目录

Vulkan Programming Guide 第一章（2）
原创 2017 年 03 月 21 日 00:46:03 25330
物理设备内存

在许多情况下，Vulkan 设备是与主主机处理器分离的硬件的物理部件，或者工作方式不同，以便以专门的方式访问存储器。 Vulkan 中的设备内存指的是设备可访问的内存，可用作纹理和其他数据的后备存储。 内存分为多种类型，每种类型都有一组属性，例如主机和设备之间的缓存标志和一致性行为。 每种类型的存储器然后由设备的堆之一支持，其中可以有几个。
要查询堆的配置和设备支持的内存类型，请调用

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

生成的内存结构写入 VkPhysicalDeviceMemoryProperties 结构，其结构在 pMemoryProperties 中传递。 VkPhysicalDeviceMemoryProperties 结构包含设备堆及其支持的内存类型的属性。 这个结构的定义是

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

内存类型的多少在 memoryTypeCount 字段中获取。 可能获取的内存类型的最大数目是 VK_MAX_MEMORY_TYPES 的值，其定义为 32. memoryTypes 数组包含描述每种内存类型的 memoryTypeCount VkMemoryType 结构。 VkMemoryType 的定义是

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

这是一个只包含一组标志和内存类型的堆索引的简单结构。 flags 字段描述了存储器的类型，并由 VkMemoryPropertyFlagBits 标志的组合构成。 标志的含义如下：
•VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT 表示内存是本地的（即物理连接到）设备。如果该位未设置，则可以假定存储器对于主机是本地的。
•VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT 表示使用的内存分配
此类型可以由主机映射和读取或写入。如果该位未被设置，则这种类型的存储器不能被主机直接访问，而是被设备独占使用。
•VK_MEMORY_PROPERTY_HOST_COHERENT_BIT 表示当此类型的内存由主机和设备同时访问时，这些访问将在两个客户端之间一致。如果未设置此位，则设备或主机可能不会看到每个执行的写入的结果，直到高速缓存被显式刷新。
•VK_MEMORY_PROPERTY_HOST_CACHED_BIT 表示此类型的内存中的数据由主机缓存。对这种类型的存储器的读取访问通常比如果该位未被设置时的读取访问更快。然而，设备的访问可以具有略高的延迟，特别是如果存储器也是一致的。

•VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT 表示使用此类型分配的内存不一定立即从关联的堆消耗空间，并且驱动程序可能推迟物理内存分配，直到内存对象用于回退资源。每种内存类型都报告在 VkMemoryType 结构的 heapIndex 字段，并且可以得到该堆消耗空间的大小。 这是从调用 vkGetPhysicalDeviceMemoryProperties（）得到 kPhysicalDeviceMemoryProperties 结构中返回的 memoryHeaps 数组的索引。memoryHeaps 数组的每个元素描述了设备的一个内存堆。 这个结构的定义是

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

同样，这是一个简单的结构。 它只包含堆的大小（以字节为单位）和描述堆的一些标志。在 Vulkan 1.0 中，唯一定义的标志是 VK_MEMORY_HEAP_DEVICE_LOCAL_BIT。 如果该位置 1，则堆对设备是本地的。 这对应于描述存储器类型的类似命名的标志。

设备队列

Vulkan 设备执行提交到队列的工作。 每个设备将有一个或多个队列，并且每个队列将属于设备的队列系列之一。 队列系列是一组具有相同功能但能够并行运行的队列。 队列系列的数量，每个系列的能力以及属于每个系列的队列的数量都是物理设备的属性。 要查询设备的队列系列，请调用 vkGetPhysicalDeviceQueueFamilyProperties（），其原型为

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

vkGetPhysicalDeviceQueueFamilyProperties () 工作有点像 vkEnumeratePhysicalDevices ()，因为它希望你调用它两次。 第一次，你传递 nullptr 作为 pQueueFamilyProperties，在 pQueueFamilyPropertyCount，你传递一个指针，将被覆盖的变量

设备支持的队列系列数。 您可以使用此数字来适当地确定 VkQueueFamilyProperties 的数组大小。 然后，在第二次调用时，在 pQueueFamilyProperties 中传递此数组，并且 Vulkan 将使用队列的属性填充它。 VkQueueFamilyProperties 的定义是

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

此结构中的第一个字段 queueFlags 描述了队列的总体功能。该字段由 VkQueueFlagBits 位的组合构成，其含义如下：

•如果设置了 VK_QUEUE_GRAPHICS_BIT，则此系列中的队列支持图形操作，例如绘制点，线和三角形。

•如果设置了 VK_QUEUE_COMPUTE_BIT，则此系列中的队列支持计算操作，例如调度计算着色器。

•如果设置了 VK_QUEUE_TRANSFER_BIT，则此系列中的队列支持传输操作，例如复制缓冲区和映像内容。
•如果设置了 VK_QUEUE_SPARSE_BINDING_BIT，则此系列中的队列支持用于更新稀疏资源的内存绑定操作。
queueCount 字段表示系列中的队列数。这可以被设置为 1，或者如果设备支持具有相同基本功能的多个队列，则可以显着更高。
timestampValidBits 字段指示当从队列获取时间戳时有多少位有效。如果此值为零，则队列不支持时间戳。如果它是非零，那么它保证至少为 36 位。此外，如果设备的 VkPhysicalDeviceLimits 结构的 timestampComputeAndGraphics 字段是 VK_TRUE，则支持 VK_QUEUE_GRAPHICS_BIT 或 VK_QUEUE_COMPUTE_BIT 的所有队列都保证支持具有至少 36 位分辨率的时间戳。在这种情况下，不需要单独检查每个队列。
最后，minImageTimestampGranularity 字段指定队列支持图像传输的单元（如果有的话）。
注意，可能是设备报告具有明显相同属性的多个队列系列的情况。一个家庭中的队列基本上是相同的。不同族中的队列可能具有不同的内部功能，不能在 Vulkan API 中轻松表达。为此，实现可以选择报告与不同族的成员相似的队列。这对这些队列之间的资源如何共享施加了额外的限制，这可能允许实现来适应这些差异。

清单 1.2 演示了如何查询物理设备的内存属性和队列系列属性。 您将需要在创建逻辑设备之前检索队列系列属性，如下一节中所述。

清单 1.2：查询物理设备属性

```
uint32_t queueFamilyPropertyCount;
std::vector<VkQueueFamilyProperties> queueFamilyProperties;
VkPhysicalDeviceMemoryProperties physicalDeviceMemoryProperties;

// Get the memory properties of the physical device.
vkGetPhysicalDeviceMemoryProperties(m_physicalDevices[deviceIndex],&physicalDeviceMemoryProperties);

/* First determine the number of queue families supported by the physical device. */
vkGetPhysicalDeviceQueueFamilyProperties(m_physicalDevices[0],
&queueFamilyPropertyCount, nullptr);
// Allocate enough space for the queue property structures.
queueFamilyProperties.resize(queueFamilyPropertyCount);
// Now query the actual properties of the queue families.
vkGetPhysicalDeviceQueueFamilyProperties(m_physicalDevices[0],&queueFamilyPropertyCount,
queueFamilyProperties.data());
```

创建逻辑设备
枚举系统中的所有物理设备后，您的应用程序应选择一个设备并创建与其对应的逻辑设备。逻辑设备表示处于初始化状态的设备。 在创建逻辑设备期间，您可以选择启用可选功能，打开所需的扩展程序等。 创建逻辑设备是通过调用 vkCreateDevice（）执行的，其原型是

```
VkResult vkCreateDevice (
```

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

新逻辑设备对应的物理设备在 physicalDevice 中传递。 有关新逻辑设备的信息通过 pCreateInfo 结构在 VkDeviceCreateInfo 结构的实例中传递。 VkDeviceCreateInfo 的定义是

```
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;
```

VkDeviceCreateInfo 结构的 sType 字段应设置为 VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO。 像往常一样，除非你使用扩展，pNext 应该设置为 nullptr。 在当前版本的 Vulkan 中，没有为结构的标志字段定义位，因此也将其设置为零。

接下来是队列创建信息。 pQueueCreateInfos 是指向一个或多个 VkDeviceQueueCreateInfo 结构的数组的指针，每个结构都允许指定一个或多个队列。 数组中的结构数量在 queueCreateInfoCount 中给出。 VkDeviceQueueCreateInfo 的定义是

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

VkDeviceQueueCreateInfo 的 sType 字段为 VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO。目前没有定义用于标志字段的标志，因此应将其设置为零。 queueFamilyIndex 字段指定要创建的队列系列。这是从 vkGetPhysicalDeviceQueueFamilyProperties（）返回的队列系列数组的索引。要在此系列中创建队列，请将 queueCount 设置为要使用的队列数。当然，设备必须至少在你选择的系列中支持这么多的队列。

pQueuePriorities 字段是可选的指针，指向浮点型数组，表示提交到每个队列的工作的相对优先级。这些数字是在 0.0 至 1.0 的范围内的归一化数字。具有较高优先级的队列可以被分配更多处理资源或者比具有较低优先级的队列更积极地调度。将 pQueuePriorities 设置为

nullptr 会使队列保持相同的默认优先级。
请求的队列按优先级顺序排序，然后分配依赖于设备的相对优先级。队列可能承担的离散优先级的数量是设备特定的参数。你可以通过
检查从调用 vkGetPhysicalDeviceProperties（）返回的 VkPhysicalDeviceLimits 结构的 discreteQueuePriorities 字段来确认。例如，如果设备仅支持低优先级和高优先级工作负载，则此字段将为 2.所有设备都至少支持两个离散优先级。然而，如果设备支持任意优先级，则该字段可以高得多。不管 discreteQueuePriorities 的值如何，队列的相对优先级仍然表示为浮点值。
返回到 VkDeviceCreateInfo 结构，enabledLayerCount，ppEnabledLayerNames，enabledExtensionCount 和 ppEnabledExtensionNames 字段用于启用层和扩展。我们将在本章后面讨论这两个主题。现在，我们将 enabledLayerCount 和 enabledExtensionCount 都设置为零，并将 ppEnabledLayerNames 和 ppEnabedExtensionNames 设置为 nullptr。
VkDeviceCreateInfo 的最后一个字段 pEnabledFeatures 是指向 VkPhysicalDeviceFeatures 结构的实例的指针，它指定您的应用程序希望使用哪些可选功能。如果你不想使用任何可选功能，你可以简单地设置为 nullptr。然而，Vulkan 在这种形式是相对有限的，其大部分有趣的功能将被禁用。要确定设备支持哪些可选功能，请如前所述调用 vkGetPhysicalDeviceFeatures（）。
vkGetPhysicalDeviceFeatures（）将设备支持的功能集写入到您传入的 VkPhysicalDeviceFeatures 结构的实例中。通过简单查询物理设备的功能，然后将相同的 VkPhysicalDeviceFeatures 结构传回 vkCreateDevice（），您可以启用每个可选功能设备支持并且不请求设备不支持的特征。
只是启用每个支持的功能，可能会带来一些性能影响。对于某些功能，Vulkan 实现可能需要分配额外的内存，跟踪附加状态，稍微不同地配置硬件，或执行一些其他操作，否则会使应用程序花费。启用不会使用的功能不是一个好主意。在优化的应用程序中，您应该从设备查询支持的功能;然后，从支持的功能部件，启用您的应用程序需要的特定功能。

清单 1.3 显示了一个简单的示例，它查询设备的支持的功能，设置应用程序需要的功能列表。支持曲面细分和几何着色器是绝对必需的，如果设备支持，则启用对多画面间接的支持。然后，代码使用其第一个队列的单个实例创建一个设备。
清单 1.3：创建逻辑设备

```
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);
```

在清单 1.3 中的代码运行并成功创建逻辑设备之后，已启用的功能集存储在 requiredFeatures 变量中。 这可以保留以供以后使得可以可选地使用特征的代码可以检查它是否被成功地启用并且优雅地回退。

Vulkan Programming Guide 第一章（3）

原创 2017 年 03 月 26 日 23:30:14

- 224
- 0
- 0

对象类型和函数约定

事实上，Vulkan 中的所有内容都被表示为一个由句柄引用的对象。句柄分为两大类：可分派对象和不可分散对象。在大多数情况下，这与应用程序无关，仅影响 API 的结构以及系统级组件（如 Vulkan 加载器和层）与这些对象的互操作性。

可分派对象是内部包含调度表的对象。这是各种组件使用的功能表，用于确定当您的应用程序调用 Vulkan 时要执行的代码部分。这些类型的对象通常是较大较复杂的结构，目前由实例（VkInstance），物理设备（VkPhysicalDevice），逻辑设备（VkDevice），命令缓冲区（VkCommandBuffer）和队列（VkQueue）组成）构成。所有其他对象被认为是不可分散的。任何 Vulkan 函数的第一个参数总是一个可调度的对象。此规则的唯一例外是与实例的创建和初始化相关的功能。

管理内存

Vulkan 提供两种类型的内存：主机内存和设备内存。由 Vulkan API 创建的对象通常需要一定数量的主机内存。这就是 Vulkan 实现将存储对象状态以及实现 Vulkan API 所需的任何数据。资源对象（如缓冲区和图像）需要一定量的设备内存。这是存储在资源中的数据的存储器。

您的应用程序可以管理 Vulkan 实现的主机内存，并且需要您的应用程序管理设备内存。为此，您将需要创建一个设备内存管理子系统。您可以查询您创建的每个资源，以获得支持所需的内存量和类型。应用程序将分配正确的内存量并将其附加到资源对象，然后才能使用它。在诸如 OpenGL 这样的更高级别的 API 中，这个“魔术”是由驱动程序代表你的应用程序执行的。然而，一些应用程序需要非常大量的小资源，而其他应用程序需要较少数量的非常大的资源。一些应用程序在执行过程中创建和销毁资源，而其他应用程序在启动时创建所有资源，并且在终止之前不释放它们。

在这些情况下使用的分配策略可能会有很大的不同。没有一刀切的策略。 OpenGL 驱动程序不知道您的应用程序将如何运行，因此必须调整分配策略以尝试适应您的使用模式。另一方面，您，应用程序开发人员，您的应用程序的行为准确无误。您可以将资源分成长期和短暂的组。您可以将将一起使用的资源分成少量池化分配。您可以决定应用程序使用的分配策略

重要的是要注意，每个“实时”内存分配会给系统带来一些成本。因此，将分配对象的数量保持在最小值是很重要的。建议设备内存分配器以大块分配内存。许多小资源可以放置在少得多的设备内存块内。第 2 章“内存和资源”中讨论了设备内存分配器的一个例子，它们更详细地讨论了内存分配。

Vulkan 的多线程

支持多线程应用程序是 Vulkan 设计的一个组成部分。 Vulkan 通常假定应用程序将确保没有两个线程同时在同一对象上进行变异。这被称为外部同步。在 Vulkan 的性能关键部分（如构建命令缓冲区）中的绝大多数 Vulkan 命令根本不提供同步。

为了具体定义各种 Vulkan 命令的线程命令，必须保护不受主机并发访问的每个参数被标记为外部同步。在某些情况下，对象或其他数据的句柄将被包含在数据结构中，包含在数组中，或以其他方式传递给命令。这些参数也必须是外部同步的。

这样做的意图是，Vulkan 实现不需要在内部采用互斥体或使用其他同步原语来保护数据结构。这意味着多线程程序很少停止或跨线程阻塞。

除了要求主机在线程间使用共享对象的同时访问之外，Vulkan 还提供了许多更高级别的功能，专门用于允许线程执行工作而不会相互阻止。这些包括以下内容：

•主机内存分配可以通过传递给对象创建功能的主机内存分配结构来处理。通过使用每个线程的分配器，该分配器中的数据结构不需要被保护。主机内存分配器在第 2 章“内存和资源”中有介绍。

•命令缓冲区是从池分配的，对池的访问是外部同步的。如果应用程序每个线程使用单独的命令池，则可以从这些池中分配命令缓冲区，而不会相互阻止。命令缓冲区和池在第 3 章“队列和命令”中有介绍。

•描述符按描述符池分配。描述符是在设备上运行的着色器使用的资源的表示。它们在第 6 章“着色器和管道”中有详细描述。如果每个线程都使用单独的池，则可以从这些池中分配描述符集，而不会使线程相互阻塞。

•二级命令缓冲区允许并行生成大型 renderpass（必须包含在单个命令缓冲区中）的内容，然后在从主命令缓冲区调用时分组。第 13 章“Multipass Rendering”中详细介绍了二级命令缓冲区。

当您构建一个非常简单的单线程应用程序时，创建从其分配对象的池的需求可能看起来很多不必要的间接。然而，随着应用程序在线程数量上的扩展，这些对象是实现高性能的必不可少的。

在本书的其余部分中，关于线程的任何特殊要求将在引入命令时注明。

数学概念

计算机图形学和大多数异构计算应用程序都是以数学作为基础。大多数 Vulkan 设备都是基于非常强大的计算处理器。 在撰写本文时，即使是适中的移动处理器也能够提供许多千兆位的处理能力，而高端桌面和工作站处理器则提供了数千倍的数字处理能力。 因此，真正有趣的应用程序将建立在偏重数学的着色器上。 此外，Vulkan 处理管道的几个固定功能部分建立在硬连接到设备和规范中的数学概念上。

向量和矩阵

任何图形应用程序的基本构建块之一是向量。无论它们是位置，方向，颜色还是其他数量的表示，在整个图形文献中都使用向量。向量的一种常见形式是均匀矢量，它是一个空间中的矢量，该空间的一维高于其所代表的数量。这些向量用于存储投影坐标。将均匀矢量乘以任何标量产生表示相同投影坐标的新矢量。要投射一个点矢量，通过其最后一个分量进行分割，产生 x，y，z，1.0 形式的向量（对于四分量矢量）。

要将矢量从一个坐标空间转换到另一个坐标空间，将该矢量乘以一个矩阵。正如 3D 空间中的一点被表示为四分量均匀矢量，在 3D 均匀矢量上运行的变换矩阵是 4×4 矩阵。

3D 空间中的点通常表示为常规称为 x，y，z 和 w 的四个分量的均匀矢量。对于一个点，w 分量通常以 1.0 开始，并且随着向量通过投影矩阵变换而改变。在通过 w 分量分配之后，该点通过以下变量进行投影：如果没有一个变换是投影变换，则 w 保持为 1.0，除以 1.0 对矢量没有影响。如果矢量进行投影变换，则 w 不等于 1.0，但是通过它将会投射点并将 w 返回到 1.0。

同时，3D 空间中的方向也表示为 w 分量为 0.0 的均匀矢量。用正确构造的 4×4 投影矩阵乘以方向矢量将使 w 分量保持在 0.0，对任何其他分量都不起作用。通过简单地丢弃附加组件，您可以将 3D 方向向量通过与 4D 均匀 3D 点向量相同的变换，并使其经常进行旋转，缩放和其他变换。

坐标系

Vulkan 通过将其端点或角点表示为 3D 空间中的点来处理图形基元（如线条和三角形）。这些图元被称为顶点。 Vulkan 系统的输入是相对于它们所属对象的原点的 3D 坐标空间中的顶点坐标（以 w 分量为 1.0 表示为均匀矢量）。这个坐标空间被称为对象空间或有时是模型空间。

通常，管道中的第一个着色器将将该顶点转换为视图空间，该空间是相对于查看器的位置。通过将顶点的位置矢量乘以变换矩阵来进行该变换。这通常被称为对象视图矩阵或模型视图矩阵。

有时，需要一个顶点的绝对坐标，例如当找到一个顶点相对于某个其他对象的坐标时。这个全球空间被称为世界空间，是相对于全球起点的顶点的位置。

从视图空间，顶点的位置被转换为剪辑空间。这是 Vulkan 的几何处理部分使用的最终空间，并且是将顶点推入典型 3D 应用程序使用的投影空间时通常会转换顶点的空间。剪辑空间是所谓的，因为它是大多数实现执行剪辑的坐标空间，其中移除位于被呈现的可见区域之外的原始图元。

从剪切空间中，顶点位置通过划分其 W 分量进行归一化。这产生称为标准化设备坐标或 NDC 的坐标空间，并且该过程通常被称为透视分割。在该空间中，坐标系的可见部分在 x 和 y 方向上为-1.0 至 1.0，在 z 方向为 0.0 至 1.0。在透视分割之前，这个地区之外的任何东西都将被剪掉。

最后，顶点的归一化设备坐标由视口变换，它描述了 NDC 如何映射到要渲染图片的窗口或图像中。

(OpenGL 视图变换：[http://www.songho.ca/opengl/index.html])

增强 Vulkan

虽然 Vulkan 的核心 API 规范相当广泛，但绝非全然。 一些功能是可选的，而更多的可用层（其修改或增强现有行为）和扩展（向 Vulkan 添加新功能）的形式。 这两个增强机制将在以下部分进行描述。

图层

层是 Vulkan 的特征，允许修改其行为。 层通常拦截全部或部分 Vulkan，并添加诸如日志记录，跟踪，提供诊断，分析等功能。 可以在实例级别添加一个层，在这种情况下，它会影响整个 Vulkan 实例以及可能由其创建的每个设备。 或者，可以在设备级别添加该层，在这种情况下，该层仅影响其启用的设备。

要发现系统上实例可用的层，请调用 vkEnumerateInstanceLayerProperties（），其原型是

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

如果 pProperties 为 nullptr，那么 pPropertyCount 应指向一个将被 Vulkan 可用层数计数覆盖的变量。 如果 pProperties 不是 nullptr，那么它应该指向一个 VkLayerProperties 结构的数组，它将填充有关系统注册的层的信息。 在这种情况下，pPropertyCount 指向的变量的初始值是由 pProperties 指向的数组的长度，并且该变量将被该命令覆盖的数组中的条目数量覆盖。

属性数组的每个元素都是 Layer Properties 结构的一个实例，它的定义是

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

每个层都有一个正式的名称，它存储在 VkLayerProperties 结构的 layerName 成员中。由于每个层的规范可能会随着时间的推移而改进，澄清或附加，因此在 specVersion 中报告了层实现的版本。

随着规格随着时间的推移而改进，这些规范的实现也是如此。实现版本存储在 VkLayerProperties 结构的 implementationVersion 字段中。这允许实现提高性能，修复错误，实现更广泛的可选功能集等。应用程序写入器可以识别层的特定实现，并且仅当该实现的版本超过特定版本时选择使用它，其中例如已知固定的关键错误。

最后，描述层的人类可读字符串存储在描述中。此字段的唯一目的是在用户界面中记录或显示，仅供参考。

清单 1.4 说明了如何查询 Vulkan 系统支持的实例层。

uint32_t numInstanceLayers = 0; std::vector<VkLayerProperties> instanceLayerProperties; // Query the instance layers. vkEnumerateInstanceLayerProperties(&numInstanceExtensions, nullptr); // If there are any layers, query their properties. if (numInstanceLayers != 0) { instanceLayerProperties.resize(numInstanceLayers); vkEnumerateInstanceLayerProperties(nullptr, &numInstanceLayers, instanceLayerProperties.data()); }

如上所述，不仅在实例级别，层可以被注入。 层也可以在设备级应用。 要确定哪些层可用于设备，请调用 vkEnumerateDeviceLayerProperties（），其原型是

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

可用于系统中的每个物理设备的层可以是不同的，因此每个物理设备可以报告不同的层集合。 要查询的层的物理设备在 physicalDevice 中传递。 对

vkEnumerateDeviceLayerProperties（）的 pPropertyCount 和 pProperties 参数与 vkEnumerateInstanceLayerProperties（）的相同命名的参数类似。 设备层也由 VkLayerProperties 结构的实例描述。
要启用实例级别的层，请将其名称包含在用于创建实例的 VkInstanceCreateInfo 结构的 ppEnabledLayerNames 字段中。 同样，为了在创建与系统中的物理设备相对应的逻辑设备时启用层，请在用于创建设备的 VkDeviceCreateInfo 的 ppEnabledLayerNames 成员中包含层名称。
官方 SDK 中包含了几个层次，其中大部分内容与调试，参数验证和日志记录有关。 这些包括以下内容：
•VK_LAYER_LUNARG_api_dump 将 Vulkan 调用及其参数和值打印到控制台。
•VK_LAYER_LUNARG_core_validation 对描述符集，流水线状态和动态状态中使用的参数和状态执行验证;验证 SPIR-V 模块和图形管道之间的接口;并跟踪并验证用于返回对象的 GPU 内存的使用情况。
•VK_LAYER_LUNARG_device_limits 确保将值传递给 Vulkan 命令作为参数或数据结构成员属于设备支持的功能集限制。
•VK_LAYER_LUNARG_image 验证图像使用情况与支持的格式一致。
•VK_LAYER_LUNARG_object_tracker 对 Vulkan 对象执行跟踪，尝试执行
捕获泄漏，使用后自由错误和其他无效对象使用。
•VK_LAYER_LUNARG_parameter_validation 确认传递给 Vulkan 函数的所有参数值都有效。
•VK_LAYER_LUNARG_swapchain 对第 5 章“演示文稿”中描述的 WSI（Windows 系统集成）扩展功能提供的功能进行验证。
•VK_LAYER_GOOGLE_threading 确保了对于线程的 Vulkan 命令的有效使用，确保没有两个线程在不同时间访问同一个对象。
•VK_LAYER_GOOGLE_unique_objects 确保每个对象都有一个唯一的句柄，以便应用程序更容易的跟踪，避免实现可能会重复使用相同参数表示对象的句柄。
除此之外，大量单独的层被分组成一个更大的单层，称为 VK_LAYER_LUNARG_standard_validation，使其易于打开。该书的应用程序框架在内置在调试模式下时可以启用此层，当内置在释放模式时，所有层都被禁用。
扩展
扩展是任何跨平台，开放 API（如 Vulkan）的基础。它们允许实施者进行实验，创新，并最终推动技术向前发展。最终，作为扩展功能最初引入的有用功能在该领域得到证明后可以进入未来版本的 API。但是，扩展不是没有成本。有些可能需要实现跟踪附加状态，在命令缓冲区构建期间进行额外的检查，或者即使扩展不是直接使用也会带来一些性能损失。因此，应用程序必须显式启用扩展，才能使用它们。这意味着不使用扩展的应用程序在性能或复杂性方面不支付费用，并且几乎不可能不小心使用扩展功能，从而提高了可移植性。
扩展程序分为两类：实例扩展和设备扩展。实例扩展是通常在平台上增强整个 Vulkan 系统的扩展。这种类型的扩展是通过与设备无关的层提供的，或者仅仅是由系统上的每个设备暴露的扩展，并被提升为一个实例。设备扩展是扩展系统中的一个或多个设备的功能，但不一定在每个设备上可用。
每个扩展可以定义新的功能，新的类型，结构，枚举等。一旦启用，扩展程序可以被认为是应用程序可用的 API 的一部分。必须在创建 Vulkan 实例时启用实例扩展，并且必须在创建设备时启用设备扩展。这些让我们有一个鸡和蛋谁先有的情况：我们如何知道在初始化 Vulkan 实例之前支持哪些扩展？
查询支持的实例扩展是在创建 Vulkan 实例之前可以使用的几个 Vulkan 功能之一。这是使用

vkEnumerateInstanceExtensionProperties（）函数执行的，其原型是
VkResult vkEnumerateInstanceExtensionProperties (
const char* pLayerName, uint32_t* pPropertyCount,
VkExtensionProperties* pProperties);
pLayerName 是可能提供扩展名的图层的名称。现在，将其设置为 nullptr。pPropertyCount 是一个指向包含查询 Vulkan 的实例扩展数量的变量的变量，pProperties 是一个指向将被填充有关受支持扩展的信息的 VkExtensionProperties 结构数组的指针。 如果 pProperties 为 nullptr，则 pPropertyCount 指向的变量的初始值将被忽略，并覆盖支持的实例扩展数。
如果 pProperties 不为 nullptr，则数组中的条目数假定为 pPropertyCount 指向的变量，并且直到该数组的许多条目都填充有关受支持扩展名的信息。 pPropertyCount 指向的变量将被实际写入 pProperties 的条目数覆盖。
要正确查询所有受支持的实例扩展，请调用 vkEnumerateInstanceExtensionProperties（）两次。 第一次调用 pProperties 设置为 nullptr 来检索支持的实例扩展的数量。 然后适当地调整数组以接收扩展属性并再次调用 vkEnumerateInstanceExtensionProperties（），此时将 pProperties 中的数组的地址传递给它。 清单 1.5 演示了如何做到这一点。
清单 1.5：查询实例扩展
uint32_t numInstanceExtensions = 0; std::vector<VkExtensionProperties> instanceExtensionProperties; // Query the instance extensions. vkEnumerateInstanceExtensionProperties(nullptr, &numInstanceExtensions, nullptr); // If there are any extensions, query their properties. if (numInstanceExtensions != 0) { instanceExtensionProperties.resize(numInstanceExtensions); vkEnumerateInstanceExtensionProperties(nullptr, &numInstanceExtensions, instanceExtensionProperties.data()); }
在清单 1.5 中的代码完成执行后，instanceExtensionProperties 将包含实例支持的扩展列表。VkExtensionProperties 数组的每个元素都描述了一个扩展。VkExtensionProperties 的定义是
typedef struct VkExtensionProperties { char extensionName[VK_MAX_EXTENSION_NAME_SIZE]; uint32_t specVersion; } VkExtensionProperties;
VkExtensionProperties 结构只包含扩展名和扩展名的版本。扩展可能随着时间的推移而增加功能，因为扩展的新修订版本将被生成。 specVersion 字段允许更新扩展，而不需要创建一个全新的扩展，以便添加次要功能。扩展名的名称存储在 extensionName 中。
如前所述，创建 Vulkan 实例时，VkInstanceCreateInfo 结构中有一个名为 ppEnabledExtensionNames 的成员，它是一个指向要启用扩展名的字符串数组的指针。如果平台上的 Vulkan 系统支持扩展，该扩展将包含在从 vkEnumerateInstanceExtensionProperties（）返回的数组中，并且可以通过 VkInstanceCreateInfo 结构的 ppEnabledExtensionNames 字段将其名称传递给 vkCreateInstance（）。
查询对设备扩展的支持是一个类似的过程。要做到这一点，调用 vkEnumerateDeviceExtensionProperties（），其原型是
VkResult vkEnumerateDeviceExtensionProperties (VkPhysicalDevice physicalDevice, const char* pLayerName, uint32_t* pPropertyCount, VkExtensionProperties* pProperties);
vkEnumerateDeviceExtensionProperties()的原型与 vkEnumerateInstanceExtensionProperties（）的原型几乎相同，只是额外添加了 physicalDevice 参数。 physicalDevice 参数是其查询扩展名的设备的句柄。与 vkEnumerateInstanceExtensionProperties () 一样，如果 pProperties 为 nullptr，则 vkEnumerateDeviceExtensionProperties（）将覆盖 pPropertyCount 与支持的

扩展数量，如果 pProprties 不为 nullptr，则使用有关所支持扩展的信息填充该数组。相同的 VkExtensionProperties 结构用于设备扩展和实例扩展。
创建物理设备时，VkDeviceCreateInfo 结构的 ppEnabledExtensionNames 字段可能包含指向从 vkEnumerateDeviceExtensionProperties（）返回的一个字符串的指针。
一些扩展提供了您可以调用的附加入口点形式的新功能。这些显示为函数指针，其值在启用扩展后必须从实例或设备查询。实例函数是对整个实例有效的函数。如果扩展扩展了实例级功能，则应该使用实例级函数指针来访问新功能。
要检索一个实例级的函数指针，调用 vkGetInstanceProcAddr（），其原型是
PFN_vkVoidFunction vkGetInstanceProcAddr (VkInstance instance, const char* pName);
实例参数是要检索新函数指针的实例的句柄。 如果您的应用程序使用多个 Vulkan 实例，则从此命令返回的函数指针仅对被引用实例拥有的对象有效。 函数的名称以 pName 传递，该名称是一个非终止的 UTF-8 字符串。 如果识别功能名称并启用扩展名，则 vkGetInstanceProcAddr（）的返回值是可从应用程序调用的函数指针。
PFN_vkVoidFunction 是指向以下声明的函数的指针的类型定义：
VKAPI_ATTR void VKAPI_CALL vkVoidFunction（void）；
Vulkan 没有功能具有这种特殊的签名，扩展不太可能引入这样的功能。 很可能，您需要将生成的函数指针转换为相应签名的指针，然后才能调用它。
假设创建对象的设备（或设备本身，如果设备上的功能调度）支持扩展，并且该设备的扩展名已启用，则实例级别的函数指针对于实例拥有的任何对象都是有效的。 因为每个设备可能会在不同的 Vulkan 驱动程序中实现，所以实例函数指针必须通过一个间接层分配到正确的模块中。 管理这种间接可能会产生一些开销；为了避免这种情况，您可以获得直接转到相应驱动程序的特定于设备的功能指针。
要获取设备级功能指针，请调用 vkGetDeviceProcAddr（），其原型是
PFN_vkVoidFunction vkGetDeviceProcAddr (VkDevice device, const char* pName);
将函数指针指向的设备传递到设备中。再次，您正在查询的函数的名称作为 nul 终止的 UTF-8 字符串在 pName 中传递。 所产生的函数指针仅在设备中指定的设备有效。 设备必须参考设备
它支持提供新功能的扩展，扩展已被启用。
由 vkGetDeviceProcAddr（）生成的函数指针特定于设备。 即使使用相同的物理设备创建具有完全相同参数的两个或多个逻辑设备，您必须使用结果函数指针与其查询的设备。
彻底关闭
在您的程序退出之前，您应该自己清理。在许多情况下，操作系统将清理您的应用程序终止时分配的资源。但是，代码结束并不总是这样，应用程序结束。如果您正在编写较大应用程序的组件，例如，该应用程序可能会终止使用 Vulkan 的呈现或计算操作，而不会实际退出。清理时，一般做法如下：
•在与 Vulkan 相关的所有线程中，完成或以其他方式终止您的应用程序在主机和设备上进行的所有工作。
•按照与创建顺序相反的顺序销毁对象。
逻辑设备可能是您在应用程序初始化期间创建的最后一个对象（除了运行时使用的对象）。在销毁设备之前，您应该确保它不会代表您的应用程序执行任何工作。要做到这一点，调用 vkDeviceWaitIdle（），其原型是
VkResult vkDeviceWaitIdle (VkDevice device);
设备的手柄通过设备传递。 当 vkDeviceWaitIdle（）返回时，代表您的应用程序提交给设备的所有工作都将保证已经完成 - 除非您在此期间向设备提交更多的工作。您应该确保可能

正在提交到设备的任何其他线程已被终止。
一旦确保设备空闲，您可以安全地销毁它。 要做到这一点，调用 vkDestroyDevice（），其原型是
void vkDestroyDevice (VkDevice device, const VkAllocationCallbacks* pAllocator);
要销毁的设备的句柄在设备参数中传递，必须外部访问其访问。请注意，对于任何其他命令，对设备的访问不需要是外部同步的。然而，应用程序应确保在其他任何访问它的命令仍在另一个线程中执行时，设备不会被销毁。
pAllocator 应指向与用于创建设备的分配结构兼容的分配结构。一旦设备对象被破坏，就不能再提交任何命令了。此外，不再可能使用设备句柄作为任何功能的参数，包括
将设备句柄作为其第一个参数的其他对象销毁功能。这是为什么你应该从创建顺序的相反顺序销毁对象的另一个原因。
一旦与 Vulkan 实例关联的所有设备都已被销毁，可以安全地销毁该实例。这是通过调用其原型的 vkDestroyInstance（）函数来实现的
void vkDestroyInstance (VkInstance instance, const VkAllocationCallbacks* pAllocator);
要破坏的实例的句柄被实例传递，与 vkDestroyDevice（）一样，指向与分配实例的分配结构兼容的指针应在 pAllocator 中传递。如果 pAllocator 参数为 vkCreateInstance()为 nullptr，那么 pAllocator 参数 vkDestroyInstance（）也应该是。
请注意，没有必要销毁物理设备。 物理设备不是通过专用的创建功能创建的，因为逻辑设备是。 相反，物理设备从调用返回到 vkEnumeratePhysicalDevices（），并被认为是由实例拥有的。 因此，当实例被销毁时，该实例与每个物理设备相关联的资源也被释放。
概要
本章向您介绍了 Vulkan。 你已经看到一个实例中包含了全部的 Vulkan 状态。 该实例提供对物理设备的访问，每个物理设备暴露了可用于执行工作的多个队列。 您已经看到如何创建与物理设备相对应的逻辑设备。 您已经了解了如何扩展 Vulkan，如何确定实例和设备可用的扩展，以及如何启用这些扩展。 您已经看到如何通过等待设备完成应用程序提交的工作，破坏设备句柄，最终摧毁实例句柄来干净地关闭 Vulkan 系统。

Vulkan 编程指南翻译 第二章 第一节 CPU 内存管理
翻译 2017 年 02 月 17 日 21:06:56 24401

你将在本章中学到：
l Vulkan 如何管理主机和设备内存
l 在应用程序中如何有效地管理内存
l Vulkan 如何使用 images 和 buffers 消费和生产数据
memory 是是几乎所有计算机系统做任何操作的基础，也包括 Vulkan。在 Vulkan 里，memory 基本上有两种类型：主机 memory 和设备 memory。Vulkan 操作的所有资源必须被设备内存支持，应用程序需要负责管理内存。此外，内存也被用作主机上存储数据。Vulkan 提供了让应用程序管理内存的机会。在本章中，您将学到 Vulkan 用来管理内存的各种机制。
主机内存管理
每当 Vulkan 创建新对象时，它可能需要内存来存储与它们相关的数据。为此，它使用主机内存，可以被 CPU 访问，是通过 malloc 或者 new 调用返回的通常意义的内存。然而，除了正常的分配器，Vulkan 有一些特殊的内存分配需求。其中最常见的，是它希望分配的内存被对齐。这是因为一些高性能 CPU 指令在对齐的内存上工作的最好。若存储 CPU 端的数据被对齐，Vulkan 可以无条件的使用这些高性能指令，提供实质性能优势。

由于上述要求，Vulkan 实现将使用高级分配器。但是，为了某些，甚至全部操作，它还为您的应用程序提供了替换默认分配器的机会。这是通过指定多数的设备创建函数的 pAllocator 参数来实现的。例如，让我们重新看一遍 vkCreateInstance() 函数，它可能是你的应用程序第一个调用的函数。原型如下：

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

pAllocator 参数是一个指向 VkAllocationCallbacks 类型数据的指针。直到目前，我们一直设置 pAllocator 为 nullptr，这告诉 Vulkan 去使用它内部提哦那个的默认内存分配器，而不是应用程序提供的内存分配器。VkAllocationCallbacks 书籍结构封装了我们提供的自定义内存分配器。这个数据结构定义如下：

```
typedef struct VkAllocationCallbacks {
void* pUserData;
PFN_vkAllocationFunction pfnAllocation;
PFN_vkReallocationFunction pfnReallocation;
PFN_vkFreeFunction pfnFree;
PFN_vkInternalAllocationNotification pfnInternalAllocation;
PFN_vkInternalFreeNotification pfnInternalFree;
} VkAllocationCallbacks;
```

你可以通过 VkAllocationCallbacks 的定义知道，它基本上是一些函数指针的集合和一个 void* pUserData。这个指针可以供应用程序使用。它可以指向任何位置。Vulkan 不会 dereference 它。事实上，它甚至不需要是一个指针。你可以放任何东西在哪儿，只要它放入一个指针占用的内存区。Vulkan 对 pUserData 做的唯一的事情，就是将它传递到包含 VkAllocationCallback 指针的回调函数。

pfnAllocation，pfnReallocation 和 pfnFree 用于通常的、对象级的内存管理。它们被定义为指向与以下声明匹配的函数的指针：

```
        void* VKAPI_CALL Allocation(
void* pUserData,
size_t size,
size_t alignment,
VkSystemAllocationScope allocationScope);
void* VKAPI_CALL Reallocation(
void* pUserData,
void* pOriginal
size_t size,
size_t alignment,
VkSystemAllocationScope allocationScope);
void VKAPI_CALL Free(
void* pUserData,
void* pMemory);
```

注意，这三个函数用一个 pUserData 作为第一个参数，这是和 VkAllocationCallbacks 数据结构的 pUserData 是同一东西。如果你的应用程序使用数据结构来管理内存，这是放置他们的地址的好地方。用一个 C ++类实现你的内存分配器（假设你

在使用 C++），并且把这个类的 this 指针放到 pUserData，是一个合理的方式。

Allocation 函数负责新的内存分配。size 参数指定了分配多少 byte。Alignment 参数指定了安几个 byte 进行要求的内存对齐，这是一个经常被忽视的参数。非常容易和原生的内存分配器 malloc 挂钩联系起来。如果这么做，你将会发现程序会正常运行一段时间，但是在某个函数中神奇的崩溃。如果你提供自己的 allocator,，你需要重视 alignment 参数。

最后一个参数 allocationScope,，告诉应用程序，内存分配的范围，生命周期是什么样的。它是 VkSystemAllocationScope 值中的某一个，

l VK_SYSTEM_ALLOCATION_SCOPE_COMMAND 意味着分配的内存将只生存于调用 Allocation 的 demand 中。因为只在一个 command 内生存，Vulkan 有可能使用它做作为临时的内存分配。

l VK_SYSTEM_ALLOCATION_SCOPE_OBJECT 内存分配和一个特定的 Vulkan 对象关联。在对象被销毁之前，分配的内存一直存在。这种类型的内存分配只发生在 command 创建（所有以 vkCreate 开头的函数）期间。

l VK_SYSTEM_ALLOCATION_SCOPE_CACHE 意味着分配的内存和内部缓存或者 VkPipelineCache 对象关联。

l VK_SYSTEM_ALLOCATION_SCOPE_DEVICE 意味着分配的内存在整个 devcie 中都有效。当 Vulkan 需要和 GPU 关联的内存，而不是和一个对象关联。比如，该内存分配器在某些 blocks 中分配内存，有很多对象或许都在这个 block 内，那么，分配的内存就不能和任何特定对象直接关联。

l VK_SYSTEM_ALLOCATION_SCOPE_INSTANCE 意味着分配的内存在一个 instance 内有效。这个与 VK_SYSTEM_ALLOCATION_SCOPE_DEVICE.相似。这种类型的内存分配通常是 layer 或者 Vulkan 启动时做的，比如 vkCreateInstance()、vkEnumeratePhysicalDevices()。

pfnInternalAllocation 和 pfnInternalFree 函数指针指向 Vulkan 使用自带分配器时的 备用的函数。他们和 pfnAllocation、pfnInternalFree 的函数签名相同，除了 pfnInternalAllocation 不返回值，且 pfnInternalFree 不应该真的释放内存。这些函数仅仅用来通知应用程序管理好 Vulkan 在使用的内存。这些函数原型如下：

```
void VKAPI_CALL InternalAllocationNotification(
void* pUserData,
size_t                                  size,
VkInternalAllocationType        allocationType,
VkSystemAllocationScope         allocationScope);

void VKAPI_CALL InternalFreeNotification(
void*                                   pUserData,
size_t                                  size,
VkInternalAllocationType        allocationType,
VkSystemAllocationScope         allocationScope);
```

对于 pfnInternalAllocation 和 pfnInternalFree 提供的信息，你并不能做什么，除了做日志和跟踪应用程序的内存使用量。这些函数指针是可选的，但如果你指定了一个，另外一个也需要指定。如果你不想用，把他们都设置为 nullptr 即可。

Listing2.1 展示了一个如何写 C++ class 作为和 Vulkan 内存分配回调函数匹配的 allocator 的例子。因为这些回调函数被 Vulkan 通过 C 函数指针调用，所以这些回调函数被声明为该 class 静态成员函数，然而，真的实现函数被声明为非静态成员函数。

```
        class allocator
{
public:
// Operator that allows an instance of this class to be used as a
// VkAllocationCallbacks structure
inline operator VkAllocationCallbacks() const
{
VkAllocationCallbacks result;
result.pUserData = (void*)this;
result.pfnAllocation = &Allocation;
result.pfnReallocation = &Reallocation;
result.pfnFree = &Free;
result.pfnInternalAllocation = nullptr;
result.pfnInternalFree = nullptr;
return result;
};
private:
// Declare the allocator callbacks as static member functions.
static void* VKAPI_CALL Allocation(
void* pUserData,
size_t size,
size_t alignment,
VkSystemAllocationScope allocationScope);
static void* VKAPI_CALL Reallocation(
void* pUserData,
void* pOriginal,
size_t size,
size_t alignment,
VkSystemAllocationScope allocationScope);
static void VKAPI_CALL Free(
void* pUserData,
void* pMemory);
// Now declare the nonstatic member functions that will actually
perform
// the allocations.
void* Allocation(
size_t size,
size_t alignment,
VkSystemAllocationScope allocationScope);
void* Reallocation(
void* pOriginal,
size_t size,
size_t alignment,
VkSystemAllocationScope allocationScope);
```

```
void Free(
void* pMemory);
};
```

在 Listing2.2 中展示了该类的实现。它把 Vulkan 的内存分配函数映射到符合 POSIX 标准的 aligned_malloc 函数。注意，这个 allocator 几乎不会比 Vulkan 内部大多的默认分配器好，这只是作为一个用自己的代码写回调函数的例子。

```
void* allocator::Allocation(
size_t size,
size_t alignment,
VkSystemAllocationScope allocationScope)
{
return aligned_malloc(size, alignment);
}

void* VKAPI_CALL allocator::Allocation(
void* pUserData,
size_t size,
size_t alignment,
VkSystemAllocationScope allocationScope)
{
return static_cast<allocator*>(pUserData)->Allocation(size,
alignment,
allocationScope);
}

void* allocator::Reallocation(
void* pOriginal,
size_t size,
size_t alignment,
VkSystemAllocationScope allocationScope)
{
return aligned_realloc(pOriginal, size, alignment);
}
void* VKAPI_CALL allocator::Reallocation(
void* pUserData,
void* pOriginal,
size_t size,
size_t alignment,
VkSystemAllocationScope allocationScope)
{
return static_cast<allocator*>(pUserData)->Reallocation(pOriginal,
size,
alignment,
allocationScope);
```

```
}

void allocator::Free(
void* pMemory)
{
aligned_free(pMemory);
}

void VKAPI_CALL allocator::Free(
void* pUserData,
void* pMemory)
{
return static_cast<allocator*>(pUserData)->Free(pMemory);
}
```

在 Listing2.2 中我们可以看到，静态成员函数内部，可以简单的把 pUserData 参数静态类型转换回该类的一个实例对象，并调用非静态成员函数。因为非静态和静态函数在同一个编译单元内，非静态函数很有可能被内联了，以致这种实现是很高效的。

Vulkan 编程指南翻译 第二章 第二节 资源

翻译 2017 年 02 月 20 日 22:46:44

- 315
- 0
- 1

第二章　内存与资源　第二节　资源

Vulkan 操纵数组。与之相比，其他东西重要性皆次之。数据被存储在 resources 中，resource 存放在内存硬件中。Vulkan 有两种基础的资源：buffers　与　images。Buffer 是一块儿简单的、连续的数据，可以用来存储任何东西—数据结构，原生数组，甚至图像数据，你应当选择 buffer。另一方面，Images，是结构化的，拥有类型信息，可以是多维的，自己也可组成数组，可以支持高级的读写操作。

两种类型的 resource 都是通过两个步骤完成构造的：一，创建 resource，然后放在内存中。这么做的原因是允许应用程序自己来管理内存。内存管理比较复杂，由驱动来做就会比较困难，也许一个程序中工作正常，在其他的程序中就不工作。因此，应该是由应用程序来做内存管理，而不是驱动。例如，一个应用程序可以使用数量很少但数据量很大的资源，使用单独的内存分配器的管理策略，让它们常驻内存，然而，其他的程序可能需要不断的创建并销毁小数据量的资源。

即使 images 数据结构比较复杂，它被创建的过程和 buffes 类似。本节将先讲解 buffer 的创建，后讲解 images。

Buffres

Buffers 是 Vulkan 中最简单但使用非常广泛的资源类型。它通常被用来存储连续的结构化的或非结构化的数据，在内存中可以有格式或者原生的 bytes。当我们将讨论到这些话题时，就会降到 buffers 对象的各种使用方式。创建 buffer 对象，需调用 vkCreateBuffer()，原型如下：

```
VkResult  vkCreateBuffer  (
        VkDevice   device,
        const  VkBufferCreateInfo*   pCreateInfo,
        const  VkAllocationCallbacks*  pAllocator,
        VkBuffer*   pBuffer);
```

如同 Vulkan 中大多数函数，它有许多个参数，这些参数被打包进一个数据结构中，通过指针传到 Vulkan。这里，pCreateInfo 参数是指向一个 instance 的 VkBufferCreateInfo 成员的指针，它的定义如下：

```
typedef  struct  VkBufferCreateInfo  {
        VkStructureType   sType;
        const  void*   pNext;
        VkBufferCreateFlags   flags;
        VkDeviceSize   size;
        VkBufferUsageFlags   usage;
        VkSharingMode   sharingMode;
        uint32_t   queueFamilyIndexCount;
        const  uint32_t* pQueueFamilyIndices;
}  VkBufferCreateInfo;
```

sType 应当被设置为 VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO，pNext 应被设置为 nullptr—除非你想要使用拓展。flags 告诉 Vulkan 有关 Buffer 的属性信息。在当前的 Vulkan 版本中，只能设置为 sparse buffes 相关的值，我们将在本章稍后部分讲解。暂时，flags 被设置为 0。

size 设定了 buffer 的大小，以 bytes 为单位。usage 告诉 Vulkan 你如何使用 buffer，它只能被设置为 VkBufferUsageFlagBits 这个枚举类型的某一个值。在某些架构中，buffer 的使用方式会影响到被创建的过程。本章中用到的设定值有下面几个：

- VK_BUFFER_USAGE_TRANSFER_SRC_BIT 、VK_BUFFER_USAGE_TRANSFER_DST_BIT 表示在转移 commands 过程中，可以被用来做源地址或者目的地。转移 operatin 会把数据从源地址 copy 到目标地址。第四章将会讲解。
- VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT、VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT 表示 buffer 可被用来存储 uniform 或者 storage texel。Texel buffer 是存放像素的格式化的数组，可以被用作源地址或者目标地址，被运行在 GPU 上的 shader 读写。Texel Buffer 将在第六章讲解。
- VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT 、 VK_BUFFER_USAGE_STORAGE_BUFFER_BIT 表示可以存储 uniform 或者 storage buffer。和 texel buffers 相反，常规的 uniform 和 storage buffers 并没有格式，可用来存储任意的数据。我们在第六章“Shaders and Pipelines“中讲解。

1 VK_BUFFER_USAGE_INDEX_BUFFER_BIT 、 VK_BUFFER_USAGE_VERTEX_BUFFER_BIT 可以用来存放索引或者顶点数据，被绘制 commands 使用。在第八章”绘制“中，我们讲解绘制命令，包含索引化的绘制命令。

1 VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT 表示可以存储间接派发与绘制命令的参数，这些 commands 从 buffers 中直接获取参数，而不是从应用程序。在第六章和第八章会讲解。

VkBufferCreateInfo 的 sharingMode 域，表示 buffer 将在 GPU 支持的多个 command queues 中如何被使用。因为 Vulkan 并行的执行多个 operation，一些 Vulkan 实现需要知道 buffer 是被一个还是被多个 command 使用。设置 sharingMode 为 VK_SHARING_MODE_EXCLUSIVE 时，表明 buffer 只会被一个 queue 使用，设置为 VK_SHARING_MODE_CONCURRENT 时表示你计划在多个 queue 中同时使用这个 buffer。使用 VK_SHARING_MODE_CONCURRENT 可能会导致在一些系统上低效率，所以，除非你的确需要才设置为这个值。

如果你真的设置为 VK_SHARING_MODE_CONCURRENT，你需要告诉 Vulkan 哪些 queue 将使用这个 buffer。这通过设置 VkBufferCreateInfo,的 pQueueFamilyIndices 域来完成，这是一个指向 queue families 数组的指针。queueFamilyIndexCount 是数组的长度，是将使用这个 buffer 的 queue families 的个数。当 sharingMode 被设置为 VK_SHARING_MODE_EXCLUSIVE 时，queueFamilyCount、pQueueFamilies 都会被忽略。

Listing 2.3 示例了如何创建一个 1MB 大小的 buffer 对象，可读可写，还有每次只被一个 queue family 使用。

Listing 2.3: Creating a Buffer Object

```
static const VkBufferCreateInfo bufferCreateInfo =
    {
        VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO, nullptr,
        0,
        1024 * 1024,
        VK_BUFFER_USAGE_TRANSFER_SRC_BIT | VK_BUFFER_USAGE_TRANSFER_
        DST_BIT,
        VK_SHARING_MODE_EXCLUSIVE,
        0, nullptr
    };VkBuffer buffer = VK_NULL_HANDLE;
    vkCreateBuffer(device, &bufferCreateInfo, &buffer);
```

在 Listing 2.3 中的代码被运行后，一个新的 VkBuffer 的 handle 被创建，并且被赋值到 buffer 变量。这个 buffer 并不能被使用，因为，他首先需要被存放到内存，这个操作将在本章的”设备内存管理“小节中讲到。

格式 和 Support？

Buffer 是相对简单的资源类型，存放的数据没有格式的概念，images 和 buffer views 包含有他们内容的相关信息。部分信息描述了资源中数据的格式。一些格式对特定的管线中使用它们有特殊的要求或者限制。例如，一个格式可读但是不可写，这在压缩格式中很常见。

为了知道对各种格式的属性和 supoort 级别，你可以调用 vkGetPhysicalDeviceFormatProperties()，原型如下：

```
void  vkGetPhysicalDeviceFormatProperties  (
      VkPhysicalDevice   physicalDevice,
      VkFormat   format,
      VkFormatProperties*   pFormatProperties);
```

因为对物理设备而非逻辑设备特定格式的支持，物理设备的 handle 是被 physicalDevice.指定的。如果应用程序需要完全支持某些格式，你可以在创建逻辑设备之前做检查，或者在应用程序启动时拒绝特定的物理设备。如果设备识别格式，它将把支持级别写到 instance 的 VkFormatProperties 类型的域 pFormatProperties.。VkFormatProperties 的定义如下：

```
typedef  struct  VkFormatProperties  {
      VkFormatFeatureFlags   linearTilingFeatures;
      VkFormatFeatureFlags   optimalTilingFeatures;
      VkFormatFeatureFlags   bufferFeatures;
}  VkFormatProperties;
```

VkFormatProperties 的三个域都是位域，可选值由 VkFormatFeatureFlagBits 这个枚举中的某些值构成。一幅图像可以是单列或者是块状模式：线性的，就是在内存中连续的排列，先按行，再按照列排放；或者最优，图像数据以最优方案被排放，可以最高效率的利用显卡内存子系统。linearTilingFeatures 域表示对一种格式线性铺排的支持级别，optimalTilingFeatures 表示对图像优化铺排方式的支持级别，bufferFeatures 表示对 buffer 使用的支持级别。

可选的 bit 值定义如下：

- VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT 这种格式被 shader 采样的只读图像使用
- VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT 过滤模式，包含线性过滤，这种格式用于被采用的 image
- VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT: 这种格式被 shader 可读写的 images 使用
- VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT: 这种格式被 支持 shader 的 atomic 操作的可读写 images 使用。
- VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT: 这种格式被 shader 只读的 texel buffer 使用
- VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT:此格式 被 shader 读写的 texel buffer 使用
- VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT:此格式被 支持 shader 原子操作的 texel buffers 使用
- VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT:此格式被 管线的顶点组装阶段用作源地址的顶点数据使用
- VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT:此格式被管线的颜色混合阶段用作颜色附件使用
- VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT:当 blending 被启用时，有此格式的 images 可被用作颜色附件

l VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT:此格式可被深度、挡板，或者深度-挡板 附件使用

l VK_FORMAT_FEATURE_BLIT_SRC_BIT: 此格式被用作 image 拷贝操作的源数据使用

l VK_FORMAT_FEATURE_BLIT_DST_BIT: 此格式被用作 image 拷贝操作的目的数据使用

许多格式是几种状态位的组合。实际上，许多格式被强制支持。在 Vulkan 技术规范文档中，有一个完整的“必须支持“格式列表。如果一种格式在此列表中，那么它不需要被强制去测试是否支持。然而，完了完整性，各种 Vulkan 实现被期待会准确的向 Vulkan 委员会汇报对各种格式，甚至是必须支持格式的兼容性。

vkGetPhysicalDeviceFormatProperties()函数会返回一个粗糙的结果集，告诉我们一个格式是否可被用在特定的场景。对 images 来说，一个特定格式和对 images 不同支持级别之间的相互影响会复杂多了。因此，当用于 images 时，为了更多的获取对格式的支持信息，你可以调用 vkGetPhysicalDeviceImageFormatProperties()，原型如下：

```
VkResult vkGetPhysicalDeviceImageFormatProperties (
        VkPhysicalDevice  physicalDevice,
        VkFormat  format,
        VkImageType  type,
        VkImageTiling  tiling,
        VkImageUsageFlags  usage,
        VkImageCreateFlags  flags,
        VkImageFormatProperties*  pImageFormatProperties);
```

与 vkGetPhysicalDeviceFormatProperties(),类似，vkGetPhysicalDeviceImageFormatProperties()接受一个 VkPhysicalDevice handle 作为第一个参数，反馈物理设备而非逻辑设备对某个格式的支持结果。你查询的这个格式通过 format 参数传递。

你想询问的 image 的类型通过 type 指定。它应当是 images 类型中的某一个：VK_IMAGE_TYPE_1D，VK_IMAGE_TYPE_2D，VK_IMAGE_TYPE_3D。不同的 image 类型也许有不同的限制条件和增强措施。Image 的 Tiling 模式是通过 tiling 参数指定的，值为 VK_IMAGE_TILING_LINEAR 或者 VK_IMAGE_TILING_OPTIMAL，表示线性或者最优铺排。

Image 的类型是通过 usage 参数指定的。这个位域表明 image 就如何被使用。如何使用 images 将在本章稍后讲到。flags 参数应当被设置为和创建 image 时同样。

如果这个格式被识别并且被 Vulkan 的实现所支持，那么支持级别将会写入到一个 VkImageFormatProperties 类型的数据结构 pImageFormatProperties. VkImageFormatProperties 的定义如下：

```
typedef  struct  VkImageFormatProperties  {
        VkExtent3D maxExtent;
        uint32_t  maxMipLevels;
        uint32_t  maxArrayLayers;
        VkSampleCountFlags  sampleCounts;
```

```
        VkDeviceSize    maxResourceSize;
}  VkImageFormatProperties;
```
VkImageFormatProperties 的成员 extent 报告了某个格式的 image 在创建时最大尺寸的大小。例如，每个像素占位（bit）更少的格式比占位更多的格式，可以创建更大的 image。Extent 是 VkExtent3D 类型的数据，定义如下：
```
typedef  struct  VkExtent3D  {
        uint32_t   width;
        uint32_t   height;
        uint32_t   depth;
        }  VkExtent3D;
```
maxMipLevels 域，报告了一个 指定格式的 image 可以支持的最大 mipmap 个数。多数情况下，maxMipLevels 要么报告 log2 (max (extent.x, extent.y, extent.z))，要么 1 表示不支持 mipmap。

maxArrayLayers 域报告了 image 支持的 array layers 最大数量。再次强调，如果 arrays 被支持，或不支持时为 1，这几可能会是一个非常大的数字。

如果 image 格式支持多重采样，那么支持的采样数通过 sampleCounts 获得，这是 yige 位字段，每一位都表示支持的采样数量。如果 n 被设置，那么支持 2^n 采样的 images 就能够被这种格式支持。如果格式完全支持，这个位字段至少有一个位是被设置过的。几乎不可能看到支持多重采样但不支持单采样的格式。

最后，maxResourceSize 域，指定了这个格式的资源，以 byte 为单位最大的大小。此域不应当和 maxExtent 混淆了。maxExtent 表示了支持的每一个维度的最大值。比如，某个 Vulkan 实现表示可以支持 16,384 × 16,384 pixels × 2,048 layers 的 image，每个像素 128 bits。那么，以每个维度的最大值来穿件图像，将会产生 8TB 的数据。该 Vulkan 实现几乎不会真的实现支持创建 8TB 的图像。然而，它可能支持创建 n 8 × 8 × 2,048 array or a 16,384 × 16,284 nonarray 图像，两者都可以放到合适的内存块中。

Images

Images 比 buffers 更加复杂，因为他们是多维的；有独特的布局和格式信息；即可被用作过滤、混合、深度或者 stencil 测试等操作的源地址 或者是目的地址。可以用 vkCreateImage()函数创建 images，原型如下：
```
VkResult  vkCreateImage  (
        VkDevice   device,
        const  VkImageCreateInfo*   pCreateInfo,
        const  VkAllocationCallbacks*  pAllocator,
VkImage*  pImage);
```
被用来创建 image 的的设备，通过 device 参数被传入。Image 相关的信息通过一个数据结构 pCreateInfo 被传入。它是一个 VkImageCreateInfo 类型 数据的指针，定义如下：
```
typedef  struct  VkImageCreateInfo  {
        VkStructureType   sType;
```

```
    const  void*   pNext;
    VkImageCreateFlags   flags;
    VkImageType   imageType;
    VkFormat   format;
    VkExtent3D   extent;
    uint32_t   mipLevels;
    uint32_t   arrayLayers;
    VkSampleCountFlagBits   samples;
    VkImageTiling   tiling;
    VkImageUsageFlags   usage;
    VkSharingMode   sharingMode;
    uint32_t   queueFamilyIndexCount;
    const  uint32_t*   pQueueFamilyIndices;
    VkImageLayout   initialLayout;
}  VkImageCreateInfo;
```

你可以看到，这是一个比 VkBufferCreateInfo 显著复杂的数据结构。常见的域，sType、 pNext，在最前面，与其他的多数 Vulkan 内部数据结构类似。sType 域应当被设置为 VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO。flags 域包含描述 image 的属性信息。可以从 VkImageCreateFlagBits 这个枚举中选择几个值来构造。前面三个-

- VK_IMAGE_CREATE_SPARSE_BINDING_BIT, VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, and VK_IMAGE_CREATE_SPARSE_ALIASED_BIT -是用来控制稀疏图像的，在本章稍后讲解。

如果被设置为 VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT，那么你可以为 image 创建具有不同格式 view。Image views（image 视图）基本上是一种特殊的图像，它可以和父图像共享数据和布局，但是可以表现出不同的格式。这就允许了 image 的数据在同时被不同的方式解读。使用图像视图，就可以使用一份数据创建多个 iamge。本章稍后讲解。如果被设置为 VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT，那么你可以创建立方纹理视图。立方纹理视图于本章稍后讲解。

imageType 域指定了你想创建的图像的类型。图像类型基本上就是图像的维度，可选值有 VK_IMAGE_TYPE_1D, VK_IMAGE_TYPE_2D, VK_IMAGE_TYPE_3D，分别表示 1D，2D，3D。

图像也有格式，它描述了像素数据是如何在内存在存放的，并且是如何被 Vulkan 读取的。图像的格式是通过 formats 指定的，而且必须是由 VkFormat 这个枚举的值组成的。Vulkan 支持很大数量的格式—在这里无法列出。我们在本书中使用某些作为例子并加以讲解。剩下的格式，请参考 Vulkan 文档。

图像的 extent 是指以像素为单位的大小。通过 extent 域指定，它是 VkExtent3D 类型的数据，有 width, height, depth 三个成员。他们应当被设置为目标图像的宽度，高度和深度。对于 1D 图像，高度是 1，对于 1D、2D 图像，depth 需被设置为 1。相较于把下一维度当作数组的大小，Vulkan 显式的指定 数组大小，通过 arrayLayers.指定。

能够创建的图像的大小依赖于每个 GPU 设备。想要获取这个最大数值，可调用 vkGetPhysicalDeviceFeatures()并且检查内嵌的 VkPhysicalDeviceLimits 结

构
的 maxImageDimension1D, maxImageDimension2D, 和 maxImageDimension 3D。maxImageDimension1D 包含一维图像的最大宽度，maxImageDimension2D、maxImageDimension3D 都包含最后一维的最大值。同样的，maxImageArrayLayers 包含了 image 可拥有的层数的最大值。
如果这是一张立方纹理，那么最后一位的最大值就被存储在 maxImageDimensionCube。
maxImageDimension1D, maxImageDimension2D, 和 maxImageDimensionCube 都被保证不小于 4096 像素，且 maxImageDimensionCube 和 maxImageArrayLayers 被保证不小于 256。如果想要创建的图像比这些维数的最大值小，那么就无需检查硬件的特征。进一步来讲，Vulkan 实现一般都会支持远高于最低标准的规格。所以，我们可以合理的假定可以创建更大的图像，而不是为了适配低端设备做容错处理。
mipLevels.指定了 mipmap 的层数。Mipmapping 是为了提高降采样的质量而预先提供的一组被过滤过的图片。这些图片以金字塔的形式组成了 mipmap 的各个层级。

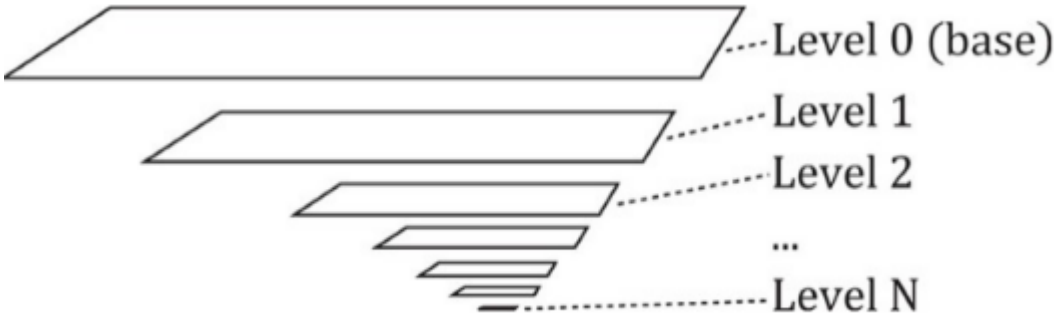

Figure 2.1: Mipmap Image Layout

在一个 mipmapped 贴图中，基础级别是最小号码的级别（通常是 0），并且拥有贴图的原装尺寸。后继的级别依次有上一层的一半大小，直到某个维度的尺寸变为一个像素。从 mipmaped 的贴图进行采用将在第六章讲解。
同样的，samples 指定了采样的次数。这个域与其他的不大相似。它必须是 VkSampleCountFlagBits 这个枚举中的某个值，是一个 bit 定义、用在位域的变量。然而，在现在的 Vulkan 中，只有 2 的幂采样数才能被使用，意味着它们只用 1 表示含义，所以，按位的枚举值就可以工作了。
余下的几个域描述了图像就会被如何使用。首先是 tiling 模式，通过 tiling 域指定。这是一个 VkImageTiling 枚举类型的变量，只有 VK_IMAGE_TILING_LINEAR、 VK_IMAGE_TILING_OPTIMAL 这两个选项。
Linear tiling 表示图像数据从左到右，从上到下被存放，如果你映射到底层内存或者通过 CPU 写入，他将形成线性的图像。同时，优化 tiling 是不透明的表示方式，由 Vulkan 在内存中存放，使 GPU 的内存子系统更高效的访问。这是你多数时候的选项，除非你打算用 CPU 来操作数据。对于绝大多数操作，优化 tiling 会比线性 tiling 表现的显著高效，而且线性 tiling 也不被一些操作和格式所支持，这取决于 Vulkan 的具体实现者。
usage 也是位域变量，描述了图像在哪儿被使用。这和 VkBufferCreateInfo 的 usage 类似。这里的 usage 由 VkImageUsageFlags 枚举值形成，如下：

l VK_IMAGE_USAGE_TRANSFER_SRC_BIT 、 VK_IMAGE_USAGE_TRANSFER_DST_BIT 表示图像图像操作的源地址和目标地址。图像转移命令将在第四章讲解。

l VK_IMAGE_USAGE_SAMPLED_BIT 表示图像可以被 shader 采样

l VK_IMAGE_USAGE_STORAGE_BIT 表示图像可以被用作通用存储，包括 shader 的写入

l VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT 表示图像可以绑定一个颜色附件并且用绘制操作卷入。Framebuffers 和它们的附件在第七章绘制管线中讲解。

l VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT 表示图像可以绑定一个深度或者 stencil 附件，且用作深度或者 stencil 测试。深度或者 stencil 操作在第十章“像素处理“中讲解。

l VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT 表示图像可以被用作临时附件，是一种特殊的图像，被用于存储绘制操作的中间结果。临时附件在第十三章”多遍渲染“中讲解。

l VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT 表示图像可被用作渲染过程中的特殊输入。输入图像和常规菜呀或存储的图像不一样的点在于 fragment shader 可以读取他们的像素点。输入输入附件在第十三章。

sharingMode 与本章早一些提到的 VkBufferCreateInfo 结构的同名成员，在功能上是相同的。若被设置为 VK_SHARING_MODE_EXCLUSIVE，这个图像将在某个时刻只能被一个队列使用。若设置为 VK_SHARING_MODE_CONCURRENT,，那么图像可以同时被多个 queue 访问到。同样，当 sharingMode 被设置为 VK_SHARING_MODE_CONCURRENT. 时，queueFamilyIndexCount 和 pQueueFamilyIndices 与上面小节的功能描述相同。

最后，图像有布局（layout），在某种程度上指定了在任意时刻它将会被如何使用。initialLayout 域决定了图像以哪种布局被创建。VkImageLayout 这个枚举类型定义了可用的布局方式，他们如下：

l VK_IMAGE_LAYOUT_UNDEFINED:

l VK_IMAGE_LAYOUT_GENERAL:

l VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL:

l VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL:

l VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL:

l VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL:

l VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL:

l VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL:

l VK_IMAGE_LAYOUT_PREINITIALIZED:

l VK_IMAGE_LAYOUT_PRESENT_SRC_KHR:

图像可从一个布局转移到另外一个，我们将会在相关章节讲到不同的布局。然而，图像初始创建时，只能是 VK_IMAGE_LAYOUT_UNDEFINED 或者 VK_IMAGE_LAYOUT_PREINITIALIZED 布局。当你在内存中有数据并且迅速绑定到图像资源时，才使用 VK_IMAGE_LAYOUT_PREINITIALIZED。当你在使用前，计划把资源转移到另外一个布局，应当使用 VK_IMAGE_LAYOUT_UNDEFINED。任何时候，当图像被移出 VK_IMAGE_LAYOUT_UNDEFINED layout 时，几乎没有性能消耗。

改变图像布局的机制也被称为”管线壁垒“，或”壁垒“。一个壁垒，不仅仅是改变资源布局的途径，也可以用来同步 Vulkan 管线不同阶段，甚至在一个 GPU 设备上同时运行的队列，对该资源的访问。由此，管线壁垒相当的复杂，正确的使用它并不简单。管线壁垒在第四章深度讲解。
Listing 2.4 展示了一个简单的图像资源创建的例子
Listing 2.4: Creating an Image Object

```
VkImage image = VK_NULL_HANDLE;
VkResult result = VK_SUCCESS;

static const VkImageCreateInfo imageCreateInfo =
{
    VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO,   // sType
    nullptr, // pNext
    0,   // flags
    VK_IMAGE_TYPE_2D, // imageType
    VK_FORMAT_R8G8B8A8_UNORM,   // format
    { 1024, 1024, 1 },   // extent
    10,   // mipLevels
    1,   // arrayLayers
    VK_SAMPLE_COUNT_1_BIT, // samples
    VK_IMAGE_TILING_OPTIMAL, // tiling
    VK_IMAGE_USAGE_SAMPLED_BIT,   // usage
    VK_SHARING_MODE_EXCLUSIVE,   // sharingMode
      0,   // queueFamilyIndexCount
      nullptr,   // pQueueFamilyIndices
      VK_IMAGE_LAYOUT_UNDEFINED   // initialLayout
};
result = vkCreateImage(device, &imageCreateInfo, nullptr, &image
);
```

在 Listing 2.4 中创建的图像是一个 1,024 × 1,024 2D 图像，单采样，VK_FORMAT_R8G8B8A8_UNORM 格式，最优 tiling。代码以未定义布局创建了它，这表示我们可以把它转移到另外一个地方兵器用数据填充它。这个图像将会被作为贴图使用被 shader 访问，所以，我们设置 usage 为 VK_IMAGE_USAGE_SAMPLED_BIT。在我们简单的应用程序中，我们只是用单一队列，所以，我们设置共享模式为 exclusive.

线性图像

如之前章节讨论过的，各种资源都有两种 tiling 模式：VK_IMAGE_TILING_LINEAR and VK_IMAGE_TILING_OPTIMAL。
VK_IMAGE_TILING_OPTIMAL 是一种不透明，实现方式各异的布局，是为了提高设备内存子系统对图像的读写性能。然而，VK_IMAGE_TILING_LINEAR 是一种不可见（transparent）的数据布局方式，是为了足够的直观，在图像内部，像素以从左到右，从上倒下的方式布局，几乎不可能把数据映射到资源并让 CPU 直接的读写。

如果想让 CPU 可以访问潜在的图像数据，除了图像的宽度，高度，深度，像素格式，还有几个其他的信息是必需的：图像的 row pitch--； array pitch - 不同 array layer 之间的距离； depth pitch - 深度切片间的距离。当然，array pitch 和 depth pitch 分别应对于 array 或 3D 图像，row pitch 只可用于 2D 和 3D 图像。
一个图像通常是由几个子资源组成的。一些格式有多于一个的 aspect，是类似于深度图像的深度或者 stencil 成分的一种成分。一个 image 下不同的子资源的布局可能是不同的，因此有不同的布局信息。这个信息可调用 vkGetImageSubresourceLayout()来查询，原型如下：

```
void vkGetImageSubresourceLayout (
        VkDevice device,
        VkImage image,
        const VkImageSubresource* pSubresource,
    VkSubresourceLayout* pLayout);
```

被查询图像所在的设备通过 device 参数传递过去，查询的图像通过 image 参数传递。子资源的信息通过一个 VkImageSubresource 类型的指针 pSubresource，传递回来，定义如下：

```
typedef struct VkImageSubresource {
        VkImageAspectFlags aspectMask;
        uint32_t mipLevel;
        uint32_t arrayLayer;
} VkImageSubresource;
```

Vulkan 编程指南翻译 第二章 第三节 GPU 设备内存管理

翻译 2017 年 02 月 20 日 22:48:09

- 189
- 0
- 2

当 Vulkan 操纵数据，数据必须存储在设备内存。这是 GPU 设备可以访问的内存。Vulkan 系统有四个级别的内存。某些系统或许只有其中的一个或几个。给定一个 CPU（应用程序运行的处理器）设备和 GPU 设备（执行 Vulkan 命令的处理器），他们都有各自的物理存储器。另外，一个处理器附带的物理存储器的某部分区域可以被另外一个处理器访问到。
某些情况下，共享内存的可见区域可能会相当的小，其他情况下，也许只有一块儿物理存储器，被 host 和 GPU 共享。Figure2.5 示例了 CPU 和 GPU 各自的物理存储器的内存映射。

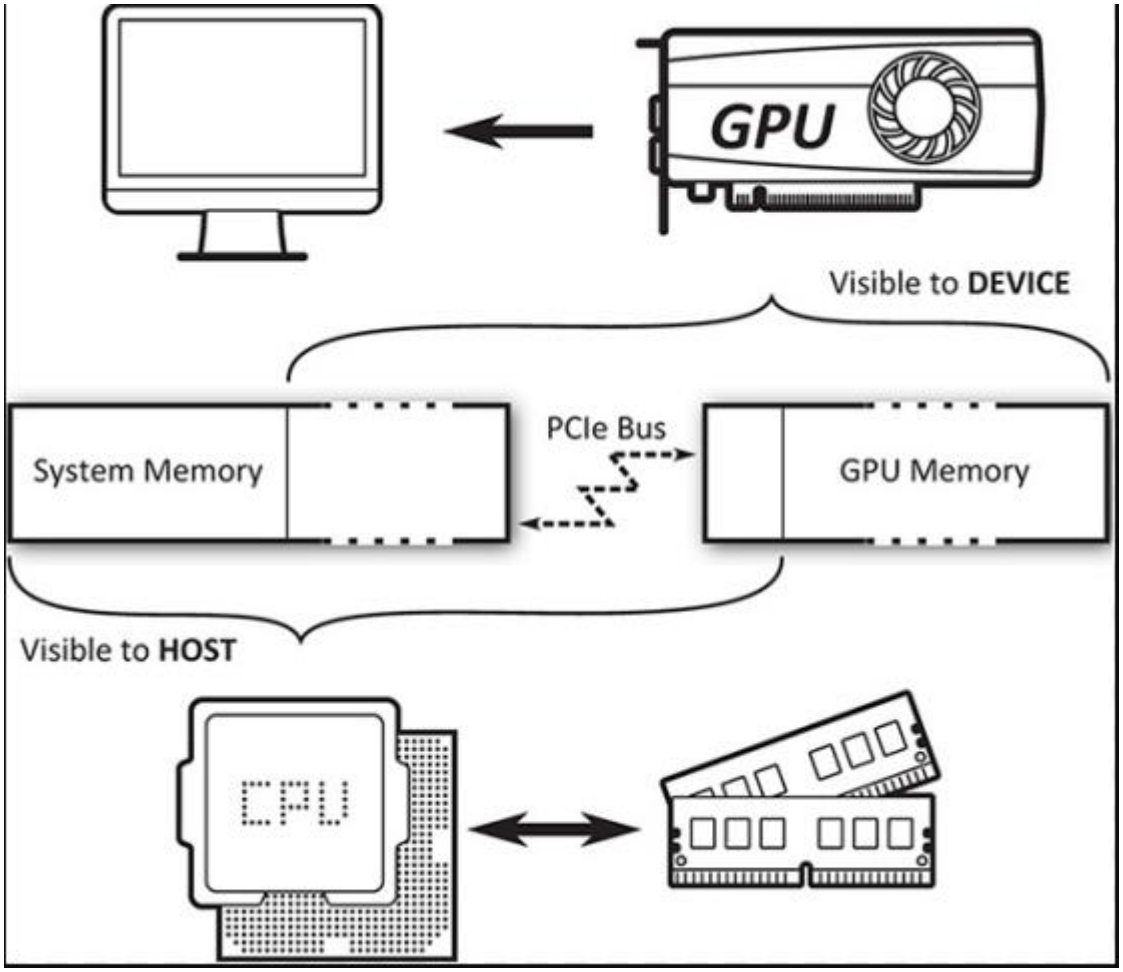

Figure 2.5: Host and Device Memory

可以被 GPU 访问的内存被称为 device memory，即使这些内存是物理连接在 CPU 端。在这种情况下，它是主机端的设备内存。这与主机端内存有区别，主机内存又被称为系统内存，是可以通过 malloc 和 new 操作获取到的普通内存。设备内粗也可以通过映射被 CPU 访问到。

一个典型的单个 GPU 通常是插在 PCI-Express 插槽的插卡中的，它有一定容量的专用内存，是通过电路直接附着在电路板上的。这个存储器的一部分只可被 GPU 访问到，一部分可以被 CPU 访问到。另外，GPU 可以访问到一些甚至全部的主机系统内存。这些内存池对 CPU 来说就是堆，。。。

另外一方面，一个典型的嵌入式 GPU——比如说那些嵌入式系统，手机，甚至笔记本电脑中可以找到——会与 CPU 共享存储器控制器和子系统。这种情况下，很有可能对主机内存的访问时一致性的，且 GPU 会暴露少一些--甚至一个内存堆。这就是通常所谓的“统一内存架构”。

分配 GPU 内存

GPU 上分配的任何内存都通过 VkDeviceMemory 类型的数据表示，它通过 vkAllocateMemory()产生，原型如下：

```
VkResult  vkAllocateMemory  (
    VkDevice  device,
    const  VkMemoryAllocateInfo*  pAllocateInfo,
```

```
    const  VkAllocationCallbacks*  pAllocator,
    VkDeviceMemory*  pMemory);
```
device 参数指定了从哪个 GPU 分配内存。pAllocateInfo 描述了新分配的内存对象，如分配成功，pMemory.将指向新分配的内存。pAllocateInfo 指向一个 VkMemoryAllocateInfo 类型的数据结构，其原型如下：
```
typedef  struct  VkMemoryAllocateInfo  {
    VkStructureType   sType;
    const  void*   pNext;
    VkDeviceSize   allocationSize;
    uint32_t   memoryTypeIndex;
}  VkMemoryAllocateInfo;
```
这个数据结构很简单，进包含内存的大小，类型。sType 应当被设置为 VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO，除非是用了需要获知更多内存分配信息的拓展，pNext 应当被设置为 nullptr。allocationSize 指定了需要分配的内存的大小，以 byte 为单位。内存的类型通过 memoryTypeIndex 指定，可调用 vkGetPhysicalDeviceMemoryProperties()用 memoryTypeIndex,从内存类型的数组查询，可以得到内存类型，我们在第一章中讲述过。
一旦你完成了 GPU 内存分配，它就可以用来存储 buffer、image 等资源。Vulkan 也许会把内存用作其他用途，比如其他类型的设备对象，内部分配或者数据结构，片段存储，诸如此类。这些分配活动由 Vulkan 驱动管理，不同 Vulkan 实现之间的差别可能会较大。
当你不再使用这些内存时，你需要释放它们。可以调用 vkFreeMemory(),原型如下：
```
void  vkFreeMemory  (
    VkDevice  device,
    VkDeviceMemory  memory,
    const  VkAllocationCallbacks*  pAllocator);
```
vkFreeMemory()需要直接传入 memory 对象。你需要保证在释放之前，在设备上没有 queue 正在使用该内存。Vulkan 将不会跟踪 memory 对象的使用情况。如果设备试图访问已经被释放的内存，结果是不可知的，这也许会轻易的导致应用程序崩溃。
深入来说，对内存的访问必须要严格保持同步。当一块内存被其他线程的 command 访问时，尝试释放它将产生不可知的结果切易导致程序崩溃。
在某些平台上，也许有单线程能够内存分配的次数的上线。如果你尝试分配更多，将导致分配失败。这个上限可以通过调用 vkGetPhysicalDeviceProperties()函数并检查返回的 maxMemoryAllocationCount 类型对象的 maxMemoryAllocationCount 成员便可获知。Vulkan 标准保证的最小值是 4096，一些平台或许高得多。即使这个值看起来很小，它被设定为如此的意图就是让你单次尽量分配大的内存块，然后，从这个大的内存块再分配小的内存块，在单次分配的内存中，尽量多放入资源。只要内存允许，资源创建的数量是没有上限的。
平常情况下，当你从堆中分配内存时，分配到的内存被 赋值给被返回的 VkDeviceMemory 类型的一个对象，它需要调用 vkFreeMemory() 被销毁。在一

些情况下，你（或者 Vulkan 实现）并不知道对于某些操作来说需要多少内存，或者是究竟是否需要内存。
特别是，在渲染时被需要用到的中间临时数据来自于图像。当图像被创建时，如果 VkImageCreateInfo 类型的数据包含 VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT，Vulkan 就知道这个图像的数据生存周期很短，一次，它很大可能从来被会被写入设备内存。
在这种情况下，你可要求 Vulkan 内存分配时导游 lazy 参数，并把分配活动推后到 Vulkan 决定真的需要使用物理存储空间的时候。若有这种需求，需要报内存类型设置为 VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT。选择其他的类型也是可以让程序正常工作的，但是，它总是事先就分配好了内存，不管你使用与否。
如果你想要知道一次内存分配究竟有多少已经真的在物理内存中分配了，和多少备用内存已经为一个内存对象分配了，你可以调用 vkGetDeviceMemoryCommitment(),原型如下：

```
void  vkGetDeviceMemoryCommitment  (
    VkDevice  device,
    VkDeviceMemory  memory,
    VkDeviceSize*  pCommittedMemoryInBytes);
```

内存所在的设备通过 device 参数传入，需要查询的内存块通过 memory 参数传入。pCommittedMemoryInBytes 是一个指向了有多少内存真正被分配了的变量的指针，指向的值被该函数修改。这个数值的大小就是从堆上计算得来的，带有内存类型信息。
对于那些不包含 K_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT 内存类型的内存对象，或者内存被完全提交的内存对象，vkGetDeviceMemoryCommitment()将只返回整个内存对象的大小。vkGetDeviceMemoryCommitment()返回的值的大小只能做参考用。很多时候，这个信息是过时的，而且你对这个值也无能为力。

CPU 访问设备内存

如前面小节所述，设备内存被分为几个区域。纯设备内存只能被设备访问。然而，有几个区域是可以被 host 和 device 都能访问的。Host 就是主应用程序所运行的处理器，且我们可以让 Vulkan 返回一个指向 CPU 可访问区域内存的指针。这叫做内存映射。
为了把设备内存映射为主机端内存地址，需要被映射的内存对象必须要从 VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT 类型的堆中分配获得。假设我们需要这么做，映射内存来获取一个 CPU 可用的指针是通过调用 vkMapMemory() 来实现的，原型如下：

```
VkResult  vkMapMemory  (
    VkDevice   device,
    VkDeviceMemory   memory,
    VkDeviceSize   offset,
    VkDeviceSize   size,
    VkMemoryMapFlags   flags,
    void**   ppData);
```

被映射的内存所在的设备是通过 device 参数传递的，内存对象的 handle 是通过 memory 参数传递。一定要在外部同步的访问这个内存对象。若需要映射一个内存对象的一部分，需要通过 offset 参数确定起始位置和 size 参数来制定区域的大小。如果你想要映射整个内存对象，直接设置 offset 为 0，size 为 VK_WHOLE_SIZE 即可。设置 offset 为非零值且 size 为 VK_WHOLE_SIZE 将会从内存对象的 offset 位置到结束做映射。offset、size 的单位都是 byte。你不应当尝试把一个内存对象映射到一个较小的区域。flags 参数是为以后保留的，当前应当设置为 0。

如果 vkMapMemory()调用成功，一个指向映射区域的指针就被写入 ppData。在应用程序中，这个指针可以被转换为任意的类型，并且被 dereference 后可以直接读写。Vulkan 保证，当指针减去 offset 时，vkMapMemory()返回的指针是对齐为设备内存对齐最小值的倍数。

这个值是通过调用 vkGetPhysicalDeviceProperties()函数返回的 VkPhysicalDeviceLimits 类型的数据的 minMemoryMapAlignment 成员获取的。它肯定至少是 64byte 的，但是，但可能是高于此的任何 2 的幂的值。在一些 CPU 架构中，可以通过让数据对齐和指令对齐得到更高的运行效率。为达到此目的，minMemoryMapAlignment 经常和 cache line size 匹配，或者和机器的最大寄存器自然对齐。如果被传入未对齐的指针，一些主机 CPU 指令会出错。一次，你可以检查 minMemoryMapAlignment，决定使用优化过的函数或者使用可处理未对齐指针的默认函数，只是这样就要承受性能损失。

当你用映射指针做完了工作，它需要被解除映射，需调用 vkUnmapMemory()，原型如下：

```
void vkUnmapMemory (
    VkDevice device,
    VkDeviceMemory memory);
```

内存所在的设备通过 device 参数传入，需要被解除映射的内存对象通过 meory 参数传入。和 vkMapMemory()一样，对内存对象的访问需要在外部同步访问。对一个相同的内存对象，不能做多次映射。也就是，你不能对一个内存对象使用不同的参数多次调用 vkMapMemory()来做内存映射，不管这些内存是否有重叠部分。在取消映射的时候，范围是不需要的，因为 Vulkan 知道映射的范围有多大。

一旦内存对象被取消了内存映射，任何指向以前通过调用 vkMapMemory()获取的指针是无效的，不应当被使用。即使你把内存对象以相同的参数范围映射，你也不能假设会得到相同的指针。

当设备内存被映射到主机内存地址空间时，就有两个地方可以修改该内存了。对于该映射内存，在 CPU 和 GPU 端都很有可能有 cache 层次的存在。这两个地方的 cache 也许不能保持一致性。为了保证 CPU 和 GPU 能看到一致性的数据视图，即使被另外一边写入了也无所谓。那么就有很有必要强制 Vulkan 把 cache 写入内存。

设备支持的每一种内存类型都有一些属性，其中的一个可能是 VK_MEMORY_PROPERTY_HOST_COHERENT_BIT。如果有一个映射的内存被设置为这种属性，Vulkan 就会保证 cache 之间的一致性。在某些情况下，cache 之间会自动一致，因为它们是被 CPU 和 GPU 共享的，或者是有一种形式的一致性协议

来保持他们之间的同步。在其他情况下，Vulkan 驱动可能会推断出什么时候 cache 需要被刷新或者被失效，进而在幕后进行上述操作。
如果一块映射内存区域的属性之一没有被设置为 VK_MEMORY_PROPERTY_HOST_COHERENT_BIT，那么你就需要显式地刷新 cache 或者使 cache 无效。为了刷新带有 pending 状态写操作的 cache，需要调用 vkFlushMappedMemoryRanges(),原型如下：

```
VkResult vkFlushMappedMemoryRanges (
    VkDevice device,
    uint32_t memoryRangeCount,
    const VkMappedMemoryRange* pMemoryRanges);
```

拥有内存对象的设备是通过 device 参数指定的。需要刷新的区域大小是通过 memoryRangeCount 参数指定的，每一个范围的信息都是封装在 VkMappedMemoryRange 类型的数据被传入的。一个指向 memoryRangeCount 的数组的指针是通过 pMemoryRanges 参数传入的。VkMappedMemoryRange 的定义如下：

```
typedef struct VkMappedMemoryRange {
    VkStructureType sType;
    const void* pNext;
    VkDeviceMemory memory;
    VkDeviceSize offset;
    VkDeviceSize size;
} VkMappedMemoryRange;
```

VkMappedMemoryRange 的 sType 域应当被设置为 VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE，pNext 应当被设置为 nullptr。每一个内存范围都指向了一个内映射的内存对象，是通过 memory 域指定的，范围是通过 offset 和 size 计算得来的。你不需要刷新那个对象的整个映射区域，所以 offset 和 size 不需要和 vkMapMemory()的参数匹配。如果内存对象没有被映射，或者 offset 和 size 确定的一个对象的区域没有被映射，那么刷新操作就不会有任何作用。为了刷新一个内存对象的任何映射区，只要把 offset 设置为 0，size 设置为 VK_WHOLE_SIZE 即可。
如果 CPU 向映射内存区域写入了数据而且需要设备看到写人的效果，刷新是必须的，然而，如果设备写入内存映射区域且你需要 CPU 能够看到写入的信息，你需要在 CPU 端主动的使任何 cache 无效，因为这些信息可能是没有更新的。你需要调用 vkInvalidateMappedMemoryRanges()，原型如下：

```
VkResult vkInvalidateMappedMemoryRanges (
    VkDevice device,
    uint32_t memoryRangeCount,
    const VkMappedMemoryRange* pMemoryRanges);
```

对 vkFlushMappedMemoryRanges()来说，device 参数就是拥有内存对象并

Vulkan 编程指南翻译 第三章 队列和命令 第 1 节 设备队列

翻译 2017 年 02 月 20 日 22:53:23

- 111
- 0
- 1

第三章　队列和命令　第 3 节　设备队列

你将在本章中学到

●·队列是什么，如何使用它

●·如何创建命令并把它们发送给 Vulkan

●·如何保证设备已完成任务

Vulkan 设备对外暴露多个队列来完成任务。在本章，我们讨论多种队列类型并详解如何以命令 buffer 的形式向他们提交任务。我们也演示如何指示一个队列完成你发送给它的任务。

设备队列

Vulkan 中每一个队列都有一个或多个队列。队列是设备中真正执行工作的。它可以被认为是对外暴露了设备功能的子设备。在一些实现中，每一个独立饿甚至是系统内物理性分离。

多个设备被分组到一个或者多个队列集中，一个队列集包含一个或者多个队列。在一个集中的多个队列基本上是相同的。他们的能力、性能级别、对资源的访问也是相同的，除了同步的时候，相互之间转移工作也是没有损耗的。如果设备哟多个具有相同能力但性能不同的核心对内存的访问或其他会导致表现不同的因素，这会让他们不同的集明显的表现各异。

如在第一章讨论过的，你可以调用 vkGetPhysicalDeviceQueueFamilyProperties()查询每一个物理设备队列集的属性。这个函数向一个你给定的 VkQueueFamilyProperties 类型的数据写入队列集的属性。

当你创建设备的时候，队列数量和类题型能够必须要确定。如你在第一章所见，你传入 vkCreateDevice()的 VkDeviceCreateInfo 结构，包含 queueCreateInfoCount 和 pQueueCreateInfos 这两个成员。第一章简介了他们，现在我们来做详细的讲解。

queueCreateInfoCount 成员包含了一个由 pQueueCreateInfos 指针指向的数组中 VkDeviceQueueCreateInfo 类型对象的数量。VkDeviceQueueCreateInfo 结构如下：

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

如多数 Vulkan 数据结构一样，sType 域是数据的类型，在此种情况下是 VK_STRUCTURE_TYPE_QUEUE_CREATE_INFO，pNext 域被拓展使用，当没有拓展时应当设置为 nullptr。flags 域包含对队列的控制标记信息，但是，当前的 Vulkan 并没有定义任何标志，所以此成员应当被设置为 0。

我们感兴趣的域是 queueFamilyIndex 和 queueCount。queueFamilyIndex 域指定了从哪个集中分配队，queueCount 域指定了需分配的队列的个数。从多个集合中分配队列时，只需向 VkDeviceCreateInfo 类型对象的 pQueueCreateInfos 成员，传入多于一个 VkDeviceQueueCreateInfo 类型对象的数组即可。

当设备被创建的时候队列就被构造了。所以，我们无需创建队列，但是，为了获取他们，我们需要调用 vkGetDeviceQueue():

```
void vkGetDeviceQueue (
        VkDevice device,
        uint32_t queueFamilyIndex,
        uint32_t queueIndex,
        VkQueue* pQueue);
```

这个函数以你需要从哪个设备获取队列的 device，集的索引 queueFamilyIndex，集合中队列的索引 queueIndex 作为参数。pQueue 参数指向了一个 VkQueue 类型 handle，就是你想要的队列的 handle。queueFamilyIndex、queueIndex 必须参考 device 被创建时的参数。如果正确的这么做了，一个正确的队列 的 handle 将会被赋值到 pQueue，否则，pQueue 的值就会是 VK_NULL_HANDLE。

创建命令 buffer

队列的意义就是在应用程序内处理任务。任务是通过一串的命令表示的，命令被记录到命令缓冲区（command buffer）中。你的应用将会创建包含任务的命令缓冲区进而提交到队列来执行。在你记录任何命令之前，你需要创建命令缓冲区。命令缓冲区并不被直接创建，需要从 pool 中分配。你可以调用 vkCreateCommandPool() 函数来创建 pool，其原型如下：

```
VkResult vkCreateCommandPool (
        VkDevice   device,
        const VkCommandPoolCreateInfo*   pCreateInfo,
        const VkAllocationCallbacks*   pAllocator,
        VkCommandPool*   pCommandPool);
```

和 Vulkan 中绝大多数对象创建一样，它的第一个参数 device，就是拥有该 pool 的设备，对该 pool 的描述信息是通过一个指向某个数据结构的指针 pCreateInfo 来传递的。这个数据结构就是 VkCommandPoolCreateInfo，定义如下：

```
typedef struct VkCommandPoolCreateInfo {
        VkStructureType   sType;
        const void*   pNext;
        VkCommandPoolCreateFlags   flags;
```

```
    uint32_t    queueFamilyIndex;
} VkCommandPoolCreateInfo;
```

与大多数 Vulkan 数据结构类似，前两个域是 sType 和 pNext，前一包含了数据结构的类型，后一个是一指向包含更多关于 pool 创建信息的数据。这里，我们设置 sType 为 K_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO。我们不需要传递任何其他信息，pNext 需被设置为 nullptr。

flags 域包含标志，它控制了 pool 和从 pool 分配得倒的命令缓冲区的行为。其值为 VkCommandPoolCreateFlagBits 枚举类型。这个枚举现在有两个值被定义了：

- VK_COMMAND_POOL_CREATE_TRANSIENT_BIT 表示命令缓冲区使用周期短，使用完后马上退回给 pool。不设置这个枚举值，就意味着告诉 Vulkan，你将长时间的持有这个命令缓冲区。
- VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT 表示允许单独的命令缓冲区可通过重置或重启而被重用。如果没有这个 bit 标志，那么只有 pool 本身能够被重置，这隐式的回收所有由它分配出的命令缓冲区。

每一个 bit 标志位都会增加一些开销，因为 Vulkan 实现需要跟踪资源或者改变其自身的分配策略。例如，设置 VK_COMMAND_POOL_CREATE_TRANSIENT_BIT 这个标志位会导致 Vulkan 实现跟踪每一个命令缓冲区的重置状态而不是简单的只是在 pool 内跟踪。在这种情形下，我们实际上需要把两个标志位都设置上。这将给我们极大的灵活性，可能在一些性能损失的条件下，我们可以批量的管理命令缓冲区。

最后，VkCommandPoolCreateInfo 的 queueFamilyIndex 域指定了从这个 pool 分配的命令缓冲区需要提交的队列的所属的集合。这是必需的，因为，一个设备上拥有相同性能并具有相同命令的两个队列，发送相同的命令到他们，表现也会各异。

pAllocator 参数是应用程序用来管理主机内存的，这部分在第二章有讲解。假设一个命令池已被成功的创建，它的 handle 被赋值给 pCommandPool，vkCreateCommandPool()就会返回 VK_SUCCESS。

一旦我们有了一个可以分配命令缓冲区的 pool，我们可以通过调用 vkAllocateCommandBuffers()得到新的命令缓冲区，其原型如下：

```
VkResult vkAllocateCommandBuffers (
        VkDevice   device,
        const VkCommandBufferAllocateInfo*   pAllocateInfo,
        VkCommandBuffer*   pCommandBuffers);
```

分配额命令缓冲区的设备通过 device 参数传递，剩余的参数描述了命令缓冲区，是通过一个 VkCommandBufferAllocateInfo 类型的数据传递的，缓冲区的地址是通过 pCommandBuffers 参数传递的。VkCommandBufferAllocateInfo 的原型如下：

```
typedef struct VkCommandBufferAllocateInfo {
        VkStructureType   sType;
        const void*   pNext;
        VkCommandPool   commandPool;
        VkCommandBufferLevel   level;
        uint32_t   commandBufferCount;
} VkCommandBufferAllocateInfo;
```

sType 域应当被赋值为 VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO，当我们仅仅使用核心功能集的时候，需要设置 pNext 为 nullptr。我们在之前创建的命令池被放入了 commandPool 参数。level 参数表示我们需要分配的命令缓冲区的级别。它可以被赋值为 VK_COMMAND_BUFFER_LEVEL_PRIMARY 或者 VK_COMMAND_BUFFER_LEVEL_SECONDARY.。Vulkan 允许主命令缓冲区来调用次命令缓冲区。在我们最初的几个例子中，我们将只使用主级别的命令缓冲区。将在本书的后面讲到次级命令缓冲区。

最后，commandBufferCount 指定了我们想要从 pool 中分配的命令缓冲区的数量。注意，我们并没有告诉 Vulkan 任何关于我们需要的命令缓冲区的大小和个数。表示设备命令的内

部内部数据将自由的变动以适应任意尺寸大小的 command。Vulkan 会替你管理好命令缓冲区的内存。

如果 vkAllocateCommandBuffers()运行成功，它会返回 VK_SUCCESS，并且把分配好的多个命令缓冲区的 handle 放到 pCommandBuffers 这个数组中。这个数组应当足够的大以容纳这些 handle。当然，如果你想仅仅分配一个命令缓冲区，你可以只声明一个普通的 VkCommandBuffer 类型的变量即可。

需要使用 vkFreeCommandBuffers()来释放命令缓冲区，其声明如下：

```
void vkFreeCommandBuffers (
    VkDevice   device,
    VkCommandPool   commandPool,
    uint32_t   commandBufferCount,
    const VkCommandBuffer*   pCommandBuffers);
```

device 参数是进行命令缓冲区分配所在的设备。commandPool 是 pool 的 handle，commandBufferCount 是需要释放的命令缓冲区的个数，pCommandBuffers 是需要被释放的命令缓冲区的 handle 组成的数组。注意，释放一个命令缓冲区并不意味着需要释放与它相关的资源，只是把它们放回了他们被创建时所在的 pool。

为了释放一个命令池所用的所有资源和它创建的所有命令缓冲区，需要调用 vkDestroyCommandPool()，原型如下：

```
void vkDestroyCommandPool (
        VkDevice   device,
        VkCommandPool   commandPool,
        const VkAllocationCallbacks* pAllocator);
```

拥有命令池的设备是通过 device 参数传入的，需要销毁的命令池通过 commandPool 参数传递。pAllocator 参数应当和 pool 创建时所用的该参数保持一致。如果 vkCreateCommandPool()的 pAllocator 参数是 nullptr，这个函数中的 pAllocator 参数也应该为 nullptr。

在 pool 被销毁之前，没有必要显式的释放所有的从它分配出来的命令缓冲区。当 pool 被销毁时，分配出来的命令缓冲区，作为 pool 的组成部分，会自动的被释放，每个缓冲区相关的资源同样会被释放。然而，这里需要注意，需要保证当 vkDestroyCommandPool()被调用时，从 pool 分配出来的命令缓冲区正在执行。

Vulkan 编程指南翻译 第三章 队列和命令 第 2 节 创建命令缓冲区

翻译 2017 年 02 月 20 日 22:56:54

- 221
- 0
- 1

第三章　队列和命令　第 3 节　创建命令缓冲区

创建命令缓冲区

队列的意义就是在应用程序内处理任务。任务是通过一串的命令表示的，命令被记录到命令缓冲区（command buffer）中。你的应用将会创建包含任务的命令缓冲区进而提交到队列来执行。在你记录任何命令之前，你需要创建命令缓冲区。命令缓冲区并不被直接创建，需要从 pool 中分配。你可以调用 vkCreateCommandPool() 函数来创建 pool，其原型如下：

```
VkResult vkCreateCommandPool (
        VkDevice   device,
        const VkCommandPoolCreateInfo*   pCreateInfo,
        const VkAllocationCallbacks*   pAllocator,
        VkCommandPool*   pCommandPool);
```

和 Vulkan 中绝大多数对象创建一样，它的第一个参数 device，就是拥有该 pool 的设备，对该 pool 的描述信息是通过一个指向某个数据结构的指针 pCreateInfo 来传递的。这个数据结构就是 VkCommandPoolCreateInfo，定义如下：

```
typedef struct VkCommandPoolCreateInfo {
        VkStructureType    sType;
        const void*    pNext;
        VkCommandPoolCreateFlags    flags;
        uint32_t    queueFamilyIndex;
} VkCommandPoolCreateInfo;
```

与大多数 Vulkan 数据结构类似，前两个域是 sType 和 pNext，前一包含了数据结构的类型，后一个是一指向包含更多关于 pool 创建信息的数据。这里，我们设置 sType 为 K_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO。我们不需要传递任何其他信息，pNext 需被设置为 nullptr。

flags 域包含标志，它控制了 pool 和从 pool 分配得倒的命令缓冲区的行为。其值为 VkCommandPoolCreateFlagBits 枚举类型。这个枚举现在有两个值被定义了：

- VK_COMMAND_POOL_CREATE_TRANSIENT_BIT 表示命令缓冲区使用周期短，使用完后马上退回给 pool。不设置这个枚举值，就意味着告诉 Vulkan，你将长时间的持有这个命令缓冲区。
- VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT 表示允许单独的命令缓冲区可通过重置或重启而被重用。如果没有这个 bit 标志，那么只有 pool 本身能够被重置，这隐式的回收所有由它分配出的命令缓冲区。

每一个 bit 标志位都会增加一些开销，因为 Vulkan 实现需要跟踪资源或者改变其自身的分配策略。例如，设置 VK_COMMAND_POOL_CREATE_TRANSIENT_BIT 这个标志位会导致 Vulkan 实现跟踪每一个命令缓冲区的重置状态而不是简单的只是在 pool 内跟踪。在这种情

形下，我们实际上需要把两个标志位都设置上。这将给我们极大的灵活性，可能在一些性能损失的条件下，我们可以批量的管理命令缓冲区。

最后，VkCommandPoolCreateInfo 的 queueFamilyIndex 域指定了从这个 pool 分配的命令缓冲区需要提交的队列的所属的集合。这是必需的，因为，一个设备上拥有相同性能并具有相同命令的两个队列，发送相同的命令到他们，表现也会各异。

pAllocator 参数是应用程序用来管理主机内存的，这部分在第二章有讲解。假设一个命令池已被成功的创建，它的 handle 被赋值给 pCommandPool，vkCreateCommandPool()就会返回 VK_SUCCESS。

一旦我们有了一个可以分配命令缓冲区的 pool，我们可以通过调用

vkAllocateCommandBuffers()得到新的命令缓冲区，其原型如下：

```
VkResult vkAllocateCommandBuffers (
        VkDevice   device,
        const VkCommandBufferAllocateInfo*   pAllocateInfo,
        VkCommandBuffer*   pCommandBuffers);
```

分配额命令缓冲区的设备通过 device 参数传递，剩余的参数描述了命令缓冲区，是通过一个 VkCommandBufferAllocateInfo 类型的数据传递的，缓冲区的地址是通过

pCommandBuffers 参数传递的。VkCommandBufferAllocateInfo 的原型如下：

```
typedef struct VkCommandBufferAllocateInfo {
        VkStructureType   sType;
        const void*   pNext;
        VkCommandPool   commandPool;
        VkCommandBufferLevel   level;
        uint32_t   commandBufferCount;
} VkCommandBufferAllocateInfo;
```

sType 域应当被赋值为

VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO，当我们仅仅使用核心功能集的时候，需要设置 pNext 为 nullptr。我们在之前创建的命令池被放入了 commandPool 参数。

level 参数表示我们需要分配的命令缓冲区的级别。它可以被赋值为

VK_COMMAND_BUFFER_LEVEL_PRIMARY 或者 VK_COMMAND_BUFFER_LEVEL_SECONDARY.。

Vulkan 允许主命令缓冲区来调用次命令缓冲区。在我们最初的几个例子中，我们将只使用主级别的命令缓冲区。将在本书的后面讲到次级命令缓冲区。

最后，commandBufferCount 指定了我们想要从 pool 中分配的命令缓冲区的数量。注意，我们并没有告诉 Vulkan 任何关于我们需要的命令缓冲区的大小和个数。表示设备命令的内部内部数据将自由的变动以适应任意尺寸大小的 command。Vulkan 会替你管理好命令缓冲区的内存。

如果 vkAllocateCommandBuffers()运行成功，它会返回 VK_SUCCESS，并且把分配好的多个命令缓冲区的 handle 放到 pCommandBuffers 这个数组中。这个数组应当足够的大以容纳这些 handle。当然，如果你想仅仅分配一个命令缓冲区，你可以只声明一个普通的 VkCommandBuffer 类型的变量即可。

需要使用 vkFreeCommandBuffers()来释放命令缓冲区，其声明如下：

```
void vkFreeCommandBuffers (
    VkDevice   device,
    VkCommandPool   commandPool,
    uint32_t   commandBufferCount,
    const VkCommandBuffer*   pCommandBuffers);
```

device 参数是进行命令缓冲区分配所在的设备。commandPool 是 pool 的 handle，commandBufferCount 是需要释放的命令缓冲区的个数，pCommandBuffers 是需要被释放的命令缓冲区的 handle 组成的数组。注意，释放一个命令缓冲区并不意味着需要释放与它相关的资源，只是把它们放回了他们被创建时所在的 pool。

为了释放一个命令池所用的所有资源和它创建的所有命令缓冲区，需要调用 vkDestroyCommandPool()，原型如下：

```
void vkDestroyCommandPool (
        VkDevice   device,
        VkCommandPool   commandPool,
        const VkAllocationCallbacks* pAllocator);
```

拥有命令池的设备是通过 device 参数传入的，需要销毁的命令池通过 commandPool 参数传递。pAllocator 参数应当和 pool 创建时所用的该参数保持一致。如果 vkCreateCommandPool()的 pAllocator 参数是 nullptr，这个函数中的 pAllocator 参数也应该为 nullptr。

在 pool 被销毁之前，没有必要显式的释放所有的从它分配出来的命令缓冲区。当 pool 被销毁时，分配出来的命令缓冲区，作为 pool 的组成部分，会自动的被释放，每个缓冲区相关的资源同样会被释放。然而，这里需要注意，需要保证当 vkDestroyCommandPool()被调用时，从 pool 分配出来的命令缓冲区正在执行。

Vulkan 编程指南翻译 第三章 队列和命令 第 3 节 记录命令

翻译 2017 年 02 月 20 日 23:00:21

- 237
- 0
- 1

记录命令

命令是通过使用 Vulkan 命令函数记录到命令缓冲区的，这些函数都接受一个命令缓冲区的 handle 作为第一个参数。对命令缓冲区的访问必须是同步的，意味着应用程序需负责保证没有两个线程同时记录命令到同一个命令缓冲区中。然而，下面的情形是可以接受的：

l 一个线程可以通过调用命令缓冲区函数，依次的记录命令进入到多个命令缓冲区中。

l 两个或多个线程可以参加建立一个命令缓冲区的过程，只要应用程序可以保证它们不同时执行命令缓冲区建立的函数。

Vulkan 的一个关键的设计原则是让程序可以多线程的工作。为此，要保证应用程序的多个线程不会互相堵塞，比如，不会因对共享资源加锁而导致堵塞其他线程。因此，最好是一个线程有一个或者多个命令缓冲区，而不是共享一个。因命令缓冲区是从 pool 中分配而来，你可以进一步的为每一个线程创建命令池，允许多个命令缓冲区都是从自己线程的命令池中获取的，这样就不会有任何冲突了。

在你往一个命令缓冲区录入命令之前，你需要启动命令缓冲区，这将导致它处于初始状态，这需要调用 vkBeginCommandBuffer()，原型如下：

```
VkResult vkBeginCommandBuffer (
VkCommandBuffer   commandBuffer,
const VkCommandBufferBeginInfo*   pBeginInfo);
```

需要开始记录命令的命令缓冲区是通过 commandBuffer 参数传递的，记录命令需要用的参数是通过一个 VkCommandBufferBeginInfo 类型的变量 pBeginInfo，通过指针传递的。VkCommandBufferBeginInfo 的定义如下：

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

VkCommandBufferUsageFlags flags;
const VkCommandBufferInheritanceInfo* pInheritanceInfo;
} VkCommandBufferBeginInfo;
sType 域应当被设置为 VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO，pNext 应被设置为 nullptr。flags 域是用来告诉 Vulkan 命令缓冲区将会被如何使用。它的值是 VkCommandBufferUsageFlagBits 枚举变量某几个值的组合得到的，枚举类型有以下选项：
l VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT 意味着命令缓冲区就会被记录，只会被执行一次，然后被销毁、回收。
l VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT 意味着这个命令缓冲区就会被用在 renderpass 内部，此类型只对次级命令缓冲区有效。如果用于主命令缓冲区，那么它会被无视。Renderpass 和次级命令缓冲区将在第十三章“”讲解。
l VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT 意味着命令缓冲区有可能被多次执行或者暂停
把 flags 设置为 0 是安全的，意味着我们可能多次执行命令缓冲区，但不同时访问也不需要次级命令缓冲区。
pInheritanceInfo 成员是用来开启次级命令缓冲区所需要的，来定义被继承的主命令缓冲区的状态。对于主命令缓冲区，这个指针参数被忽略。我们将在第十三章讲解次级命令缓冲区时介绍 VkCommandBufferInheritanceInfo 类型的结构
现在，我们需要创建首个命令来。在第二章中，已经学到了缓冲，图像，内存。vkCmdCopyBuffer()命令是用来在两个 buffer 对象之间拷贝数据的，原型如下：

void vkCmdCopyBuffer (
VkCommandBuffer commandBuffer,
VkBuffer srcBuffer,
VkBuffer dstBuffer,
uint32_t regionCount,
const VkBufferCopy* pRegions);
这是一个常见形式的 Vulkan 命令。第一个参数 commandBuffer，指定了命令需要追加到的哪个命令缓冲区。srcBuffer 和 dstBuffer 指定了源地址和目的地址。最后，一个 regions 的数组是通过 pRegions 指定的。每一个 region 代表着一个 VkBufferCopy 类型的数据，定义如下：
typedef struct VkBufferCopy {
VkDeviceSize srcOffset;
VkDeviceSize dstOffset;
VkDeviceSize size;
} VkBufferCopy;这个数组的每一个元素都仅包含源和目标的偏移值，和每个 region 从 srcOffset 到 dstOffset 需要拷贝的大小。当命令被执行时，在 pRegions 中的每一个 region，size 大小的数据将会从 srcBuffer 的 srcOffset 拷贝到 dstBuffer 的 dstOffset。这些偏移值都是以 byte 为单位的。
关于 Vulkan 操作的很基础的事实是命令在被调用时并不是被立即执行的。他们仅仅是被添加到命令缓冲区中了。如果你正在从或者到一个 CPU 可访问的内存区域拷贝数据，那么，你需要确认以下几件事：
l 保证在命令被设备执行前，数据一直都在源区域
l 保证在命令在设备上执行后，源区域的数据是有效的

l 保证直到命令在设备上执行命令后，

Vulkan 编程指南翻译 第四章 队列和命令 第 1 节 管理资源的状态

原创 2017 年 02 月 22 日 22:35:26

- 144
- 0
- 2

在本章，你将学到：

l 如何管理资源被 Vulkan 使用时的状态

l 如何在资源间复制数据，用已知数据填充缓冲区和图像

l 如何进行位块操作以拉伸或缩放图像数据

图形和计算操作总体上是数据密集型的。Vulkan 引入了几个对象，可以提供存储和操纵数据的途径。经常需要把数据移入或者转出这些对象，有几个命令可以用来做这个工作：复制数据，填充缓冲区和图像对象。进一步，在任何时刻，一个资源可能出狱多个状态，Vulkan 管线的多个部分可能需要访问他们。本章将讲解数据移动命令—可以用来复制数据与填充内存—当它们被应用程序访问的时候，命令需要用来管理资源的状态。

第三章，展示了命令被放在了一个命令缓冲区中并提交到设备的某一个队列以被执行。这非常重要，因为这表示命令并不是你的应用程序调用它们的时候就被执行的，却是当你把它们提交到队列，队列进入设备的时候被执行的。之前介绍给你的第一个相关函数，vkCmdCopyBuffer()，在两个 buffer 之间或者同一 buffer 不同的区域之间复制数据。这是众多能够改变 buffer，图像，其他的 Vulkan 对象的命令之一。本章将讲解填充、复制、清除 buffer 与图像的相关命令。

4.1 管理资源状态

在一个程序执行中在任何给定的时刻，每一个资源都可以处于一个或者多个不同的状态。例如，如果图形管线正在绘制一个图像或者使用它作为一个贴图，或者 Vulkan 正在从 CPU 端复制数据到一个图像，这些场景都是不同的。对于一些 Vulkan 实现，上述的一些状态之间也许没有任何差别，对于其他的，准确的知道一个时间点资源所处的状态会决定你的应用程序正常的工作或者是做垃圾工作。

因为命令缓冲区内的命令负责访问资源，且多个命令缓冲区被创建的顺序有可能不同于他们被提交执行的顺序，故由 Vulkan 跟踪资源的状态并且保证在每一场景都正确的工作是不现实的。特别是，一个资源会因命令缓冲区的执行从一个状态转为另一个状态。驱动无法跟踪资源的状态，当命令缓冲区被提交执行的时候，在命令缓冲区之间跟踪状态有显著的损耗。因此，这个责任落到了应用程序身上。资源的状态，对图像来说可能非常重要，因为他们是复制、结构化的资源。

一个图像的状态可以粗略的分为两种正交的集合：布局，记录。布局决定了数据在内存中的存放布局，在本书前面讲过。记录，谁最后一次写入信息，将影响 cache 和数据的一致性。图像初始的布局是在创建时被指定的，然后在其生命周期内发生改变，要么显式的使用 barriers 或者隐式的使用 renderpass。Barriers 掌控这 Vulkan 渲染管线的不同部分对资源的访问，在某些情况下，在另一个 midpipeline 同步工作中，barriers 可以一个资源从一个布局转变到另一布局。

每一种布局的准确使用将在本书后面详细讲到。然而，转移数据从一个状态到另一状态的基础操作就是 barrier，且，在应用程序中准确的理解并高效率的使用它，是极其重要的。

4.1.1 管线屏障

屏障是同步机制的一种，被用来管理内存访问和资源的在 Vulkan 管线的一个 stage 内状态迁移。资源访问同步和状态迁移主要的命令是 vkCmdPipelineBarrier()，原型如下：

```
void vkCmdPipelineBarrier (
VkCommandBuffer commandBuffer,
VkPipelineStageFlags srcStageMask,
VkPipelineStageFlags dstStageMask,
VkDependencyFlags dependencyFlags,
uint32_t memoryBarrierCount,
const VkMemoryBarrier* pMemoryBarriers,
uint32_t bufferMemoryBarrierCount,
const VkBufferMemoryBarrier* pBufferMemoryBarriers,
uint32_t imageMemoryBarrierCount,
const VkImageMemoryBarrier* pImageMemoryBarriers);
```

需要执行屏障的命令缓冲区是通过 commandBuffer 指定的。接下来的两个参数，srcStageMask 和 dstStageMask，指定了哪个阶段的管线最后向资源写入和哪个阶段接下来要从资源读数据。亦即，屏障通过它们指定了数据流通的源和目的地。每一个值都是 VkPipelineStageFlagBits 枚举类型的值构造的。

• VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT: The top of pipe is considered to be hit as soon as the device starts processing the command.

• VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT: When the pipeline executes an indirect command, it fetches some of the parameters for the command from memory. This is the stage that fetches those parameters.

• VK_PIPELINE_STAGE_VERTEX_INPUT_BIT: This is the stage where vertex attributes are fetched from their respective buffers. After this, content of vertex buffers can be overwritten, even if the resulting vertex shaders have not yet completed execution.

• VK_PIPELINE_STAGE_VERTEX_SHADER_BIT: This stage is passed when all vertex shader work resulting from a drawing command is completed.

• VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT: This stage is passed when all tessellation control shader invocations produced as the result of a drawing command have completed execution.

• VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT: This stage is passed when all tessellation evaluation shader invocations produced as the result of a drawing command have completed execution.

• VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT: This stage is passed when all geometry shader invocations produced as the result of a drawing command have completed execution.

• VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT: This stage is passed when all fragment shader invocations produced as the result of a drawing command have completed execution. Note that there is no way to know that a primitive has been completely rasterized while the resulting fragment shaders have not yet completed. However, rasterization does not access memory, so no information is lost here.

• VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT: All per-fragment tests that

might occur before the fragment shader is launched have completed.
• VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT: All per-fragment tests that might occur after the fragment shader is executed have completed. Note that outputs to the depth and
stencil attachments happen as part of the test, so this stage and the early fragment test stage include the depth and stencil outputs.
• VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT: Fragments produced by the pipeline have been written to the color attachments.
• VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT: Compute shader invocations produced as the result of a dispatch have completed.
• VK_PIPELINE_STAGE_TRANSFER_BIT: Any pending transfers triggered as a result of calls to vkCmdCopyImage() or vkCmdCopyBuffer(), for example, have completed.
• VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT: All operations considered to be part of the graphics pipeline have completed.
• VK_PIPELINE_STAGE_HOST_BIT: This pipeline stage corresponds to access from the host.
• VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT: When used as a destination, this special flag means that any pipeline stage may access memory. As a source, it's effectively equivalent to VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT.
• VK_PIPELINE_STAGE_ALL_COMMANDS_BIT: This stage is the big hammer. Whenever you just don't know what's going on, use this; it will synchronize everything with everything. Just use it wisely
因为 srcStageMask 和 dstStageMask 内的标志位用来指示事情什么时候发生，Vulkan 实现把它们放在一边或者多种方式解读它们。srcStageMask 指定了什么时候源阶段已经完成了向资源读或写数据。结果，在管线中稍后移动这个阶段有效的位置并不改变访问已经完成的事实。
这意味着实现等待的时间比它需要完成的时间还久。
同样的，dstStageMask 指定了管线在处理之前等着的时间点。如果一个 Vulkan 实现把等待的时间点提前了，也是能够工作的。在逻辑上后面的管线部分开始执行前，被等待的事件也最终会被完成。这种实现仅仅是失去了当它在等待时却能够工作的机会。
dependencyFlags 参数指定了标志位的一个集合，描述了由屏障表示的依赖关系如何影响到屏障引用的资源。唯一被定义的标志类型是 VK_DEPENDENCY_BY_REGION_BIT，它表示屏障只影响被 source stage 改变的区域，此区域被目标阶段所消耗。
vkCmdPipelineBarrier()调用可用来触发多个屏障操作。有三种类型的屏障操作：全局内存屏障，buffer 屏障，图像屏障。全局内存屏障影响到诸如映射内存被 CPU、GPU 同步访问之类的工作。Buffer 和 image 屏障主要影响设备对 buffer 和 images 资源的访问。、

4.1.2 全局内存屏障

vkCmdPipelineBarrier()可触发的全局内存屏障是通过 memoryBarrierCount 参数指定的。如果这是一个非零值，那么 pMemoryBarriers 指向一个大小为 memoryBarrierCount 的 VkMemoryBarrier 类型的数组，每一个都代表一个内存屏障。VkMemoryBarrier 定义如下：

```
typedef struct VkMemoryBarrier {
VkStructureType sType;
const void* pNext;
VkAccessFlags srcAccessMask;
```

VkAccessFlags dstAccessMask;
} VkMemoryBarrier;

这两周以来，我都在加紧速度翻译 Vulkan 编程指南，完成了 1/3 的内容，每天都忙到深夜，中午吃饭的时间都用上了，但是，显然我的速度还不够快。我对自己这种速度不满意，原来以为两三天就能够完成一章的，现在所蚯时间都加倍了。效率低啊。

我想，这种落差感来源于我对自己英语水平的误判。我这几年学习的教材大多是英文版的，读起来没有什么难度，我便以为我的英文和中文一样流利了。还颇以为满意。这里便暴露出了我在有两个问题：

1，我其实读的还不够多，这点阅读量，估计连人家的高中生都比不过，更别说还有四年的大学教育。翻译过程中，查资料才知道我对于一些词汇的理解有偏差，常用的短语不知其意，不能很好的处理长句。这是阅读训练的量还不够。

2，读的快，并不代表我理解的就快。我之前阅读英文教材的速度很快，是以牺牲吸收速度和准确性为代价的。如果记不住，无法很好的在头脑中形成知识图谱。

所以，以后在这方面还是需要锻炼，坚持不懈的训练。这应该是常态，而没有一个终极的目标。

这段时间的翻译也让我有了一个总结，就是程序员如何检验自己的英语阅读水平是否过关呢？判断的标准必须足够简单，可测量。我认为，就是翻译一本你所关注领域的英文书籍，或者同等工作量的官方文档。如果能完成，不管是否痛苦，就表明英文的水平足以胜任工作了。

http://vk-spec.knowthyself.cn/ 这是新的翻译任务，等把 Vulkan 编程指南翻译完成之后，再开始这项工作。也会逐节的发布到 CSDN 上的，让更多的人能够看到。希望大家帮我找错。

Vulkan 编程指南翻译 第四章 队列和命令 第 2 节 清空和填充缓冲区

翻译 2017 年 03 月 01 日 11:59:20

- 166
- 0
- 1

4.2 清空和填充缓冲区

在第二章“内存和资源”中已经介绍过缓冲区对象了。一个缓冲区就是内存中一个连续的区域，包含数据。为了能够使用它，需要向里面填充数据。在某些情况下，清空整个缓冲区并设置为某一个值，是你仅需要做的事。这允许你，比如，初始化一个缓冲区，并最终在 shader 里或在其他的操作里向它写入数据。

向缓冲区填入某个值，需要调用 vkCmdFillBuffer()，其原型如下：

void vkCmdFillBuffer (
VkCommandBuffer commandBuffer,
VkBuffer dstBuffer,
VkDeviceSize dstOffset,
VkDeviceSize size,
uint32_t data);

接受命令的缓冲区是通过 commandBuffer 参数指定。dstBuffer 指定了需要填充数据的缓冲区。需要通过 dstOffset 指定填充操作开始的位置和 size 指定填充区域的大小。dstOffset 和 size 必须是 4 的倍数。从 dstOffset 到 buffer 的尾部，可以指定 size 参数为特殊的值 VK_WHOLE_SIZE。它允许填充整个缓冲区，只需要把 dstOffset 置为 0， size 置为

VK_WHOLE_SIZE。
你想要填充的数据通过 data 参数指定。这是一个 uint32_t 类型的变量，填充操作不断重复填入的值。就好象缓冲区被视为 uint32_t 类型的数组，从 dstOffset 到结束的区域每一个元素都被设置为这个值。为了清除缓冲区内的浮点数，你也可以故意把浮点数的值转换为 uint32_t 在传入到 vkCmdFillBuffer()，示例如下：
Listing 4.2: Filling a Buffer with Floating-Point Data
```
void FillBufferWithFloats(VkCommandBuffercmdBuffer,
VkBuffer dstBuffer,
VkDeviceSize offset,
VkDeviceSize length,
const float value)
{
vkCmdFillBuffer(cmdBuffer,
dstBuffer,
0,
1024,
*(const uint32_t*)&value);
}
```
有些时候，用已知数值填充缓冲区是不够的，还需要更显式的把数据放入缓冲区对象。当大量数据需要被转移进或转移出缓冲区，要么对缓冲区做映射并通过 CPU 写入或者调用 vkCmdCopyBuffer()从其他的缓冲区复制数据更合适一些。然而，对于小量更新，比如更新数组或者小的数据结构的数据，调用 vkCmdUpdateBuffer()来吧数据直接放入缓冲区对象。
vkCmdUpdateBuffer()函数原型如下：
```
void vkCmdUpdateBuffer (
VkCommandBuffer commandBuffer,
VkBuffer dstBuffer,
VkDeviceSize dstOffset,
VkDeviceSize dataSize,
const uint32_t* pData);
```
vkCmdUpdateBuffer()直接从 CPU 内存复制数据到缓冲区对象。一旦 vkCmdUpdateBuffer()调用完成，主机内存就被消耗掉了，可以释放掉或者覆盖掉。注意， vkCmdUpdateBuffer()命令被提交到设备并开始执行，在没有执行完成之前，数据的写入是没有完成的。因此，Vulkan 会复制一份你提供的数据，把它放在命令缓冲区附加的数据结构中或者直接放在命令缓冲区内部。
再有，此命令所提交的命令缓冲区通过 commandBuffer 参数传递，目标缓冲区大小通过 dstBuffer 参数传递。数据开始存放的位置的偏移量通过 dstOffset 参数传递，放到缓冲区的数据的大小通过 dataSize 参数传递。dstOffset 和 dataSize 都是以 byte 为单位，和 vkCmdFillBuffer()一样，都必须是 4 的倍数。vkCmdUpdateBuffer()的 size 参数不接受 VK_WHOLE_SIZE 这个值，因为 size 也饿被用作主机端数据之源的内存区域。vkCmdUpdateBuffer()能放到缓冲区的数据最大为 65,536 bytes。
pDate 指向主机端最终将被放入缓冲区的数据。即使这个参数的类型被期望是 uint32_t 型指针，任何类型数据都可被放入缓冲区。只需要把主机端可读的数据强制转换为 uint32_t*，并传递到 pData 即可。确保数据区域的大小至少是 size byte。比如，构造 C++对象数据以匹配 uniform 布局或 shader 块存储布局，或者复制所有数据后在 shader 中被使用到的缓冲

区 的布局，是合理的选择。
再有，小心使用 vkCmdFillBuffer()。它适合小的，及时的缓冲区更新。比如，向 uniform 缓冲区写入单个值，使用 vkCmdFillBuffer()来做会比调用 vkCmdCopyBuffer()高效率的多。

Vulkan 编程指南翻译 第四章 队列和命令 第 3 节 清空和填充图像
翻译 2017 年 03 月 01 日 12:04:30

- 195
- 0
- 1

4.3 清空和填充图像
和缓冲区一样，也可以把数据直接复制到图像，或使用一个值填充。图像是更大、复杂、不透明的数据结构，所以原生的偏移量和数据通常对应用程序来说是不可见的。[2]
2. Of course, it's possible to map the memory that is used for backing an image. In particular, when linear tiling is used for an image, this is standard practice. However, in general, this is not recommended.
通过 vkCmdClearColorImage()函数调用，可清楚图像数据并填充，原型如下：
void vkCmdClearColorImage (
VkCommandBuffer commandBuffer,
VkImage image,
VkImageLayout imageLayout,
const VkClearColorValue* pColor,
uint32_t rangeCount,
const VkImageSubresourceRange* pRanges);
清除命令被提交到的命令缓冲区通过 commandBuffer 参数指定。需要被清除数据的图像通过 image 参数指定，执行清除操作时图像可选的布局通过 imageLayout 参数指定。
imageLayout 可接受的布局为 e VK_IMAGE_LAYOUT_GENERAL 和 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL。清除不同布局的图像，在执行清除命令之前，有必要使用管线屏障把它们转移到这两个布局中某一个。
清除图像数据并填充的数据是 VkClearColorValue 类型的实例，定义如下：
typedef union VkClearColorValue {
float float32[4];
int32_t int32[4];
uint32_t uint32[4];
} VkClearColorValue;VkClearColorValue 是一个简单的 union 类型，有三个可选值，每个都是含有四个元素的数组。一个是浮点类型，一个是带符号整型，一个是无符号。Vulkan 将会按照图像格式所明确的类型去读数据。应用程序可以可以按照既定数据类型写入数据。vkCmdClearColorImage()不会执行任何的数据转换，需要应用程序负责正确的填充 VkClearColorValue 类型的数据。
单次调用 vkCmdClearColorImage()可以清除目标图像的任意大小的区域，虽然所有被清除的内存都会被填充相同的内容。如果你需要清除多个区域并填充不同的颜色值，你需要多次调用该函数。然而，如果你想要清除所有区域并填充同样颜色，通过 rangeCount 指定区域的数量，传入一个指针 pRanges，指向大小为 rangeCount、类型为 VkImageSubresourceRange 的数组。VkImageSubresourceRange 定义如下：
typedef struct VkImageSubresourceRange {

VkImageAspectFlags aspectMask;
uint32_t baseMipLevel;
uint32_t levelCount;
uint32_t baseArrayLayer;
uint32_t layerCount;
} VkImageSubresourceRange;这个数据结构在第二章“内存和资源”有所介绍，讨论图像视图的创建的小节。这里，它被用来定义图像里你想清空并填充数据的区域。因为我们正在清空颜色图像，aspectMask 须置为 VK_IMAGE_ASPECT_COLOR_BIT。baseMipLevel 和 levelCount 域用来指定 mipmap 起始层和需要被清空数据的层数，如果图像是 array image，baseArrayLayer 和 layerCount 域用来指定起始层和需要被清空的层数。如果图像不是 array image，这些域应各置为 0 和 1。
清空 depth-stencil 图像与清空颜色图像相似，除了一个 VkClearDepthStencilValue 类型的数据用来指定填充的数据。vkCmdClearDepthStencilImage()的原型和 vkCmdClearColorImage()原型类似，其原型如下：
void vkCmdClearDepthStencilImage (
VkCommandBuffer commandBuffer,
VkImage image,
VkImageLayout imageLayout,
const VkClearDepthStencilValue* pDepthStencil,
uint32_t rangeCount,
const VkImageSubresourceRange * pRanges);
同样，将执行清除操作的命令缓冲区通过 commandBuffer 参数指定，需要被清空的图像由 image 参数指定，清空操作执行时期待的图像布局通过 imageLayout 参数指定。和 vkCmdClearColorImage()一样，imageLayout 值为 VK_IMAGE_LAYOUT_GENERAL 或 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL 二者其一。对于清空操作没有其他的有效可选。
清空后填充的数据通过一个 VkClearDepthStencilValue 类型的数据传递，它包含深度和 stencil 数值。它的定义如下：
typedef struct VkClearDepthStencilValue {
float depth;
uint32_t stencil;
} VkClearDepthStencilValue;
和 vkCmdClearColorImage()一样，一次 vkCmdClearDepthStencilImage()调用可以清空多个图像。可以清空的数量通过 rangeCount 指定，pRanges 参数指向了一个大小为 rangeCount、类型为 VkImageSubresourceRange 的数组，数组定义可以清空的范围。
因为 depth-stencil 图像包含深度和 stencil aspect，pRanges 的每一个成员的 aspectMask 都可包含 VK_IMAGE_ASPECT_DEPTH_BIT, VK_IMAGE_ASPECT_STENCIL_BIT 中一个或者两个。如果 aspectMask 包含 VK_IMAGE_ASPECT_DEPTH_BIT，那么存储在 VkClearDepthStencilValue 结构的深度域将会被用来填充深度 aspect 的那块区域。同样，如果 aspectMask 包含 VK_IMAGE_ASPECT_STENCIL_BIT，那么深度 aspect 的那块区域将会被清空并填充 VkClearDepthStencilValue 类型数据 stencil 域的数据。
注意，通常给单个区域指定 VK_IMAGE_ASPECT_DEPTH_BIT 和 VK_IMAGE_ASPECT_STENCIL_BIT 属性，回避单独指定一个属性要高效率的多。

Vulkan 编程指南翻译 第四章 队列和命令 第 4 节 复制图像数据

翻译 2017 年 03 月 01 日 12:13:46

- 145
- 0
- 1

4.4 复制图像数据

在前一节，我们讨论了如何清空图像并通过一个简单的数据结构来填充数据。在很多情况下，你需要把纹理数据上传到图像或者复制图像数据到另一图像。Vulkan 支持从缓冲区向图像复制数据，在图像之间复制数据，和从图像向缓冲区复制数据。

调用 vkCmdCopyBufferToImage()，可从缓冲区向图像的一个或者多个区域复制数据。其原型如下：

```
void vkCmdCopyBufferToImage (
        VkCommandBuffer commandBuffer,
        VkBuffer srcBuffer,
        VkImage dstImage,
        VkImageLayout dstImageLayout,
        uint32_t regionCount,
        const VkBufferImageCopy* pRegions);
```

将要执行复制操作命令的命令缓冲区通过 commandBuffer 参数指定，源缓冲区 srcBuffer 参数指定，数据将要复制到的目标图像由 dstImage 参数指定。和需要清空的目标图像一样，目标图像的布局须

为 VK_IMAGE_LAYOUT_GENERAL 或 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL，通过 dstImageLayout 参数指定。

需要更新的区域数量由 regionCount 参数给定，pRegions 参数是一个指针，指向一个大小为 regionCount、类型为 VkBufferImageCopy 的数组，每一个元素定义图像中需要向其复制数据的一个区域。VkBufferImageCopy 的定义如下：

```
typedef struct VkBufferImageCopy {
        VkDeviceSize bufferOffset;
        uint32_t bufferRowLength;
        uint32_t bufferImageHeight;
        VkImageSubresourceLayers imageSubresource;
        VkOffset3D imageOffset;
        VkExtent3D imageExtent;
    } VkBufferImageCopy;
```

bufferOffset 域包含了缓冲区中数据的偏移量，单位是 byte。缓冲区内的数据从左到右、从上到下排放，如图 4.1 所示。bufferRowLength 域指定了源图像中纹素的个数，bufferImageHeight 图像数据的行数。如果 bufferRowLength 为 0，图像就被认为是在缓冲区中紧致压缩的，因此和 imageExtent.width 相等。同样，如果 bufferImageHeight 为 0，那么源图像的行数就被认为和图像 extend 的高度，亦即 imageExtent.height 相等。

图 4.1 缓冲区中图像的数据排放

需要复制数据到的子资源，是通过一个 VkImageSubresourceLayers 类型的数据来指定的。其原型如下：

```
typedef struct VkImageSubresourceLayers {
    VkImageAspectFlags aspectMask;
    uint32_t mipLevel;
    uint32_t baseArrayLayer;
    uint32_t layerCount;
} VkImageSubresourceLayers;
```

VkImageSubresourceLayers 的 aspectMask 域包含复制数据操作目标图像的 aspect 或多个 aspect。通常，这将会是 VkImageAspectFlagBits 枚举中的某一个。如果目标图像是颜色图像，那么将是 VK_IMAGE_ASPECT_DEPTH_BIT，如果图像是 stencil-only 图像，它将是 VK_IMAGE_ASPECT_STENCIL_BIT。如果图像是复合的 depth-stencil 图像，那么你需要指定 VK_IMAGE_ASPECT_DEPTH_BIT 和 VK_IMAGE_ASPECT_STENCIL_BIT 这两个标志位，以同时复制数据到深度和 stencil aspect。

目标 mipmap 层数通过 mipLevel 参数指定。你可以用 pRegions 数组中每一元素复制到 mipmap 的某一层，即使你可以指定多个元素，每个对应不同的层。

如果目标图像是 array 图像，那么你就可以通过 baseArrayLayer 和 layerCount 指定图像的起始层和层数。如果图像不是 array 图像，那么这两个域就应该分别置为 0 和 1.

每个区域可以对应一整个 mipmap 层或者一层中更小的一块窗口。窗口的偏移量通过 imageOffset 指定，窗口的大小通过 imageExtent 指定。可以通过设置 imageOffset.x 和 imageOffset.y 为 0，imageExtent.width 和 imageExtent.height 分别为 mipmap level 的大小，来覆盖掉整个 mipmap 层。需要你自己来做计算，Vulkan 不会替你做这个工作的。

也可以反方向的做复制操作--从一个图像对象复制数据到缓冲区。需要调用 vkCmdCopyImageToBuffer()命令，其原型如下：

```
void vkCmdCopyImageToBuffer (
    VkCommandBuffer commandBuffer,
    VkImage srcImage,
    VkImageLayout srcImageLayout,
    VkBuffer dstBuffer,
    uint32_t regionCount,
    const VkBufferImageCopy* pRegions);
```

做这个操作的命令缓冲区通过 commandBuffer 指定，含有源数据的图像通过 srcImage 参数传递，复制操作的目标图像通过 dstImage 参数传递。再有，两个图像的布局也需要传递到复制命令。srcImageLayout 是复制操作中期望的源图像布局，值为 VK_IMAGE_LAYOUT_GENERAL 或 VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL（当它是转移操作的源）。同样的，dstImageLayout 是期望的目标图像布局，值为 VK_IMAGE_LAYOUT_GENERAL 或 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL。

和“缓冲区至图像”，“图像至缓冲区”命令一样，vkCmdCopyImage()可同时复制多个区域。regionCount 指定了复制区域的个数，每一个都通过 VkImageCopy 类型数组内的一个实例表示，数组的地址通过 pRegions 传递。VkImageCopy 定义如下：

```
typedef struct VkImageCopy {
```

```
    VkImageSubresourceLayers  srcSubresource;
    VkOffset3D  srcOffset;
    VkImageSubresourceLayers  dstSubresource;
    VkOffset3D  dstOffset;
    VkExtent3D  extent;
} VkImageCopy;
```

每一个 VkImageCopy 实例包含子资源信息和源、目标的窗口起始位置。vkCmdCopyImage()不能改变数据的尺寸，所以源和目标图像的 extent 是相同的，且包含在 extent 域。srcSubresource 包含子对源数据子资源的定义，和传递给 vkCmdCopyImageToBuffer()命令的 VkBufferImageCopy 的 imageSubresource 域是相同的意思。同样，dstSubresource 包含了对目标区域的子资源的定义，和传递给 vkCmdCopyImageToBuffer()命令的 VkBufferImageCopy 的 imageSubresource 域意思相同。

Vulkan 编程指南翻译 第四章 队列和命令 第 5 节 复制压缩图像数据

翻译 2017 年 03 月 01 日 12:18:36

- 153
- 0
- 1

4.5 复制压缩图像数据

如在第二章"内存和资源"中讨论过的，Vulkan 支持多种压缩图像格式。所有目前定义的压缩格式都是基于固定大小的块的格式。对于其中的许多种，块大小是 4 × 4 纹素。对于 ASTC 格式，块大小是随图像变动的。

当从缓冲区复制数据到图像时，只有整数大小的块能被复制。因此，每一个图像区域，以纹素为单位，必须是图像所用块大小的整数倍。还有，复制操作的源区域必须是块大小的整数倍。

也可以在两个压缩图像间或者压缩图像与非压缩图像间复制数据，需调用 vkCmdCopyImage()命令。当你这么做时，源图像和目标图像格式必须拥有相同的块大小。这就是，如果压缩块的大小是 64 bits，那么源格式和目标格式也必须是块大小为 64 bits 的压缩图像，或者非压缩图像格式须是纹素为 64-bit 的格式。

当从非压缩图像向压缩图像复制数据时，每一个源纹素被当作一个原生数据，和压缩图像一个块大小。这个值直接被写入压缩数据，即使它是压缩数据。纹素的值并没有被 Vulkan 压缩。这允许你在应用程序或者 shader 中创建压缩图像数据，然后把它复制到压缩图像中以供后续处理。Vulkan 并不会替你要所原生的图像数据。再有，对于非压缩到压缩的复制，VkImageCopy 类型数据的 extent 域在源图像中以纹素为单位，但是要和目标图像要求的块大小匹配。

当从压缩格式向非压缩格式复制数据时，上述的也相反掉。Vulkan 不会解压缩图像数据。相反，它从源图像中获取原生的 64-bit 或 128-bit 压缩块的值，并把它们分解到目标图像。这种情况下，目标图像每纹素的大小应该和拥有源图像每块的大小相同。对于压缩到非压缩的复制，VkImageCopy 类型数据的 extent 域在目标图像中以纹素为单元度量，但是必须符合源图像的块大小。

在两个块压缩图像格式之间复制数据是允许的，只要两种图像格式每块拥有的 bits 数相同。然而，这个值是可以商讨的，因为压缩格式下的图像数据通常以另一种格式解读并解码出来也没有什么意义。不管它的价值如何，当进行这项操作时，被复制的区域依然以纹素为单位，但是所有的偏移量和 extents 必须是常见的块大小的整数倍。

Vulkan 编程指南翻译 第四章 队列和命令 第 6 节 展开图像

翻译 2017 年 03 月 01 日 12:26:24

- 213
- 0
- 1

4.6 展开图像

所有目前讲到的图片相关的命令，没有一个支持格式转换和改变复制区域的尺寸。你可以调用 vkCmdBlitImage()实现这个功能，它可以接受不同格式的图像并展开或缩小需要复制的区域，当这些区域写入到目标图像时。术语 blit 是 block image transfer 的缩写，指不仅需要复制图像数据，并且也可能需要在此过程中处理数据。vkCmdBlitImage()的原型如下：

```
void vkCmdBlitImage (
VkCommandBuffer  commandBuffer,
VkImage  srcImage,
VkImageLayout  srcImageLayout,
VkImage  dstImage,
VkImageLayout  dstImageLayout,
uint32_t  regionCount,
const VkImageBlit*  pRegions,
VkFilter  filter);
```

将执行此命令的命令缓冲区通过 commandBuffer 参数传递。源图像和目标图像各通过 srcImage 和 dstImage 参数传递。再有，和 vkCmdCopyImage()一样，期望的源图像和目标图像的布局通过 srcImageLayout 和 dstImageLayout 参数传递。源图像的布局是 VK_IMAGE_LAYOUT_GENERAL 或者 VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL，目标图像的布局是 VK_IMAGE_LAYOUT_GENERAL 或者 VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMA。

和其他的复制命令一样，vkCmdBlitImage()可复制源图像的任意数量区域到目标图像，每一个区域都通过一个数据结构表示。区域的数量由 regionCount 参数传递，pRegion 指向了一个大小为 regionCount 的数组，每一个元素定义了一个需要复制的区域。VkImageBlit 的定义为：

```
typedef struct VkImageBlit {
VkImageSubresourceLayers  srcSubresource;
VkOffset3D  srcOffsets[2];
VkImageSubresourceLayers  dstSubresource;
VkOffset3D  dstOffsets[2];
} VkImageBlit;
```

VkImageBlit 的 srcSubresource 和 dstSubresource 域定义了源图像和目标图像的子资源。然而，在 VkImageCopy 中每一个区域都被一个 VkOffset3D 类型的数据定义，并共享一个 VkExtent3D，在 VkImageBlit 中每一个区域都通过一对 VkOffset3D 数据（两个元素的数组）定义。

srcOffsets 和 dstOffsets 数组的第一个元素定义了将被复制的区域的一侧，数组的第二个元素定义了区域的另外一侧。源图像中 srcOffsets 定义的区域被复制到了 dstOffsets 定义的目标图像。如果这两个区域其中一个对于另外一个是“上下颠倒的”，那么复制的区域将会被垂直翻转。同理，如果一个区域相对域另外一个“前后颠倒的”，那么图像就会被水平翻转。如果这两种情况都满足，那么复制的区域和原区域相比就被旋转 180 度了。

如果在源区域和目标区域的巨型尺寸不相同，那么图像就会被相应的放大或缩小。这种情况下，vkCmdBlitImage()命令 filter 参数定义的过滤模式将会被应用到数据的过滤上。Filter 必须是 VK_FILTER_NEAREST 或 VK_FILTER_LINEAR 其一，分别用于点采样或线性采样。
源图像的格式必须是支持 VK_FORMAT_FEATURE_BLIT_SRC_BIT 特性的其中一种。在绝大多数 Vulkan 实现中，这几乎包含了所有的图像格式。还有，目标图像格式必须是支持 VK_FORMAT_FEATURE_BLIT_DST_BIT 的其中一种。通常，这是任何可以在 shader 内渲染到或者设备可写入的格式。Vulkan 设备不大有可能会支持在压缩图像格式上 blit 操作。
总结
本章讲解了如何清空图像并填充数据。我们使用命令缓冲区中的命令把少量的数据直接写入缓冲区对象，并解释了 Vulkan 是如何把在缓冲区和图像互相之间、和一对图像之间复制数据。最后，我们介绍了 blit 的概念，它是一个在复制数据时允许把数据缩放和转换图像格式的操作。这些操作提供了获取大量数据进出 Vulkan 设备以供后续处理的基础。

Vulkan 编程指南翻译 第五章 展现 第 1 节 拓展
翻译 2017 年 03 月 01 日 12:27:09

- 265
- 0
- 1

你将在本章中学到：
l 如何在屏幕上展现你的应用程序的结果
l 如何决定系统的显示设备
l 如何改变显示模式并和原生的窗口系统交流

Vulkan 主要是一个图形 API，它的大多数功能专注于生成和处理图片。绝大多数 Vulkan 应用程序被设计来给用户展现结果的。这就是常被称为“展现”的过程。然而，因为 Vulkan 所运行的平台各异，展示并不是它核心 API，但被移交给一套拓展。本章讨论如何开启并使用这些拓展来获取屏幕上的图像。

5.1 展现相关的拓展
Vulkan 里展示并不是核心 API 的一部分。实际上，一个 Vulkan 实现或许根本不支持展现。理由如下：
l 并不是所有的 Vulkan 应用程序都需要向用户展示图像。比如，计算集中型应用程序，也许会产生非可视化的数据或者产生只需要存储到磁盘而不是实时显示的图像。
l 展现通常是通过操作系统的窗口系统，或者其他特定的库处理的，它们在不同平台上的差别很大。
由于上述原因，展现是通过一套被称为 WSI 拓展（**W**indow **S**ystem **I**ntegration systems）处理的。Vulkan 中拓展在使用前必须显式地开启，且每一个平台需要的拓展或许会有点不同，如一些函数会接受平台特定的参数。在你可以执行任何展现相关的操作前，你需要使用第一章描述的机制来开启合适的展现相关的拓展。
在 Vulkan 中展现是由一套拓展处理的。在所有通过拓展以图形化形式输出给用户的平台上的功能都是类似的，且每个平台相关的功能都通过一些更小的、平台相关的拓展来支持。

Vulkan 编程指南翻译 第五章 展现 第 2 节 展现表面

翻译 2017 年 03 月 01 日 12:28:01

- 103
- 0
- 1

5.2 展现表面

被渲染的图形数据以备展示的对象被被称为”表面“，由一个 VkSurfaceKHR 类型的 handle 表示。这种特殊的对象由 VK_KHR_surface 拓展引入。这个拓展提供了处理 surface 对象的功能，但是它在各平台基础上高度定制化，以提供 surface 与窗口的对接的功能。Interfaces are defined for Microsoft Windows, Mir and Wayland, X Windows via either the XCB or Xlib interface, and Android. 以后有更多的平台加入。

拓展中平台特定的原型和数据类型被包含在最主要的 vulkan.h 头文件中，但是被平台特定的宏保护着。本书中的代码支持 Windows 平台和 Linux 平台（通过 Xlib 或者 Xcb 接口）。为了启用这些代码，在包含 vulkan.h 之前，我们必须定义 VK_USE_PLATFORM_WIN32_KHR, VK_USE_PLATFORM_XLIB_KHR, 或者 VK_USE_PLATFORM_LIB_XCB_KHR 这些宏之一。本书的源代码构建系统将会使用一个编译器命令行帮助你完成这个工作。

5.2.1 在 Microsoft Windows 上展现

在我们可以展示之前，我们需要知道在设备上是否有队列可以支持展示的操作。展示能力是每个队列族都有的能力。在 Windows 平台，调用 vkGetPhysicalDeviceWin32PresentationSupportKHR()函数来获知一个队列是否支持展示操作，函数原型如下：

```
VkBool32 vkGetPhysicalDeviceWin32PresentationSupportKHR(
    VkPhysicalDevice   physicalDevice,
    uint32_t   queueFamilyIndex);
```

被查询的物理设备通过 physicalDevice 参数传递，队列族的索引通过 queueFamilyIndex 传递。如果至少有一个队列族支持展示，那么我们就可以继续来在设备上创建表面对象。为了创建表面，使用 vkCreateWin32SurfaceKHR()函数，其原型如下：

```
VkResult vkCreateWin32SurfaceKHR(
    VkInstance   instance,
    const VkWin32SurfaceCreateInfoKHR*   pCreateInfo,
    const VkAllocationCallbacks*   pAllocator,
    VkSurfaceKHR*   pSurface);
```

这个函数把一个原生的窗口 handle 和一个新创建的 surface 对象关联起来，并通过 pSurface 参数返回这个对象。只有一个 Vulkan 实例是必须的，它的 handle 通过 instance 参

数传递。这个新的 surface 对象的描述信息通过 pCreateInfo 参数传递，它是一个 VkWin32SurfaceCreateInfoKHR 类型的指针，其定义如下：

```
typedef  struct  VkWin32SurfaceCreateInfoKHR  {
    VkStructure  Type   sType;
    const  void*   pNext;
    VkWin32SurfaceCreateFlagsKHR   flags;
    HINSTANCE   hinstance;
    HWND   hwnd;
}  VkWin32SurfaceCreateInfoKHR;
```

VkWin32SurfaceCreateInfoKHR 的 sType 域应被置为 VK_STRUCTURE_TYPE_DISPLAY_SURFACE_CREATE_INFO_KHR， 除非有其他的拓展使用 pNext 域，否则应该置为 nullptr。flags 被保留未来使用，应置为 0.

hinstance 参数应当置为应用程序或者 module 的 HINSTANCE，它是用来创建原生窗口的。这是应用程序传递给 WinMain 的第一个参数或者调用 Win32 函数 GetModuleHandle 获取到。hwnd 成员是原生窗口的 handle，它可以把 Vulkan 的 surface 关联起来。这就是渲染结果将要交换的呈现的窗口。

5.2.2 在基于 Xlib 的平台上展现

在 Xlib-Based 系统上创建 surface 的过程是相似的。第一，我们需要知道平台是否支持在 X Server 上展现一个 Xlib surface。调用函数 vkGetPhysicalDeviceXlibPresentationSupportKHR() 可以获取这个信息，其原型如下：

```
VkBool32  vkGetPhysicalDeviceXlibPresentationSupportKHR(
    VkPhysicalDevice   physicalDevice,
    uint32_t   queueFamilyIndex,
    Display*   dpy,
    VisualID   visualID);
```

物理设备的 handle 通过 physicalDevice 参数指定，队列族通过 queueFamilyIndex 参数指定，vkGetPhysicalDeviceXlibPresentationSupportKHR()报告了在那个族的队列是否支持在一个 X Server 上的 Xlib surface 展现结果。与 X Server 之间的联系由 dpy 参数表示，在每一种格式上展示都是支持的。在 Xlib 中，格式通过 visual ID 表示，surface 的目标格式是通过 visualID 参数指定。

假设设备上至少一个队列族支持我们想要使用的格式来展示，我们就可以给一个 Xlib 窗口创建 surface，需调用 vkCreateXlibSurfaceKHR()函数，它的原型是：

```
VkResult  vkCreateXlibSurfaceKHR(
    VkInstance   instance,
    const  VkXlibSurfaceCreateInfoKHR*   pCreateInfo,
    const  VkAllocationCallbacks*   pAllocator,
    VkSurfaceKHR*   pSurface);
```

vkCreateXlibSurfaceKHR()创建一个和 Xlib 窗口关联的 surface。Vulkan 实例因该通过 instance 参数传递，剩下的参数控制 surface 的创建，通过 pCreateInfo 参数传递。

pCreateInfo 是 VkXlibSurfaceCreateInfoKHR 类型数据实例的指针，该类型定义如下：

```
typedef  struct  VkXlibSurfaceCreateInfoKHR  {
    VkStructureType   sType;
```

```
    const  void*   pNext;
    VkXlibSurfaceCreateFlagsKHR   flags;
    Display*   dpy;
    Window   window;
} VkXlibSurfaceCreateInfoKHR;
```

sType 域应被置为 VK_STRUCTURE_TYPE_XLIB_SURFACE_CREATE_INFO_KHR，pNext 应置为 nullptr。flags 域保留使用且应置为 0。

Dpy 域是 Xlib Display，表示和 X Server 的连接，window 是 Xlib Window 类型的 handle，这将和新创建的 surface 关联起来。

如果 vkCreateXlibSurfaceKHR()需要主机内存，它将使用 pAllocator 参数传递过来的主机内存分配器。如果 pAllocator 为 nullptr，那么内置的内存分配器将会被使用。

如果 surface 创建成功，返回的 VkSurface 类型数据的 handle 将会写入 pSurface 这个指针。

5.2.3 在 Xcb 上展现

Xcb 是针对 X 协议的一个比 Xlib 更轻量级、底层次接口，也是对要求低延迟的应用来说是更好的选择。和 Xlib、其他的平台，在 Xcb 系统上创建显示对象，我们需要知道是否有队列支持展现。可以调用 vkGetPhysicalDeviceXcbPresentationSupportKHR()，其原型如下：

```
VkBool32  vkGetPhysicalDeviceXcbPresentationSupportKHR(
    VkPhysicalDevice   physicalDevice,
    uint32_t   queueFamilyIndex,
    xcb_connection_t*   connection,
    xcb_visualid_t   visual_id);
```

被查询的物理设备通过 physicalDevice 参数传递，队列族的索引通过 queueFamilyIndex 参数传递。和 X Server 的连接通过 connection 参数传递。再有，在每个 visual ID 基础上都有不同展示能力，需要查询的 visual ID 通过 visual_id 参数传递。

当你已经知道了设备上至少有一个队列族支持在你选择的 visual ID 上展示，你就可以调用 vkCreateXcbSurfaceKHR() 函数来创建可以放渲染结果的 surface，函数原型如下：

```
VkResult  vkCreateXcbSurfaceKHR(
    VkInstance   instance,
    const  VkXcbSurfaceCreateInfoKHR*   pCreateInfo,
    const  VkAllocationCallbacks*   pAllocator,
    VkSurfaceKHR*   pSurface);
```

Vulkan 实例通过 instance 传递，剩下的参数通过一个 VkXcbSurfaceCreateInfoKHR 类型数据的指针 pCreateInfo 来传递，它控制 surface 的创建。VkXcbSurfaceCreateInfoKHR 定义如下：

```
typedef  struct  VkXcbSurfaceCreateInfoKHR  {
    VkStructureType   sType;
    const  void*   pNext;
    VkXcbSurfaceCreateFlagsKHR   flags;
    xcb_connection_t*   connection;
    xcb_window_t   window;
} VkXcbSurfaceCreateInfoKHR;
```

VkXcbSurfaceCreateInfoKHR 的 sType 域应置为 VK_STRUCTURE_TYPE_XCB_SURFACE_CREATE_INFO_KHR，pNext 应置为 nullptr。flags 域保留使用且应置为 0。与 X Server 的连接通过 connection 与来传递，窗口的 handle 通过 window 传递。如果 vkCreateXcbSurfaceKHR()调用成功，它将新创建的 surface 的 handle 写入到 pSurface 所指向的变量。如果它需要主机内存来构造这个 handle 且 pAllocator 不为 nullptr，那么它将使用你提供的内存分配器来获取内存。

支持 Vulkan 的移动 GPU

原创 2017 年 03 月 02 日 15:51:46

去年差不多这个时候，Vulkan 标准发布，NVIDIA 和 AMD 随之发布了显卡的 Vulkan 驱动，虽然都是实验版本。但是毕竟能够工作的。Intel 的速度就慢了不少。时隔一年，Intel 终于推出了 Vulkan 认证的驱动，虽然之前就有了实验性质的测试版本。

Intel Core Processor 第五代和 第六代开始支持 Vulkan，当然，最近发布的第七代肯定是支持的了。Intel 还有一点很好，他把 Vulkan 的驱动开源了。说是这样，我似乎还没有找到代码库。

ARM 很早就宣布支持 Vulkan 了，目前官网公布的可支持硬件有：

ARM® Mali™-T760

ARM® Mali™-T820

ARM® Mali™-T830

ARM® Mali™-T860

ARM® Mali™-T880

ARM® Mali™-G51

ARM® Mali™-G71

我的魅蓝 Note 都带了 T-860，可想而知有多少中低端的设备可以跑 Vulkan。然而，并没有什么卵用。Android 7 中系统才会带有驱动。Flyme OS6 竟然是基于 Android 5.1，等到我的机器用废了，估计也没有操作系统来适配了。悲伤！看了看，似乎小米做的还不错，更新的比较快。不要黑，我对手机没有什么概念，打电话就可以。如果大家想要搞一搞移动端的 Vulkan 开发，可以参考下面链接[5]。MTK 平台的机器都还不错，只等着操作系统厂商跟进

Android 7 了。估计 17 年过了，主流的手机大厂都进入到 Android 7，估计到那时候有一条广告语就是“Vulkan Supported！”。

P.S.　看到了任天堂竟然也支持了 Vulkan，真是可喜可贺，可喜可贺啊。希望这个 API 能够蓬勃的发展下去，有更多的人参与，把它做得更好。

ref

1. http://blogs.intel.com/evangelists/2016/02/16/intel-open-source-graphics-drivers-now-support-vulkan/
2. https://01.org/zh/linuxgraphics/blogs/jekstrand/2016/open-source-vulkan-drivers-intel-hardware?langredirect=1
3. https://01.org/zh/linuxgraphics/documentation/development/source-code?langredirect=1
4. https://01.org/zh/linuxgraphics/documentation/hardware-specification-prms
5. https://developer.arm.com/graphics/vulkan/vulkan-drivers
6. http://www.ithome.com/html/android/280099.htm
   国内已确认升级安卓 7.0 手机全汇总：大品牌更靠谱
7. http://www.igao7.com/news/201608/rxHDorvIV0HHh20I.html
   骁龙 800/801 设备无缘安卓 7.0 的原因竟然是
8. http://www.qualcomm.cn/news/releases-2016-02-18-0　Qualcomm 宣布 Adreno 530 GPU 支持 Vulkan API

Vulkan 编程指南翻译 第五章 展现 第 3 节 交换链

翻译　2017 年 03 月 02 日 17:24:27

- 144
- 0
- 1

5.4 全屏画幕

在前一节提到的平台特定的拓展允许创建的 VkSurface 对象代表一个操作系统或者窗口系统原生的一个窗口。这些拓展通常被用来在一个桌面上可见的窗口做渲染工作。即使我们可以创建一个不带边框、占满真个屏幕的窗口，这样渲染效率更高。

这个功能由 VK_KHR_display 和 VK_KHR_display_swapchain 拓展提供。这些拓展提供了一个平台无关的机制，来获取系统的显示器，判定他们的属性和支持模式等等。

如果 Vulkan 实现支持 VK_KHR_display，你就可以通过调用 vkGetPhysicalDeviceDisplayPropertiesKHR()函数来获知屋里设备连接了几个显示器。其原型如下：

```
VkResult vkGetPhysicalDeviceDisplayPropertiesKHR(
VkPhysicalDevice
physicalDevice,
uint32_t*  pPropertyCount,
VkDisplayPropertiesKHR*
pProperties);
```

显示器是连接在物理设备上的，你想获取哪个显示器的信息需要通过

physicalDevice 参数指定。pPropertyCount 是指向一个变量，将被函数覆盖为连接到设备
的显示器数量。如果 pProperties 为 nullptr，那么被 pPropertyCount 指向的变量的初始值被忽略，仅被覆盖为连接到设备的显示器总数。然而，如果 pPropertyCount
不为 nullptr，那么 pProperties 就是一个指向 VkDisplayPropertiesKHR 类型数组的指针。数组的长度通过 pPropertyCount 传递。VkDisplayPropertiesKHR 定义为

```
typedef struct VkDisplayPropertiesKHR {
VkDisplayKHR display;
const char* displayName;
VkExtent2D physicalDimensions;
VkExtent2D physicalResolution;
VkSurfaceTransformFlagsKHR supportedTransforms;
VkBool32 planeReorderPossible;
VkBool32 persistentContent;
} VkDisplayPropertiesKHR;
```

VkDisplayPropertiesKHR 的 display 成员是之后将被用到的显示器的 handle。displayName 是人可读的显示器描述字符串。physicalDimensions 域给出了显示器的维度，
以毫米为单位，physicalResolution 给出了显示器的分辨率，以像素为单位。
一些显示器（或者显示器控制器）在显示时支持反转或者旋转。如果是这种情形，这些能力信息包含在 supportedTransforms 域中。这个位域由 VkSurfaceTransformsFlagsKHR
枚举类型的多个值构成。
如果显示器支持多个显示面板，那么当这些面板可被排序时 planeReorderPossible 将被置为 VK_TRUE，。如果这些面板值能以固定的顺序显示，那么
planeReorderPossible 被置为 VK_FALSE。
最后，一些显示器可以接受部分或者低频率更新，这样可以提高能源利用率。如果显示器支持以这种方式更新，persistentContent 将被置为 VK_TRUE，否则为
VK_FALSE.。
所有 GPU 设备都支持与其相连的每一个显示器上有至少一个显示面板。一个面板可以向用户展示图像。在一些场景下，一个 GPU 设备可以支持把多个面板拼装起来用来
最终显示一幅图像。这些面板也通常被称为叠加面板，因为每一个面板可以逻辑上叠加在另外一个之上或之下。当一个 Vulkan 应用显示时，它可以向显示器的一个面板
显示图像。也可在一个应用程序中向多个面板显示。
被支持的面板数量被认为是设备的一部分，通常意义上的显示设备（并不是物理显示器），可以用来从多个面板中融合信息并显示单独一张图像。

一个物理设备就可以显示它所支持的面板上图像的一部分。可以调用 vkGetPhysicalDeviceDisplayPlanePropertiesKHR()来查询 GPU 设备可以支持面板的个数
和类型，其原型如下：

```
VkResult vkGetPhysicalDeviceDisplayPlanePropertiesKHR(
VkPhysicalDevice physicalDevice,
uint32_t* pPropertyCount,
VkDisplayPlanePropertiesKHR* pProperties);
```

被查询的覆盖能力的物理设备通过 physicalDevice 传递。如果 pProperties 为空，那么 pPropertyCount 是一个其值将被覆盖为设备能够支持的

Vulkan 编程指南翻译 第五章 展现 第 3 节 全屏画幕

翻译 2017 年 03 月 02 日 17:25:55

- 172
- 0
- 1

5.4 全屏画幕

在前一节提到的平台特定的拓展允许创建的 VkSurface 对象代表一个操作系统或者窗口系统原生的一个窗口。这些拓展通常被用来在一个桌面上可见的窗口做渲染工作。即使我们可以创建一个不带边框、占满真个屏幕的窗口，这样渲染效率更高。

这个功能由 VK_KHR_display 和 VK_KHR_display_swapchain 拓展提供。这些拓展提供了一个平台无关的机制，来获取系统的显示器，判定他们的属性和支持模式等等。

如果 Vulkan 实现支持 VK_KHR_display，你就可以通过调用 vkGetPhysicalDeviceDisplayPropertiesKHR()函数来获知屋里设备连接了几个显示器。其原型如下：

```
VkResult vkGetPhysicalDeviceDisplayPropertiesKHR(
VkPhysicalDevice
physicalDevice,
uint32_t*  pPropertyCount,
VkDisplayPropertiesKHR*
pProperties);
```

显示器是连接在物理设备上的，你想获取哪个显示器的信息需要通过 physicalDevice 参数指定。pPropertyCount 是指向一个变量，将被函数覆盖为连接到设备
的显示器数量。如果 pProperties 为 nullptr，那么被 pPropertyCount 指向的变量的初始值被忽略，仅被覆盖为连接到设备的显示器总数。然而，如果 pPropertyCount
不为 nullptr，那么 pProperties 就是一个指向 VkDisplayPropertiesKHR 类型数组的指针。数组的长度通过 pPropertyCount 传递。VkDisplayPropertiesKHR 定义为

```
typedef struct VkDisplayPropertiesKHR {
VkDisplayKHR display;
const char* displayName;
VkExtent2D physicalDimensions;
VkExtent2D physicalResolution;
VkSurfaceTransformFlagsKHR supportedTransforms;
VkBool32 planeReorderPossible;
VkBool32 persistentContent;
} VkDisplayPropertiesKHR;
```

VkDisplayPropertiesKHR 的 display 成员是之后将被用到的显示器的 handle。displayName 是人可读的显示器描述字符串。physicalDimensions 域给出了显示器的维度，
以毫米为单位，physicalResolution 给出了显示器的分辨率，以像素为单位。
一些显示器（或者显示器控制器）在显示时支持反转或者旋转。如果是这种情形，这些能力信息包含在 supportedTransforms 域中。这个位域由
VkSurfaceTransformsFlagsKHR
枚举类型的多个值构成。
如果显示器支持多个显示面板，那么当这些面板可被排序时
planeReorderPossible 将被置为 VK_TRUE，。如果这些面板值能以固定的顺序显示，那么
planeReorderPossible 被置为 VK_FALSE。
最后，一些显示器可以接受部分或者低频率更新，这样可以提高能源利用率。如果显示器支持以这种方式更新，persistentContent 将被置为 VK_TRUE，否则为
VK_FALSE.。
所有 GPU 设备都支持与其相连的每一个显示器上有至少一个显示面板。一个面板可以向用户展示图像。在一些场景下，一个 GPU 设备可以支持把多个面板拼装起来用来
最终显示一幅图像。这些面板也通常被称为叠加面板，因为每一个面板可以逻辑上叠加在另外一个之上或之下。当一个 Vulkan 应用显示时，它可以向显示器的一个面板
显示图像。也可在一个应用程序中向多个面板显示。
被支持的面板数量被认为是设备的一部分，通常意义上的显示设备（并不是物理显示器），可以用来从多个面板中融合信息并显示单独一张图像。
一个物理设备就可以显示它所支持的面板上图像的一部分。可以调用
vkGetPhysicalDeviceDisplayPlanePropertiesKHR()来查询 GPU 设备可以支持面板的个数
和类型，其原型如下：

```
VkResult vkGetPhysicalDeviceDisplayPlanePropertiesKHR(
VkPhysicalDevice physicalDevice,
uint32_t* pPropertyCount,
VkDisplayPlanePropertiesKHR* pProperties);
```

被查询的覆盖能力的物理设备通过 physicalDevice 传递。如果 pProperties 为空，那么 pPropertyCount 是一个其值将被覆盖为设备能够支持的

Vulkan 编程指南翻译 第五章 展现 第 5 节 开始呈现

5.6 开始显示

显示，是在队列上下文中发生的操作。一般，提交到队列的命令缓冲区中的命令被执行时，会产生用来显示
图像，所以，这些图像只有在渲染操作完成后才能展示给用户。一个系统中一个设备可以支持多个队列，
它们不必都支持显示。在你可以使用队列来在画幕上显示之前，你必须判断队列是否支持在某个画幕上
显示。
要想知道一个队列是否支持显示，传递物理设备，画幕，队列组到
vkGetPhysicalDeviceSurfaceSupportKHR()函数
并调用，函数原型如下：

```
VkResult vkGetPhysicalDeviceSurfaceSupportKHR(
VkPhysicalDevice physicalDevice,
uint32_t queueFamilyIndex,
VkSurfaceKHR surface,
VkBool32* pSupported);
```

需要查询的物理设备通过 physicalDevice 传递。所有的队列都属于一个队列族，所有属于一个队列族
的队列都被认为有相同的属性。因此，只需要判断一个队列所属的族是否支持显示。队列组索引通过
queueFamilyIndex 传递。
队列显示的能力取决于画幕。例如，一些队列可能在操作系统上的两个窗口显示，但是并没有全屏显示这样的物理设备
直接访问权限。因此，你想显示所在的画幕通过 surface 传递。
如果 vkGetPhysicalDeviceSurfaceSupportKHR()调用成功，在某个队列族的队列在 surface 这个画幕上
的能力通过会被写入的 pSupported 指针表示--VK_TRUE 表示支持，VK_FALSE 表示缺乏支持。如果有错误，
vkGetPhysicalDeviceSurfaceSupportKHR()将返回一个错误码，pSupported 的值就不会被覆写。
在图像能够被显示之前，它必须被正确的布局。这个布局状态是
VK_IMAGE_LAYOUT_PRESENT_SRC_KHR。

图像可从一个布局迁移到另一个布局，我们在第二章“内存和资源”中讲解过。” Listing5.2
将展示如何使用图像内存屏障把图像从
VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL 迁移到
VK_IMAGE_LAYOUT_PRESENT_SRC_KHR 布局。
Listing 5.2: Transitioning an Image to Present Source

```
const VkImageMemoryBarrier barrier =
{
VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER, // sType
nullptr, // pNext
VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT, // srcAccessMask
VK_ACCESS_MEMORY_READ_BIT, // dstAccessMask
VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL, // oldLayout
VK_IMAGE_LAYOUT_PRESENT_SRC_KHR, // newLayout
0, // srcQueueFamilyIndex
0, // dstQueueFamilyIndex
sourceImage, // image
{ // subresourceRange
VK_IMAGE_ASPECT_COLOR_BIT, // aspectMask
0, // baseMipLevel
1, // levelCount
0, // baseArrayLayer
1, // layerCount
}
};

vkCmdPipelineBarrier(cmdBuffer,
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
0,
0, nullptr,
0, nullptr,
1, &barrier);
```

注意，图像内存屏障在一个命令缓冲区内执行，这个命令缓冲区应该提交到一个队列以被执行。
一旦图像处于 VK_IMAGE_LAYOUT_PRESENT_SRC_KHR 布局，它就可以通过调用 vkQueuePresentKHR()被显示给用户了
其原型为：

```
VkResult vkQueuePresentKHR(
VkQueue queue,
const VkPresentInfoKHR* pPresentInfo);
```

图像提交以显示的队列通过 queue 指定。这个命令剩下的参数通过一个 VkPresentInfoKHR
类型的实例传递，该类型定义为:

```
typedef struct VkPresentInfoKHR {
VkStructureType sType;
const void* pNext;
uint32_t waitSemaphoreCount;
const VkSemaphore* pWaitSemaphores;
uint32_t swapchainCount;
const VkSwapchainKHR* pSwapchains;
const uint32_t* pImageIndices;
VkResult* pResults;
} VkPresentInfoKHR;
```

VkPresentInfoKHR 的 sType 应置为 VK_STRUCTURE_TYPE_PRESENT_INFO_KHR，pNext 应置为 nullptr。在图像
被显示之前，Vulkan 将选择性等待一个或者多个信号量来使渲染图像和显示保持同步。需等待的信号量个数
通过 waitSemaphoreCount 成员传递，pWaitSemaphores 成员指向了一个包含所有需要等待的信号量的 hanlde 的数组。
一次 vkQueuePresentKHR()调用会同时向多个交换链中提交多个图像。这很有用，例如，一个应用程序正在向
多个窗口同时渲染。需要呈现的图像个数通过 swapchainCount 指定，pSwapchains 是呈现图像的交换链对象数组。
呈现到每一个交换链的图像不通过 VkImage 类型的 handle 被引用，需通过它们在交换链对象中
获取到的交换链图像数组中的索引来引用。对于每一个被呈现图像的交换链，图像的索引通过对应数组 pImageIndices
中的元素的索引传递。

每一个单独的通过调用 vkQueuePresentKHR()触发的呈现操作可以产生自己的返回码。记住，VkResult 中的一些值表示成功，。
pResults 是指向了一个大小为 swapchainCount 的数组。多个 VkResult 变量将会被呈现操作的结果填充。

Vulkan 编程指南翻译 第五章 展现 第 6 节 清扫工作

翻译 2017 年 03 月 03 日 16:05:30

- 168
- 0
- 1

5.6 清扫工作

不管你在应用程序中用什么方法来呈现，你都需要正确的做清扫工作。第一，你应该销毁正在呈现的交换链。需要
调用 vkDestroySwapchainKHR()函数，其原型如下：

```
void vkDestroySwapchainKHR(
VkDevice  device,
VkSwapchainKHR
swapchain,
const VkAllocationCallbacks*
pAllocator);
```

拥有该交换链的设备通过 device 传递，需要被销毁的交换链通过 swapchain 传递。如果之前用
自定义的内存分配器来创建交换链，那么就需要一个配对的内存分配器通过 pAllocator 传入。
当交换链被销毁，所有与其关联的可显示的图像都被销毁了。因此，你需要保证没有将向这些画幕写入
的待命的任务，也没有从画幕读取数据的待命操作。最简单的方式是调用 vkDeviceWaitIdle()。通常是不被推荐的，因为销毁交换链通常并不会在应用程序性能苛刻的部分出现，
所以，在这种情况下，简单的就是最好的。
当使用 vkAcquireNextImageKHR()从交换链中取得图像时，或者使用 vkQueuePresentKHR()呈现
图像时，信号量头传递到这些函数中去激发或等待。需要注意，信号量需要存活的时间需要比交换链在
被销毁前完成任何激发操作所需时间都要长。为了保证这条，最好在销毁任何信号量之前销毁可能使用这些
信号量的交换链。

总结

在本章，你学习了如何包图像放到显示器的 Vulkan 操作。我们讲解了各种窗口系统的展现操作，你如何
确定图像该往哪个显示器渲染的机制，如何遍历和控制连接到系统的显示器设备。我们简单的介绍了呈现
相关的同步，将在后面章节继续深入同步原语。我们也讨论了配置显示器同步的方法。依本章的

信息，你应对如何把图像展示给用户有了相当的理解。

Vulkan 编程指南翻译 第六章 着色器和管线 第 1 节 GLSL 简介

翻译 2017 年 03 月 04 日 12:02:26

- 218
- 0
- 1

第六章 着色器和管线

在本章你将学到

l 什么事着色器及如何使用它

l SPIR-V -- Vulkan 渲染语言的基础知识

l 如何构造着色器管线并用它做工作

着色器是在设备上执行的小程序。他们是任何复杂 Vulkan 程序构建的基础单元。对你的应用程序中操作来讲，着色器比 Vulkan API 更重要。本章介绍着色器和着色器模块，展示他们如何从 SPIR-V 二进制文件构建，并讲解如何使用标准工具从 GLSL 生成这些二进制文件。讲解由着色器组成的管线和需要运行它们的其他信息，然后展示如何在设备上执行着色器来完成工作。

着色器是设备上执行工作的基础的构建单位。Vulkan 着色器通过 SPIR-V 表示，它是一个二进制中间码，可表示程序代码。SPIR-V 可以用编译器离线生成，或直接在程序内部在线的生成，或者直接通过高级语言传递给运行时库。本书示例程序使用第一种方式：离线的编译后从磁盘载入生成的 SPIR-V 二进制文件。

原始的着色器程序由 GLSL 语言书写，参照 Vulkan 标准。这是一个相较于 OpenGL 使用的着色语言有修改和增强的版本。因此，大多数例子，依 GLSL 而言讨论 Vulkan 的特征。然而，应该明白 Vulkan 本身和 GLSL 没有任何关系，它也不关心 SPIR-V 着色器程序从哪里来。

6.1 GLSL 总览

尽管并不是 Vulkan 标准的一部分，Vulkan 和 OpenGL 特性部分相似。在 OpenGL 中，官方支持的高级语言是 GLSL--OpenGL Shading Language。因此，在 SPIR-V 设计期间，很多工作集中在保证至少有一个高级语言适合生成 SPIR-V 着色器。GLSL 经少量修改就可以被 Vulkan 使用。一些新添加的特性让 GLSL 着色器可以清晰的和 Vulkan 系统交互，OpenGL 不被引入 Vulkan 的遗留特征被从 Vulkan GLSL 规范中去除了。

结果就是一个简洁版的 GLSL 支持 Vulkan 绝大多数特征，但也同时保持了 OpenGL 和 Vulkan 之间高度的兼容性。简而言之，如果你坚持在 OpenGL 程序中使用高级特征，你在着色器中写的多数代码都将可使用官方的编译器直接编译为 SPIR-V 中间码。淡然，你也可以使用自己写的编译器和工具，或者使用第三方的编译器来从任意语言产生 SPIR-V 模块。

对 GLSL 的修改允许它被用来产生 Vulkan 可用 SPIR-V 着色器程序，这些修改记录在 GL_KHR_vulkan_glsl 拓展的文档中。

在本节，我们简单的介绍 GLSL。这里假设读者对于高级着色语言是有一些熟悉的，并且有能力来深入的研究 GLSL。
Listing 6.1 展示一个最简单的 GLSL 着色器。它只是一个空函数，返回 void，什么操作也不做。这实际上对于 Vulkan 管线任何阶段来说是一个有效的着色器程序，即使在一些阶段执行它会造成为定义的行为。
Listing 6.1: Simplest Possible GLSL Shader

```
#version 450 core
void main (void)
{
// Do nothing!
}
```

所有的 GSL 着色器都应一个#version 宏开头，来告知 GLSL 编译器我们正在使用哪个版本的 GLSL 这允许编译器进行适当的错误检查，并允许随着时间语言引进新的结构。
当被编译为 SPIR-V 以供 Vulkan 使用时，编译器应向在用的 GL_KHR_vulkan_glsl 拓展自动定义 VULKAN 宏，以便你可以在 GLSL 着色器的 #ifdef VULKAN or #if VULKAN >{version} 代码块中包含 Vulkan 特定的构造或者功能，这样着色器就可以被 OpenGL 和 Vulkan 共同使用了。
贯穿这本书，当讨论 GLSL 语境下 Vulkan 特定的特征时，都默认假设缩写代码要么只为 Vulkan 所写，要么被包含在合适的#ifdef 预处理条件中，以被编译为 Vulkan 所用。
GLSL 是 C 风格的语言，它的语法和许多语义符号借鉴子 C 或 C++。如果你是一个 C 程序员，你应该对下面的结构感到熟悉 for、while 循环类似的构造；流程控制关键词如 break，continue；switch 语言；条件操作符如==, <, and >；二元操作符 a ？ b ： c 等等。这些都可以在 GLSL 着色器中使用
GLSLjichu 的数据类型是有符号和无符号整型和浮点数，通过 int, uint, and float 表示。双精度浮点数一刻通过 double 来声明。在 GLSL 内部，它们没有预定义的位宽，和 C 类似。GLSL 没有 stdint 这个类似物，所以不支持定义任意位宽的变量，即使 GLSL 和 SPIR-V 规范提供了 Vulkan 所用变量的数值范围和精度的最小保障。然而，位宽和布局只被定义来从内存中读取或写入。整型在内存中以二进制补码的形式存储，浮点数尽量遵循 IEEE 标准，除了精度要求上的小差别，来处理反常和不是数字（NaN）的值等等。
除了基本标量整型和浮点类型，GLSL 内置最多四个成员的向量和 4×4 的矩阵作为第一等公民。可以声明基本类型（有符号、无符号整型和浮点类型标量）的向量和矩阵。Vec2，vec3 和 vec4 类型，比如，是二元、三元和四元浮点类型的向量。整型向量可用 i 或者 u 作为前缀以表示有符号和无符号。故，ivec4 是一个四元有符号整型向量，uvec4 是四元无符号整型向量。d 前缀用来表示双精度浮点类型。故，dvec4 是四元双精度浮点类型的向量。
矩阵用 matN 或者 matNxM 这样的形式书写，各代表 N x N 方阵和 N x M 矩阵。d 前锥也可以和矩阵类型一起使用表示双精度矩阵。因此，dmat4 是 4 x 4 的双精度浮点方阵类型。然而，并没有整型的矩阵。矩阵被认为是列矩阵，可以被当作向量的数组。故，m 为 mat4 的
m[3]表示一个四元素的浮点向量（vec4），它是 m 最后一列。

Boolean 类型也是 GLSL 的第一等公民，可以形成向量（但并不能形成矩阵），和浮点、整型变量一样。Boolean 变量使用 bool 类型来声明。对向量的的关系运算产生 Boolean 类型数组，每一个元素都代表一个比较的结果。内置函数 any()和 all()被用来判断数组中是否有一个为 true 和是否所有的元素都是 true。
系统产生的数据通过内置的变量传递给 GLSL 着色器。比如 gl_FragCoord, gl_VertexIndex 等变量，它们每一个都将在本书相关的章节讲到。内置变量一般拥有特殊的语义，对它们的读写会改变着色器程序的行为。
用户自定义数据通常通过内存传递给着色器。变量可以被绑定在一个 block 里，然后绑定到设备可访问的资源，存在在程序可以读写的内存里。这允许你向着色器程序传递大量的数据。针对小但频繁更新的数据，“push constants”这种特殊类型的变量我们在本书稍后讲解。
GLSL 提供了大量的内置函数。然而，和 C 语言相比，GLSL 并没有头文件。所以并不必须要使用#include 宏。GLSL 的标准库已经内置到编译器里面了。它包括了很多数学函数，纹理访问函数，和一些特殊的流程控制函数--它们可以控制设备上着色器程序的执行。

Vulkan 编程指南翻译 第六章 着色器和管线 第 2 节 SPIR-V 概述

翻译 2017 年 03 月 04 日 17:55:27

- 138
- 0
- 1

6.2 SPIR-V 概述

SPIR-V 着色器嵌入在 module 里。每一个 module 都可以包含一个或多个着色器。每一个着色器都有一个固定名字入口点和着色器类型，这用来定义着色器在哪个着色阶段来使用。入口点之着色器开始运行时的起始位置。一个 SPIR-Vmodule 都随着一些附带信息传递 给 Vulkan，Vulkan 返回一个对象来表示这个 module。此 module 可以用来构造一个管线，管线是附带信息的着色器经过编译的版本，可以在设备上运行。

6.2.1 如何表示 SPIR-V

SPIR-V 是 Vulkan 官方唯一支持的着色语言。在 API 层面被 Vulkan 接受，最终用来构建管线，管线对象可配置一个设备来完成应用程序的工作。
SPIR-V 被设计为可被工具和驱动很简单就能使用。这减少了不同实现之间的不同意提高兼容性。SPIR-V module 在内存的 32bi 每字数据流形式存储的。除非你是工具作者或者计划自己生成 SPIR-V，你不大可能会去直接处理二进制编码的 SPIR-V。相反，你要么会去看 SPIR-V 的文本形式，或者使用诸如 glslangvalidator（Chronos 官方 GLSL 编译器）的工具来生成 SPIR-V。
把着色器程序以.comp 后缀名的方式保存为文本文件，告诉 glslangvalidator 把它当作计算着色器编译。我们可以通过使用 glslangvalidator 命令行的方式来编译着色器：

```
glslangvalidator simple.comp -o simple.spv
```

这会产生一个名为 simple.spv　的 SPIR-V 二进制文件。我们可以使用 SPIR-V 反汇编工具，spirv-dis，来反编译这个二进制文件，会输出一份人可读的反汇编代码。如 Listing 6.2 所示：

Listing 6.2: Simplest SPIR-V

```
; SPIR-V
; Version: 1.0
; Generator: Khronos Glslang Reference Front End; 1
; Bound: 6
; Schema: 0
OpCapability Shader
%1 = OpExtInstImport "GLSL.std.450"
OpMemoryModel Logical GLSL450
OpEntryPoint GLCompute %4 "main"
OpExecutionMode %4 LocalSize 1 1 1
OpSource GLSL 450
OpName %4 "main"
%2 = OpTypeVoid
%3 = OpTypeFunction %2
%4 = OpFunction %2 None %3
%5 = OpLabel
OpReturn
OpFunctionEnd
```

你可以看到 SPIR-V 的文本形式看起来像汇编语言的变体。我们可深入这段反汇编，来看看他和原始的输入文件如何关联。反汇编文件的每一行待变一个 SPIR-V 指令，可能由多个 token 组成。

第一个指令 OpCapability Shader，要求这个着色器开启兼容模式。SPIR-V 功能粗略的分为“指令”和“特征”。在你的着色器程序可以使用这些特征之前，shader 必须要声明使用的特征是哪一部分的。Listing 6.2 的着色器程序是一个图形着色器，因此使用 Shader 能力。这是最基础的能力了。没有这个，我们不能编译图形着色器。随着我们引入更多 SPIR-V 和 Vulkan 功能，我们将介绍这些不同能力所依赖哦特征。

下一行，我们看到　%1 = OpExtInstImport "GLSL.std.450"。 这就是引入了一些附加的指令，对应着 GLSL 450 包含的功能，这也是原始着色器程序所写入的版本。注意这个指令以 %1 = 开头。这把指令计算的结果使用 ID 命名。OpExtInstImport 的结果是一个有效的库。当我们需要调用这个库的函数，我们可以使用 OpExtInst 指令，它接受一个库（OpExtInstImport 指令的结果）和一个指令索引。这允许 SPIR-指令集被任意的拓展。

下一行，我们看到一些附加的声明，OpMemoryModel 指定了这个 module 的工作内存的模型，这是对应 GLSL 450 的逻辑上的内存模型。这意味着所有的内存访问是通过资源而不是通过指针访问内存的物理内存模型。

下一行是这个 module 入口点的声明。OpEntryPoint GLCompute %4 "main" 指令表明对应的 OpenGL 计算着色器有一个可用的入口，以函数名字 main 导出为 ID 4。这个名字被用来当把结果着色器 module 返回 Vulkan 时的参考入口点。

我们在后续的指令中使用这个 ID。OpExecutionMode %4 LocalSize 1 1 1，它定义了这个着色器的工作组大小为 1 × 1 × 1 的 work item。 如果没有局部大小的 layout 限定符，这在 GLSL 中是隐式的。
下面两个指令是简单的声明类型的。OpSource GLSL 450 表示这个 module 从 GLSL 450 版本编译而来，OpName 4 "main"提供了 ID 为 4 的 token 的名字。现在，我们看到这个函数的真身了。第一， %2 = OpTypeVoid 声明了我们想要以 void 类型使用 ID 2. 所有东西在 SPIR-V 中都有一个 ID，甚至是类型声明。大型、复合的类型可以通过连续的小的，简单的类型组成。然而，我们需要从某处开始，并且指定我们开始的地方类型为 void。
%3 = OpTypeFunction %2 便是我们定义了 ID 3 为一个函数类型，类型为 void，不接受参数。我们在下一行使用它，%4 = OpFunction %2 None %3。这表示我们声明 ID 4（知其那被命名为“main”）为函数 3 的一个实例（在上一行声明），返回 void（如 ID 2），且没有特殊的声明，这通过指令中的 None 表明，而且可以被用来表示诸如 inline，不管变量是否为常量（常量性）等等。
最后，我们可以看到一个 label（没有被用到，编译器操作的副作用）的声明，隐式的返回语句，和函数的结束。这是我们 SPIR-Vmodule 的结尾。
这个着色器程序的二进制 dump 是 192 字节长。SPIR-V 是非常详尽的，因为 192 字节比原来的着色器要长。然而，SPIR-V 把原来的着色语言中隐式的东西变得显式了。比如，在 GLSL 中无需声明内存模型，因为它只支持一种逻辑内存模型。更有，这里编译的 SPIR-V module 有一些冗余信息，我们不关心 main 函数的名字，ID 5 的 label 从来没有被使用，且着色器引入了 GLSL.std.450 库，但是从来没有使用过。我们可以把这个 module 中多余的信息抽离出去。即使在此之后，因为 SPIR-V 编码方式相对稀疏，产生的二进制文件使用一个通用的压缩器也相当容易被压缩，而且使用一个专门的压缩库可以把它压缩的更紧致 。
所有的 SPIR-V 编码都通过 SSA 来书写（single static assignment），这表明每一个虚拟寄存器（在上面的 List 中写作%n 的 token）都被仅写入一次。几乎所有的工作的指令产生一个结果标识符。当我们开始写更复杂的着色器时，你将看到机器离线生成的 SPIR-V 有点笨拙，因为它的详尽特性和 SSA 形式，非常难以手写。非常建议你在应用程序中以 lib 形式使用编译器来离线生成 SPIR-V。
如果你计划自己生成或者解释 SPIR-V module，你可以使用预定义的二进制编码器来构建工具，来解析或生成它们。然而，它们都有格式良好的二进制存储格式，我们在本章稍后讲解。
所有的 SPIR-V module 文件都以一个 magic number 开始，这可以用来简单的炎症二进制块，实际上就是 SPIR-V module。这个魔法数字以无符号整型来看是 0x07230203。这个数字也可以用来退单 module 的字节顺序。因为每一个 SPIR-V toke 都是一个 32-bit word，如果一个 SPIR-V module 通过磁盘或网络传输到有不同字节顺序的主机端，这些在一个 word 内的字节都被交换位置了，它的值给改变了。例如，如贵哦一个 SPIR-V module 存储为小端格式，被加载到一个大端格式主机 CPU，那么魔法书记就会被读为 0x03022307，所以这个 CPU 就知道需要在这个 module 里交换字节顺序。
在魔法数字的后面有几个 word，描述了 module 的属性。第一个是 SPIR-V 使用的的版本数字。它被编码在一个 32-bit word 里，其中 16-23 bit 为包含主版

本号，8-15 包含次版本号。 SPIR-V 1.0 因此使用编码 0x00010000。版本号剩下的 bit 位是保留的。下一个 token 包含生成 SPIR-V module 的工具的版本号。这个值由工具自定义。
下一个是本 module 中最大的 ID 号码。所有的变量，函数，和 SPIR-V module 的其他成员都被赋值到一个比这个数字小的 ID，所以，在最前面包含这个数字允许工具分配好数组来存放它们，而不是随时的分配内存。头部最后一个 word 是保留的，应被置为 0。接下来的是指令流。

6.2.2 把 SPIR-V 传递给 Vulkan

Vulkan 并不关心 SPIR-V 着色器和 module 从哪里来。通常，它们会被离线编译好作为应用程序的一部分，或者在应用程序中直接生成。一旦你有了一个 SPIR-V module，你需要把它传递给 Vulkan，以便可以使用它创建一个着色器 module 对象。可以调用 vkCreateShaderModule()来做到，其原型如下：

```
VkResult vkCreateShaderModule (
VkDevice device,
const VkShaderModuleCreateInfo* pCreateInfo,
const VkAllocationCallbacks* pAllocator,
VkShaderModule* pShaderModule);
```

和所有的 Vulkan 对象创建函数一样，vkCreateShaderModule()接受一个设备 handle 输入，和一个指向包含创建对象信息的数据的指针。此时，就是 VkShaderModuleCreateInfo 类型，其定义为：

```
typedef struct VkShaderModuleCreateInfo {
VkStructureType sType;
const void* pNext;
VkShaderModuleCreateFlags flags;
size_t codeSize;
const uint32_t* pCode;
} VkShaderModuleCreateInfo;
```

VkShaderModuleCreateInfo 的 sType 域应置为 VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO，pNext 应置为 nullptr。flags 域保留使用应置为 0。codeSize 域包含了 SPIR-V module 字节单位的大小，代码通过 pCode 传入。
如果这个 SPIR-V 代码是有效的，且能够被 Vulkan 理解，那么 vkCreateShaderModule()将返回 VK_SUCCESS，并

Vulkan 编程指南翻译 第六章 着色器和管线 第 3 节 管线

翻译 2017 年 03 月 07 日 00:26:35

- 152
- 0
- 1

6.3 管线

如你在前面小节所读到的，Vulkan 使用着色器 module 来表示一系列的着色器程序通过把 module 代码交给 vkCreateShaderModule()可以创建着色器 module，但是，在它们可以在设备上被用来工作之前，你需要创建管线。在 Vulkan 中有两种管线：计算和图形。图形管线相对复杂，且包含许多和着色器

相关的状态。然而，计算管线在概念上简单多了，且出来着色器代码也不包含其他的什么东西。

6.3.1 计算管线

在我们讨论创建计算管线之前，我们应该讲讲计算管线的基础知识。着色器和它的执行时 Vulkan 的核心。Vulkan 也提供了对各种功能的固定块大小的访问，比如复制和处理像素数据。然而，着色器将会是任何有意义程序的核心。计算着色器提供了对 Vulkan 设备计算能力的直接访问。设备可以被视为许多处理相关数据的宽向量处理单元的集合。一个计算着色器一般被认为会连续的，单轨道的执行。然而，有办法可以让多个轨道上一起指向。实际上，这也是大多数 Vulkan 设备如何被构造的。每一个执行轨道被称为一个调用。

当计算着色器被执行时，许多调用马上开始。这些调用被子和到一个固定大小的本地工作组，然后，一个或多个组被一起发射，它有时被称为全局工作组的。逻辑上，本地工作组和全局工作组都是三维的。然而，设定三维的任一维度的大小减少了组的维度。

本地工作组的大小在计算着色器内部设置。在 GLSL 中，这是通过使用 layout 限定符做到的，限定符被翻译为传递给 Vulkan 的着色器的 OpExecutionMod 描述符上的 LocalSize 修饰符。Listing 6.3 展示了在着色器中使用的大小修饰符，Listing 6.4 展示了生成的精简的 SPIR-V 反汇编代码

Listing 6.3: Local Size Declaration in a Compute Shader (GLSL)

```
#version 450 core
layout (local_size_x = 4, local_size_y = 5, local_size_z 6) in;
void main(void)
{
// Do nothing.
}
```

Listing 6.4: Local Size Declaration in a Compute Shader (SPIR-V)

```
...
OpCapability Shader
%1 = OpExtInstImport "GLSL.std.450"
OpMemoryModel Logical GLSL450
OpEntryPoint GLCompute %4 "main"
OpExecutionMode %4 LocalSize 4 5 6
OpSource GLSL 450
...
```

你可以看到，Listing 6.4 的 OpExecutionMode 指令设置着色器的本地大小为{4, 5, 6}，这在 Listing 6.3 中指定。

一个着色器程序的本地工作组的最大个数由调用 vkGetPhysicalDeviceProperties()可以获取 VkPhysicalDeviceLimits 的数据的 maxComputeWorkGroupSize 域指定，在第一章“Vulkan 简介”中解释过。还有，本地工作组的最大调用数之和也是同一个类型数据的 maxComputeWorkGroupInvocations 域指定的。一个 Vulkan 实现也许会拒绝超过这些限制条件的 SPIR-V 着色器，即使只是会出现一些未知的运行结果而已。

6.3.2 创建管线

可调用 vkCreateComputePipelines()来创建一个或多个管线，函数原型如下：

```
VkResult vkCreateComputePipelines (
VkDevice device,
VkPipelineCache pipelineCache,
uint32_t createInfoCount,
const VkComputePipelineCreateInfo* pCreateInfos,
const VkAllocationCallbacks* pAllocator,
VkPipeline* pPipelines);
```

vkCreateComputePipelines()的参数 device，就是负责使用管线并扶着分配管线对象的设备。pipelineCache 是用来加速管线创建的一个对象的 handle，在本章稍后涉及到。创建一个新管线的参数信息通过一个 VkComputePipelineCreateInfo 类型的数据表示。该数据结构的个数（亦即需要创建的管线的个数）通过 createInfoCount 传入，这写数据组成的数组的地址通过 pCreateInfos 传入。VkComputePipelineCreateInfo 的定义如下：

```
typedef struct VkComputePipelineCreateInfo {
VkStructureType sType;
const void* pNext;
VkPipelineCreateFlags flags;
VkPipelineShaderStageCreateInfo stage;
VkPipelineLayout layout;
VkPipeline basePipelineHandle;
int32_t basePipelineIndex;
} VkComputePipelineCreateInfo;
```

VkComputePipelineCreateInfo 的 sType 应置为 VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO，pNext 应置为 nullptr。flags 域保留使用，在当前的 Vulkan 版本中应置为 0。stage 域是一个嵌入式的结构，包含着色器本身的信息，它是 VkPipelineShaderStageCreateInfo 类型的一个实例，定义如下：

```
typedef struct VkPipelineShaderStageCreateInfo {
VkStructureType sType;
const void* pNext;
VkPipelineShaderStageCreateFlags flags;
VkShaderStageFlagBits stage;
VkShaderModule module;
const char* pName;
const VkSpecializationInfo* pSpecializationInfo;
} VkPipelineShaderStageCreateInfo;
```

VkPipelineShaderStageCreateInfo 的 sType 是 VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO，pNext 应置为 nullptr。flags 域保留使用，在当前的 Vulkan 版本中应置为 0。

VkPipelineShaderStageCreateInfo 类型数据在所有阶段的管线创建中都使用。尽管图形管线拥有多个阶段（在第七章“图形管线”中学到），计算管线只有一个阶段，所以，stage 应置为 VK_SHADER_STAGE_COMPUTE_BIT。
module 是之前创建的着色器 module 的 handle，它包含了你想要创建的管线所需的代码。因为单一一个着色器 module 可以包含多个入口点和多个着色器，表示这个特别的管线的入口点通过 VkPipelineShaderStageCreateInfo 的 pName 域指定。这也是少数的几个被 Vulkan 使用的人可断的字符串。

6.3.3 特殊常量

VkPipelineShaderStageCreateInfo 最后一个域是一个指向 VkSpecializationInfo 类型数据的指针。这个结构包含特化一个着色器所需的信息，“特化”是指包含常量的着色器构造的过程。
一个典型的 Vulkan 实现会管线代码的最终生成，直到调用 vkCreateComputePipelines()函数。
这允许特化常量的值在着色器优化的最后一个 pass 中才被求值。常见使用和特化的程序使用的常量包含：

- Producing special cases through branching: Including a condition on a Boolean specialization constant will result in the final shader taking only one branch of the if statement. The nontaken branch will probably be optimized away. If you have two similar versions of a shader that differ in only a couple of places, this is a good way to merge them into one.
- Special cases through switch statements: Likewise, using an integer specialization constant as the tested variable in a switch statement will result in only one of the cases ever being taken in that particular pipeline. Again, most Vulkan implementations will optimize out all the nevertaken cases.
- Unrolling loops: Using an integer specialization constant as the iteration count in a for loop may result in the Vulkan implementation making better decisions about how to unroll the loop or whether to unroll it at all. For example, if the loop counter ends up with a value of 1, then the loop goes away and its body becomes straight-line code. A small loop iteration count might result in the compiler unrolling the loop exactly that number of times. A larger iteration count may result in the compiler unrolling the loop by a factor of the count and then looping over that unrolled section a smaller number of times.

• Constant folding: Subexpressions involving specialization constants can be folded just as with
any other constant. In particular, expressions involving multiple specialization constants may
fold into a single constant.
• Operator simplification: Trivial operations such as adding zero or multiplying by one disappear,
multiplying by negative one can be absorbed into additions turning them to subtractions,
multiplying by small integers such as two can be turned into additions or even absorbed into
other operations, and so on.

在 GLSL 中，特化常量被声明为一个普通的常量，在一个 layout 限定符中被给予一个 ID。在 GLSL 中特化常量可以是 Boolean，整型，浮点类型或者诸如数组、结构、向量、矩阵等复合类型。当被翻译为 SPIR-V 时，这些都变成了 OpSpecConstant token。Listing 6.5 展示了一个 GLSL 声明了一些特化常量的例子，Listing 6.6 展示了 GLSL 编译器生成的 SPIR-V。

Listing 6.5

```
layout (constant_id = 0) const int numThings = 42;
layout (constant_id = 1) const float thingScale = 4.2f;
layout (constant_id = 2) const bool doThat = false;
```

Listing 6.6

```
...
OpDecorate %7 SpecId 0
OpDecorate %9 SpecId 1
OpDecorate %11 SpecId 2
%6 = OpTypeInt 32 1
%7 = OpSpecConstant %6 42
%8 = OpTypeFloat 32
%9 = OpSpecConstant %8 4.2
%10 = OpTypeBool
%11 = OpSpecConstantFalse %10
...
```

Listing 6.6 被编辑过，删除了和特化常量的无关的代码。然而，你可以看到，, %7 被 OpSpecConstant 指令声明为一个特化常量，类型为% 6（一个 32 位的整型）， 初始值为 42。下一行，% 9 被声明为一个特化常量，类型为% 8（32 位浮点类型），初始值为 4.2.最后，%11 被声明为一个 Boolean 类型（在这个 SPIR-V 中类型为%10），初始值为 false。注意，Boolean 型被 OpSpecConstantTrue 或 OpSpecConstantFalse 声明，取决于他们的初始值为 true 还是 false。

注意，在 GLSL 着色器和生成的 SPIR-V 着色器中，特化常量都被赋予了初值。实际上，它们必须要被赋初值。这些常量也许和着色器中其他常量一样的被使用。特殊情况下，它们可以被用来指定数组大小--只允许编译时常量能被使用。如果新的值没有被包含在传递给 vkCreateComputePipelines()的 VkSpecializationInfo 类型的数据中，那么这些默认值就被使用。然而，这些常量可以被创建管线时传递的新值覆盖。VkSpecializationInfo 定义为：

```
typedef struct VkSpecializationInfo {
uint32_t mapEntryCount;
const VkSpecializationMapEntry* pMapEntries;
size_t dataSize;
const void* pData;
} VkSpecializationInfo;
```

在 VkSpecializationInfo 内部，mapEntryCount 包含了需要被设置新值的特化常量的个数，这也是 pMapEntries 所指向的 VkSpecializationMapEntry 类型数组元素的个数。每一个都表示一个特化常量。VkSpecializationMapEntry 定义为：

```
typedef struct VkSpecializationMapEntry {
uint32_t constantID;
uint32_t offset;
size_t size;
} VkSpecializationMapEntry;
```

constantID 是特化常量的 ID，被用来匹配着色器 module 中使用常量 ID。在 GLSL 中通过使用 constant_id 布局限定符来设置值，在 SPIR-V 中使用 SpecID 描述符来设置值。Offset 和 size 域是包含特化常量的值的原生数据的偏移量和大小。VkSpecializationInfo 类型数据的 pData 域指向了原生数据，数据大小通过 dataSize.给定。Vulkan 使用此数据来初始化特化常量。当管线被构造时，如果着色器中一个或者多个特化常量没有在该数据中指定，它就会使用默认值。

当你使用完了管线并不再需要它时，可以销毁它来释放它关联的资源。可调用 vkDestroyPipeline()来销毁管线对象，函数原型如下：

```
void vkDestroyPipeline (
VkDevice device,
VkPipeline pipeline,
const VkAllocationCallbacks* pAllocator);
```

拥有管线的设备通过 device 指定，需要被销毁的管线通过 pipeline 传递。如果在创建管线是使用了主机内存分配器，那么需要使用 pAllocator 来传递一个匹配的分配器；否则，pAllocator 应置为 nullptr。

在管线被销毁后，它不应该再被使用了。这包含可能还没有完成执行的命令缓冲区中的任何对它的引用。应用程序有责任保证任何提交的引用到该管线的命令缓冲区已经完成执行，和绑定该管线的任何命令缓冲区在管线被销毁后不被提交。

6.3.4 加速管线的创建

创建管线可能是你的应用程序最昂贵的操作。尽管 SPIR-V 代码被 vkCreateShaderModule()消耗，但是直到你调用 vkCreateGraphicsPipelines() 或 vkCreateComputePipelines()，Vulkan 才能看到所有的着色器阶段和其他的域管线相关能影响到最终在设备上运行的代码的阶段。因为此原因，一个 Vulkan 实现也许会延迟涉及创建一个准备执行的管线对象的工作，直到尽可能最后一刻。这包括着色器编译和代码生成，这些是典型的相当密集型的操作。
因为应用程序运行很多次也只是使用一遍又一遍的使用相同的管线，Vulkan 提供了一个多次运行时缓存管线创建结果的机制。这允许应用程序在启动时就构建所有的管线来迅速启动。管线缓存通过下面函数创建的一个对象表示，

```
VkResult vkCreatePipelineCache (
VkDevice device,
const VkPipelineCacheCreateInfo* pCreateInfo,
const VkAllocationCallbacks * pAllocator,
VkPipelineCache* pPipelineCache);
```

用来创建管线缓存的设备由 device 指定。创建管线缓存所需的剩下的参数通过一个 VkPipelineCacheCreateInfo 类型数据的指针传递。该类型定义如下：

```
typedef struct VkPipelineCacheCreateInfo {
VkStructureType sType;
const void * pNext;
VkPipelineCacheCreateFlags flags;
size_t initialDataSize;
const void * pInitialData;
} VkPipelineCacheCreateInfo;
```

VkPipelineCacheCreateInfo 的 sType 域应置为 VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO，pNext 应置为 nullptr。flags 域保留使用，应置为 0。如果存在程序上一次运行产生的数据，数据的地址可以通过 pInitialData 传递。数据的大小通过 initialDataSize 传递。如果没有数据，initialDataSize 应置为 0，pInitialData 应置为 nullptr。
当创建了缓存之后，初始出具用来填充缓存。如果有必要，Vulkan 会复制一份数据。pInitialData 指向的数据没有被修改。当更多的管线创建时，描述它们的数据被添加到缓存，随着时间增长。可以调用 vkGetPipelineCacheData()从缓存中取出数据。其原型如下：

```
VkResult vkGetPipelineCacheData (
VkDevice device,
VkPipelineCache pipelineCache,
size_t* pDataSize,
void* pData);
```

拥有该管线缓存的设备通过 device 指定，需被获取数据的管线缓存应通过 pipelineCache 传递。如果 pData 不是 nullptr，那么他指向将接受缓存数据的内存区域。这种情况下，pDataSize 指向的变量的初始值是内存区域以字节为单位的大小。这个变量就会被数据的真实大小所覆盖。

如果 pData 为 nullptr，那么 pDataSize 所指向的变量初始值就被忽略，变量被覆盖为用来存储缓存数据的内存的大小。可以调用 vkGetPipelineCacheData()两次来存储所有的缓存数据；第一次，设置 pData 为 nullptr，pDataSize 指向一个接受缓存数据大小的变量。然后，创建一个缓冲区来存储生成的缓存数据并再次调用 vkGetPipelineCacheData()，这一次向 pData 传递一个指向这个地址区域的指针。Listing 6.7 举例说明如何保存管线数据到文件。

Listing 6.7: Saving Pipeline Cache Data to a File

```
VkResult SaveCacheToFile(VkDevice device, VkPipelineCache cache,
const char* fileName)
{
size_t cacheDataSize;
VkResult result = VK_SUCCESS;
// Determine the size of the cache data.
result = vkGetPipelineCacheData(device,
cache,
&cacheDataSize,
nullptr);
if (result == VK_SUCCESS && cacheDataSize != 0)
{
FILE* pOutputFile;
void* pData;
// Allocate a temporary store for the cache data.
result = VK_ERROR_OUT_OF_HOST_MEMORY;
pData = malloc(cacheDataSize);
if (pData != nullptr)
{
// Retrieve the actual data from the cache.
result = vkGetPipelineCacheData(device,
cache,
&cacheDataSize,
pData);
if (result == VK_SUCCESS)
{
// Open the file and write the data to it.
pOutputFile = fopen(fileName, "wb");
if (pOutputFile != nullptr)
{
fwrite(pData, 1, cacheDataSize, pOutputFile);
fclose(pOutputFile);
}
free(pData);
}
}
```

```
}
return result;
}
```

一旦你接收了管线数据，你可以把它存储到磁盘或者打包以供再次运行程序时使用。缓存的内容没有预定的结构，这是有实现决定的。然而，缓存数据的前几个 word 形成的头部可用来验证大块的数据是否为有效的缓存和哪个设备创建了它。

缓存头部的布局可以使用下面的 C 结构体来表示：

```
// This structure does not exist in official headers but is included
here
// for illustration.
typedef struct VkPipelineCacheHeader {
uint32_t length;
uint32_t version;
uint32_t vendorID;
uint32_t deviceID;
uint8_t uuid[16];
} VkPipelineCacheHeader;
```

尽管结构体的成员都是 uint32_t 类型变量，缓存的数据并不必要是 uint32_t 类型的。缓存总是以小端排序的，不管主机的字节排序是那种。这意味着如果你想在大端排序的主机上解释这个结构，你需要翻转 uint32_t 类型域的排序。

lenth 域是头部结构的大小，以字节为单位。在当前的技术规范版本中，这个长度应该为 32。version 域是结构的版本。已定义的版本只有 1。vendorID 和 deviceID 域应该和通过调用 vkGetPhysicalDeviceProperties()返回的 VkPhysicalDeviceProperties 结构的 vendorID 和 deviceID 域匹配。uuid 域是一个不透明的字符串类型，用来唯一标识这个 GPU 设备。如果 vendorID, deviceID, 或 uuid 域有一个不匹配 Vulkan 驱动期望的值，那么它会拒绝缓存数据并把它置为空。一个驱动也许会内置摘要加密或其他数据到缓存里，来保证无效的缓存数据不会加载到设备。

如果你有两个缓存对象并希望那个融合它们，可调用 vkMergePipelineCaches()来完成，其原型如下：

```
VkResult vkMergePipelineCaches (
VkDevice device,
VkPipelineCache dstCache,
uint32_t srcCacheCount,
const VkPipelineCache* pSrcCaches);
```

device 参数是拥有被融合缓存数据的设备的 handle。dstCache 是目标缓存的 handle，它最终会变成源缓存数组中每一条的合体。将被融合的缓存的个数通过 srcCacheCount 指定，pSrcCaches 是一个指向 VkPipelineCache 类型数组的指针，数组中每一个 handle 是需要被融合的缓存。

在 vkMergePipelineCaches()执行后，dstCache 将包含 pSrcCaches 所指定的源缓存数组中每一条缓存。然后才能调用 vkGetPipelineCacheData()来获取一个单一的、大型的缓存数据结构，它可表示其他所有的缓存。
这一点特别有用，例如，在多线程创建管线时。尽管对管线缓存的访问是线程安全　的，Vulkan 实现也许在内部采用锁来防止对多个缓存的同时写入。如果你创建多个管线缓存--每线程一个--并在管线的创建初始时使用它们。稍后，当管线都被创建完，你可以合并多个管线，以把它们的数据存储在一个大型的资源里。
当你使用完管线并不再需要很长，就需要销毁它，因为它可能会很大。可调用 vkDestroyPipelineCache()来销毁管线缓存对象，其原型是：

```
void vkDestroyPipelineCache (
VkDevice device,
VkPipelineCache pipelineCache,
const VkAllocationCallbacks* pAllocator);
```

device 是拥有管线缓存的设备 handle，pipelineCache 是需要被销毁的管线缓存对象。在管线缓存被销毁后，它不应该再把使用了，尽管用缓存创建的管线依然有效。通过调用 vkGetPipelineCacheData()从缓存中获取到的任何数据也还是有效的，可以用来构建新的缓存。

6.3.5　绑定管线

在你可使用管线之前，它必须保额绑定到执行互指或分发命令的命令缓冲区。当一个命令被执行是，当前的管线（即其中所有的着色器程序）被用来处理这个命令。可调用 vkCmdBindPipeline()来把管线绑定到一个命令缓冲区，其原型如下：

```
void vkCmdBindPipeline (
VkCommandBuffer commandBuffer,
VkPipelineBindPoint pipelineBindPoint,
VkPipeline pipeline);
```

正在绑定的命令缓冲区通过 commandBuffer 指定，被绑定的管线通过 pipeline 指定。在每一个命令缓冲区上有两个管线绑定点：图形和计算绑定点。计算绑定点是计算管线应该绑定的点。图形管线在下一章讲解，它应被绑定到图形管线绑定点。
把管线绑定到计算绑定点，需设置 pipelineBindPoint 为 VK_PIPELINE_BIND_POINT_COMPUTE；把管线绑定到图形管线绑定点，应设置 pipelineBindPoint 为 VK_PIPELINE_BIND_POINT_GRAPHICS。
当前与计算和图形管线的绑定是命令缓冲区的一个状态。当新的命令缓冲区开始时，这个状态是未定义的。因此，你必须在管线被用来工作前把管线绑定到一个绑定点。

Vulkan 编程指南翻译 第六章 着色器和管线 第 4 节 执行工作

翻译　2017 年 03 月 10 日 08:16:01

- 246
- 0
- 3

6.4 执行工作

在前一节，你看到了如何使用 vkCreateComputePipelines()构造一个计算管线并把大绑定到一个命令缓冲区。一旦管线被绑定，你可以用它来执行工作。

计算着色器作为计算管线的一部分，以分组的形式来执行，分组成为本地工作组。这些工作逻辑上速度一致的执行，在着色器里被指定了固定的大小。本地工作组最大个数一般比较小，但是至少是 128 调用 × 128 调用× 64 调用。还有，在单一一个本地工作组里最大调用个数也比总容量较小，且要求只能是 128 调用。因此原因，本地工作组从较大的组开始，优势被称为全局工作组或者分发大小。从计算着色器中开始任务因此被称为分发任务，或者分发。本地工作组逻辑上是一个三维结构，或者调用立体，计算一个或两个温度可以是单个调用，以保持工作组在该方向上扁平。同样，这些本地工作组在三维上一起被分发，即使一个或多个维度是一个工作组深度。

用计算管线可用来分发任务的命令是 vkCmdDispatch()，其原型如下：

```
voidvkCmdDispatch (
VkCommandBuffer commandBuffer,
uint32_t x,
uint32_t y,
uint32_t z);
```

将要执行任务的命令缓冲区通过 commandBuffer 传递。x,y,z 每一个维度中的本地工作组数量通过 x,y,z 参数传递。一个有效的计算管线必须绑定到命令缓冲区的 VK_PIPELINE_BIND_POINT_COMPUTE 绑定点。当命令被设备执行是，一个 x × y× z 大小的全局工作组开始执行所绑定管线的着色器。

极有可能一个本地工作组在有效维度上与全局工作组不同。例如，可以存在一个 32 × 32 × 1 分发 64 × 1 × 1 的本地工作组。

在 vkCmdDispatch()中能用参数指定工作组的个数之外，也可能做到间接分发，在工作组中分发的个数来自缓冲区对象。这允许分发大小在使用一个缓冲区来间接分发构造命令缓冲区 然后把缓冲区内容写入主机端之后计算。缓冲区的内容可被设备更新，来给设备提供工作。

vkCmdDispatchIndirect()的原型如下：

```
voidvkCmdDispatchIndirect (
VkCommandBuffer commandBuffer,
VkBuffer buffer,
VkDeviceSize offset);
```

包含该命令的命令缓冲区通过 commandBuffer 传递。和 vkCmdDispatch()中传递的分发个数不一样，工作组中每一个维度的大小被存储在三个连续的 uint32_t 的变量里，其在缓冲区对象 buffer 中起始偏移位置是 offset。缓冲区中参数基本上代表了一个 VkDispatchIndirectCommand 类型的数据，定义为：

```
typedefstruct VkDispatchIndirectCommand {
uint32_t x;
uint32_t y;
uint32_t z;
}VkDispatchIndirectCommand;
```

缓冲区中的内容在设备执行的命令缓冲区中 vkCmdDispatchIndirect()命令到达之前不会被读取。
工作组每一个维度的最大个数可通过检查 vkGetPhysicalDeviceProperties()调用得来的设备的 VkPhysicalDeviceLimits 类型数据的 maxComputeWorkGroupCount 成员来获知，在第一章“Vulkan 简介”中奖结果。在 vkCmdDispatch()调用中超过了这个限制或者在 vkCmdDispatchIndirect()里使用了超出限定值的将会导致为定义（很有可能是坏的）的行为。

Vulkan 编程指南翻译 第六章 着色器和管线 第 5 节 在着色器中访问资源
翻译 2017 年 03 月 10 日 08:16:33

- 275
- 0
- 2

6.5 在着色器中访问资源

应用程序中的着色器有两种方式来消耗和生产数据。第一种是通过固定管线，第二种是直接读取和写入资源。你在第二章”内存和资源“中看到如何创建缓冲区和图像。在本节，我们介绍描述符集合，它表示着色器程序可以操作的资源的集合。

6.5.1 描述符集合

描述符集合是被绑定到管线的一系列资源形成的组。多个集合可以同时绑定到管线。每一个集合都有布局，描述了集合中资源的排放顺序和类型。两个拥有相同布局的集合被视为兼容的和可互交换的。集合的布局铜鼓一个对象表示，集合都是参照这个对象被创建的。甚至，可被管线访问的集合的集合组成了另一个对象：管线布局。管线通过参照这个管线布局对象来创建。
描述符集合布局和管线布局在 Figure 6.1 展示。你可以在此图中看到，定义了两个描述符集合，第一个有纹理，采样器和两个缓冲区。第二个包含四个纹理，两个采样器和三个缓冲区。这些描述符集合的布局堆积在一个管线布局上。然后，可以参考管线布局来创建一个管线，参考描述符集合布局来创建描述符集合。那些描述符集合和兼容的管线一道被绑定到命令缓冲区，来允许管线访问到他们的资源。

在任何时刻，应用程序都可以绑定一个新的描述符集合到命令缓冲区的某一个具有相同布局的绑定点。相同的描述符集合布局可以被用来创建多个管线。因此，如果你一系列的具有相同资源的对象，但是每一个对象有额外不同的资源，当你的应用程序需要从一个对象转移到另外一个对象来做渲染时，你可以把相同的集合绑定并替换独有的资源。可调用 vkCreateDescriptorSetLayout()来创建描述符集合布局对象，其原型如下：

```
VkResultvkCreateDescriptorSetLayout (
                                VkDevice

                              device,
                                constVkDescriptorSetLayoutCreate
Info*          pCreateInfo,
```

```
                                    constVkAllocationCallbacks*
                                                pAllocator,
                                    VkDescriptorSetLayout*
                                                            pSetLayo
ut);
```
和往常一样，用来狗仔描述符集合布局对象的信息通过一个数据的指针来传递。这个数据是 VkDescriptorSetLayoutCreateInfo 类型的，其定义如下：
```
                typedefstruct VkDescriptorSetLayoutCreateInfo {
                                    VkStructureType
                                                            sTyp
e;
                                    constvoid*

            pNext;
                                    VkDescriptorSetLayoutCreateFlags
          flags;
                                    uint32_t

                bindingCount;
                                    constVkDescriptorSetLayoutBindin
g* pBindings;
                }VkDescriptorSetLayoutCreateInfo;
```

VkDescriptorSetLayoutCreateInfo 的 sType 域应被置为 VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO，pNext 应置为 nullptr。flags 被保留使用，应置为 0。资源被绑定到描述符集合的绑定点。VkDescriptorSetLayoutCreateInfo 的 bindingCount 和 pBindings 成员包含了该集合的绑定点个数，和一个包含有它们描述的数组的指针。每一个绑定都通过一个 VkDescriptorSetLayoutBinding 类型的实例来表示，其定义如下：
```
                typedefstruct VkDescriptorSetLayoutBinding {
                                    uint32_t
                                                binding;
                                    VkDescriptorType
                      descriptorType;
                                    uint32_t
                                                descriptorCount;
                                    VkShaderStageFlags
                stageFlags;
                                    constVkSampler*
                      pImmutableSamplers;
                }VkDescriptorSetLayoutBinding;
```

每个着色器可访问的资源都被给定了一个绑定号。这个绑定号存储在 VkDescriptorSetLayoutBinding
的 binding 域，在一个集合中序号可以不连续。然而，很推荐你不要创建稀疏防止的集合，因为这回浪费 GPU 设备资源。这个绑定点的描述符的类型存储在 descriptorType。这是 VkDescriptorType 枚举类型的一个成员。我们在稍后章节讨论不同的资源类型，它们包括：

• VK_DESCRIPTOR_TYPE_SAMPLER: A sampler is an object that can be usedto perform
operations such as filtering and samplecoordinate transformations when reading data from an
image.
• VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE: A sampled image is an image thatcan be used
in conjunction with a sampler to providefiltered data to a shader.
• VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER: A combined image-sampler object is a sampler and an image pairedtogether. The same sampler is always used to sample
from the image, which can be more efficienton some architectures.
• VK_DESCRIPTOR_TYPE_STORAGE_IMAGE: A storage image is an image thatcannot be
used with a sampler but can be written to.This is in contrast to a sampled image, which cannot
be written to.
• VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER: A uniform texel buffer isa buffer
that is filled with homogeneous formatteddata that cannot be written by shaders. Knowing that
buffer content is constant may allow someVulkan implementations to optimize access to the
buffer better.
• VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER: A storage texel buffer isa buffer
that contains formatted data much like auniform texel buffer, but a storage buffer can be written
to.
• VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER and VK_DESCRIPTOR_TYPE_STORAGE_BUFFER: Theseare similar to VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER and VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER,except that the data is unformatted
and described by structures declared in theshader.
• VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC and VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC:These are similar to

VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER and
VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, butinclude an offset and size that are passed
when the descriptor set is bound to thepipeline rather than when the descriptor is bound into the
set. This allows a single buffer in asingle set to be updated at high frequency.
• VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT: An input attachment is aspecial type of
image whose content is generated by earlieroperations on the same image in a graphics
pipeline.

Listing 6.8 illustrates how a selection ofresources is declared inside a GLSL shader.

```
                #version450 core
layout (set = 0, binding = 0) uniformsampler2DmyTexture;
layout (set = 0, binding = 2) uniformsampler3DmyLut;
layout (set = 1, binding = 0) uniformmyTransforms
{
mat4 transform1;
mat3 transform2;
};
void main(void)
{
// Do nothing!
}
```

Listing 6.9 展示了 Listing 6.8 中一个着色器被 GLSL 编译器编译为紧密格式。

Listing 6.9: Declaring Resources in SPIR-V

```
; SPIR-V
; Version: 1.0
; Generator: Khronos Glslang Reference Front End; 1
; Bound: 22
; Schema: 0
OpCapability Shader
%1 = OpExtInstImport "GLSL.std.450"
OpMemoryModel Logical GLSL450
OpEntryPoint GLCompute %4 "main"
OpExecutionMode %4 LocalSize 1 1 1
```

```
OpSource GLSL 450
OpName %4 "main"
OpName %10 "myTexture"
OpName %14 "myLut"
OpName %19 "myTransforms"
OpMemberName %19 0 "transform1"
OpMemberName %19 1 "transform2"
OpName %21 ""
OpDecorate %10 DescriptorSet 0
OpDecorate %10 Binding 0
OpDecorate %14 DescriptorSet 0
OpDecorate %14 Binding 2
OpMemberDecorate %19 0 ColMajor
OpMemberDecorate %19 0 Offset 0
OpMemberDecorate %19 0 MatrixStride 16
OpMemberDecorate %19 1 ColMajor
OpMemberDecorate %19 1 Offset 64
OpMemberDecorate %19 1 MatrixStride 16
OpDecorate %19 Block
OpDecorate %21 DescriptorSet 1
OpDecorate %21 Binding 0
%2 = OpTypeVoid
%3 = OpTypeFunction %2
%6 = OpTypeFloat 32
%7 = OpTypeImage %6 2D 0 0 0 1 Unknown
%8 = OpTypeSampledImage %7
%9 = OpTypePointer UniformConstant %8
%10 = OpVariable %9 UniformConstant
%11 = OpTypeImage %6 3D 0 0 0 1 Unknown
%12 = OpTypeSampledImage %11
%13 = OpTypePointer UniformConstant %12
%14 = OpVariable %13 UniformConstant
%15 = OpTypeVector %6 4
%16 = OpTypeMatrix %15 4
%17 = OpTypeVector %6 3
%18 = OpTypeMatrix %17 3
%19 = OpTypeStruct %16 %18
%20 = OpTypePointer Uniform %19
%21 = OpVariable %20 Uniform
%4 = OpFunction %2 None %3
%5 = OpLabel
OpReturn
OpFunctionEnd
```

在一个管线中可以使用多个描述符集合布局。你可以在 Listing 6.8 和 6.9 看到，资源被放到了两个集合，第一个包含"myTexture" and "myLut"（两个采样器），第二个包含"myTransforms"(a uniform buffer). 把两个或多个描述符集合打包来给管线使用，我们需要把它们汇集到一个 VkPipelineLayout 对象。可以调用 vkCreatePipelineLayout()来做到，其原型如下：

```
VkResultvkCreatePipelineLayout (
                                VkDevice

        device,
                                constVkPipelineLayoutCreateInfo*
    pCreateInfo,
                                constVkAllocationCallbacks*
                            pAllocator,
                                VkPipelineLayout*
                                                        pPipelin
eLayout);
```

这个函数使用 device 参数指定的设备，使用 pCreateInfo 指针所指向的 VkPipelineLayoutCreateInfo 类型的数据的信息，来创建新的 VkPipelineLayout 对象。
VkPipelineLayoutCreateInfo 定义如下：

```
        typedefstruct VkPipelineLayoutCreateInfo {
                                VkStructureType
                                          sType;
                                constvoid*
                                                      pNext;
                                VkPipelineLayoutCreateFlags
    flags;
                                uint32_t

setLayoutCount;
                                constVkDescriptorSetLayout*
   pSetLayouts;
                                uint32_t

pushConstantRangeCount;
                                constVkPushConstantRange*
 pPushConstantRanges;
        }VkPipelineLayoutCreateInfo;
```

VkPipelineLayoutCreateInfo 的 sType 域应置为 VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO，pNext 应置为 nullptr。

flags 域被当前版本 Vulkan 保留，应置为 0。描述符集合布局的数量（）通过 setLayoutCount 给定，pSetLayouts 是一个指向之前调用 vkCreateDescriptorSetLayout()创建的 VkDescriptorSetLayout 类型 handle 数组的指针。一次可绑定的描述符集合的最大数（）至少为 4.一些 Vulkan 实现也许会支持更大的个数。你可以通过调用 vkGetPhysicalDeviceProperties()来获取的 VkPhysicalDeviceLimits 类型的数据，检查它的 maxBoundDescriptorSets 来获知布局的最大个数。最后两个参数，pushConstantRangeCount 和 pPushConstantRanges，用来描述管线中用到的 pushconstants。Push Constants 是一种特殊的资源类型，它可以被当作着色器直接使用的常量。对 push constants 的更新非常快，并且无需同步。我们在稍后章节讨论 push constants。
当 VkDescriptorSetLayout 对象被创建了，一个管线内所有集合使用的资源都被集合了，并且需要符合设备自身的上限值。一个管线可访问的资源的个数和类型是有上限的。
可以通过调用 vkGetPhysicalDeviceProperties()得到设备的 VkPhysicalDeviceLimits 类型数据，
通过检查该数据中相关的成员来来获取到每一个上限值。
VkPhysicalDeviceLimits 和管线布局最大数相关的成员如下所列：

如果你的着色器程序或者生成的管线需要使用比上述规定最大值还大的资源数量，那么你需要检查资源上限，并准备当超过了是如何正确的处理失败。然而，如果你的资源需求在允许的范围内，那么便没有理由去检查 Vulkan 支持的最低标准。
一个描述符集合可以使用两个管线，如果这两个管线的布局是兼容的。对描述符集合兼容的两个管线布局，需符合：

•Use the same number of push constant ranges

•Use the same descriptor set layouts (or identical layouts) in the same order

如两个管线布局对于前面几个集合使用相同（）的集合布局，后面不相同，那这两个管线布局被认为是部分兼容的。在这种情况下，管线是兼容的，直至描述符集合布局不同的地方。
当管线被绑定到一个命令缓冲区时，它可继续使用一个管线布局内与集合绑定兼容的已绑定的描述符集合中任一个。且，交换两个部分兼容的管线无需重绑定任何集合到管线共享布局的点上。如果你有一系列的资源并向在全局访问，比如包含每帧所需常量的 uniform block，或者在每个着色器都需要访问的纹理，那就把这些资源放在第一个集合里。频繁被访问的资源可以放到高序号的集合里。
Listing 6.10 展示了创建描述符集合布局与描述该集合的管线布局创建所需的应用程序端代码：

Listing6.10: Creating a Pipeline Layout

```
// This describes our combinedimage-samplers. One set, two disjoint
```

```
bindings.
static const VkDescriptorSetLayoutBindingSamplers[] =
{
                {
                0,
                                                        // Start from
binding
                0
                VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,//
Combined imagesampler
                1,
                                                        // Create one
binding
                VK_SHADER_STAGE_ALL,            // Usable in all
                stages
                nullptr
                                        // Nostatic samplers
                },
                {
                2,
                                                        // Start from
binding
                2
                VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER,//
Combined imagesampler
                1,
                                                    //Create one
binding
                VK_SHADER_STAGE_ALL,                // Usable in all
                stages
                nullptr
                                        // Nostatic samplers
}
};
// This is our uniform block. One set, onebinding.
static const VkDescriptorSetLayoutBindingUniformBlock =
{
                0,// Start from binding
                0
                VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,// Uniform block
                1,// One binding
                VK_SHADER_STAGE_ALL,// All stages
                nullptr// No static samplers
};
```

```
// Now create the two descriptor setlayouts.
static constVkDescriptorSetLayoutCreateInfo createInfoSamplers =
{
                VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,
                nullptr,
                0,
                2,
                &Samplers[0]
};
static constVkDescriptorSetLayoutCreateInfo createInfoUniforms =
{
                VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,
                nullptr,
                0,
                1,
                &UniformBlock
};
// This array holds the two set layouts.
VkDescriptorSetLayout setLayouts[2];
vkCreateDescriptorSetLayout(device,&createInfoSamplers,

                                        nullptr,&setLaouts[0]);
vkCreateDescriptorSetLayout(device,&createInfoUniforms,

                                        nullptr,&setLayouts[1]);
// Now create the pipeline layout.
const VkPipelineLayoutCreateInfopipelineLayoutCreateInfo =
{
                VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO,nullpt
r,
                0,
                2,setLayouts,
                0,nullptr
};
VkPipelineLayout pipelineLayout;
vkCreatePipelineLayout(device,&pipelineLayoutCreateInfo,

                        nullptr,pipelineLayout);
```

在 Listing 6.10 中所创建的管线布局与 Listing 6.9 与 Listing 6.10 中着色器代码所期待的布局匹配。当我们创建一个使用该着色器的计算管线，我们把 Listing 6.10 创建的管线布局对象作为 VkComputePipelineCreateInfo 类型数据的 layout 域传递给 vkCreateComputePipelines()。当管线布局不再被需要

时，应该调用 vkDestroyPipelineLayout()来销毁它。这将释放管线布局对象关联的任何资源。vkDestroyPipelineLayout()原型如下：

```
voidvkDestroyPipelineLayout (
                                VkDevice
                                        device,
                                VkPipelineLayout
                        pipelineLayout,
                                constVkAllocationCallbacks*
pAllocator);
```

在管线布局对象被销毁后，它不应被再次使用了。然而，通过该管线布局对象创建的管线依然是有效的，直到它们被销毁。因此，没有必要在创建了管线之后依然让管线布局对象继续存在。可调用下面函数来销毁描述符集合布局对象并释放它的资源：

```
voidvkDestroyDescriptorSetLayout (
                                VkDevice
                                        device,
                                VkDescriptorSetLayout
        descriptorSetLayout,
                                constVkAllocationCallbacks*
pAllocator);
```

拥有该描述符集合布局的设备应通过 device 传递，描述符集合布局通过 descriptorSetLayout 传递。pAllocator 应指向一个主机内存分配器，和用来创建该描述符集合布局的内存分配器匹配，或当 vkCreateDescriptorSetLayout()的 pAllocator 为 nullptr 时这个 pAllocator 也为 nullptr。

6.5.2 绑定资源到描述符集合

资源是通过描述符表示的，被绑定到管线上--首先把描述符绑定到集合上，然后把描述符集合绑定到管线。这允许花费很少时间消耗就能绑定一个很大的资源，因为被特定绘制命令使用的资源的集合可以提前就指定了，并提前创建好存放它们的描述符集合。描述符是从“描述符池”分配出来的。因为对不同资源类型的描述符都拥有相同的数据结构--不管在那种 Vulkan 实现里。以缓存池的方式类存储描述符让驱动可以更加高效的使用内存。可以使用 vkCreateDescriptorPool()来创建描述符缓存池，其原型为：

```
VkResultvkCreateDescriptorPool (
VkDevice
                                                        devi
ce,
constVkDescriptorPoolCreateInfo*            pCreateI
nfo,
constVkAllocationCallbacks*
          pAllocator,
```

```
                VkDescriptorPool*
                                        pDescriptorPool);
```
创建描述符缓存池的设备通过 device 指定，剩下的参数描述了新的缓存池，通过 VkDescriptorPoolCreateInfo 类型的 pCreateInfo 来传递。
VkDescriptorPoolCreateInfo 原型如下：
```
                typedefstruct VkDescriptorPoolCreateInfo {
                                        VkStructureType
                                                  sType;
                                        constvoid*
                                                                  pNext;
                                        VkDescriptorPoolCreateFlags
          flags;
                                        uint32_t

maxSets;
                                        uint32_t

poolSizeCount;
                                        constVkDescriptorPoolSize*
                                    pPoolSizes;
                }VkDescriptorPoolCreateInfo;
```
VkDescriptorPoolCreateInfo 的 sType 域应置为 VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO，pNext 应置为 nullptr。flags 用来传递关于分配策略的附加的信息，这个策略用在管理资源池的。当前唯一已定义的标志位是 VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT，它表示应用程序可以释放从池子中获取的单个描述符，所以应该为此准备好内存分配器。如果你不打算把单个描述符还给缓存池，把 flags 置为 0 即可。
maxSets 域指定了可从池中分配出集合的数量的最大值。注意，这是集合的总个数，无关每个集合的大小或者池子的总大小。下两个域，poolSizeCount 和 pPoolSize，指定了存储在集合中每种类型资源描述的个数。pPoolSize 是一个指向大小为 poolSizeCount 的 VkDescriptorPoolSize 类型的数组的指针，数组每一个元素指定了可以从池子中分配出的一种类型描述符的个数。
VkDescriptorPoolSize 定义如下：
```
                typedefstruct VkDescriptorPoolSize {
VkDescriptorType                        type;
uint32_t                                                        descr
iptorCount;
} VkDescriptorPoolSize;
```
VkDescriptorPoolSize 的第一个域，type，指定了资源的类型，第二个域 descriptorCount 指定了在池子中该种类型资源的个数。type 是 VkDescriptorType 枚举的一个枚举值。如果 pPoolSizes 数组中没有元素指定了一个特殊类型的资源，那么就没有那种类型的描述符可以从池子重分配出来。如果一个特定类型的资源在数组中出现两次，那么他们两个的

descriptorCount 域之和指定了缓存池中该类型资源的个数。当资源被分配出去后，资源池中的资源个数就相应的减少。
如果成功的创建了缓存池，那么一个新的 VkDescriptorPool 类型的对象的 handle 就会被写入到 pDescriptorPool 所指的变量里。为了从缓存池中分配多块描述符，我们可以调用 vkAllocateDescriptorSets()来创建描述符集合对象，该函数原型如下：

```
VkResultvkAllocateDescriptorSets (
VkDevice
                                        device,
const VkDescriptorSetAllocateInfo*          pAllocateInfo,
VkDescriptorSet*
                          pDescriptorSets);
```

拥有该描述符缓存池并可以从其分配出集合的设备通过 device 传递。剩下的关于集合分配的信息通过一个 VkDescriptorSetAllocateInfo 类型数据的指针 pDescriptorSets 来传递。VkDescriptorSetAllocateInfo 的定义如下：

```
typedefstruct VkDescriptorSetAllocateInfo {
VkStructureType sType;
const void* pNext;
VkDescriptorPool descriptorPool;
uint32_t descriptorSetCount;
const VkDescriptorSetLayout* pSetLayouts;
} VkDescriptorSetAllocateInfo;
```

VkDescriptorSetAllocateInfo 的 sType 域应该置为 VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO，pNext 应置为 nullptr。可以分配出集合的描述符缓存池的 handle 通过 descriptorPool 指定，这个 handle 通过调用 vkCreateDescriptorPool()来产生。对于 descriptorPool 的访问应该在外部保持同步。创建的集合的个数通过 descriptorSetCount 指定。每一个集合的布局通过 VkDescriptorSetLayout 类型数组 pSetLayouts 中每一个元素来传递。我们在之前讲过如何创建 VkDescriptorSetLayout 类型对象。当调用成功，vkAllocateDescriptorSets()从指定的缓存池消耗集合和描述符，把新的描述符集合存储到一个 pDescriptorSets 指向的数组中。对于每一个描述符集合从缓存池中消耗的描述符的个数由描述符集合布局的 pSetLayouts 决定，我们在之前讲过如何创建该布局。
如果创建描述符缓存池是带有的 VkDescriptorSetCreateInfo 类型参数包含 VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT 标志位，那么描述符集合可以通过被释放得以返回缓存池。可调用 vkFreeDescriptorSets()来释放一个或者多个描述符集合，其原型如下：

```
VkResultvkFreeDescriptorSets (
VkDevice device,
VkDescriptorPool                                        descriptorP
ool,
uint32_t descriptorSetCount,
const VkDescriptorSet* pDescriptorSets);
```

拥有该描述符缓存池的设备通过 device 指定，描述符集合需要返回的缓存池通过 descriptorPool 指定。对于 descriptorPool 的访问需要在外部同步。需释放的描述符集合的个数通过 descriptorSetCount 传递，pDescriptorSets 指向一个 VkDescriptorSet 类型的数组，每个元素都是需要被释放的对象。当描述符集合被释放，他们的资源都被返回到它们所来自的那个缓存池，这些资源将来可能被分配到一个新的集合。
即使在创建描述符缓存池时 VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT 没有被指定，也可以把所有集合的资源回收到它们所分配出来的缓存池。可以调用 vkResetDescriptorPool()函数来做到重置缓存池。对于这个命令，有必要显式的指定每一个才哦你缓存池中分配出来的集合。函数原型如下：

```
VkResultvkResetDescriptorPool (
VkDevice device,
VkDescriptorPool descriptorPool,
VkDescriptorPoolResetFlags flags);
```

Device 是拥有该描述符缓存池的设备的 handle，descriptorPool 是需要被重置的描述符缓存池的 handle。对于描述符缓存池的访问需要在外部同步。
Flags 被保留使用，应置为 0。
不管集合是通过调用 vkFreeDescriptorSets()单独的释放，还是通过调用 vkResetDescriptorPool()来整块的释放，必须要注意保证在释放这些集合后不再引用它们。特别是，任何包含可能引用将被释放的到描述符集合的命令的命令缓冲区，应该已经完成了执行或者丢弃未提交的命令。
为了释放和资源关联的描述符缓存池，你应该通过调用 vkDestroyDescriptorPool()来销毁，其原型如下：

```
voidvkDestroyDescriptorPool(
VkDevice device,
VkDescriptorPool descriptorPool,
const VkAllocationCallbacks* pAllocator);
```

拥有该缓存池的设备的 handle 通过 device 传递，需要被销毁的缓存池的 handle 通过 descriptorPool 传递。pAllocator 指向一个主机端内存分配器，它应和创建缓存池时使用的分配器相匹配，或者如果 vkCreateDescriptorPool()的 pAllocator 参数为 nullptr 时就设置为 nullptr。
当描述符缓存池被销毁了，它里面所有的资源也被释放了，包含从它分配出来的任何集合。在调用 vkResetDescriptorPool()来销毁缓存池之前，没有必要显式的释放从缓存池分配出来的描述符集合。然而，当描述符集合被显示释放，你必须保证你的应用程序在缓存池被销毁后不会再次访问从缓存池中分配出来的描述符集合。这包含设备命令缓冲区中已提交但未完成的正在执行的任务。
我们可以直接写入描述符集合或者从其他一个描述符集合复制绑定，来把自然绑定到描述符集合。这种情况下，我们可以使用 vkUpdateDescriptorSets()命令，其原型如下：

```
voidvkUpdateDescriptorSets (
VkDevice device,
```

```
uint32_t descriptorWriteCount,
const VkWriteDescriptorSet* pDescriptorWrites,
uint32_t descriptorCopyCount,
const VkCopyDescriptorSet* pDescriptorCopies);
```
拥有需要被更新的描述符集合的设备通过 device 传递。直接写入的个数通过 descriptorWriteCount 传入，描述符复制的个数通过 descriptorCopyCount 传入。每一个写入的参数都包含在一个 VkWriteDescriptorSet 类型的数据里，每一个复制的参数包含在一个 VkCopyDescriptorSet 类型的数据。
pDescriptorWrites 和 pDescriptorCopies 参数包含

Vulkan 编程指南翻译 第六章 着色器和管线 第 5 节 第 6 段 图像采样

翻译 2017 年 03 月 11 日 18:48:50

- 171
- 0
- 1

这一节实在太长了，分篇来翻译。
Sampled Images

当着色器从图像读数据时，他们可以可以使用两种方式。第一种是原生加载，从图像的指定位置直接读取格式化的或非格式化的数据，第二种是使用采样器对图像采样。采样包括：在图像坐标空间做基础变形或者纹素过滤来光滑图像返回到着色器。
采样器的状态通过一个采样器对象表示，它可像图像或者缓冲区一样被绑定到描述符集合。可调用函数 vkCreateSampler()来创建一个采样器对象，其原型如下：

```
VkResultvkCreateSampler (
VkDevice
                                device,
const VkSamplerCreateInfo*                  pCreateInfo,
const VkAllocationCallbacks*                pAllocator,
VkSampler*
                        pSampler);
```

将创建采样器的设备通过 device 传递，采样器剩下的参数通过一个 VkSamplerCreateInfo 类型数据的指针 pCreateInfo 传递。一个设备上可以创建的采样器个数的上限取决于与 Vulkan 实现。可以保证的是至少有 4000 个。如果你的应用程序有可能创建超过这个数量的采样器，那么你需要检查设备本身支持创建多少个采样器。一个设备可以管理的采样器个数包含在 VkPhysicalDeviceLimits 类型数据的 maxSamplerAllocationCount 域，可调用 vkGetPhysicalDeviceProperties()函数来获取 VkPhysicalDeviceLimits 数据。VkSamplerCreateInfo 原型如下：

```
typedefstruct VkSamplerCreateInfo {
VkStructureType sType;
const void* pNext;
VkSamplerCreateFlags flags;
```

```
VkFilter magFilter;
VkFilter minFilter;
VkSamplerMipmapMode mipmapMode;
VkSamplerAddressMode addressModeU;
VkSamplerAddressMode addressModeV;
VkSamplerAddressMode addressModeW;
float mipLodBias;
VkBool32 anisotropyEnable;
float maxAnisotropy;
VkBool32 compareEnable;
VkCompareOp compareOp;
float minLod;
float maxLod;
VkBorderColor borderColor;
VkBool32 unnormalizedCoordinates;
} VkSamplerCreateInfo;
```

VkSamplerCreateInfo 的 sType 域应置为 VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO，pNext 应置为 nullptr。Flags 域被保留应置为 0。

图像过滤

magFilter 和 minFilter 域指定了图像在被放大或缩小时将被使用的过滤模式。图像是放大还是缩小是通过比较被计算的相邻像素间采样坐标来决定。如果采样坐标的梯度大于 1，那么图像是被缩小的。否则，它是被放大的。

magFilter 和 minFilter 是 VkFilter 枚举的成员。VkFilter 的成员如下：

• VK_FILTER_NEAREST: When sampling, the nearest texel in the image ischosen and returned
directly to the shader.
• VK_FILTER_LINEAR: A 2 × 2 footprint containing the texelcoordinates is used to produce a
weighted average of four texels, and thisaverage is returned to the shader

VK_FILTER_NEAREST 模式将使 Vulkan 在从一张图像采样时简单选择指定坐标临近的纹素。在很多情况下，这就导致 blocky 喝着带锯齿的图像，导致渲染出来的图像闪烁。VK_FILTER_LINEAR 告诉 Vulkan 当采样时对图像采用线性过滤。当你正在使用线性过滤来处理图像时，需要获取的采样点也许在 1D 纹理的两个纹素中间，或者 2D 纹理的 4 个纹素中间。Vulkan 将从周围的纹素读取数据，然后基于到每个点的距离给予不同的权重计算出结果。这在 Figure 6.2 中给出了图示，采样点在 x，它处于四个纹素 A，B，C 和 D 的中间。尽管纹理坐标的整数部分是{u, v}，纹理坐标的小数部分是{α, β}。

要获取纹素 A、B 的线性权重之和，可按照下面的关系来简单的计算

$$Tu0 = \alpha A + (1 - \alpha) B$$

这也可以写成

Tu0 = B + α (B−A)

同理，C 与 D 的权重之和通过下面的公式来算

Tu1 = αC + (1 −α) D or

Tu1 = D− α (D−C)

然后，两个临时量 Tu0 and Tu1 可被结合来算单一一个权重之和，结合 β 和相似的机制

T = βTu0 + (1 −β) Tu1 or

T = Tu1 + β(Tu1 −Tu0)

这可以拓展到任何维数，尽管在 Vulkan 中纹理的维数最多只有 3。

Mipmapping

mipmapMode 域指定了当图像被采样时使用多少层 mipmaps。其值是 VkSamplerMipmapMode 枚举的一个成员，枚举成员如下：

• VK_SAMPLER_MIPMAP_MODE_NEAREST: The computed level-of-detail isrounded to the
nearest integer, and that level is used toselect the mipmap level. If sampling from the base
level, then the filtering mode specified inmagFilter is used to sample from that level;
otherwise, the minFilter filter is used.

• VK_SAMPLER_MIPMAP_MODE_LINEAR: The computed level of detail isrounded both up
and down, and the two resulting mipmaplevels are sampled. The two resulting texel values are
then blended and returned to the shader.

要从一个图像中选择 mipmap，Vulkan 将计算纹理中被采样的坐标的导数。关于数学的部分在 Vulkan 规范里有讲述。简单来说，被选择的 level 是纹理坐标维数中导数最大的一个做 $\log_2$ 的结果。这个 level 可以通过采样器或者着色器的参数来调整，或者完全在着色器里指定。不管源是什么，结果可能不是整数。

当 mipmap 模式是 VK_SAMPLER_MIPMAP_MODE_NEAREST，被选择的 mipmap level 是简单的取其整数部分，然后那单一一层被采样，尽管只是一层图像。当 mipmap 模式是 VK_SAMPLER_MIPMAP_MODE_LINEAR 时，采样点从邻近的上一层和下一层取得，使用 minFilter 域指定的过滤模式，然后这两个采样点依据其权重来合称为一个点，和前面讲述的线性采样相近。

注意，过滤模式只适用于缩小，亦即此过程在 mipmap 层采样而并不在原图像采样。到纹理坐标导数的 $\log_2$ 比 1 小时，那么第 0 层就被使用，所以只有一层

能够被采样使用了。这就是放大，使用 magFilter 指定的过滤模式来从原图像中采样。
VkSamplerCreateInfo 中接下来的三个域 addressModeU,addressModeV, 和 addressModeW，用来指定纹理坐标系之外采样时选用的变形。可选的模式如下：

• VK_SAMPLER_ADDRESS_MODE_REPEAT: As the texture coordinateprogresses from 0.0 to
1.0 and beyond, it is wrapped back to 0.0,effectively using only the fractional part of the
coordinate to sample from the image. Theeffect is to tile the image indefinitely.
• VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT: The texture coordinateprogresses
from 0.0 to 1.0 as normal, and then in therange 1.0 to 2.0, the fractional part is subtracted from
1.0 to form a new coordinate moving backtoward 0.0. The effect is to alternately tile the normal
and mirror-image version of a texture.
• VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE: Texture coordinates beyond1.0 are
clamped to 1.0, and negative coordinatesare clamped to 0.0. This clamped coordinate is used to
sample from the image. The effect is thatthe texels along the edge of the image are used to fill
any area that would be sampled outside theimage.
• VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER: Sampling from the textureoutside
its bounds will result in texels of theborder color, as specified in the borderColor field,
being returned rather than data from theimage.
• VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE: This is a hybrid modethat
first applies a single mirroring of thetexture coordinate and then behaves like
VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE.

在 Figure 6.3 展示了每一种应用到图像的采样器寻址模式的效果。在图中，左上角的图像展示了 VK_SAMPLER_ADDRESS_MODE_REPEAT 寻址模式。你可以看到，在帧之间纹理只是简单的重复。右上角的图像展示 VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT 模式的结果。每一个重复都会在 X 或 Y 方向上镜像。
Figure6.3: Effect of Sampling Modes

左下角的图像表示对纹理使用了 VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE 寻址模式。这里，最后一行或者一列的像素在采样坐标离开纹理范围后被无限重复。最后，右下角的图像展示了 VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER 模式的结果。这个纹理在创建时使用了黑色的边框，所以，在纹理之外的部分就是空白的，但是 Vulkan 会从黑色采样。这允许你可以看清原来的纹理。
当过滤模式是 VK_FILTER_LINEAR，wrapping 或者 clamping 纹理坐标被应用到每一个在 2×2 范围内生成的用来产生纹素的坐标。结果就是图像被 wrapped 时过滤也被应用上了。
对于 VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER 过滤模式，当纹素是从边界采样时（），边界的颜色被替换了而非从图像中读取。被使用的颜色取决于 borderColor 域的值。这不是一个完整的颜色规范，只是 VkBorderColor 枚举类型的一个成员，它允许一个小的、预定义的颜色集合可选。它们是：

• VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK: Returns floating-pointzeros to the
shader in all channels
• VK_BORDER_COLOR_INT_TRANSPARENT_BLACK: Returns integer zeros to theshader in
all channels
• VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK: Returns floating-point zeros inthe R, G,
and B channels, and floating-point one in A
• VK_BORDER_COLOR_INT_OPAQUE_BLACK: Returns integer zeros in R, G,and B, and
integer one in A
• VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE: Returns floating-point ones tothe shader in
all channels
• VK_BORDER_COLOR_INT_OPAQUE_WHITE: Returns integer ones to theshader in all
Channels

VkSamplerCreateInfo 的 mipLodBias 域指定了在被添加到选择 mipmap 之前计算的 level ofdetail 的一个浮点差值。这允许你你把在 mipmap 链中把 level of detail 调高或调低，使生成的被过滤的纹理看起来更加锐利或者模糊。
如果你想要使用各向异性过滤，设置 anisotropyEnable 为 VK_TRUE。各向异性过滤的细节是依赖于 Vulkan 实现的。各向异性管理通常在投影范围内做采样，而不是在固定的 2×2 范围内。通过在范围内使用更多的采样点来逼近范围内采样。因为采样的个数可能非常大，各向异性过滤可能对性能产生负面影响。同样，在一些极端情况下，投影的范围可能非常大，这会导致影响到很大区域和模糊的过滤效果。为了限制这些效果，你可以对各向异性的范围做剪裁，设置 maxAnisotropy 在 1.0 和设备允许的最大值之间。可调用 vkGetPhysicalDeviceProperties()并检查 VkPhysicalDeviceLimits 类型数据的 maxSamplerAnisotropy 域来获知该最大值。

当采样子用于深度图像时，可以配置来进行比较操作并返回比较的结果而不是存储在图像原生的值。当这种模式启用时，从图像中取出的采样点之间进行比较，结果便是通过采样后数据的分数部分。这可用来实现一个名为 spercentage closer filtering, or PCF 的技术。要开启这个模式，需设置 compareEnable 为 VK_TRUE，并给 compareOp 设置比较操作。
compareOp 是 VkCompareOp 枚举的一个成员，该枚举在 Vulkan 中用于很多地方。如将在第七章“图形管线”中见到，这个枚举会被用于指定深度测试操作。可用的操作和它们在着色器访问深度资源时的语境在 Table 6.2 中展示。
采样器可被配置来限制在一个带有 mipmaps 的图像 mip levels 中一个子集。被限制的 mipmap 范围通过 minLod and maxLod 指定，它俩包含了应被采样的最低（最高分辨率）和最高（最低分辨率）的 mipmaps。要在整个 mipmaps 链上做采样，设置 minLod to 0.0, 并设置 maxLod 为高到不能再进行 clamp 的 level of detail 层。
最后，unnormalizedCoordinates 是一个标志位，当被设置为 VK_TRUE 时，表示图像被用于采样的坐标是以原生纹素为单位。而不是从 0.0 到 1.0 归一化的值。这允许可从图像里显式的取出纹素。然而，这种模式存在几个限制。当 unnormalizedCoordinates 是 VK_TRUE 时，minFilter 和 magFilter 必须是相同的，mipmapMode 必须是 VK_SAMPLER_MIPMAP_MODE_NEAREST，anisotropyEnable 必须是 VK_FALSE。
当你使用完了采样器，你应该调用 vkDestroySampler()销毁它，其原型如下：

```
voidvkDestroySampler (
VkDevice device,
VkSampler sampler,
const VkAllocationCallbacks* pAllocator);
```

device 是拥有该采样器对象的设备，sampler 是需被销毁的采样器对象。如果在创建采样器对象时使用了主机内存分配器，一个兼容的内存分配器也需要通过 pAllocator 参数传入。否则，pAllocator 应置为 nullptr。

总结

本章覆盖了 Vulkan 支持的着色语言的基础、SPIR-V，包含了如何 Vulkan 如何接收 SPIR-V 着色器 module，包含这些着色器的管线如何构造。你看到可如何狗仔计算着色器并用它来创建计算管线，如何分发工作到管线，如何让管线访问资源以产生数据。在接下来的章节里，我们将继续深入管线的概念来创建拥有多阶段的管线对象并使用更多的高级特性。

Vulkan 编程指南翻译 第七章 图形管线 第 1 节 逻辑图形管线

翻译 2017 年 03 月 12 日 21:47:37

- 181
- 0
- 2

第七章 图形管线

在本章你将学到
图形管线是什么样子的

如何创建图形管线对象
如何使用 Vulkan 绘制图元

或许 Vulkan 最常被当作图形管线 API 来使用。图形是 Vulkan 和驱动最基础的功能，是任何视觉效果程序的核心。Vulkan 中图形处理可以被看作是管线，管线接受多个贯穿多个阶段图形命令，并在显示器上产生图像。本章将覆盖 Vulkan 中图形管线的基础并介绍第一个管线的例子。

7.1 逻辑图形管线

Vulkan 中图形管线可被看作为一条生产线，命令进入管线的前端，在多个阶段中被处理。每一个阶段执行一种转变，接受命令和它们关联的数据并把它们转化为其他的东西。在管线的末端，命令经过转换变位多彩的像素组成了最终输出的图片。
图形管线的很多部分是可选的，可以被禁用，甚至不被 Vulkan 实现所支持。管线中只有一个部分必须要在应用程序中开启的 顶点着色器。完整的 Vulkan 图形管线在 Figure 7.1 中展示了。然而，不要惊慌，我们在本章中慢慢的介绍每一个阶段，在本书的后续部分再深入的挖掘更多细节。

Figure 7.1: The Full Vulkan Graphics Pipeline

下面是管线的每一个管线的简单介绍和它所做的工作
• 绘制：这是命令进入 Vulkan 图形管线的地方。通常，Vulkan 设备里一个很小的处理器或者专用的硬件对命令缓冲区中的命令做解释，并直接和硬件交互来完成工作
• 输入组装：这个阶段读取索引缓冲区和顶点缓冲区，它俩包含了组成你想要绘制的图形的顶点信息。
• 顶点着色器：这是顶点着色器执行的地方。它接受作为输入的属性顶点，并为下一级准备变换和处理的顶点数据。
• 细分控制着色器：这个可编程的着色阶段负责生产细分因子，和被固定细分管线引擎使用的其他 per-patch 数据。
• 细分图元生成：在 Figure 7.1 中没有展示，这个固定功能管线使用在锌粉控制着色器产生的细分因子来吧 patch 图元分解成许多更小的，简单的图元以供细分求值着色器使用。
• 细分其中着色器：这个着色器阶段在细分图元生成器产生的每一个顶点上工作。它和顶点着色器的操作类似--除了顶点来自生成的而非内存。
• 几何着色器：这个着色阶段在所有的图元上运行。图元可能是：点，直线或者三角形，或它们的变种（在周围有附件顶点）。这个阶段也有改变图元类型的能力。
• 图元组装：这个阶段把顶点、细分或几何阶段的顶点分组，组成图元以供光栅化。它也剔除或剪裁图元并把图元变形到合适的视口。
• 裁剪和提出：这个固定功能阶段决定了图元的那一部分可能组成输出图像的一部分并抛弃那些不组成图像的部分，把可见的图元发送给光栅器。

• 光栅器：光栅器是 Vulkan 所有的图形的基础核心。光栅器接受被组装的图元（仍然用一系列顶点表示），并把它们变成单独的片元，片元将变成组装图像的像素。
• 片元操作：在计算之前一旦片元的位置被知晓，就可以在它上面进行好几个操作。这些操作包括深度和 stencil 测试（当开启了这两个测试时）。
• 片元组装：在图中并没有展示，片元组装接受光栅器的输出的逐片元的数据，形成一组，向片元着色阶段传递。
• 片元着色器：这个阶段在管线的最后运行，负责计算最终将送往最后固定功能处理阶段的数据。
• 后片元操作：在一些环境中，片元着色器将修改进入片元操作之前的数据。这些情况下，这些操作转移到后片元阶段并在此执行。
• 颜色混合：颜色操作接收片元着色器和后片元操作的结果，并把它们更新到 framebuffer。颜色操作包括混合与逻辑操作。

如你所知，　在图形管线中有很多关联的阶段。不像第六章“着色器与管线”所介绍的计算管线，图形管线不仅包含很大一部分固定管线的配置，也包括最到五个着色器阶段。更有，取决于 Vulkan 实现，一些逻辑上的固定功能阶段实际上被驱动生成的着色器代码所示现。
在 Vulkan 中把图形管线当作一个对象的目的是给 Vulkan 实现提供尽量多的信息把固定管线的部分转移到可编程管线着色器核心。如果在同一个对象中这些信息无法提供，就表示一个 Vulkan 实现也许需要根据可配置状态重新编译着色器。包含在图形管线中的状态的集合已经被仔细的选择出来防止这样的状况，让状态的切换尽量的快。
Vulkan 中绘制的基础单元是顶点。顶点可以组成图元并被 Vulkan 管线处理。Vulkan 中最简单的绘制命令是 vkCmdDraw()，原型如下：

```
void vkCmdDraw (
VkCommandBuffer commandBuffer,
uint32_t vertexCount,
uint32_t instanceCount,
uint32_t firstVertex,
uint32_t firstInstance);
```

就像其他的 Vulkan 命令，vkCmdDraw()在即将被设备执行的命令缓冲区后面追加一个命令。这个被追加的命令缓冲区通过 commandBuffer 指定。附加到管线的顶点的数量通过 vertexCount 指定。如果你想要一遍又一遍的重复绘制一个集合的顶点，只有一些参数稍稍不同，你可通过 instanceCount 指定实例的个数。这就是几何体实例化，我们在本章后面讲解。现在，我们把它设置为 1 即可。也可以从非 0 的顶点或者实例开始绘制。要想这样，需要使用 firstVertex 和 firstInstance。同样，在后面讲解。现在，把这两个参数都设置为 0.
在绘制任何东西之前，你必须要把一个管线绑定到命令缓冲区，在此之前，你需要创建管线。如果没有绑定管线就试图绘制，会产生未知的后果（通常是糟糕的后果）。

当你调用 vkCmdDraw()时，vertexCount 数量的顶点被产生并送入当前的 Vulkan 绘制管线。对于每一个顶点，在顶点着色器之前都执行输入组装。Declaring inputs beyond what is provided for you by Vulkan is optional, but having a vertex
shader is not.故，最简单的图形管线只有顶点着色器即可。

Vulkan 编程指南翻译 第七章 图形管线 第 2 节 Renderpasses（未完成）

翻译 2017 年 03 月 22 日 00:53:46

- 427
- 0
- 1

7.2 renderpass

Vulkan 图形管线和计算管线的区别之一是，你使用图形管线来渲染出像素，组成图像以供处理或显示给用户。在复杂的图形应用程序中，图片经过很多遍构建，每一遍都生成场景的一部分，应用全帧效果如后期处理、合成、渲染用户界面元素等等。

这样的一遍可以使用 Vulkan 中 renderpass 对象表示。一个单一的的 renderpass 对象封装了多个 pass 或者一系列最终图像的几个渲染阶段，renderpass 对象包含输出图像所需的信息。

所有的绘制必须被包含在一个 renderpass 中。甚至，图形管线需要知道他们把渲染结果发往哪儿，因此，有必要在创建图形管线之前创建一个 renderpass 对象，一边我们可以告诉正在生成图像的管线有关图像的信息。在第十三章将会深度的讲解 renderpass。在本章，我们创建一个最简单的 renderpass 对象，我们可以把渲染结果写到一张图像。

可调用 vkCreateRenderPass()来创建 renderpass 对象，其原型如下：

```
VkResult vkCreateRenderPass (
VkDevice device,
const VkRenderPassCreateInfo* pCreateInfo,
const VkAllocationCallbacks* pAllocator,
VkRenderPass* pRenderPass);
```

vkCreateRenderPass()的 device 参数指定了将要创建 renderpass 对象的设备，pCreateInfo 指向了一个定义 renderpass 的结构。它是 VkRenderPassCreateInfo 类型的一个实例，定义如下：

```
typedef struct VkRenderPassCreateInfo {
VkStructureType sType;
const void* pNext;
VkRenderPassCreateFlags flags;
uint32_t attachmentCount;
const VkAttachmentDescription* pAttachments;
uint32_t subpassCount;
const VkSubpassDescription* pSubpasses;
uint32_t dependencyCount;
const VkSubpassDependency* pDependencies;
} VkRenderPassCreateInfo;
```

VkRenderPassCreateInfo 的 sType 应置为 VK_STRUCTURE_TYPE_RENDERPASS_CREATE_INFO，pNext 应置为 nullptr。flags 域被保留使用，应置为 0。pAttachments 是一个指向大小为 attachmentCount 的 VkAttachmentDescription 类型数组的指针，数组定义了和 renderpass 关联的多个附件。数组中的每一个元素定义了在一个 renderpass 中多个 subpasses 里一张用作输入、输出或者二者兼具的图像。如果真的没有和 renderpass 关联的附件，你可以设置 attachmentCount 为 0，pAttachments 为 nullptr。然而，除了一些高级使用场景，几乎所有的图形渲染都将使用至少一个附件。
VkAttachmentDescription 定义是：

```
typedef struct VkAttachmentDescription {
VkAttachmentDescriptionFlags flags;
VkFormat format;
VkSampleCountFlagBits samples;
VkAttachmentLoadOp loadOp;
VkAttachmentStoreOp storeOp;
VkAttachmentLoadOp stencilLoadOp;
VkAttachmentStoreOp stencilStoreOp;
VkImageLayout initialLayout;
VkImageLayout finalLayout;
} VkAttachmentDescription;
```

flags 用来给 Vulkan 提供关于附件的附加信息。当前只有一个定义 VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT，如果被设置了这个值，表示附件可能被同一个 renderpass 引用的其他附件共享一块内存。这告诉 Vulkan 不要对做任何可能导致附件的数据不一致的事情。这个标志位可以用于一些高级场景，如内存很稀少，你视图优化它的使用。绝大多数场景下，flags 可以设置为 0。
format 域指定了附件的格式。这是 VkFormat 枚举类型的一个成员，应该和用作附件的图像的格式相同。同样，samples 表示图像的采样的个数，被用于多重采样。当不使用多重采样时，把 samples 设置为 VK_SAMPLE_COUNT_1_BIT。
接下来的四个域指定了在 renderpass 的开始与结束时如何处理附件。加载操作告诉 Vulkan 在 renderpass 开始时如何处理附件。可以设置为如下值：

- VK_ATTACHMENT_LOAD_OP_LOAD indicates that the attachment has data in it already and
that you want to keep rendering to it. This causes Vulkan to treat the contents of the attachment
as valid when the renderpass begins.
- VK_ATTACHMENT_LOAD_OP_CLEAR indicates that you want Vulkan to clear the attachment
for you when the renderpass begins. The color to which you want to clear the attachments is
specified when the renderpass has begun.
- VK_ATTACHMENT_LOAD_OP_DONT_CARE indicates that you don’t care about the content of

the attachment at the beginning of the renderpass and that Vulkan is free to do whatever it
wishes with it. You can use this if you plan to explicitly clear the attachment or if you know that
you’ll replace the content of the attachment inside the renderpass.

同样，存储操作告诉 Vulkan 在 renderpass 结束时如何处理附件的内容。可以设置为如下值：
- VK_ATTACHMENT_STORE_OP_STORE indicates that you want Vulkan to keep the contents
of the attachment for later use, which usually means that it should write them out into memory.
This is usually the case for images you want to display to the user, read from later, or use as an
attachment in another renderpass (with the VK_ATTACHMENT_LOAD_OP_LOAD load
operation).
- VK_ATTACHMENT_STORE_OP_DONT_CARE indicates that you don’t need the content after
the renderpass has ended. This is normally used for intermediate storage or for the depth or
stencil buffers

如果附件是一个被绑定的 depth-stenci 附件，那么 stencilLoadOp 和 stencilStoreOp 与告诉 Vulkan 如何处理附件的 stencil 部分（常规的 loadOp 和 storeOp 域指定了如何处理附件等哦 depth 部分），和 depth 部分不同。initialLayout 和 finalLayout 域告诉 Vulkan 期待的 image 在 renderpass 开始时域结束时处于哪种布局。注意，renderpass 对象不会自动把图像转变到初始布局。这个布局是当 renderpass 使用它时期待的布局。尽管，renderpass 结束时会改变图像的最终布局。
你可以使用屏障，显式的把图像从一个布局转移到另外一个布局，但是，可能的话，最好尝试在 renderpass 内部转移布局。这可以让 Vulkan 有很好的机会可以在其他渲染工作进行时并行的改变图像的布局。这个域的高级使用方式和 renderpass 在第十三章讲解。
在你定义了将在 renderpass 中使用的所有的附件后，你需要定义所有的 subpasses。每一个 subpass 都引用了一些数量的附件（）作为输入或者输出。这些描述通过一个 VkSubpassDescription 类型的数组描述，每一个元素对应 renderpass 中的一个 subpass。VkSubpassDescription 定义如下：
```
typedef struct VkSubpassDescription {
VkSubpassDescriptionFlags flags;
VkPipelineBindPoint pipelineBindPoint;
uint32_t inputAttachmentCount;
const VkAttachmentReference* pInputAttachments;
uint32_t colorAttachmentCount;
```

```
const VkAttachmentReference* pColorAttachments;
const VkAttachmentReference* pResolveAttachments;
const VkAttachmentReference* pDepthStencilAttachment;
uint32_t preserveAttachmentCount;
const uint32_t* pPreserveAttachments;
} VkSubpassDescription;
```

Vulkan 编程指南翻译 第七章 图形管线 第 3 节 帧缓冲区

翻译 2017 年 03 月 16 日 19:56:12

- 198
- 0
- 2

7.3 Framebuffer

帧缓冲区是一个用来表示一些存储渲染结果的图像的对象。这影响管线最后几个阶段：深度和 stencil 测试，混合，logic 操作，多采样等等。一个帧缓冲区对象通过使用 renderpass 的引用来创建，可以和多个有类似附件排放的 renderpass 一同使用。调用 vkCreateFramebuffer()可创建一个帧缓冲区对象，其原型如下：

```
VkResultvkCreateFramebuffer (
VkDevice
                                                device,
const VkFramebufferCreateInfo*                   pCreateInfo,
const
VkAllocationCallbacks*                                    pAllocator,
VkFramebuffer*
                              pFramebuffer);
```

将用来创建帧缓冲区的设备通过 device 传递，剩下的参数描述了新创建的哦帧缓冲区对象，通过一个 VkFramebufferCreateInfo 类型的数据 pCreateInfo 来传递。VkFramebufferCreateInfo 的定义：

```
typedefstruct VkFramebufferCreateInfo {
VkStructureType                                             sType;
const
void*
pNext;
VkFramebufferCreateFlags      flags;
VkRenderPass                                                  ren
derPass;
uint32_t
              attachmentCount;
const VkImageView*                          pAttachments;
uint32_t
              width;
```

```
uint32_t
                height;
uint32_t
                layers;
}VkFramebufferCreateInfo;
```
VkFramebufferCreateInfo 的 sType 域应被置为 VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO， pNext 应置为 nullptr。flags 被保留使用，应置为 0。
和被创建的帧缓冲区兼容 renderpass 对象的 handle 通过 renderPass 传递。为了和帧缓冲区对象保持兼容，若两个 renderpass 的附件相同，它们也应保持兼容。
和帧缓冲区对象绑定的一系列图像通过一个 VkImageView 类型数组的指针 pAttachments 传递。pAttachments 数组的长度通过 attachmentCount 指定。组成 renderpass 的 pass 引用图像附件，这些引用通过 pAttachments 数组的索引来指定。如果你知道特定的 renderpass 不使用某一些附件，但是你想要帧缓冲区和多个 renderpass 对象保持兼容或者和应用程序中的图像的布局保持一致，pAttachments 中的一些图像 handle 可以是 VkNullHandle。
尽管帧缓冲区中每一张图像都有原生的宽度、高度，和（array 图像意义上）层数，你必须指定帧缓冲区的位数。这些维数通过 VkFramebufferCreateInfo 类型数据的 width, height,and layers 域传递。渲染到一些图像外部的帧缓冲区的区域导致没有在继续时向附加图像的位于图像外部的那些部分呈现渲染到图像的那些部分。
支持的帧缓冲区最大数量依赖于设备。可查询 VkPhysicalDeviceLimits 类型数据的 maxFramebufferWidth, maxFramebufferHeight, 和 maxFramebufferLayers 域来获知帧缓冲区的最大个数。这些参数提供了支持的最大宽度、高度和层数。Vulkan 标准保证最小支持宽度和高度是 4096 像素，层数至少为 256.然而，绝大多数桌面级硬件支持 16,384 像素的宽度和高度，2048 层。
也可以创建不带附件的帧缓冲区。它也被称作 attachmentless framebuffer。在这种情况下，帧缓冲区的维数只通过 width,height, 和 layers 域指定。这种类型的帧缓冲区通常和片元着色器一起使用产生其他的效果，比如存储图像或者遮挡查询，这些可以为其他方面的渲染做度量而无需把渲染结果存储在其他地方。
如果 vkCreateFramebuffer()调用成功，它将把新的 VkFramebuffer handle 写入 pFramebuffer 所指向的变量。如果它要求使用主机内存，就将会用到 pAllocator 所指向的分配器。如果 pAllocator 不为 nullptr，那么在销毁帧缓冲区时就需要一个兼容的分配器。
如你将在第八章“绘制”所见，我们将和 renderpass 结合起来使用帧缓冲区来绘制到帧缓冲区绑定的图像。当你使用完了帧缓冲区，你应该调用 vkDestroyFramebuffer()来销毁它，其原型如下：
```
                voidvkDestroyFramebuffer (
VkDevice
                                device,
```

```
VkFramebuffer
                framebuffer,
const VkAllocationCallbacks*                    pAllocator);
```

device 是创建帧缓冲区的设备的 handle，framebuffer 是需被销毁的帧缓冲区对象昂的 handle。如果在创建帧缓冲区时使用了内存分配器，那么一个匹配的内存分配器通过 pAllocator 传递。

销毁帧缓冲区对象并不影响附着到它上面的图像。图像可以同时附着到多个帧缓冲区上，可同时以多种范式被使用。然而，即使图形没有被销毁，也不应该使用帧缓冲区，包括在设备的命令缓冲区中访问它。你应该保证任何使用这个帧缓冲区的已提交或在帧缓冲区销毁后未提交的命令缓冲区都已经完成了执行。

Vulkan 编程指南翻译 第七章 图形管线 第 4 节 创建简单的图形管线

翻译 2017 年 03 月 20 日 10:08:22

- 256
- 0
- 2

7.4 创建简单的图形管线

可调用和第六章“着色器和管线”中创建计算管线的函数类似的函数来创建图形管线。然而，你可以看到，图形管线包含很多个着色阶段和固定功能处理单元，所以，对于图形管线的相应的描述就复杂的多了。可调用 vkCreateGraphicsPipelines()来创建图形管线，其原型如下：

```
                VkResultvkCreateGraphicsPipelines (
VkDevice
                                                    device,
VkPipelineCache
                            pipelineCache,
uint32_t
                                                      createInfoCount,
const VkGraphicsPipelineCreateInfo*    pCreateInfos,
const
VkAllocationCallbacks*                                  pAllocator,
VkPipeline*
                                                pPipelines);
```

如你所见，vkCreateGraphicsPipelines()的原型和 vkCreateComputePipelines()很类似。都接受一个设备（device），管线缓存的 handle（pipelineCache），一个 createInfo 数组，包含数组的长度（pCreateInfosand createInfoCount）。这才是这个函数难搞的地方。VkGraphicsPipelineCreateInfo 是一个很大、复杂的数据类型，它包含了多个其他结构的指针和你已经创建的对象的 handle。深呼吸一下。

VkGraphicsPipelineCreateInfo 的定义是：

```
                typedefstruct VkGraphicsPipelineCreateInfo {
```

```
VkStructureType
                                                                    sT
ype;
const
void*

        pNext;
VkPipelineCreateFlags
                                                  flags;
uint32_t

                    stageCount;
const
VkPipelineShaderStageCreateInfo*                          pStages;
const
VkPipelineVertexInputStateCreateInfo*          pVertexInputState;
const VkPipelineInputAssemblyStateCreateInfo*   pInputAssemblyState;
const
VkPipelineTessellationStateCreateInfo*         pTessellationState;
const
VkPipelineViewportStateCreateInfo*                   pViewportState;
const VkPipelineRasterizationStateCreateInfo*    pRasterizationState;
const VkPipelineMultisampleStateCreateInfo*        pMultisampleState;
const
VkPipelineDepthStencilStateCreateInfo*       pDepthStencilState;
const
VkPipelineColorBlendStateCreateInfo*             pColorBlendState;
const
VkPipelineDynamicStateCreateInfo*                    pDynamicState;
VkPipelineLayout
                                                                  layo
ut;
VkRenderPass

   renderPass;
uint32_t

                    subpass;
VkPipeline

                basePipelineHandle;
int32_t

                        basePipelineIndex;
```

}VkGraphicsPipelineCreateInfo;
上面提醒过的，VkGraphicsPipelineCreateInfo 是一个很大的数据结构，带有很多个其他数据的指针。然而，可简单的把它分解成小块，很多个创建附加信息是可选的，可设置为 nullptr。和 Vulkan 中其他的创建信息类型一样，VkGraphicsPipelineCreateInfo 从 sType 域和 pNext 开始。
VkGraphicsPipelineCreateInfo 的 sType 是 VK_GRAPHICS_PIPELINE_CREATE_INFO，pNext 可以置为 nullptr，除非使用了拓展。
flags 域包含管线如何被使用的信息。在当前的 Vulkan 版本中已经定义了三个标志位，如下：
• VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT tells Vulkan that thispipeline is
not going to be used inperformance-critical applications and that you would prefer to receive a
ready-to-go pipeline object quickly ratherthan have Vulkan spend a lot of time optimizing the
pipeline. You might use this for thingslike simple shaders for displaying splash screens or user
interface elements that you want to displayquickly.
• VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT and VK_PIPELINE_CREATE_DERIVATIVE_BIT are usedwith derivative pipelines. This is a
feature whereby you can group similarpipelines and tell Vulkan that you’ll switch rapidly
among them. TheVK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT flag tells Vulkan
that you will want to create derivatives ofthe new pipeline, and VK_PIPELINE_CREATE_DERIVATIVE_BIT tellsVulkan that this pipeline is a pipeline.

7.4.1 图形着色器阶段

VkGraphicsPipelineCreateInfo 接下来两个域，stageCount 和 pStages，是你把着色器传递到管线的目标位置。pStages 是一个指向长度为 stageCount、类型为 VkPipelineShaderStageCreateInfo 的数组的指针，数组每一个元素描述了一个着色阶段。这些和你在 VkComputePipelineCreateInfo 定义中看到的类似，除了现在变成了一个数组。VkPipelineShaderStageCreateInfo 的定义是：

```
typedefstruct VkPipelineShaderStageCreateInfo {
VkStructureType sType;
const
void*
                                    pNext;
VkPipelineShaderStageCreateFlags            flags;
```

```
VkShaderStageFlagBits
          stage;
VkShaderModule
                        module;
const
char*
                                  pName;
const
VkSpecializationInfo*                                          pSpeci
alizationInfo;
}VkPipelineShaderStageCreateInfo;
```

所有的图形管线至少包含一个顶点着色器，且这个顶点着色器总是管线的第一个着色阶段。因此，VkGraphicsPipelineCreateInfo 的 pStages 应该指向一个描述了顶点着色器的 VkPipelineShaderStageCreateInfo 类型数据。VkPipelineShaderStageCreateInfo 中的参数应该和你在第六章“着色器和管线”中创建计算管线时的参数一样含义。module 应该是一个着色器模块，至少包含了顶点着色器，pName 应该是该模块中顶点着色器的执行入口。
因为在我们简单的管线

Listing 7.2:Creating a Simple Graphics Pipeline

```
VkPipelineShaderStageCreateInfoshaderStageCreateInfo =
{
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO,// sType
nullptr, // pNext
0, // flags
VK_SHADER_STAGE_VERTEX_BIT, // stage
module, // module
"main", // pName
nullptr // pSpecializationInfo
};

static const
VkPipelineVertexInputStateCreateInfovertexInputStateCreateInfo =
{
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO,// sType
nullptr, // pNext
0, // flags
0, //
vertexBindingDescriptionCount
nullptr, // pVertexBindingDescriptions
0, //
vertexAttributeDescriptionCount
nullptr // pVertexAttributeDescriptions
```

```
};
static const
VkPipelineInputAssemblyStateCreateInfoinputAssemblyStateCreateInfo =
{
VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO,//sType
nullptr, // pNext
0, // flags
VK_PRIMITIVE_TOPOLOGY_POINT_LIST, //topology
VK_FALSE // primitiveRestartEnable
};
static const
VkViewportdummyViewport =
{
0.0f, 0.0f, // x, y
1.0f, 1.0f, // width, height
0.1f, 1000.0f // minDepth, maxDepth
};
static const
VkRect2DdummyScissor =
{
{ 0, 0 }, // offset
{ 1, 1 } // extent
};

static const
VkPipelineViewportStateCreateInfoviewportStateCreateInfo =
{
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO,// sType
nullptr, // pNext
0, // flags
1, // viewportCount
&dummyViewport, // pViewports
1, // scissorCount
&dummyScissor // pScissors
};
static const
VkPipelineRasterizationStateCreateInforasterizationStateCreateInfo =
{
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO,// sType
nullptr, // pNext
0, // flags
VK_FALSE, // depthClampEnable
VK_TRUE, // rasterizerDiscardEnable
VK_POLYGON_MODE_FILL, // polygonMode
```

```
VK_CULL_MODE_NONE, // cullMode
VK_FRONT_FACE_COUNTER_CLOCKWISE, //frontFace
VK_FALSE, // depthBiasEnable
0.0f, // depthBiasConstantFactor
0.0f, // depthBiasClamp
0.0f, // depthBiasSlopeFactor
0.0f // lineWidth
};
static const
VkGraphicsPipelineCreateInfographicsPipelineCreateInfo =
{
VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO,// sType
nullptr, // pNext
0, // flags
1, // stageCount
&shaderStageCreateInfo, // pStages
&vertexInputStateCreateInfo, //pVertexInputState
&inputAssemblyStateCreateInfo, //pInputAssemblyState
nullptr, // pTessellationState
&viewportStateCreateInfo, //pViewportState
&rasterizationStateCreateInfo, //pRasterizationState
nullptr, // pMultisampleState
nullptr, // pDepthStencilState
nullptr, // pColorBlendState
nullptr, // pDynamicState
VK_NULL_HANDLE, // layout
renderpass, // renderPass
0, // subpass
VK_NULL_HANDLE, // basePipelineHandle
0, // basePipelineIndex
};
result =vkCreateGraphicsPipelines(device,
VK_NULL_HANDLE,
1,
&graphicsPipelineCreateInfo,
nullptr,
&pipeline);
```

当然，大多数时候，你不会使用一个仅包含顶点着色器的图形管线。如本章前面所介绍的，管线最多由五个着色器阶段组成。这些阶段如下：

- The vertex shader, specified as VK_SHADER_STAGE_VERTEX_BIT,processes one vertex at a time and passes it to the next logicalstage in the pipeline.
- The tessellation control shader, specified as

VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT,processes one control point at a
time but has access to all of the data thatmakes up the patch. It can be considered to be a patch
shader, and it produces the tessellationfactors and per-patch data associated with the patch.
• The tessellation evaluation shader, specified using
VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT,processes one tessellated vertex at a time. In many applications, itevaluates the patch function at each point—hence, the
name. It also has access to the full patchdata produced by the tessellation control shader.
• The geometry shader, specified using
VK_SHADER_STAGE_GEOMETRY_BIT,executes once
for each primitive that passes through thepipeline: points, lines, or triangles. It can produce new
primitives or throw them away rather thanpassing them on. It can also change the type of a
primitive as it passes by.
• The fragment shader, specified using
VK_SHADER_STAGE_FRAGMENT_BIT,executes once
per fragment, after rasterization. It isprimarily responsible for computing the final color of each
pixel.

大多数直接渲染将包含顶点着色器和片元着色器。每一个阶段从上一个阶段消耗数据并向下一个阶段传递数据，形成一个管线。在一些情况下，着色器的输入由固定功能单元提供，优势输出被固定功能单元消耗。不管数据的来源和去向，在着色器内声明输入和输出的方式是相同的。
在 SPIR-V 中向着色器声明一个输入，变量声明时必须被 Input 修饰。同样，在着色器内声明输出，变量声明时需要被 Output 修饰。不像 GLSL，特定用途的输入和输出并没有 SPIR-V 预定义的名字。它们以自己的目的被修饰。然后，你用 GLSL 写着色器，并用编译器编译为 SPIR-V。编译器将识别内置的变量，并把它们翻译为生成的 SPIR-V 着色器里合适的带有修饰符声明的输入、输入变量。

7.4.2 顶点输入状态

要渲染真实的几何对象，你需要给 Vulkan 管线提供数据。你可以使用 SPIR-V 提供的顶点和实例索引，来可控的生成几何数据或者从缓冲区取得几何数据。或者，你可描述集合数据在内存中的布局，Vulkan 可以帮你获取，直接给入到着色器内。

要想这么做，需要使用 VkGraphicsPipelineCreateInfo 的 pVertexInputState 成员，它是 VkPipelineVertexInputStateCreateInfo 类型的数据，定义如下：

```
typedefstruct VkPipelineVertexInputStateCreateInfo {
VkStructureType sType;
const
void*
                                                    pNext;
VkPipelineVertexInputStateCreateFlags              flags;
uint32_t

   vertexBindingDescriptionCount;
const
VkVertexInputBindingDescription*          pVertexBindingDescriptions;
uint32_t

   vertexAttributeDescriptionCount;
const
VkVertexInputAttributeDescription*    pVertexAttributeDescriptions;
} VkPipelineVertexInputStateCreateInfo;
```

VkPipelineVertexInputStateCreateInfo 结构同样以 sType 和 pNext 域开始，各自应置为 VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO 和 nullptr。VkPipelineVertexInputStateCreateInfo 的 flags 域被保留，应置为 0。

顶点输入状态可划分为可绑定到含有数据缓冲区的顶点绑定集，和描述顶点在缓冲区中如何布局的顶点属性集。和顶点缓冲区绑定的缓冲区有时被称为顶点缓冲区。注意，真的不存在这个所谓的可存储顶点数据的顶点缓冲区。用来存储顶点数据的缓冲区的唯一要求就是创建的时候带有 VK_BUFFER_USAGE_VERTEX_BUFFER_BIT 参数。

vertexBindingDescriptionCount 是管线使用的顶点绑定的个数，pVertexBindingDescriptions 是指向了一个 VkVertexInputBindingDescription 类型的数组的指针，每一个元素描述了一个绑定。VkVertexInputBindingDescription 定义是：

```
typedefstruct VkVertexInputBindingDescription {
uint32_t                    binding;
uint32_t                    stride;
VkVertexInputRate inputRate;
}VkVertexInputBindingDescription;
```

binding 域是这个结构数据描述的绑定的索引。每一个管线可以访问到一定数量的顶点缓冲区绑定，他们的索引并不需要是连续的。只要管线使用的每一个绑定都被描述了，就无需在指定的管线中描述绑定。

VkVertexInputBindingDescription 数组能寻址到的最后一个绑定索引必须比设备支持的最大绑定个数要小。这个限制被 Vulkan 标准保证至少是 16。对于一些机器，可能会更高。然而，你可以检查设备的 VkPhysicalDeviceLimits 数据的 maxVertexInputBindings 成员来获知这个最高值。可调用

vkGetPhysicalDeviceProperties()函数来获取 VkPhysicalDeviceLimits 数据。
每一个绑定可以被看作一个缓冲区对象中一个数组。数组的步长—亦即每一个元素的起始位置的距离，以 byte 为单位—通过 stride 指定。如果顶点数组通过一个数据类型的数组指定，stride 参数必须包含该数据类型的大小，即使着色器并不会使用每一个它的每一个成员。对于任何类型的绑定，stride 的最大值都是 Vulkan 实现决定的，但是 Vulkan 标准保证至少是 2048 字节。如果想要使用更大步长的顶点数据，你需要查询设备是否支持这个步长。
可检查 VkPhysicalDeviceLimits 的 maxVertexInputBindingStride 域来获知受支持的最大步长。
还有，Vulkan 可以顶点索引的函数或者实例索引的函数来遍历数组。这是通过 inputRate 域来指定的，它的值可以是 VK_VERTEX_INPUT_RATE_VERTEX 或者 VK_VERTEX_INPUT_RATE_INSTANCE。
每一个顶点属性都是存储在顶点缓冲区的一个数据类型的数据。一个顶点缓冲区的每一个顶点属性都有相同的步进速率和数组步长，但是在数据内部有自己的数据类型和偏移量。这是通过 VkVertexInputAttributeDescription 类型数据来描述的。数组的指针通过 VkPipelineVertexInputStateCreateInfo 的 pVertexAttributeDescriptions 域来传递，数组中元素的个数（顶点属性的个数）通过 vertexAttributeDescriptionCount 传递。
VkVertexInputAttributeDescription 定义是：

```
typedefstruct VkVertexInputAttributeDescription {
uint32_t                    location;
uint32_t                    binding;
VkFormat              format;
uint32_t                    offset;
}VkVertexInputAttributeDescription;
```

每一个属性都有 location，可在顶点着色器内引用到该属性。顶点属性位置并不需要是连续的，也无需描述每一个顶点属性位置，只要管线使用的所有属性都被描述了。属性的位置通过 VkVertexInputAttributeDescription 的 location 成员来描述。
指定了被绑定的缓冲区的此次绑定，亦即可从之获取属性数据绑定，通过 binding 参数指定，并且它应和之前描述过的 VkVertexInputBindingDescription 类型数据的每个元素指定的绑定想匹配。
顶点数据的格式通过 format 指定，每个数据的便宜量通过 offset 指定。
如同数据结构的总大小有上限一样，每一个属性在数据结构内的偏移量也有上限。Vulkan 标准保证至少是 2047 个字节，这个大小足以保证每一个结构的最后面能刚好放一个 byte，从而数据结构大小不超过最大值 2048 自己。如果你需要使用比这个更大的数值，你需要检查设备的能力是否可以处理。设备的 VkPhysicalDeviceLimits 的 maxVertexInputAttributeOffset 域包含了 offset 可以使用的最大值。你可以调用 vkGetPhysicalDeviceProperties()函数来获取该值。
Listing 7.3 展示了如何创建一个 C++数据结构，并使用 VkVertexInputBindingDescription 和 VkVertexInputBindingDescription 来描述它，以便你可以用它来把顶点数据传递给 Vulkan。

Listing7.3: Describing Vertex Input Data

```
typedefstruct vertex_t
{
vmath::vec4    position;
vmath::vec3    normal;
vmath::vec2    texcoord;
} vertex;

static const
VkVertexInputBindingDescription vertexInputBindings[] =
{
{ 0, sizeof(vertex),VK_VERTEX_INPUT_RATE_VERTEX } // Buffer
};

static const
VkVertexInputAttributeDescription vertexAttributes[] =
{
{ 0, 0, VK_FORMAT_R32G32B32A32_SFLOAT,
0 },                                    //Position
{ 1, 0, VK_FORMAT_R32G32B32_SFLOAT, offsetof(vertex,
normal) },//Normal
{ 2, 0, VK_FORMAT_R32G32_SFLOAT, offsetof(vertex, texcoord) } //
TexCoord
};
static const
VkPipelineVertexInputStateCreateInfo vertexInputStateCreateInfo =
{
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO, // sType
nullptr, // pNext
0, // flags
vkcore::utils::arraysize(vertexInputBindings),
//vertexBindingDescriptionCount
vertexInputBindings, //pVertexBindingDescriptions
vkcore::utils::arraysize(vertexAttributes),//vertexAttributeDescripti
onCount
vertexAttributes //
pVertexAttributeDescriptions
};
```

在单个顶点着色器内可以使用的输入属性的最大个数依赖于设备，但是被保证至少是 16.这是 pVertexInputAttributeDescriptions 数组中 VkVertexInputAttributeDescription 元素个数的上限。一些 Vulkan 实现也许支持更大的数量。可检查设备的 VkPhysicalDeviceLimits 数据的 maxVertexInputAttributes 域，来获知顶点着色器可以使用的输入的最大个数。

顶点数据是从你绑定到命令缓冲区的顶点缓冲区中读取的，然后传递给顶点着色器。对于将处理这些顶点数据的顶点着色器，它必须声明和已定义的顶点属性对应的输入。在你的 SPIR-V 顶点着色器中创建一个带 Input 存储标志的变量即可做到。在 GLSL 着色器中，使用 in 类型的变量即可表达。
每一个输入必须被赋予一个 location。在 GLSL 中这是通过 location 限定符来指定的，翻译到 SPIR-V 中就是应用到输入的 Location 修饰符。Listing 7.4 展示了声明了多个输入的 GLSL 顶点着色器的片段。glslangvalidator 生成的 SPIR-V 如 Listing 7.5 所示。

Theshader shown in Listing 7.5 is incomplete, as it has been edited to make thedeclared inputs clearer.

```
#version450 core
layout (location = 0) in vec3 i_position;
layout (location = 1) in vec2 i_uv;
void main(void)
{
gl_Position = vec4(i_position, 1.0f);
}
```

Listing 7.5: Declaring Inputs to a Vertex Shader (SPIR-V)

```
; SPIR-V
; Version: 1.0
; Generator: Khronos Glslang Reference Front End; 1
; Bound: 30
; Schema: 0
OpCapability Shader
%1 = OpExtInstImport "GLSL.std.450"
OpMemoryModel Logical GLSL450
OpEntryPoint Vertex %4 "main" %13 %18 %29
OpSource GLSL 450
OpName %18 "i_position" ;; Name of i_position
OpName %29 "i_uv" ;; Name of i_uv
OpDecorate %18 Location 0 ;; Location of i_position
OpDecorate %29 Location 1 ;; Location of i_uv
...
%6 = OpTypeFloat 32 ;; %6 is 32-bit floating-point type
%16 = OpTypeVector %6 3 ;; %16 is a vector of 3 32-bit floats
(vec3)
%17 = OpTypePointer Input %16
%18 = OpVariable %17 Input ;; %18 is i _position - input pointer to
vec3
%27 = OpTypeVector %6 2 ;; %27 is a vector of 2 32-bit floats
%28 = OpTypePointer Input %27
%29 = OpVariable %28 Input ;; %29 is i _uv - input pointer to vec2
...
```

也可以

7.4.3 输入组装

7.4.4 细分状态

7.4.5 视口状态

7.4.6 栅格化状态

7.4.7 多采样状态

7.4.8 深度和 stencil 测试状态

7.4.9 颜色混合状态

Vulkan 编程指南翻译 第七章 图形管线 第 4 节 创建简单的图形管线（下）

翻译 2017 年 03 月 22 日 00:51:23

- 386
- 0
- 3

这一节实在太长了，拖了好久。还是分开发吧。

7.4.3 输入组装

图形管线的输入组装阶段接受顶点数据，并把它们分组，组成图元，以供管线接下来的部分处理。它是通过一个 VkPipelineInputAssemblyStateCreateInfo 类型的数据描述的，通过 VkGraphicsPipelineCreateInfo 类型数据的 pInputAssemblyState 成员传递。VkPipelineInputAssemblyStateCreateInfo 的定义是：

```
typedef struct VkPipelineInputAssemblyStateCreateInfo {
VkStructureType sType;
const void* pNext;
VkPipelineInputAssemblyStateCreateFlags flags;
VkPrimitiveTopology topology;
VkBool32 primitiveRestartEnable;
} VkPipelineInputAssemblyStateCreateInfo;
```

sType 域应置为 VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO，pNext 应置为 nullptr。flags 域被保留使用，应置为 0。图元拓扑类型由 topology 指定，应该是受 Vulkan 支持的图元拓扑类型之一。它们是 VkPrimitiveTopology 类型的枚举。枚举中最简单的成员是 list 拓扑，如下所示：

• VK_PRIMITIVE_TOPOLOGY_POINT_LIST: Each vertex is used to construct an
independent point.
• VK_PRIMITIVE_TOPOLOGY_LINE_LIST: Vertices are grouped into pairs, each pair
forming a line segment from the first to the second vertex.
• VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST: Vertices are grouped into triplets forming
Triangles.

接下来是 strip 和 fan 图元。这些是顶点组成的图元（线段或者三角形）再组成的，每一个线段或者三角形都与上一个共享一个或者两个顶点。strip 和 fan 图元如下所示：
• VK_PRIMITIVE_TOPOLOGY_LINE_STRIP: The first two vertices in a draw form a single
line segment. Each new vertex after them forms a new line segment from the last processed
vertex. The result is a connected sequence of lines.
• VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP: The first three vertices in a draw form a
single triangle. Each subsequent vertex forms a new triangle along with the last two vertices.
The result is a connected row of triangles, each sharing an edge with the last.
• VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN: The first three vertices in a draw form a
single triangle. Each subsequent vertex forms a new triangle along with the last vertex and the
first vertex in the draw.
Strip and fan 拓扑并不复杂，但是如果对它们不熟悉可能不容易绘制出来。Figure 7.2 展示了它们图形化的布局。

Figure 7.2: Strip (Left) and Fan (Right) Topologies

接下来是邻接图元，通常是几何着色器启用了才被使用，可以携带有关于在同一个 mesh 里邻近图元的附加信息。邻接图元拓扑有：
• VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY: Every four vertices in the
draw form a single primitive, with the center two vertices forming a line and the first and last
vertex in each group of four being presented to the geometry shader, when present.
• VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY: The first four vertices in

the draw form a single primitive, with the center two vertices forming a line segment and the
first and last being presented to the geometry shader as adjacency information. Each subsequent
vertex essentially slides this window of four vertices along by one, forming a new line segment
and presenting the new vertex as adjacency information.
• VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY: Similar to lines with
adjacency, each group of six vertices is formed into a single primitive, with the first, third, and
fifth in each group constructing a triangle and the second, fourth, and sixth being presented to
the geometry shader as adjacency information.
• VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY: This is perhaps the
most confusing primitive topology and certainly needs a diagram to visualize. Essentially, the
strip begins with the first six vertices forming a triangle with adjacency information as in the list
case. For every two new vertices, a new triangle is formed, with the odd-numbered vertices
forming the triangle and the even-numbered vertices providing adjacency information.

邻接图元很难被视觉化出来--特别是
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY 拓扑。Figure 7.3 示例了在 VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY 拓扑中顶点的布局。在这张图中你可以看到从 12 个顶点中形成了两个三角形。顶点由外层包裹一个三角形，奇数序号的顶点形成了中心三角形（A 和 B），偶数序号的顶点形成了并不会被渲染的虚拟的三角形，但是，携带有邻接的信息。这个概念也同样在三角形 strip 图元中适用，Figure 7.4 展示了如何应用到 VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY。

Figure 7.3: Triangles with Adjacency Topology

Figure 7.4: Triangle Strip with Adjacency Topology
邻接拓扑通常在几何着色器存在时才被使用，因为几何着色器是唯一能够看到邻接顶点的阶段。然而，也可能没有几何着色器的情况下使用邻接图元，邻接的顶点直接被扔掉了而已。

7.4.4 细分状态

7.4.5 视口状态

7.4.6 栅格化状态

7.4.7 多采样状态

多采样是为图像内每一个像素生成多个样本的过程。用来对抗走样，在图像直接使用时能很大程度上提升图像质量。当你进行多采样时，颜色和深度-stencil 附件必须是多采样图像，并且管线的多采样状态应该通过 VkGraphicsPipelineCreateInfo 的 pMultisampleState 域合理设置。这个一个指向 VkPipelineMultisampleStateCreateInfo 类型数据的指针，该类型定义为：

```
typedef struct VkPipelineMultisampleStateCreateInfo {
VkStructureType sType;
const void* pNext;
VkPipelineMultisampleStateCreateFlags flags;
VkSampleCountFlagBits rasterizationSamples;
VkBool32 sampleShadingEnable;
float minSampleShading;
const VkSampleMask* pSampleMask;
VkBool32 alphaToCoverageEnable;
VkBool32 alphaToOneEnable;
} VkPipelineMultisampleStateCreateInfo;
```

VkPipelineMultisampleStateCreateInfo 的 sType 域应置为 VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO，pNext 应置为 nullptr。flags 域被保留使用，应置为 0。

7.4.8 深度和 stencil 测试状态

深度-stencil 状态控制了如何进行深度和 stencil 测试和如何决定片元如何通过这些测试。深度和 stencil 测试可以在片元着色器运行之前或之后进行。默认情况下，深度测试在片元着色器运行之后发生。1

1. Most implementations will only keep up the appearance that the depth and stencil tests are running after the fragment shader and, if possible, run the tests before running the shader to avoid running shader code when the test would fail.

要在深度测试之前运行片元着色器，我们可以设置我们的片元着色器入口为 SPIR-V EarlyFragmentTests 执行模式。

Depth-stencil 状态通过 VkGraphicsPipelineCreateInfo 类型的成员 pDepthStencilState，一个指向 VkPipelineDepthStencilStateCreateInfo 类型数据的指针来配置。VkPipelineDepthStencilStateCreateInfo 的定义是：

```
typedef struct VkPipelineDepthStencilStateCreateInfo {
VkStructureType sType;
const void* pNext;
VkPipelineDepthStencilStateCreateFlags flags;
VkBool32 depthTestEnable;
VkBool32 depthWriteEnable;
VkCompareOp depthCompareOp;
VkBool32 depthBoundsTestEnable;
VkBool32 stencilTestEnable;
VkStencilOpState front;
VkStencilOpState back;
float minDepthBounds;
float maxDepthBounds;
} VkPipelineDepthStencilStateCreateInfo;
```

VkPipelineDepthStencilStateCreateInfo 的 sType 域应置为 VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_CREATE_INFO，pNext 应置为 nullptr。flags 域被保留使用，应置为 0。

如果 depthTestEnable 被设置为 VK_TRUE，那么深度测试被开启。如果深度测试被启用，那么测试算法可通过 depthCompareOp 选择，它是 VkCompareOp 枚举值之一。可用的深度测试操作将在第十章“片元处理”中深入的讲解。如果 depthTestEnable 被设置为 VK_FALSE，那么深度测试被关闭。depthCompareOp 被启用，所有的片元被认为已经通过了深度测试。需要注意，当深度测试被关闭是，将不会有深度缓冲区写入操作。

如果通过了深度测试（或者深度测试被关闭），那么片元测试继续到 stencil 测试。如果 VkPipelineDepthStencilCreateInfo 的 stencilTestEnable 被设置为 VK_TRUE，那么 stencil 测试被开启。当 stencil 测试被开启时，front and back 成员给正面和背面图元提供了单独的状态。如果 stencil 测试被禁用了，所有的图元被认为通过了 stencil 测试。

关于深度和 stencil 测试，将会在第十章“片元处理”中深入讲解。

7.4.9 颜色混合状态

Vulkan 图形管线的最后一个阶段是颜色混合阶段。这个阶段负责把片元写入颜色附件。在很多情况下，这是一个非常简单的哦操作，仅仅是把附件的数据覆盖为片元着色器的输出即可。然而，颜色混合可以把帧缓冲区中已存在的值做混合，在片元着色器的输出和当前的帧缓存区的内容之间进行简单的逻辑操作。

颜色混合状态通过 VkGraphicsPipelineCreateInfo 的 pColorBlendState 成员指定。这是一个指向 VkPipelineColorBlendStateCreateInfo 类型实例的指针，该类型定义是：

```
typedef struct VkPipelineColorBlendStateCreateInfo {
VkStructureType sType;
```

```
const void* pNext;
VkPipelineColorBlendStateCreateFlags flags;
VkBool32 logicOpEnable;
VkLogicOp logicOp;
uint32_t attachmentCount;
const VkPipelineColorBlendAttachmentState* pAttachments;
float blendConstants[4];
} VkPipelineColorBlendStateCreateInfo;
```

VkPipelineColorBlendStateCreateInfo 的 sType 域应置为 VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO，pNext 应置为 nullptr。flags 域被保留使用，应置为 0。

logicOpEnable 域指定了在片元着色器的输出和颜色附件的内容之间是否进行逻辑操作。当 logicOpEnable 是 VK_FALSE 时，逻辑操作被禁用，片元着色器的输出被未经改变的写入到颜色附件。当 logicOpEnable 为 VK_TRUE 时，逻辑操作被支持他的附件启用。采用的逻辑操作对于每个附件来说都是一样的，它是 VkLogicOp 枚举类型的一个成员。每一个枚举的意义和关于逻辑操作的更多信息在第十章“片元处理”中给出。

每一个附件可以有不同的格式，可以支持不同混合操作。这些是通过一个 VkPipelineColorBlendAttachmentState 类型数组来指定的，数组的地址通过 VkPipelineColorBlendStateCreateInfo.类型数据的成员 pAttachments 来传递。附件个数通过 attachmentCount 设置。

VkPipelineColorBlendAttachmentState 的定义是：

```
typedef struct VkPipelineColorBlendAttachmentState {
VkBool32 blendEnable;
VkBlendFactor srcColorBlendFactor;
VkBlendFactor dstColorBlendFactor;
VkBlendOp colorBlendOp;
VkBlendFactor srcAlphaBlendFactor;
VkBlendFactor dstAlphaBlendFactor;
VkBlendOp alphaBlendOp;
VkColorComponentFlags colorWriteMask;
} VkPipelineColorBlendAttachmentState;
```

对已每一个颜色附件，VkPipelineColorBlendAttachmentState 的成员控制了是否开启混合，源和目标的比例因子，何种混合操作（），需要更新输出图像的哪个通道。

如果 VkPipelineColorBlendAttachmentState 的 colorBlendEnable 域为 VK_TRUE，那么剩余的参数控制了混合的状态。混合将会在第十章讲到。当 colorBlendEnable 是 VK_FALSE 是，VkPipelineColorBlendAttachmentState 中的混合相关的参数被忽略，该附件的混合被禁用。

不管 colorBlendEnable 的状态如何，最后一个域：colorWriteMask，控制了输出图像的哪个通道写入到福建。这是一个位域，由 VkColorComponentFlagBits 枚举的一个或多个值组横。四个通道，由 VK_COLOR_COMPONENT_R_BIT, VK_COLOR_COMPONENT_G_BIT,

VK_COLOR_COMPONENT_B_BIT, and VK_COLOR_COMPONENT_A_BIT 表示，可以被单独的设置。如果对应某个通道的标志位没有被包含在 colorWriteMask，那么这个通道将不会被修改。只有被 colorWriteMask 包含的通道将通过渲染到附件来更新。

Vulkan 编程指南翻译 第七章 图形管线 第 5 节 动态状态

翻译 2017 年 03 月 22 日 19:52:52

- 288
- 0
- 3

7.5

如之前章节所见，图形管线是大型的、复杂的，包含很多状态的对象。在很多图形应用程序中，我们期望能够高频率的改变一些状态。如果每一个状态的每一次更改都需要你创建一个新的图形管线对象，那么你的应用程序中对象的个数将会迅速的变得很大。

要精细粒度的管理状态的改变，Vulkan 提供了以便定图形管线某部分为动态的能力，这意味着它们可以被命令缓冲区的命令而不是一个对象直接在运行时修改。因为这减少了 Vulkan 优化或者吸收状态的一部分的机会，所以，有必要显式的指定你想要变为动态的部分。这通过 VkGraphicsPipelineCreateInfo 类型数据的 pDynamicState 成员可做到，它是一个指向 VkPipelineDynamicStateCreateInfo 类型数据的指针，该类型定义是：

```
typedefstruct VkPipelineDynamicStateCreateInfo {
VkStructureType
                              sType;
const
void*
                                        pNext;
VkPipelineDynamicStateCreateFlags        flags;
uint32_t
                                                  dynamicStateCount;
const
VkDynamicState*
   pDynamicStates;
} VkPipelineDynamicStateCreateInfo;
```

VkPipelineDynamicStateCreateInfo 的 sType 应置为 VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO，pNext 应置为 nullptr。Flags 域被保留，应置为 0。

你希望变为动态的状态的个数通过 dynamicStateCount 指定。这是 pDynamicStates 所指向的数组的长度，该数组的元素是 VkDynamicState 枚举的值。包含了该枚举的一个成员的 pDynamicStates 数组告诉 Vulkan 你想要使用对应的动态状态设置命令。VkDynamicState 的成员和含义如下：

• VK_DYNAMIC_STATE_VIEWPORT:The viewport rectangle is dynamic and will be updated

using vkCmdSetViewport().
•VK_DYNAMIC_STATE_SCISSOR: The scissor rectangle is dynamic and will be updatedusing
vkCmdSetScissor().
•VK_DYNAMIC_STATE_LINE_WIDTH: The line width is dynamic and will be updatedusing
vkCmdSetLineWidth().
•VK_DYNAMIC_STATE_DEPTH_BIAS: The depth bias parameters are dynamic and will be
updated using vkCmdSetDepthBias().
•VK_DYNAMIC_STATE_BLEND_CONSTANTS: The color blend constants are dynamic and
will be updated using vkCmdSetBlendConstants().
•VK_DYNAMIC_STATE_DEPTH_BOUNDS: The depth bounds parameters are dynamic and will
be updated using vkCmdSetDepthBounds().
•VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK,
VK_DYNAMIC_STATE_STENCIL_WRITE_MASK, and
VK_DYNAMIC_STATE_STENCIL_REFERENCE: The corresponding stencilparameters are
dynamic and will be updated using vkCmdSetStencilCompareMask(),
vkCmdSetStencilWriteMask(), and vkCmdSetStencilReference(), respectively

如果一个状态被指定为动态的，那么在绑定管线是你需要负责设置这个状态。如果状态没有被标志为动态，那么就被认为是静态的，在绑定管线是被设置。绑定管线到静态状态会导致动态状态未定义。原因是如果静态状态没有被使用或者被认定为有效时，Vulkan 实现也许优化静态状态进入管线对象内部，且不会真正把他们编程到硬件。当一个带有标记为动态的管线被随即绑定，动态状态在硬件中是否一致是未定义的。
然而，当你在标记为相同的动态状态的不同管线中切换时，状态仍跨越不同的绑定保持一致。Table7.1 展示如下：
Table 7.1: Dynamic and Static StateValidity
如你在 Table 7.1 中所见，状态变得未定义的唯一一种情形发生在你在管线里从标记为静态状态切换到状态标记为动态时。在其他所有情形下，状态都是良好的被定义，状态来自于通过管线的状态或者使用合适的命令设置的动态状态。
如果你设置一个当前绑定为静态状态的管线为动态状态，且然后用这个管线绘制，那么会造成未知的结果。这个效果也许是忽略状态设置命令，继续使用静态状态版本的管线，或者是突出状态设置命令并使用新的管线状态，又或者是损坏状态并导致程序崩溃。这个效果在不同的 Vulkan 实现之间是不同的，取决于哪个被错误的覆盖的状态。
设置动态状态，然后和一个管线绑定，将导致动态状态被使用。然而，最好是先绑定管线，然后在绑定任何状态来避免潜在的为定义行为发生。

总结

本章提供了对 Vulkan 图形管线快速的讲解。管线由多个阶段组成，一些阶段可以被配置，其中一些是固定功能但可以配置，和一些由外部提供的功能强大的着色器程序。在第六章“着色器和管线”给出的管线对象的基础概念之上，这里介绍了图形管线对象。这个对象包括大量的固定功能状态。尽管本章内示例构造的管线相当简单，为后面的章节将会讲到的更加复杂和强表述能力的管线打下了基础。

Vulkan 编程指南翻译 第八章 图形管线 第 1 节 准备绘制

翻译 2017 年 03 月 24 日 13:18:32

- 186
- 0
- 1

第八章 绘制

在本章你将学到：

l Vulkan 中不同绘制名利的细节

l 如何通过实例化来绘制多分复制的数据

l 如何通过缓冲区传递绘制参数

绘制是 Vulkan 中基础的操作，由图形管线触发工作被执行。Vulkan 包含多条绘制命令，每一个都以稍微不同的方式声称图形工作。本章深入研究 Vulkan 所支持的绘制命令。首先，我们在回顾一下第七章“”讨论过的基本绘制命令；然后，我们探索 indexed 和 instanced 绘制命令。最后，我们讨论如何为绘制命令从设备内存中取出参数，甚至在设备上生成参数。

回到第七章“”，你见到了第一个绘制命令，vkCmdDraw()。这个命令仅仅是把顶点放入 Vulkan 图形管线。当我们介绍这个命令时，我们粗略的讲解了它的参数。我们了透露了其他的绘制命令的存在。为了参考，vkCmdDraw()的原型如下：

```
                voidvkCmdDraw (
VkCommandBuffer commandBuffer,
uint32_t
                                                vertexCount,
uint32_t
                                                instanceCount,
uint32_t
                                                firstVertex,
uint32_t
                                                firstInstance);
```

和其他在设备上执行的命令一样，第一个参数是一个 VkCommandBuffer 类型数据的 handle。每一次绘制的顶点个数通过 vertexCount 指定，顶点数据中起始顶点的索引通过 firstVertex 指定。需要传送到管线的顶点是从 firstVertex 位置开始 vertexCount 个连续的顶点的数据。如果你使用顶点缓冲区和顶点属性来自动的吧数据传递到顶点着色器，那么着色器就会看到从数组中获取的连

续数据。如果你在着色器中直接使用顶点的索引，你将看到从 firstVertex 之后的数据。

8.1 准备绘制

如我们在第七章“”中提到的，所有的绘制命令都包含在一个 renderpass 里。尽管 renderpass 对象可以封装多个 subpass，简单的绘制到一个输出图像也需要是 renderpass 的一部分。可通过调用 vkCreateRenderPass()来创建 renderpass 对象。要准备绘制，我们需要调用 vkCmdBeginRenderPass()，它将设置当前的 renderpass 对象，并且，更重要的是，配置将要绘制的输出图像集合。vkCmdBeginRenderPass() 的原型是：

```
                voidvkCmdBeginRenderPass (
VkCommandBuffer
commandBuffer,
const VkRenderPassBeginInfo*        pRenderPassBegin,
VkSubpassContents
contents);
```

在 renderpass 内包含被发送命令的命令缓冲区通过 commandBuffer 来传递。描述 renderpass 的一堆参数通过一个 VkRenderPassBeginInfo 类型数据的指针 pRenderPassBegin 来传递。VkRenderPassBeginInfo 的定义是：

```
                typedefstruct VkRenderPassBeginInfo {
VkStructureType                                                 sType;
const
void*
pNext;
VkRenderPass                                                     ren
derPass;
VkFramebuffer                                                   fram
ebuffer;
VkRect2D
         renderArea;
uint32_t
              clearValueCount;
const VkClearValue*                                 pClearValues;
} VkRenderPassBeginInfo;
```

VkRenderPassBeginInfo 的 sType 域应置为 VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO，pNext 应置为 nullptr。被开始的 renderpass 通过 renderPass 指定，渲染所到的帧缓冲区通过 framebuffer 指定。如我们在第七章“”所讨论过的，帧缓冲区是图形命令绘制结果的一系列图像。

不管用什么方式使用 renderpass，我们可以选择只渲染到附着的图像的一部分区域。要这样做，需使用 VkRenderPassBeginInfo 的 renderArea 成员来指定渲染目标矩形区域。设置 renderArea.offset.x and renderArea.offset.y 为

0 ，renderArea.extent.width 和 renderArea.extent.height 为帧缓冲区的宽度和高度告诉 Vulkan 你将渲染的矩形区域。
如果 renderpass 的任何一个附件有 VK_ATTACHMENT_LOAD_OP_CLEAR 这个加载操作，那么你想清除的颜色或者值通过 pClearValues.所指向的 VkClearValue 数组指定。pClearValues 的个数通过 clearValueCount 传递。VkClearValue 的定义是：

```
typedefunion VkClearValue {
VkClearColorValue                                    color;
VkClearDepthStencilValue          depthStencil;
} VkClearValue;
```

如果附件是颜色附件，那么存储在 VkClearValue 的 color 成员的值就被使用，如果附件是深度、stencil，或者 depth-stencil 附件，那么 depthStencil 成员的值被使用。Color 和 depthStencil 是 VkClearColorValue 和 VkClearDepthStencilValue 类型的实例，各自定义如下：

```
typedefunion VkClearColorValue {
float                                float32[4];
int32_t                        int32[4];
uint32_t                     uint32[4];
} VkClearColorValue;
typedef struct VkClearDepthStencilValue {
float                                depth;
uint32_t                     stencil;
} VkClearDepthStencilValue;
```

每一个附件的索引用来在 VkClearValue 数组中索引位置。这意味着如果一些附件有 VK_ATTACHMENT_LOAD_OP_CLEAR 这个加载操作，那么数组中就有未使用的元素。pClearValues 数组中元素的个数至少有带有 VK_ATTACHMENT_LOAD_OP_CLEAR 操作的附件的最高的索引值那么多个。
对于每一个带有 VK_ATTACHMENT_LOAD_OP_CLEAR 操作的附件，如果它是一个附件，那么 float32,int32, or uint32 数组的值被用来清除附件的内容，依数组元素类型是否为浮点、归一化格式、有符号整型或者无符号整型格式而定。如果附件是深度、stencil 或者是深度-stencil 附件，那么 VkClearValue 的 depthStencil 的 depth 和 stencil 成员用来清除附件里对应的内容。
一旦 renderpass 开始，你可以把绘制命令放入命令缓冲区。所有的绘制结果都将直接进入 vkCmdBeginRenderPass()参数 VkRenderPassBeginInfo 所指定的帧缓冲区。要终止 renderpass 中的渲染，可以调用 vkCmdEndRenderPass()函数，其原型如下：

```
voidvkCmdEndRenderPass (
VkCommandBuffer                                    commandBuffer);
```

vkCmdEndRenderPass()执行后，renderpass 中任何绘制都已经完成了，帧缓冲区的内容已经更新。在此之前，帧缓冲区的内容是为定义的。只有带有 VK_ATTACHMENT_STORE_OP_STORE 操作的附件将会影响到 renderpass 中心的绘制产生的内容。如果一个附件有 VK_ATTACHMENT_STORE_OP_DONT_CARE 操作，那么它的内容在 renderpass 完成后是为未定义的。

Vulkan 编程指南翻译 第八章 图形管线 第 2 节 顶点数据

翻译 2017 年 03 月 24 日 14:04:07

- 236
- 0
- 1

8.2 顶点数据

如果你所使用的图形管线需要顶点数据，在执行任何绘制操作前，你需要绑定用来获取数据的哦缓冲区。当缓冲区被用来作为顶点数据的来源时，它们有时也被称为顶点缓冲区。把缓冲区当作顶点数据来用的命令是 vkCmdBindVertexBuffers()，它的原型是：

```
                voidvkCmdBindVertexBuffers (
VkCommandBuffer                  commandBuffer,
uint32_t                                                    first
Binding,
uint32_t                                                    bindi
ngCount,
const VkBuffer*                              pBuffers,
const VkDeviceSize*            pOffsets);
```

和缓冲区所绑定命令缓冲区通过 commandBuffer 指定。一个给定的管线可能会引用很多个顶点缓冲区，vkCmdBindVertexBuffers()能够更新特定的命令缓冲区的一部分。需要更新的第一个绑定的索引通过 firstBinding 指定，需要更新的连续的绑定个数通过 bindingCount 指定。要更新非连续的顶点缓冲区范围，你需要多次调用 vkCmdBindVertexBuffers()。

pBuffers 参数是一个指向 bindingCount 个 VKBufferhandle 的数组的指针，pOffsets 是一个指向缓冲区对象内 bindingCount 个偏移量的数组，偏移量亦即每个绑定数据开始的位置。pOffsets 的值以字节为单位。一个很合理的做法是把相同的缓冲区对象以不同的位置（如果有需要甚至是相同的开始位置）绑定到一个命令缓冲区，在 pBuffers 数组中多次包含同一个 VkBuffer 类型的 handle 即可。

缓冲区内的数据的布局和格式被将要使用顶点数据的管线所定义。因此，数据的格式在这里没有被指定，但是在用来创建管线的 VkPipelineVertexInputStateCreateInfo 的 VkPipelineVertexInputStateCreateInfo 数据中定义。回到第七章“”，我们在 Listing7.3 里展示了一个用 C++数据结构构造间隔顶点数据的例子。

Listing8.1 展示了一个稍微高级的例子，使用一个缓冲区来存储位置数据，另一缓冲区来存储顶点法相和纹理坐标。

```
                Listing8.1 分离顶点属性的构造
                typedefstruct vertex_t
{
vmath::vec3                         normal;
vmath::vec2                         texcoord;
} vertex;
static const
VkVertexInputBindingDescription vertexInputBindings[] =
{
```

```
{ 0, sizeof(vmath::vec4), VK_VERTEX_INPUT_RATE_VERTEX },        //
Buffer 1
{ 1, sizeof(vertex),
VK_VERTEX_INPUT_RATE_VERTEX }                                    //
Buffer 2
};
static const
VkVertexInputAttributeDescription vertexAttributes[] =
{
{ 0, 0, VK_FORMAT_R32G32B32A32_SFLOAT,
0 },                                                             //
Position
{ 1, 1, VK_FORMAT_R32G32B32_SFLOAT,
0 },
     // Normal
{ 2, 1, VK_FORMAT_R32G32_SFLOAT,
sizeof(vmath::vec3) }                  // Tex Coord
};
static const
VkPipelineVertexInputStateCreateInfo vertexInputStateCreateInfo =
{
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO,       //
sType
nullptr,


                                                    //pNext
0,


                                                                 //fl
ags
vkcore::utils::arraysize(vertexInputBindings),            //vertexBin
dingDescription-Count
vertexInputBindings,
                                                          //
pVertexBinding-Descriptions
vkcore::utils::arraysize(vertexAttributes),
// vertexAttribute-DescriptionCount
vertexAttributes
                                                                 //pV
ertexAttribute- Descriptions
};
```

在 Listing 8.1 中，我们已经用两个缓冲区定义了三个顶点属性。在第一个缓冲区里，只有一个简单的 vec4 变量，用来表示位置。这个缓冲区的步长因此是 vec4 的大小，是 16 字节。在第二个缓冲区里，我们存储了间隔的每一个顶点的法相和纹理坐标。我们以 vertex 类型结构表示，让编译器自动计算步长。

Vulkan 编程指南翻译 第八章 图形管线 第 3 节 索引化绘制

翻译 2017 年 03 月 29 日 09:28:38

- 138
- 0
- 1

8.3 索引化绘制

简单的吧连续的顶点给入管线并不总是你所想要的。在大多数几何网格中，很多顶点不止被使用一次。一个完全连接的网格也许会让多个三角形共用一个顶点。甚至一个立方体中三个相邻的三角形共享一个顶点。在你的顶点缓冲区中为每一个顶点指定三次是非常浪费的。除了这个，一些 Vulkan 实现也许很聪明，如果看到相同的一个顶点出现多次，在后面的过程中会跳过对相同的顶点的处理，并使用之前顶点着色器调用的结果。

要想这样做，Vulkan 允许索引化绘制。vkCmdDraw()使用索引化和 vkCmdDrawIndexed()是等价的，vkCmdDrawIndexed()的原型是：

```
voidvkCmdDrawIndexed (
VkCommandBuffer commandBuffer,
uint32_t indexCount,
uint32_t instanceCount,
uint32_t firstIndex,
int32_t vertexOffset,
uint32_t firstInstance);
```

vkCmdDrawIndexed()的第一个参数是将被执行的命令所在的命令缓冲区的 handle。然而，相比于简单从零开始，vkCmdDrawIndexed()从一个索引缓冲区中取出索引。索引缓冲区是一个通常的缓冲区对象，通过 vkCmdBindIndexBuffer()绑定到命令缓冲区，命令原型为：

```
voidvkCmdBindIndexBuffer (
VkCommandBuffer commandBuffer,
VkBuffer buffer,
VkDeviceSize offset,
VkIndexType indexType);
```

索引缓冲区所绑定的命令缓冲区通过 commandBuffer 指定，包含索引数据的缓冲区通过 buffer 指定。从 offset 开始的缓冲区的一部分可以绑定到命令缓冲区。绑定的部分总是拓展到缓冲区对象的结尾。在索引缓冲区上没有绑定检

查，Vulkan 将读取你告诉它的索引数量。然而，读取数据时将不会突破缓冲区对象的尾部。
缓冲区内的索引的类型通过 indexType 指定。这是 VkIndexType 枚举的一个成员，所有成员如下：

•VK_INDEX_TYPE_UINT16: Unsigned 16-bit integers
•VK_INDEX_TYPE_UINT32: Unsigned 32-bit integers

当你调用 vkCmdDrawIndexed()时，Vulkan 将从当前绑定的索引缓冲区中 offset 位置开始读取数据

offset+ firstIndex * sizeof(index)

其中，sizeof(index)对于 VK_INDEX_TYPE_UINT16 来说是 2，对 VK_INDEX_TYPE_UINT32 来说是 4。代码将会从索引缓冲区中读取 indexCount 个连续的整形，然后把它们加上 vertexOffset。加法操作总是以 32 位执行的，不管当前绑定的索引缓冲区的索引类型。如果加法操作在无符号 32 整形上溢出了，结果是未知的，所以要避免这种状况。
Figure 8.1 语义化的展示了数据流
Figure8.1: Index Data Flow

注意，当索引类型是 VK_INDEX_TYPE_UINT32 时，索引数值的最大范围也许并不受支持。可查看调用 vkGetPhysicalDeviceProperties(). 获取的 VkPhysicalDeviceLimits 结构的 maxDrawIndexedIndexValue 域来做检查。这个数值被保证至少是 2^{24}-1，也可以高达 2^{32}-1。
为了演示如何高效率的使用索引数据，Listing 8.2 展示了用索引和非索引方式绘制一个立方体所需数据的不同。

Listing8.2: Indexed Cube Data

```
//Raw, non-indexed data
static const float vertex_positions[] =
{
-0.25f, 0.25f, -0.25f,
-0.25f, -0.25f, -0.25f,
0.25f, -0.25f, -0.25f,
0.25f, -0.25f, -0.25f,
0.25f, 0.25f, -0.25f,
-0.25f, 0.25f, -0.25f,
0.25f, -0.25f, -0.25f,
0.25f, -0.25f, 0.25f,
0.25f, 0.25f, -0.25f,
0.25f, -0.25f, 0.25f,
0.25f, 0.25f, 0.25f,
0.25f, 0.25f, -0.25f,
0.25f, -0.25f, 0.25f,
-0.25f, -0.25f, 0.25f,
0.25f, 0.25f, 0.25f,
-0.25f, -0.25f, 0.25f,
```

```
-0.25f, 0.25f, 0.25f,
0.25f, 0.25f, 0.25f,
-0.25f, -0.25f, 0.25f,
-0.25f, -0.25f, -0.25f,
-0.25f, 0.25f, 0.25f,
-0.25f, -0.25f, -0.25f,
-0.25f, 0.25f, -0.25f,
-0.25f, 0.25f, 0.25f,
-0.25f, -0.25f, 0.25f,
0.25f, -0.25f, 0.25f,
0.25f, -0.25f, -0.25f,
0.25f, -0.25f, -0.25f,
-0.25f, -0.25f, -0.25f,
-0.25f, -0.25f, 0.25f,
-0.25f, 0.25f, -0.25f,
0.25f, 0.25f, -0.25f,
0.25f, 0.25f, 0.25f,
0.25f, 0.25f, 0.25f,
-0.25f, 0.25f, 0.25f,
-0.25f, 0.25f, -0.25f
};
static const uint32_t vertex_count = sizeof(vertex_positions) /
(3 * sizeof(float));
// Indexed vertex data
static const float indexed_vertex_positions[] =
{
-0.25f, -0.25f, -0.25f,
-0.25f, 0.25f, -0.25f,
0.25f, -0.25f, -0.25f,
0.25f, 0.25f, -0.25f,
0.25f, -0.25f, 0.25f,
0.25f, 0.25f, 0.25f,
-0.25f, -0.25f, 0.25f,
-0.25f, 0.25f, 0.25f,
};
  // Index buffer
static const uint16_t vertex_indices[] =
{
0, 1, 2,
2, 1, 3,
2, 3, 4,
4, 3, 5,
4, 5, 6,
6, 5, 7,
```

```
6, 7, 0,
0, 7, 1,
6, 0, 2,
2, 4, 6,
7, 5, 3,
7, 3, 1
};
  static const uint32_tindex_count =
vkcore::utils::arraysize(vertex_indices);
```

如你在 Listing 8.2 中所见，绘制一个立方体的所需数据很小。只有存储八个不同顶点的数据，和 36 个用来引用它们的索引数据。随着场景的复杂几何尺寸在增长，存储量也可能相当大。在这个简单的例子中，非索引化的顶点数据有 36 个顶点，每一个都包含三个 4 字节元素，总共是 432 字节的数据。同时，索引化数据有 12 个顶点，每一个都包含三个 4 字节元素，加上 36 个索引，每一个消耗 2 字节。索引化的立方体产生了 168 字节的数据。
除了使用索引化数据来节省空间，很多 Vulkan 实现包含了顶点缓存，可以重复利用顶点着色器之前计算的结果。如果顶点数据是非索引化的，那么管线必须假设他们都是不同的。然而，当顶点被索引化，两个具有相同索引的顶点是相同的。在任何一个闭合的网格中，同一个顶点将会出现多次，因为有多个图元会共享它。重用可以节省不少的计算量。

8.3.1 只索引绘制

在你的顶点着色器中可以直接访问到当前顶点的原生索引。这个索引在 SPIR-V 中是以 VertexIndex 修饰的变量，通过 GLSL 中内置的变量 gl_VertexIndex 生成的。这包含了索引缓冲区（）的内容加上传递给 vkCmdDrawIndexed()的 vertexOffset 的值。
你可以使用

8.3.2 重置索引

Vulkan 编程指南翻译 第八章 图形管线 第 4 节 实例化

翻译 2017 年 03 月 29 日 09:29:13

- 187
- 0
- 1

8.4 实例化

vkCmdDraw() and vkCmdDrawIndexed()的两个参数我们已经粗略的讲解过了。它们是 firstInstance and instanceCount，用来控制实例化的。这是一种技术，一个几何对象的很多份复制可以发送到图形管线。每一份复制都被称为一个实例。首先，这看似没有什么用，但是有两种方式可以让你的应用程序在几何物体的每一份实例上做一些改变：
• Use the InstanceIndex built-in decoration on a vertex shader inputto receive the index of

the current instance as an input to theshader. This input variable can then be used to fetch
parameters from a uniform buffer orprogrammatically compute per-instance variation, for
example.
• Use instanced vertex attributes to have Vulkan feed your vertexshader with unique data for
each instance

Listing 8.4 展示了通过 GLSL 内置的变量 gl_InstanceIndex 来使用实例索引的例子。这个例子使用实例化绘制了许多不同的立方体，立方体的每一个实例都有不同的颜色和形变。每一个立方体实例的形变矩阵和颜色都存储在一对 union 缓冲区中。着色器然后可以通过内置变量 gl_InstanceIndex 来索引这个数组。这个着色器渲染的结果在 Figure8.3 展示。

Listing8.4: Using the Instance Index in a Shader

```
#version450 core
layout (set = 0, binding = 0) uniform matrix_uniforms_b
{
mat4 mvp_matrix[1024];
};
layout (set = 0, binding = 1) uniform color_uniforms_b
{
vec4 cube_colors[1024];
};
layout (location = 0) in vec3 i_position;
out vs_fs
{
flat vec4 color;
};
void main(void)
{
float f = float(gl_VertexIndex / 6) / 6.0f;
vec4 color1 = cube_colors[gl_InstanceIndex];
vec4 color2 = cube_colors[gl_InstanceIndex & 512];
color = mix(color1, color2, f);
gl_Position = mvp_matrix[gl_InstanceIndex] * vec4(i_position, 1.0f);
}
```

Figure 8.3: Many Instanced Cubes

Vulkan 编程指南翻译 第八章 图形管线 第 5 节 间接绘制

翻译 2017 年 03 月 29 日 09:30:06

- 234
- 0
- 4

8.5 间接绘制

在 vkCmdDraw() and vkCmdDrawIndexed()命令中，命令的参数(vertexCount,vertexOffset, 等等)以立即参数直接传递给命令本身。这意味这你需要知道在你的应用程序中构建命令缓冲区时每一次绘制调用的准确参数。然而，在一些情况下，你并不知道每一次绘制的准确参数。比如以下例子：

•几何物体的所有结构是已知的，但是顶点的个数和在顶点缓冲区的位置是未知的，一遍一个对象总是参加渲染但是 LoD 的层数会改变。

•绘制命令由设备生成，而非主机。在这个情况下，顶点数据的个数和布局永远被会被主机端所知。

在这些情况下，你可以使用间接绘制，此时，绘制命令可以从设备可访问的内存获取参数，而非把参数随着命令嵌入在命令缓冲区中。第一个间接绘制命令是 vkCmdDrawIndirect()，它执行非索引化绘制，使用的参数包含在一个缓冲区中。它的原型是：

```
voidvkCmdDrawIndirect (
VkCommandBuffer                                    commandBuffer,
VkBuffer
            buffer,
VkDeviceSize
offset,
uint32_t
              drawCount,
uint32_t
              stride);
```

这个命令本身仍然被放入命令缓冲区中，和 vkCmdDraw()是一样的。commandBuffer 是命令被放入的命令缓冲区。然而，命令的参数从 buffer 指定的缓冲区中，offset 为开始的位置开始获取。在这个缓冲区的偏移量位置，应该有一个 VkDrawIndirectCommand 结构，包含了命令真是的参数。VkDrawIndirectCommand 的定义是：

```
typedefstruct VkDrawIndirectCommand {
uint32_t                  vertexCount;
uint32_t                  instanceCount;
uint32_t                  firstVertex;
uint32_t                  firstInstance;
} VkDrawIndirectCommand;
```

VkDrawIndirectCommand 的成员和 vkCmdDraw()命令中的参数有相似的含义。vertexCountand instanceCount 是顶点和索引的个数，firstVertex and firstInstance 是顶点和实例索引的起始值。

vkCmdDrawIndirect()执行非索引化、间接绘制，使用缓冲区对象中数据为参数。也可以使用 vkCmdDrawIndexedIndirect()执行索引化的间接绘制。该命令原型如下：

```
voidvkCmdDrawIndexedIndirect (
VkCommandBuffer                commandBuffer,
VkBuffer                                                buffe
r,
```

```
VkDeviceSize                                    offset,
uint32_t                                        drawC
ount,
uint32_t                                        strid
e);
```
vkCmdDrawIndexedIndirect()的参数和 vkCmdDrawIndirect()的参数是一样的。commandBuffer 是命令被放入的命令缓冲区，buffer 是包含参数的缓冲区对象；offset 是以字节为单位的在缓冲区中参数位置的偏移量，。然而，包含 vkCmdDrawIndexedIndirect()参数的数据结构不一样。它是一个 VkDrawIndexedIndirectCommand 类型的结构，定义是：

```
typedefstruct VkDrawIndexedIndirectCommand {
uint32_t                indexCount;
uint32_t                instanceCount;
uint32_t                firstIndex;
int32_t                    vertexOffset;
uint32_t                firstInstance;
} VkDrawIndexedIndirectCommand;
```
VkDrawIndexedIndirectCommand 的参数和 vkCmdDrawIndexed()中同名的参数有相同的意义。indexCountand instanceCount 是发送到管线的顶点索引和实例的个数；firstIndex 成员指定了从索引缓冲区中开始取出索引的位置，vertexOffset 指定了叠加到索引数据的偏移值；firstInstance 指定了实例计数器起始值。

需要记住的重点是，间接绘制命令是在命令没有被设备执行前，缓冲区对象和便宜值被放入命令缓冲区，绘制所需的参数不必写入到缓冲区对象。随着设备执行命令缓冲区，当进行到该命令时，它会读取缓冲区中的数据并使用，就如同这些参数已经被一个通常的绘制命令指定了一样。就管道的其余部分而言，在直接和间接绘制之间没有区别。

这意味着如下几条：

• You canbuild command buffers with indirect draws long before they’re needed, fillingin the final parameters for the draw (in the buffer object rather than thecommand buffer) before the command buffer is submitted for execution.

• You can create a command buffer containing an indirect draw, submitit, overwrite the parameters in the buffer object, and submitthe same command buffer again. This effectively patches new parameters into what could be along, complex command buffer.

• You can write parameters into a buffer object by using stores froma shader object, or by using a command such as vkCmdFillBuffer() orvkCmdCopyBuffer() to generate drawing parameters on the device itself—either inthe same command buffer or in another submitted just before the one containing the drawcommands.

以可能已经注意到了 vkCmdDrawIndirect() and vkCmdDrawIndexedIndirect() 都接受 drawCountand a stride 参数。这些参数允许你传递绘制命令的数组到 Vulkan。一次 vkCmdDrawIndirect() or vkCmdDrawIndexedIndirect()调用将会激发 drawCount 个独立的绘制，每一次绘制都各从 VkDrawIndirectCommand or VkDrawIndexedIndirectCommand 取出参数。
该数据类型的数组从缓冲区对象内 offset 字节位置开始，每一个结构都和上一个相距 stride 字节。如果 stride 是 0，那么每一次绘制都是用相同的参数。[1]

1.Note that this behavior differs from OpenGL, in which a stride of zero causesthe device to assume a tightly packed array, and it is impossible to source thesame parameters over and over.