученных от `vkEnumeratePhysicalDevices()` в качестве параметра `physicalDevice`, а в качестве `pProperties` передайте указатель на структуру `pPhysicalDeviceProperties`. Это большая структура, содержащая большое число полей, описывающих свойства физического устройства. Ее определение приводится ниже.

```
typedef struct VkPhysicalDeviceProperties {
  uint32_t                            apiVersion;
  uint32_t                            driverVersion;
  uint32_t                            vendorID;
  uint32_t                            deviceID;
  VkPhysicalDeviceType                deviceType;
  char                                deviceName [VK_MAX_PHYSICAL_DEVICE_NAME_SIZE];
  uint8_t                             pipelineCacheUUID[VK_UUID_SIZE];
  VkPhysicalDeviceLimits              limits;
  VkPhysicalDeviceSparseProperties    sparseProperties;
} VkPhysicalDeviceProperties;
```

Поле `apiVersion` содержит наибольшую версию Vulkan, поддерживаемую устройством, поле `driverVersion` содержит версию драйвера, используемую для управления устройством. Версия драйвера зависит от производителя, и поэтому нет никакого смысла сравнивать версии у разных производителей. Поля `vendorID` и `deviceID` задают изготовителя устройства и само устройство, обычно это PCI-идентификаторы производителя и устройства[1].

Поле `deviceName` содержит строку с именем устройства. Поле `pipelineCacheUUID` используется для кэширования конвейеров, что будет рассмотрено в главе 6 «Шейдеры и конвейеры».

Кроме только что рассмотренных полей, структура `VkPhysicalDeviceProperties` включает в себя `VkPhysicalDeviceLimits` и `VkPhysicalDeviceSparseProperties`, содержащие минимальные и максимальные величины для физического устройства и свойства, связанные с разреженными текстурами. В этих структурах содержится большое количество информации, и мы отдельно рассмотрим эти поля по мере изучения соответствующих возможностей (а не прямо здесь).

В дополнение к базовым возможностям, некоторые из которых могут иметь заданные границы, у Vulkan есть ряд необязательных возможностей, которые могут поддерживаться физическим устройством. Если устройство заявляет о поддержке какой-либо возможности, то она должна быть просто включена (примерно как

[1] Нет какого-либо официального репозитория PCI идентификаторов производителя и устройства. PCI SIG (http://pcisig.com) назначает идентификаторы производителя своим членам, и эти члены назначают идентификаторы своим устройствам. Достаточно полный список, удобный для чтения как человеком, так и программой, доступен на http://pcidatabase.com.

расширение), но после того как она включена, она является полноценной частью API, такой как и все базовые возможности. Для определения того, какие возможности поддерживает заданное физическое устройство, нужно вызвать vkGetPhysicalDeviceFeatures(), описание которой приводится ниже.

```
void vkGetPhysicalDeviceFeatures (
  VkPhysicalDevice              physicalDevice,
  VkPhysicalDeviceFeatures*     pFeatures);
```

Структура VkPhysicalDeviceFeatures также очень велика и содержит по булеву полю для каждой необязательной возможности, поддерживаемой Vulkan. Там слишком много полей, чтобы их всех перечислить и описать здесь, но простой пример приложения, приводимого в конце этой главы, запрашивает информацию о поддерживаемых возможностях и печатает эту информацию.

Память физического устройства

Во многих случаях устройство Vulkan является или отдельным устройством для главного процессора, или работает другим образом, так что оно будет обращаться к памяти особенным образом. В Vulkan *память устройства* обозначает память, которая доступна устройству и может быть использована для хранения текстур и других данных. Память разделяется на различные *типы*, каждый из которых обладает набором свойств, таких как флаги кэширования и когерентность между устройством и CPU. Каждому типу памяти соответствует одна из *куч* (heap) устройства, которых может быть несколько.

Для получения информации о кучах и поддерживаемых типах памяти используется вызов следующей функции.

```
void vkGetPhysicalDeviceMemoryProperties (
  VkPhysicalDevice                    physicalDevice,
  VkPhysicalDeviceMemoryProperties*   pMemoryProperties);
```

Организация памяти устройства записывается в структуру VlPhysicalDeviceMemoryProperties, адрес которой передается в pMemoryProperties. Структура VkPhysicalDevicememoryProperties содержит свойства как куч устройства, так и поддерживаемых типов памяти. Ниже приводится определение этой структуры.

```
typedef struct VkPhysicalDeviceMemoryProperties {
  uint32_t          memoryTypeCount;
  VkMemoryType      memoryTypes[VK_MAX_MEMORY_TYPES];
  uint32_t          memoryHeapCount;
  VkMemoryHeap      memoryHeaps[VK_MAX_MEMORY_HEAPS];
} VkPhysicalDeviceMemoryProperties;
```

Количество типов памяти возвращается в поле memoryTypeCount. Максимальное число типов памяти, которое может быть возвращено, равно VK_MAX_MEMORY_TYPES, что равно 32. Массив memoryTypes содержит memoryTypeCount структур VkMemoryType, описывающих каждый из типов памяти. Ниже приводится определение VkMemoryType.

```
typedef struct VkMemoryType {
  VkMemoryPropertyFlags     propertyFlags;
  uint32_t                  heapIndex;
} VkMemoryType;
```

Это простая структура, содержащая только набор флагов и индекс соответствующей этому типу кучи. Поле flags описывает тип памяти и является комбинацией флагов VkMemoryPropertyFlagBits. Значения флагов приводятся ниже:

- VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT означает, что память является локальной для устройства (т. е. физически подсоединенной к нему). Если этот бит не установлен, то память считается локальной по отношению к CPU;
- VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT означает, что выделенная память этого типа может быть отображена и читаться или записываться CPU. Если этот бит не установлен, то память этого типа не может быть напрямую доступна CPU и предназначена скорее исключительно для устройства;
- VK_MEMORY_PROPERTY_HOTS_COHERENT_BIT означает, что когда происходит одновременное обращение к памяти данного типа со стороны CPU и GPU, то эти обращения будут согласованы (когерентны) между ними. Если этот бит не установлен, то CPU или GPU могут не увидеть результатов записи до тех пор, пока кэши явно не сброшены;
- VK_MEMORY_PROPERTY_HOST_CACHED_BIT означает, что данные для этого типа памяти кэшируются на стороне CPU. Чтение из этого типа памяти обычно будет быстрее, чем если бы этот бит не был установлен. Однако доступ со стороны GPU может иметь более высокую латентность , особенно если память когерентна;
- VK_MEMORY_PROPERTY_LAZILIY_ALLOCATED_BIT означает, что выделенная память данного типа не обязательно сразу же занимает место в соответствующей куче и драйвер может отложить выделение физической памяти до того момента, когда эта память будет использована для какого-либо ресурса.

Каждый тип памяти в поле heapIndex структуры VkMemoryType сообщает, из какой кучи он выделяет память. Это индекс в массив memoryHeaps структуры VkPhysicalDeviceMemoryProperties из вызова vkGetPhysicalDeviceMemoryProperties(). Каждый элемент массива memoryHeaps описывает одну из куч для выделения памяти устройства. Описание этой структуры приводится ниже.

```
typedef struct VkMemoryHeap {
  VkDeviceSize        size;
  VkMemoryHeapFlags   flags;
} VkMemoryHeap;
```

Опять это является довольно простой структурой. Она содержит только размер в байтах кучи и некоторые флаги, описывающие эту кучу. В Vulkan 1.0 единственным таким флагом является VK_MEMORY_HEAP_DEVICE_LOCAL_BIT. Если этот бит установлен, то эта куча является локальной для устройства. Это полностью соответствует аналогично названному флагу для типов памяти.

Очереди устройства

Устройства Vulkan выполняют задания, которые помещены в *очереди*. У каждого устройства имеется одна или несколько очередей, и каждая из этих очередей относится к одному из семейств очередей устройства. *Семейство очередей* (queue family) – это группа очередей, которые обладают одинаковыми свойствами, но выполняются параллельно. Число семейств очередей, возможности каждого семейства и число очередей внутри каждого семейства определяются физическим устройством. Для получения списка всех семейств очередей устройства вызовите функцию `vkGetPhysicalDeviceQueueFamilyProperties()`, прототип этой функции приводится ниже.

```
void vkGetPhysicalDeviceQueueFamilyProperties (
  VkPhysicalDevice              physicalDevice,
  uint32_t*                     pQueueFamilyPropertyCount,
  VkQueueFamilyProperties*      pQueueFamilyProperties);
```

`vkGetPhysicalDeviceQueueFamilyProperties()` работает аналогично `vkEnumeratePhysicalDevices()` в том отношении, что ожидается, что вы вызовите ее два раза. Первый раз вы передаете `nullptr` в качестве `pQueueFamilyProperties` и в `pQueueFamilyPropertyCount` передаете указатель на переменную, в которой вы получите число семейств очередей, поддерживаемых устройством. Вы можете использовать это число для того, чтобы задать правильный размер для массива `VkQueueFamilyProperties`. Тогда, при втором вызове, передайте этот массив в `pQueueFamilyProperties`, и Vulkan запишет в него свойства очередей. Ниже приводится определение `VkQueueFamilyProperties`.

```
typedef struct VkQueueFamilyProperties {
  VkQueueFlags    queueFlags;
  uint32_t        queueCount;
  uint32_t        timestampValidBits;
  VkExtent3D      minImageTransferGranularity;
} VkQueueFamilyProperties;
```

Первое поле этой структуры, `queueFlags`, описывает общие возможности очереди. Это поле состоит из комбинации битов `VkQueueFlagBits`, значения которых приводятся ниже:

- `VK_QUEUE_GRAPHICS_BIT` – если этот бит выставлен, то очереди в этом семействе поддерживают графические операции, такие как рендеринг точек, отрезков и треугольников;
- `VK_QUEUE_COMPUTE_BIT` – если этот бит выставлен, то очереди в этом семействе поддерживают вычислительные операции, такие как запуски вычислительных шейдеров;
- `VK_QUEUE_TRANSFER_BIT` – если этот бит выставлен, то очереди в этом семействе поддерживают операции копирования, такие как копирование содержимого буферов и изображений;

- VK_QUEUE_SPARSE_BINDING_BIT – если этот бит выставлен, то очереди в этом семействе поддерживают операции привязывания (binding) памяти для работы с разреженными ресурсами.

Поле queueCount содержит число очередей в семействе. Оно может быть равно 1 или же может быть больше, если устройство поддерживает несколько очередей с одной и той же базовой функциональностью.

Поле timestampValidBits говорит о том, сколько битов валидны в метках времени (timestamp), взятых из очереди. Если это значение равно нулю, то данная очередь не поддерживает меток времени. Если оно не равно нулю, то гарантируется поддержка как минимум 36 бит. Кроме того, если поле timestampComputeAndGraphics структуры VkPhysicalDeviceLimits равно VK_TRUE, то все очереди, поддерживающие VK_QUUE_GRAPHICS_BIT или VK_QUEUE_COMPUTE_BIT, гарантированно поддерживают метки времени с как минимум 36 битами точности. В этом случае нет необходимости отдельно проверять каждую очередь.

Наконец, поле minImageTimestampGranularity сообщает, в каких единицах очередь поддерживает копирование изображений (если вообще поддерживает).

Обратите внимание, что может быть так, что устройство поддерживает более чем одно семейство очередей с вроде бы одинаковыми свойствами. Очереди внутри семейства практически одинаковы. Очереди в разных семействах могут иметь разные внутренние свойства, которые не могут быть легко выражены в Vulkan API. По этой причине реализация может решить сообщать похожие очереди как члены разных семейств. Это накладывает дополнительные ограничения на то, как ресурсы разделяются между этими очередями, что может позволить реализации учитывать эти различия.

В листинге 1.2 показано, как можно запросить свойства физической памяти устройства и свойства семейств очередей. Вам нужно будет получить свойства семейства очередей перед созданием логического устройства, что будет рассматриваться в следующем разделе.

Листинг 1.2 ❖ Получение свойств физического устройства

```
std::vector<VkQueueFamilyProperties> queueFamilyProperties;
VkPhysicalDeviceMemoryProperties physicalDeviceMemoryProperties;

// Получаем свойства памяти физического устройства
vkGetPhysicalDeviceMemoryProperties(m_physicalDevices[deviceIndex],
                                    &physicalDeviceMemoryProperties);

// Сначала определяем число семейств очередей,
// поддерживаемых физическим устройством
vkGetPhysicalDeviceQueueFamilyProperties(m_physicalDevices[0],
                                         &queueFamilyPropertyCount,
                                         nullptr);

// Выделяем достаточно места для структур, описывающих свойства очередей
queueFamilyProperties.resize(queueFamilyPropertyCount);
```

```
// Теперь запрашиваем свойства семейств очередей
vkGetPhysicalDeviceQueueFamilyProperties(m_physicalDevices[0],
                                         &queueFamilyPropertyCount,
                                         queueFamilyProperties.data());
```

Создание логического устройства

После получения списка всех физических устройств в системе ваше приложение должно выбрать одно устройство и создать соответствующее ему *логическое устройство*. Логическое устройство представляет собой устройство в проинициализированном состоянии. Во время создания логического устройства вы получите возможность задать необязательные возможности, включить нужные расширения и т. п. Логическое устройство создается при помощи вызова `vkCreateDevice()`, описание которого приводится ниже.

```
VkResult vkCreateDevice (
  VkPhysicalDevice                physicalDevice,
  const VkDeviceCreateInfo*       pCreateInfo,
  const VkAllocationCallbacks*    pAllocator,
  VkDevice*                       pDevice);
```

Физическое устройство, на основе которого создается логическое устройство, передается в параметре `physicalDevice`. Информация о новом логическом устройстве передается в структуре `VkDeviceCreateInfo`, на которую указывает параметр `pCreateInfo`. Описание `VkCreateDeviceInfo` приводится ниже.

```
typedef struct VkDeviceCreateInfo {
  VkStructureType                       sType;
  const void*                           pNext;
  VkDeviceCreateFlags                   flags;
  uint32_t                              queueCreateInfoCount;
  const VkDeviceQueueCreateInfo*        pQueueCreateInfos;
  uint32_t                              enabledLayerCount;
  const char* const*                    ppEnabledLayerNames;
  uint32_t                              enabledExtensionCount;
  const char* const*                    ppEnabledExtensionNames;
  const VkPhysicalDeviceFeatures*       pEnabledFeatures;
} VkDeviceCreateInfo;
```

Поле `sType` структуры `VkDeviceCreateInfo` должно быть равно `VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO`. Как обычно, если вы не используете расширения, поле `pNext` должно быть равно `nullptr` . В текущей версии Vulkan не определено никаких битов для поля `flags`, так что оно также должно быть равно нулю.

Дальше идет информация по созданию очереди. `pQueueCreateInfos` – это указатель на массив из одной или более структур `VkDeviceQueueCreateInfo`, каждая из которых позволяет задать одну или несколько очередей. Количество структур в массиве задается полем `queueCreateInfoCount`. Определение структуры `VkDeviceQueueCreateInfo` приводится ниже.

```
typedef struct VkDeviceQueueCreateInfo {
  VkStructureType              sType;
  const void*                  pNext;
  VkDeviceQueueCreateFlags     flags;
  uint32_t                     queueFamilyIndex;
  uint32_t                     queueCount;
  const float*                 pQueuePriorities;
} VkDeviceQueueCreateInfo;
```

Поле `sType` структуры `VkDeviceQueueCreateInfo` равно `VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO`. Сейчас не определено никаких флагов для использования в поле `flags`, так что это поле должно быть равно нулю. Поле `queueFamilyIndex` задает семейство очередей, которое вы хотите создать. Это индекс в массив семейств очередей, возвращенный вызовом `vkGetPhysicalDeviceQueueFamilyProperties()`. Для создания очередей этого семейства задайте в `queueCount` число очередей, которое вы хотите использовать. Конечно, для этого устройство должно поддерживать как минимум столько же очередей в выбранном вами семействе.

Поле `pQueuePriorities` является необязательным указателем на массив из вещественных чисел, представляющих собой относительные приоритеты заданий, помещаемых в каждую из этих очередей. Эти числа являются нормализованными значениями в диапазоне от 0,0 до 1,0. Очереди с более высоким приоритетом могут получить больше ресурсов работы или использовать более агрессивное планирование, чем очереди с низким приоритетом. Если задать в качестве значения для `pQueuePriorities` значение `nullptr`, то в результате все очереди получат один и тот же приоритет по умолчанию.

Запрашиваемые очереди сортируются в порядке приоритета, а затем получают зависимые от конкретного устройства относительные приоритеты. Число различных приоритетов, которые может получить очередь, зависит от конкретного устройства. Вы можете получить его в поле `discreteQueuePriorities` структуры `VkPhysicalDeviceLimits`, возвращаемой `vkGetPhysicalDevicePriorities()`. Например, если устройство поддерживает только низкий и высокий приоритеты, то это поле будет равно 2. Все устройства поддерживают как минимум два различных приоритета. Однако если устройство поддерживает произвольные приоритеты, то это поле может быть заметно больше. Независимо от значения `discreteQueuePriorities` относительные приоритеты очередей выражаются через вещественные числа.

Возвращаясь к структуре `VkDeviceCreateInfo`, поля `enabledLayerCount`, `ppEnabledlayerNames`, `enabledExtensionCount` и `ppEnabledExtensionNames` служат для включения слоев и расширений. Мы рассмотрим эти темы далее в этой главе. Сейчас мы зададим `enabledLayerCount` и `enabledExtensionCount` равными нулю, а `ppEnabledLayerNames` и `ppEnabledExtensionNames` – равными `nullptr`.

Последнее поле `VkDeviceCreateInfo`, `pEnabledFeatures`, является указателем на структуру `VkPhysicalDeviceFeatures`, которая задает, какие необязательные возможности ваше приложение хочет использовать. Если вы не хотите использовать

никаких таких возможностей, то просто установите это поле равным nullptr. Однако в этом случае функциональность Vulkan будет сильно ограничена и много интересного функционала будет просто выключено.

Для определения того, какие необязательные возможности поддерживает устройство, вызовите vkGetPhysicalDevicefeatures(), как рассмотрено раньше. vkGetPhysicalDevicefeatures() записывает множество поддерживаемых устройством возможностей в передаваемую структуру VkPhysicalDeviceFeatures. Если вы просто передадите полученную структуру VkPhysicalDeviceFeatures в vkCreateDevice(), вы включите все поддерживаемые возможности и не запросите ничего того, чего устройство не поддерживает.

Однако такое включение всех поддерживаемых возможностей может повлиять на быстродействие. Для поддержки некоторых возможностей реализации Vulkan может понадобиться выделить дополнительную память, отслеживать дополнительное состояние, немного иначе сконфигурировать состояние или выполнить какие-то другие операции, обладающие ценой для вашего приложения. Так что это не лучшая идея требовать возможностей, которых вы не будете использовать. В оптимизированном приложении вам следует запросить возможности у устройства; затем из поддерживаемых возможностей включить именно те, которые нужны вашему приложению.

В листинге 1.3 приведен простой пример получения от устройства поддерживаемых возможностей и установки списка возможностей, которые действительно нужны приложению. В этом примере требуется поддержка тесселляционных и геометрических шейдеров, а возможности неявного вывода сразу нескольких групп геометрии (multidraw indirect) запрашиваются только в том случае, когда она поддерживается устройством. Далее в коде создается устройство, использующее всего одну (первую) очередь.

Листинг 1.3 ❖ Создание логического устройства

```
VkResult result;
VkPhysicalDeviceFeatures supportedFeatures;
VkPhysicalDeviceFeatures requiredFeatures = {};

vkGetPhysicalDeviceFeatures(m_physicalDevices[0],
                            &supportedFeatures);

requiredFeatures.multiDrawIndirect  = supportedFeatures.multiDrawIndirect;
requiredFeatures.tessellationShader = VK_TRUE;
requiredFeatures.geometryShader     = VK_TRUE;

const VkDeviceQueueCreateInfo deviceQueueCreateInfo =
{
  VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO, // sType
  nullptr,                                    // pNext
  0,                                          // flags
  0,                                          // queueFamilyIndex
  1,                                          // queueCount
```

```
  nullptr                                    // pQueuePriorities
};
const VkDeviceCreateInfo deviceCreateInfo =
{
  VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO, // sType
  nullptr,                              // pNext
  0,                                    // flags
  1,                                    // queueCreateInfoCount
  &deviceQueueCreateInfo,               // pQueueCreateInfos
  0,                                    // enabledLayerCount
  nullptr,                              // ppEnabledLayerNames
  0,                                    // enabledExtensionCount
  nullptr,                              // ppEnabledExtensionNames
  &requiredFeatures                     // pEnabledFeatures
};

result = vkCreateDevice(m_physicalDevices[0],
                        &deviceCreateInfo,
                        nullptr,
                        &m_logicalDevice);
```

После выполнения кода из листинга 1.3 и успешного создания логического устройства множество разрешенных возможностей хранится в переменной `requiredFeatures`. Это может понадобиться для кода, который использует какую-либо возможность, если она включена и использует запасной вариант в противном случае.

Соглашения о типах объектов и функций

Практически все в Vulkan представлено при помощи объектов, и мы ссылаемся на каждый объект при помощи дескриптора (handle). Все дескрипторы делятся на две большие категории: диспетчеризуемые объекты и недиспетчеризуемые объекты. В основном это не важно для приложения и касается только того, каким образом организован Vulkan и как системные компоненты, такие как загрузчик и слои, взаимодействуют с этими объектами.

Диспетчеризуемые объекты – это объекты, которые внутри себя хранят таблицу диспетчеризации. Это таблица функций, используемая различными компонентами для определения того, какие части кода выполнить, когда приложение вызывает Vulkan. Такие объекты обычно являются довольно тяжеловесными и на данный момент состоят из экземпляра (`VkInstance`), физического устройства (`VkPhysicalDevice`), логического устройства (`VkDevice`), буфера команд (`VkCommandBuffer`) и очереди (`VkQeue`). Все остальные объекты считаются недиспетчеризируемыми.

Первым аргументом любой функции Vulkan всегда является диспетчеризируемый объект. Единственными исключениями из этого правила являются функции, служащие для создания и инициализации экземпляра.

Управление памятью

Vulkan предоставляет два типа памяти – память CPU (host memory) и память GPU (device memory). Объекты, создаваемые Vulkan API, обычно требуют некоторого количества памяти CPU. Это то место, где реализация Vulkan будет хранить состояние объекта и все данные, необходимые Vulkan API. Ресурсные объекты, такие как буферы и изображения, требуют некоторого объема памяти GPU. Это та память, где хранятся данные этого ресурса.

Ваше приложение может само управлять памятью CPU для реализации Vulkan, и требуется, чтобы ваше приложение само управляло памятью GPU.Для этого вам понадобится создать подсистему управления памятью GPU. Каждый создаваемый вами ресурс может вернуть информацию об объеме и типе памяти, которая требуется для его хранения. Ваше приложение само должно выделить правильное количество памяти и присоединить его к ресурсному объекту перед его использованием.

В высокоуровневых API, таких как OpenGL, вся эта «магия» выполняется драйверами от имени вашего приложения. Однако некоторым приложениям нужно очень большое количество маленьких ресурсов, а другим нужно небольшое количество очень больших ресурсов. Некоторые приложения создают и уничтожают ресурсы во время своей работы, в то время как другие приложения создают все свои ресурсы в начале работы программы и не освобождают их до конца работы программы.

Используемые стратегии для выделения памяти в этих случаях могут сильно отличаться. Нет какой-либо универсальной стратегии, которая подойдет всем. Драйвер OpenGL не знает, как ваше приложение будет себя вести, и должен подстраивать свои стратегии выделения в попытке подстроиться под вашу работу. С другой стороны *вы*, разработчик приложения, точно знаете, как поведет себя ваше приложение. Вы можете разделить все ресурсы на долгоживущие и короткоживущие. Вы можете группировать ресурсы, которые будут использованы совместно, в небольшое число блоков памяти. Именно вам удобнее всего решить, какую стратегию выделения памяти использует ваша программа.

Важно заметить, что каждое «живое» выделение памяти обладает некоторой стоимостью для системы. Поэтому важно минимизировать число выделенных объектов. Рекомендуется выделять память GPU большими блоками. Много маленьких ресурсов может быть помещено внутри небольшого числа выделенных блоков памяти. Пример аллокатора памяти GPU будет рассмотрен в главе 2 «Память и ресурсы», которая рассматривает выделение памяти гораздо подробнее.

Многонитевость в Vulkan

Поддержка многонитевых приложений является важной частью дизайна Vulkan. Обычно Vulkan считает, что приложение гарантирует, что никакие две нити не будут одновременно изменять один и тот же объект. Это называется *внешней син-*

хронизацией. Подавляющее большинство команд Vulkan, критичных для быстродействия (таких как построение командных буферов), вообще не предоставляет никакой синхронизации.

Для того чтобы точно определить, каковы требования различных команд Vulkan на поведение отдельных нитей, каждый параметр, который должен быть защищен от одновременного доступа CPU, помечается как внешне синхронизируемый. В некоторых случаях дескрипторы объектов и другие данные вложены в структуры, в массивы или передаются каким-то другим неявным способом. Все эти параметры также должны быть внешне синхронизированы.

Целью этого является то, что реализации Vulkan никогда не нужен внутренний мьютекс или какой-то другой синхронизационный примитив для защиты данных. Это означает, что многонитевые программы крайне редко сталкиваются с блокировками или ожиданиями нитей.

Кроме требования к CPU синхронизировать доступ к общим объектам, когда они используются из разных нитей, Vulkan включает в себя некоторое количество высокоуровневых возможностей, которые позволяют нитям выполнять работу, не блокируя других нитей. Они включают в себя следующие возможности:

- выделение памяти на стороне CPU может выполняться через специальную структуру для выделения памяти CPU, передаваемую функциям создания объектов. Используя свой аллокатор на каждую нить, структуры данных в этом аллокаторе не нужно защищать. Аллокаторы памяти CPU рассматриваются в главе 2 «Память и ресурсы»;
- командные буферы выделяются из пулов (pool), и доступ к этим пулам синхронизируется внешне. Если приложение использует отдельный пул командных буферов на каждую нить, то выделение командных буферов из этих пулов будет происходить без блокировок. Командные буферы и пулы рассматриваются в главе 3 «Очереди и команды»;
- дескрипторы ресурсов (descriptor) выделяются из пулов дескрипторов ресурсов. Дескрипторы ресурсов являются представлением ресурсов, используемых шейдерами, выполняемыми на устройстве. Они подробно рассматриваются в главе 6 «Шейдеры и конвейеры». Если используется отдельный пул для каждой нити, то множества дескрипторов могут выделяться из этих пулов без необходимости блокирования нитей;
- командные буферы второго уровня позволяют содержимому большого прохода рендеринга (renderass) (которое должно содержаться в одном командном буфере) создаваться параллельно и затем группироваться по мере вызова из первичного командного буфера. Вторичные командные буферы подробно рассмотрены в главе 13 «Многопроходный рендеринг».

Когда вы пишете очень простое однонитевое приложение, то требование создавать пулы, из которых будут выделяться объекты, может показаться слишком сложным и ненужным. Однако по мере масштабирования приложения по числу нитей эти объекты становятся крайне важными для достижения высокого быстродействия.

Математические понятия

Приложения, относящиеся к компьютерной графике и большинству разнородных вычислений, обычно используют очень много математики. Большинство устройств, поддерживающих Vulkan, основано на крайне мощных вычислительных процессорах. На момент написания даже скромные мобильные процессоры способны предоставлять гигафлопы вычислительной мощности, в то время как высокоуровневые процессоры для рабочих станций могут давать целые терафлопы вычислительной мощности. Как следствие действительно интересные приложения используют математически сложные шейдеры. Кроме того, некоторые части фиксированного конвейера основаны на математических понятиях, которые жестко заложены в устройство и его спецификации.

Векторы и матрицы

Одним из краеугольных камней любого графического приложения являются векторы. Векторы используются в графической литературе для обозначения таких понятий, как положение, направление, цвет, и ряда других. Одной из форм векторов являются однородные векторы, которые являются векторами в пространстве с размерностью на единицу больше, чем размерность представляемой величины. Эти векторы используются для хранения проективных координат. Умножение однородного вектора на любой скаляр дает вектор с теми же проективными координатами. Для проективного такого вектора разделите его на его последнюю компоненту, получая вектор вида x, y, z, 1, 0 (для четырехкомпонентного вектора).

Для преобразования вектора из одного координатного пространства в другое умножьте вектор на матрицу. Так же как точка в трехмерном пространстве представляется четырехкомпонентным вектором, матрица преобразования трехмерных однородных векторов имеет размер 4×4.

Точка в трехмерном пространстве обычно представляется как однородный вектор из четырех компонент, обычно называемых x, y, z и w. Для точки компонента w вначале равна 1,0 и изменяется при преобразовании вектора при помощи проективной матрицы. После деления на компоненту w точка проектируется соответствующим преобразованием. Если преобразование не является проективным, то w остается равной 1,0 и деление на 1,0 никак не влияет на вектор. Если вектор был подвергнут проективному преобразованию, то w не будет равно 1,0, но деление на него спроектирует точку, и мы придем к w, равному 1,0.

Направление в трехмерном пространстве обычно представляется однородным вектором, у которого компонента w равна 0,0. Умножение вектора направления на правильно построенную проективную матрицу 4×4 оставит компоненту w равной 0,0, и она не будет влиять на остальные компоненты. Просто отбрасывая дополнительную компоненту, можно подвергнуть трехмерный вектор направления тем же самым преобразованиям, что и однородный трехмерный вектор.

Системы координат

Vulkan работает с графическими примитивами, такими как отрезки и треугольники, представляя их концы или вершины как точки в трехмерном пространстве. Они тогда называются *вершинами*. На вход Vulkan поступают координаты вершин в трехмерном пространстве (заданные как однородные векторы с компонентой w, равной 1,0) относительно начала объекта, частью которого они являются. Это координатное пространство называется *пространством объектов*, или *модельным пространством*.

Обычно первый шейдер в конвейере преобразует вершину в *видовое пространство* (view space), т. е. положение относительно наблюдателя. Это преобразование выполняется при помощи умножения вектора координат на матрицу преобразования. Она часто называется *объектно-видовой* или *модельно-видовой* матрицей.

Иногда требуются абсолютные координаты вершины, например для нахождения относительных координат вершины по отношению к некоторому объекту. Это глобальное пространство называется *мировым пространством* (world space), и в нем вершины заданы своим положением по отношению к глобальному началу координат.

Из видового пространства координаты вершины преобразуются в пространство отсечения (clip space). Это последнее пространство, используемое частью обработки геометрии Vulkan. Пространство отсечения так названо потому, что это то координатное пространство, в котором большинство реализаций выполняет *отсечение*, которое удаляет части примитивов, лежащие вне видимой области.

Из пространства отсечения координаты вершины нормируются путем деления на компоненту w. В результате получается пространство, называемое пространством *нормализованных координат устройства*, или NDC (Normalized Device Coordinates), и сам этот процесс часто называют *перспективным делением*. В этом пространстве видимая часть координатной системы лежит от –1,0 до 1,0 по координатам x и y и от 0,0 до 1,0 по z. Все, лежащее вне этой области, отсекается до перспективного деления.

Наконец, нормализованные координаты устройства преобразуются в координаты области в окне (viewport) или в изображении, в котором производится рендеринг.

Расширяем Vulkan

Хотя базовые спецификации API довольно значительны, они не охватывают всего. Некоторая функциональность является необязательной, а некоторая доступна в виде слоев (которые изменяют или расширяют текущее поведение). Оба этих механизма расширения описываются далее.

Слои

Слои – это способ, при помощи которого поведение Vulkan может быть изменено. Слои перехватывают или весь Vulkan, или отдельные его части и добавляют такую функциональность, как логгирование, отслеживание, предоставление диаг-

ностики, профилировка и т. п. Слой может быть добавлен на уровне экземпляра, в этом случае он воздействует на весь Vulkan и, возможно, на каждое созданное им устройство. Или же слой может быть добавлен на уровне устройства, в этом случае он воздействует только на соответствующее устройство.

Для того чтобы найти, какие слои доступны для экземпляра в системе, вызовите функцию `vkEnumerateInstanceLayerProperties()`, прототип которой приводится ниже.

```
VkResult vkEnumerateInstanceLayerProperties (
  uint32_t*               pPropertyCount,
  VkLayerProperties*      pProperties);
```

Если `pProperties` равно `nullptr`, то `pPropertyCount` должно указывать на переменную, в которую будет записано число доступных Vulkan слоев. Если `pProperties` не равно `nullptr`, то оно должно указывать на массив структур `VkLayerProperties`, в которые будет записана информация о зарегистрированных в системе слоях. В этом случае начальное значение переменной, на которую указывает `pPropertyCount`, является длиной массива, на который указывает `pProperties`, и в эту переменную будет записано число слоев, для которых информация была записана в этот массив.

Каждый элемент массива `pProperties` является структурой `VkLayerProperties`, определение которой приводится ниже.

```
typedef struct VkLayerProperties {
  char           layerName[VK_MAX_EXTENSION_NAME_SIZE];
  uint32_t       specVersion;
  uint32_t       implementationVersion;
  char           description[VK_MAX_DESCRIPTION_SIZE];
} VkLayerProperties;
```

У каждого слоя должно быть свое имя, которое хранится в поле `layerName` структуры `VkLayerProperties`. Так как со временем спецификации слоя могут улучшаться, уточняться или расширяться, то в поле `specVersion` содержится версия спецификации.

Не только спецификации могут улучшаться со временем, но и реализации этих спецификаций. Версия реализации хранится в поле `implementationVersion` структуры `VkLayerProperties`. Это позволяет реализациям улучшать быстродействие, исправлять ошибки, реализовывать более широкий набор необязательных возможностей и т. п. Приложение может знать определенную версию реализации слоя и выбрать слой, только если номер версии больше заданного, например где была исправлена критическая ошибка.

Наконец, в поле `description` содержится в читаемом виде строка с описанием данного слоя. Единственной целью этого поля является его использование в логгировании или для показа в пользовательском интерфейсе, служащее только для информации.

В листинге 1.4 показано, как получить поддерживаемые системой слои уровня экземпляра.

Листинг 1.4 ❖ Получение слоев экземпляра

```
uint32_t numInstanceLayers = 0;
std::vector<VkLayerProperties> instanceLayerProperties;

// Запрашиваем число слоев экземпляров
vkEnumerateInstanceLayerProperties(&numInstanceExtensions,
                                   nullptr);

// Если слои есть, то запрашиваем их свойства
if (numInstanceLayers != 0)
{
  instanceLayerProperties.resize(numInstanceLayers);
  vkEnumerateInstanceLayerProperties(nullptr,
                                     &numInstanceLayers,
                                     instanceLayerProperties.data());
}
```

Как уже было отмечено, устанавливать слои можно не только на уровне экземпляра. Слои также могут применяться и на уровне устройства. Для определения того, какие слои доступны для устройств, вызовите функцию `vkEnumerateDeviceLayerProperties()`, прототип которой приводится ниже.

```
VkResult vkEnumerateDeviceLayerProperties (
  VkPhysicalDevice          physicalDevice,
  uint32_t*                 pPropertyCount,
  VkLayerProperties*        pProperties);
```

Слои, доступные различным физическим устройствам, могут отличаться, поэтому каждое физическое устройство сообщает свой набор слоев. В параметре `physicalDevice` передается физическое устройство, для которого запрашиваются поддерживаемые слои. Параметры `pPropertyCount` и `pProperties` полностью аналогичны одноименным параметрам функции `vkEnumerateInstanceLayerProperties()`. Слои для устройств также описываются при помощи структуры `VkLayerProperties`.

Для того чтобы включить слой на уровне экземпляра, просто добавьте его имя в `ppEnabledLayerNames` структуры `VkInstanceCreateInfo`, используемой для создания экземпляра. Аналогично, для того чтобы включить слой при создании логического устройства, соответствующего заданному физическому устройству, добавьте имя слоя в поле `ppEnabledLayerNames` структуры `VkDeviceCreateInfo`, используемой для создания устройства.

Несколько слоев включено в официальный SDK, большинство из них относится к отладке, проверке параметров и логгированию. Эти слои включают в себя:

- `VK_LAYER_LUNARG_api_dump` – выводит на консоль вызовы Vulkan и передаваемые параметры;
- `VK_LAYER_LUNARG_core_validation` – осуществляет базовую проверку параметров и состояния, используемых для множеств дескрипторов ресурсов, состояние конвейера и динамического состояния; проверяет интерфейсы

между модулями SPIR-V и графическим конвейером; отслеживает и проверяет использование памяти GPU в объектах;

- `VK_LAYER_LUNARG_device_limits` – проверяет, что значения, передаваемые командам Vulkan как параметры или члены структур, лежат в поддерживаемых устройством пределах;
- `VK_LAYER_LUNARG_image` – проверяет, что изображение соответствует поддерживаемым форматам;
- `VK_LAYER_LUNARG_object_tracker` – отслеживает объекты Vulkan, позволяя обнаруживать утечки, использование после уничтожения и прочие ошибки, связанные с некорректным использованием объектов;
- `VK_LAYER_LUNARG_parameter_validation` – проверяет, что все параметры функций Vulkan валидны;
- `VK_LAYER_LUNARG_swapchain` – проверяет функциональность расширений WSI (Window System Integration), описываемых в главе 5 «Показ»;
- `VK_LAYER_GOOGLE_threading` – проверяет валидное использование команд Vulkan по отношению к нитям, проверяет, что никакие две нити не обращаются к одному и тому же объекту тогда, когда они не должны этого делать;
- `VK_LAYER_GOOGLE_unique_objects` – гарантирует, что каждый объект будет иметь уникальный дескриптор, что облегчает отслеживание приложением, позволяя избегать тех ситуаций, когда реализация может выдавать один и тот же дескриптор разным объектам с одинаковыми параметрами.

Кроме этого, большое количество отдельных слоев собрано вместе в один большой слой с именем `VK_LAYER_LUNARG_stardard_validation`, облегчая их использование. Библиотека к этой книге использует этот слой при сборке в отладочном режиме, в релизной сборке все слои выключены.

Расширения

Расширения крайне важны для всех кросс-платформенных открытых API, таких как Vulkan. Они позволяют разработчикам реализации экспериментировать, создавать новое и продвигать технологию вперед. В конце концов, полезные возможности, вначале введенные как расширения, становятся частью новых версий после своего опробования в деле. Однако расширения отнюдь не бесплатны. Некоторые из них могут потребовать от реализации отслеживать дополнительное состояние, выполнять дополнительные проверки при построении командных буферов или негативно влиять на быстродействие даже в том случае, когда расширение напрямую не используется. Поэтому приложение должно явно включать расширения перед их использованием. Это означает, что приложение, которое не использует расширения, не платит за него в смысле быстродействия, и практически невозможно использовать возможности из расширения, что хорошо влияет на портируемость.

Расширения делятся на две категории: расширения экземпляра и расширения устройства. Расширение *экземпляра* (instance extension) обычно задействует всю систему Vulkan на заданной платформе. Этот тип расширения либо предостав-

ляется через не зависящий от устройства слой, либо просто является расширением, доступным каждому устройству в системе, и поэтому был переведен в расширения экземпляра. Расширения *устройства* расширяют возможности одного или нескольких устройств в системе, но не обязательно присутствуют в каждом устройстве.

Каждое расширение может определять новые функции, новые структуры, перечисления и т. п. После того как расширение было включено, оно может рассматриваться как часть API, доступная приложению. Расширения экземпляра включаются при создании экземпляра, расширения устройства включаются при создании устройства. Это приводит нас к проблеме курицы и яйца: откуда мы знаем, какие расширения поддерживаются перед инициализацией экземпляра.

Опрос существующих расширений экземпляра является одной из нескольких частей функциональности Vulkan, которая может использоваться до создания экземпляра. Для этого используется функция `vkEnumerateInstanceExtensionProperties()`, прототип которой приводится ниже.

```
VkResult vkEnumerateInstanceExtensionProperties (
  const char*                   pLayerName,
  uint32_t*                     pPropertyCount,
  VkExtensionProperties*        pProperties);
```

`pLayerName` – это имя слоя, который может предоставить расширения. Пока мы просто установим его равным `nullptr`. `pPropertyCount` – это указатель на переменную, содержащую количество расширений Vulkan, о которых мы хотим получить информацию, и `pProperties` – это указатель на массив структур `VkExtensionProperties`, в которые будет записана информация о поддерживаемых расширениях. Если `pProperties` равно `nullptr`, то начальное значение `pPropertyCount` игнорируется и вместо него записывается число всех поддерживаемых расширений.

Если `pProperties` не равно `nullptr`, то считается, что число элементов в этом массиве задается в переменной, на которую указывает `pPropertyCount`, и в данный массив запишется информация о поддерживаемых расширениях, но не более этого числа. После этого в переменную, на которую указывает `pPropertyCount`, будет записано, сколько всего записей в этом массиве было записано.

Для того чтобы получить информацию обо всех поддерживаемых расширениях уровня экземпляра, необходимо вызвать функцию `vkEnumerateInstanceExtensionProperties()` два раза. Первый раз вызываем с `pProperties`, равным `nullptr`, для получения числа поддерживаемых расширений экземпляра. После этого устанавливаем правильный размер для массива, в который будет записана информация о расширениях, и снова вызываем `vkEnumerateInstanceExtensionProperties()`, но на этот раз в `pProperties` передаем адрес массива. В листинге 1.5 показано, как это делается.

Листинг 1.5 ❖ Получение расширений экземпляра

```
uint32_t numInstanceExtensions = 0;
std::vector<VkExtensionProperties> instanceExtensionProperties;
```

```
// Запрашиваем расширения экземпляра
vkEnumerateInstanceExtensionProperties(nullptr,
                                       &numInstanceExtensions,
                                       nullptr);

// Если расширения есть, то запрашиваем их свойства
if (numInstanceExtensions != 0)
{
  instanceExtensionProperties.resize(numInstanceExtensions);
  vkEnumerateInstanceExtensionProperties(nullptr,
                                         &numInstanceExtensions,
                                         instanceExtensionProperties.data());
}
```

После того как код с листинга 1.5 завершит выполнение, `instanceExtensionProperties` будет содержать список расширений, поддерживаемых экземпляром. Каждый элемент массива из `VkExtensionProperties` описывает одно расширение. Ниже приводится определение этой структуры.

```
typedef struct VkExtensionProperties {
  char          extensionName[VK_MAX_EXTENSION_NAME_SIZE];
  uint32_t      specVersion;
} VkExtensionProperties;
```

Структура `VkExtensionProperties` просто содержит имя расширения и версию расширения. По мере появления новых версий расширения может добавляться новая функциональность. Поле `specVersion` обновляет расширения без необходимости создания полностью нового расширения для добавления незначительной функциональности. Имя расширения хранится в поле `extensionName`.

Как вы видели ранее, для создания экземпляра Vulkan в структуре `VkInstanceCreateInfo` есть поле `ppEnabledExtensionNames`, являющееся указателем на массив строк с именами расширений, которые нужно включить. Если Vulkan на заданной платформе поддерживает расширение, то оно будет возвращено при вызове `vkEnumerateInstanceExtensionProperties()` и его имя может быть передано в `vkCreateInstance` через поле `ppEnabledExtensionNames` структуры `VkInstanceCreateInfo`.

Аналогично можно получить расширения устройства. Для этого вызовите функцию `vkEnumerateDeviceExtensionProperties()`, прототип которой приводится ниже.

```
VkResult vkEnumerateDeviceExtensionProperties (
  VkPhysicalDevice          physicalDevice,
  const char*               pLayerName,
  uint32_t*                 pPropertyCount,
  VkExtensionProperties*    pProperties);
```

Описание `vkEnumerateDeviceExtensionProperties()` практически аналогично `vkEnumerateInstanceExtensionProperties()`, за исключением параметра `physicalDevice`. Параметр `physicalDevice` – это дескриптор устройства, для которого запра-

шиваются расширения. Как и в vkEnumerateInstanceExtensionProperties(), функция vkEnumerateDeviceExtensionProperties() записывает в pPropertyCount число поддерживаемых расширений, если pProperties равно nullptr, если pProperties не равен nullptr, то заполняет этот массив информацией о поддерживаемых расширениях. Для информации о расширениях устройства и расширениях экземпляра используется одна и та же структура VkExtensionProperties.

Когда вы создаете устройство ppEnabledExtensionNames, структура VkDeviceCreateInfo может содержать указатель на одну из строк, которая была возвращена, – vkEnumerateDeviceExtensionProperties().

Некоторые расширения предоставляют новую функциональность в виде точек вызова. Они передаются как указатели на функции, значения которых вы должны получить у экземпляра или устройства, после того как соответствующее расширение будет включено. Функции экземпляра валидны для всего экземпляра. Если расширение расширяет функциональность на уровне экземпляра, то вы должны использовать указатель на функцию уровня экземпляра для доступа к новым возможностям.

Для получения указателя на функцию уровня экземпляра используйте функцию vkGetInstanceProcAddr(), прототип которой приводится ниже.

```
PFN_vkVoidFunction vkGetInstanceProcAddr (
  VkInstance       instance,
  const char*      pName);
```

Параметр instance является дескриптором экземпляра, для которого вы хотите получить указатель на функцию. Если ваше приложение использует более одного экземпляра Vulkan, то функция, возвращенная этой командой, валидна только для объектов, принадлежащих данному экземпляру. Имя функции передается в pName и является завершенной нулевым байтом строкой в UTF-8. Если имя функции узнается и соответствующее расширение поддерживается, то значение, возвращенное vkGetInstanceProcAddr(), является указателем на функцию, которую вы можете вызвать.

PFN_vkVoidFunction – это тип указателя на функцию со следующим описанием:

```
VKAPI_ATTR void VKAPI_CALL vkVoidFunction(void);
```

В Vulkan нет функций с этой сигнатурой, и вряд ли расширение введет такую функцию. Скорее всего, вам понадобится выполнить преобразование типа указателя на указатель на функцию подходящего типа, перед тем как вы сможете ее вызывать.

Указатели на функции уровня экземпляра валидны только для объектов, принадлежащих экземпляру, считая, что устройство, создавшее объект (или само устройство, если функция диспетчеризируется на уровне устройства), поддерживает расширение и для этого устройства расширение включено. Поскольку каждое устройство может быть реализовано внутри отдельного драйвера Vulkan, указатели на функции уровня экземпляра должны диспетчеризироваться через дополнительный уровень косвенной адресации (indirection), чтобы выйти на

нужный модуль. Поскольку это обладает определенной ценой, то можно получить указатель на функцию для данного устройства, который ссылается прямо в соответствующий драйвер.

Для получения указателя на функцию уровня устройства служит функция vkGetDeviceProcAddr(), прототип которой приводится ниже.

```
PFN_vkVoidFunction vkGetDeviceProcAddr (
  VkDevice          device,
  const char*       pName);
```

В параметре device передается устройство, для которого данная функция будет вызываться. Имя функции опять передается как имя функции в виде строки в UTF-8, завершенной нулевым байтом в параметре pName. Полученный указатель валиден только для того устройства, которое было задано в device. Device должен ссылаться на устройство, поддерживающее расширение, предоставляющее данную функцию, и данное расширение должно быть включено.

Указатель на функцию, возвращенный vkGetDeviceProcAddr(), привязан к device. Даже если одно и то же физическое устройство было использовано для создания двух или более логических устройств с теми же самыми параметрами, вы все равно должны использовать указатель на функцию только с тем устройством, для которого он был получен.

Аккуратное завершение работы

Прежде чем ваша программа завершится, вы должны аккуратно почистить за собой. Во многих случаях операционная система по завершении вашего приложения сама освободит ресурсы, которые вы выделили. Однако далеко не всегда конец вашего кода – это конец приложения. Если вы пишете компоненту большого приложения, например, то приложение может закончить рендеринг или расчеты с использованием Vulkan без завершения работы.

При очистке хорошей практикой является следующее:

- завершить или каким-то образом закончить всю работу, которую ваше приложение выполняет и на CPU, и на GPU, на всех нитях, связанных с Vulkan;
- уничтожить объекты в порядке, обратном тому, в котором они были созданы.

Логическое устройство, скорее всего, будет последним объектом, который вы создали во время инициализации вашего приложения. Прежде чем вы уничтожите устройство, вам нужно убедиться в том, что оно не выполняет никакой работы для вашего приложения. Для этого вызовите vkDeviceWaitIdle(), прототип этой функции приводится ниже.

```
VkResult vkDeviceWaitIdle (
  VkDevice              device);
```

Дескриптор устройства передается в параметре device. Когда vkDeviceWaitIdle() возвращает управление, то вся работа, переданная устройству от имени вашего

приложения, гарантированно завершилась, конечно, если в это время не передали новую работу. Вам необходимо убедиться в том, что никакая другая нить не передает работу уничтожаемому устройству.

После того как вы убедились, что устройство ничем больше не занято, вы можете спокойно его уничтожить. Для этого служит функция `vkDestroyDevice ()`, прототип которой приводится ниже.

```
void vkDestroyDevice (
  VkDevice                          device,
  const VkAllocationCallbacks*      pAllocator);
```

Дескриптор уничтожаемого устройства передается в параметре `device`, доступ к которому должен быть внешне синхронизирован. Обратите внимание, что доступ к устройству не должен быть внешне синхронизирован для любой другой команды. Приложение должно убедиться, что устройство не уничтожается в то время, как к нему обращаются другие нити.

`pAllocator` должен указывать на структуру аллокатора, совместимую с той, которая использовалась при создании устройства. После того как устройство было уничтожено, ему нельзя передавать никаких команд. Также нельзя использовать дескриптор этого устройства в качестве аргумента какой-либо функции, включая функции уничтожения объектов, принимающие устройство в качестве первого аргумента. Это еще одна причина, почему вы должны уничтожать объекты в обратном порядке, нежели в том, в котором они создавались.

После того как все устройства, связанные с экземпляром Vulkan, были уничтожены, можно уничтожить и сам экземпляр. Это осуществляется вызовом функции `vkDestroyInstance()`, прототип которой приводится ниже.

```
void vkDestroyInstance (
  VkInstance                        instance,
  const VkAllocationCallbacks*      pAllocator);
```

Дескриптор экземпляра, который подлежит уничтожению, передается в параметре `instance`, и, как и при уничтожении устройства, в `pAllocator` передается указатель на структуру аллокатора, совместимую с той, которая использовалась при создании экземпляра. Если в `vkCreateInstance()` параметр `pAllocator` был равен `nullptr`, то и в `vkDestroyInstance()` этот параметр тоже должен быть равен `nullptr`.

Обратите внимание, что не нужно уничтожать физические устройства. Физические устройства не создаются отдельными функциями создания, в отличие от логических устройств. Можно считать, что физические устройства, которые были возвращены `vkEnumeratePhysicalDevice()`, принадлежат экземпляру. Соответственно, при уничтожении экземпляра они тоже автоматически уничтожаются.

Резюме

Это глава познакомила вас с Vulkan. Вы увидели, как все состояние Vulkan содержится в экземпляре. Экземпляр предоставляет доступ к физическим устройствам, и каждое физическое устройство предоставляет некоторый набор очередей, которые могут быть использованы для выполнения работы. Вы увидели, как создать логическое устройство, соответствующее физическому устройству. Вы увидели, как можно расширять Vulkan, как определить доступные экземпляру и устройству расширения и как включать эти расширения. Вы увидели, как аккуратно завершить работу с системой Vulkan путем ожидания завершения переданной вашим приложением работы, уничтожением дескрипторов устройств и, наконец, уничтожением дескриптора экземпляра.