Для тебя, читатель.
– Грехем Селлерс

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 сразу 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.

Благодарности

В первую и самую главную очередь я хотел бы поблагодарить всех членов рабочей группы по Vulkan. Сквозь утомительный и крайне длинный процесс мы создали то, что, как я считаю, будет надежной основой компьютерной графики и ускоренных вычислений на многие годы. Я хотел бы особенно отметить вклад моих коллег из AMD, которые создали спецификации Mantle, из которых и был создан Vulkan.

Я хотел бы поблагодарить наших рецензентов Дэна Гинсбурга и Криса «Xenon» Нэнсона. Благодарю вас за ваш ценный вклад, без которого эта книга наверняка содержала бы больше ошибок и упущений, чем есть сейчас. Я также хотел бы поблагодарить моего коллегу Мэйса Алнассера за отличный вклад в улучшение качества этой книги. Также я благодарю всю команду Vulkan в AMD, чья работа позволила мне протестировать основную часть примеров до того, как Vulkan стал доступен широкой публике.

Изображение на обложке Домиником Агоро-Омбакой из Agoro Design (http://agorodesign.com). Благодарю его за то, что он уложился в такой жесткий график.

Огромная благодарность моему редактору Лауре Левин и всей команде в Addison-Wesley за то, что позволили мне неоднократно нарушать график, делать изменения в последний момент и вообще терпели меня. Я ценю ваше доверие ко мне в этом проекте.

Наконец, я хотел бы поблагодарить мою семью – мою жену Крис и моих детей Джереми и Эмилию. «Папа, ты все еще пишешь свою книгу?» – постоянно звучало в нашем доме. Я ценю ваше терпению, любовь и поддержку во время написания книги.

– Грехем Селлерс

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.

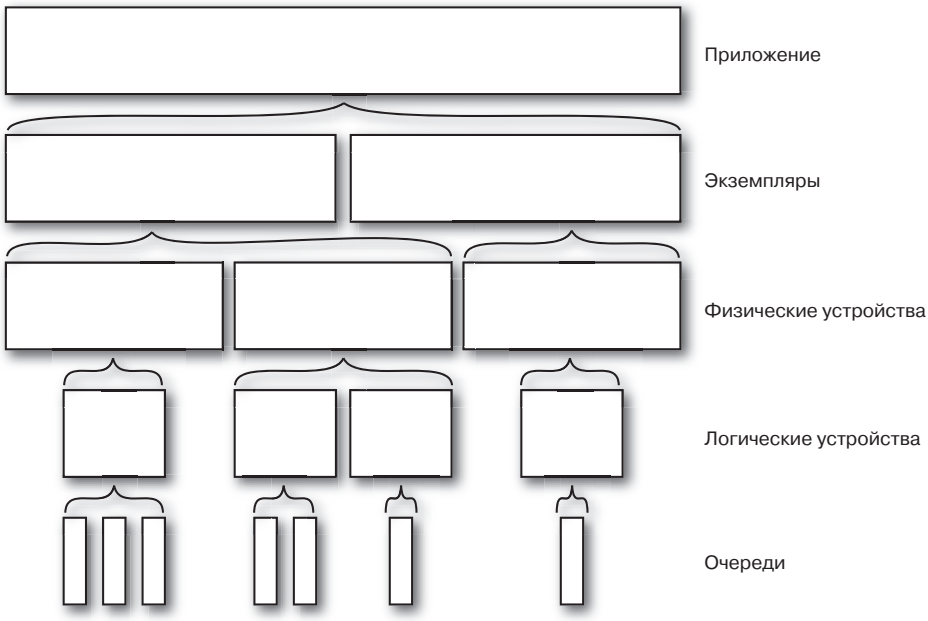

Рис. 1.1 ❖ Иерархия Vulkan из экземпляров, устройств и очередей

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, если 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.

Конец ознакомительного фрагмента.
Приобрести книгу можно
в интернет-магазине «Электронный универс»
(e-Univers.ru)